

Christian Schulte (Ed.)

LNCS 8124

# Principles and Practice of Constraint Programming

19th International Conference, CP 2013  
Uppsala, Sweden, September 2013  
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, UK*

Takeo Kanade

*Carnegie Mellon University, Pittsburgh, PA, USA*

Josef Kittler

*University of Surrey, Guildford, UK*

Jon M. Kleinberg

*Cornell University, Ithaca, NY, USA*

Alfred Kobsa

*University of California, Irvine, CA, USA*

Friedemann Mattern

*ETH Zurich, Switzerland*

John C. Mitchell

*Stanford University, CA, USA*

Moni Naor

*Weizmann Institute of Science, Rehovot, Israel*

Oscar Nierstrasz

*University of Bern, Switzerland*

C. Pandu Rangan

*Indian Institute of Technology, Madras, India*

Bernhard Steffen

*TU Dortmund University, Germany*

Madhu Sudan

*Microsoft Research, Cambridge, MA, USA*

Demetri Terzopoulos

*University of California, Los Angeles, CA, USA*

Doug Tygar

*University of California, Berkeley, CA, USA*

Gerhard Weikum

*Max Planck Institute for Informatics, Saarbruecken, Germany*

Christian Schulte (Ed.)

# Principles and Practice of Constraint Programming

19th International Conference, CP 2013  
Uppsala, Sweden, September 16-20, 2013  
Proceedings

## Volume Editor

Christian Schulte  
KTH Royal Institute of Technology  
School of Information and Communication Technology  
P.O. Box Forum 120  
16440 Kista, Sweden  
E-mail: cschulte@kth.se

ISSN 0302-9743 e-ISSN 1611-3349  
ISBN 978-3-642-40626-3 e-ISBN 978-3-642-40627-0  
DOI 10.1007/978-3-642-40627-0  
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013947084

CR Subject Classification (1998): F.4.1, G.1.6, F.2.2, F.3, G.2.2, D.3.2, F.1, E.1, I.2.8

LNCS Sublibrary: SL 2 – Programming and Software Engineering

© Springer-Verlag Berlin Heidelberg 2013

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. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

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.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

*Typesetting:* Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

# Preface

This volume contains the proceedings of the 19<sup>th</sup> International Conference on Principles and Practice of Constraint Programming (CP 2013) that was held in Uppsala, Sweden, September 16–20, 2013. Detailed information on the conference can be found on its website [cp2013.a4cp.org](http://cp2013.a4cp.org).

The CP conference is the annual international conference on constraint programming. It is concerned with all aspects of computing with constraints, including theory, algorithms, environments, languages, models, systems, and applications such as decision making, resource allocation, and agreement technologies. Besides the main technical track, CP 2013 featured an application track, which focused on uses of constraint technology and its comparison and integration with other optimization techniques (MIP, local search, SAT, ...).

The interest of the research community in this conference was witnessed by the large number of submissions received this year. We received 170 (long and short) papers as follows: 138 papers submitted to the main track and 32 to the application track. Both long and short papers were reviewed to the same high standards of quality and no long papers were accepted as short papers. The reviewing process for the main track used a two-level Program Committee, consisting of senior Program Committee members and Program Committee members. Senior Program Committee members were responsible for managing a set of papers in their respective areas of expertise. They met in Stockholm during June 8–9, 2013. Each paper received at least three reviews, was extensively discussed, and additional reviews were added when needed. At the end of the reviewing process, we accepted 47 papers for the main technical track and 12 papers for the application track. All papers were presented at the conference.

Amongst the accepted papers, Jimmy Lee, Toby Walsh, and I selected a best technical track paper (“Parallel Discrepancy-Based Search” by Thierry Moisan, Jonathan Gaudreault, and Claude-Guy Quimper), a best application track paper (“Bin Packing with Linear Usage Costs – An Application to Energy Management in Data Centres” by Hadrien Cambazard, Deepak Mehta, Barry O’Sullivan, and Helmut Simonis), and a best student paper (“Filtering AtMostNValue with Difference Constraints: Application to the Shift Minimisation Personnel Task Scheduling Problem” by Jean-Guillaume Fages and Tanguy Lapègue). I am grateful to Jimmy Lee and Toby Walsh for their expert help in the selection process.

The conference program featured three invited talks, an invited public lecture, and an invited system presentation by distinguished scientists. This volume includes abstracts for the invited talks by Michela Milano, Torsten Schaub, and Peter Stuckey and the invited public lecture by Pascal Van Hentenryck. It also includes a paper for the invited system presentation by Pascal Van Hentenryck and Laurent Michel. The conference program included four tutorials: “MaxSAT

Latest Developments” by Carlos Ansótegui, “Replication and Recomputation in Scientific Experiments” by Ian Gent and Lars Kotthoff, “Constraint Programming for Vehicle Routing Problems” by Phil Kilby, and “Constraint Programming for the Control of Discrete Event Dynamic Systems” by Gérard Verfaillie. As an additional part of the program, Peter Stuckey and Håkan Kjellerstrand organized the first international “Lightning Model and Solve Competition”. The winners of the 2013 ACP Research Excellence Award and Doctoral Research Award presented their award talks. Many thanks to all of them for making essential contributions to an exciting conference program!

The conference included a Doctoral Program, which allowed doctoral students to come to the conference, present their work, and meet a mentor with similar research interests. I am very grateful to Christopher Mears and Nina Narodytska for doing a wonderful job in organizing the Doctoral Program.

The conference would not have been possible without the high-quality and interesting submissions from authors, which made the decision process so challenging. I would like to thank the whole Program Committee for the time spent in reviewing papers and in discussions. I am grateful to the additional reviewers, often recruited on very short notice. A special thank you goes to the senior Program Committee members for driving discussions, writing metareviews, and coming to the meeting in Stockholm. I would like to thank Gilles Pesant, who handled papers where I had a conflict of interest.

The conference would not have been possible without the great job done by Mats Carlsson, Pierre Flener, and Justin Pearson as Conference Chairs. They expertly took care of the local organization and I enjoyed our smooth and efficient collaboration. For conference publicity, I very much thank Guido Tack, who did a great job in advertising the conference and for a professional and always up-to-date website. I am very grateful to Laurent Michel, who acted as Workshop and Tutorial Chair and put together an exciting workshop and tutorial program. I am also grateful to Jimmy Lee and Peter Stuckey for sharing their past experiences as CP Program Chairs with me. I gratefully acknowledge local help in sponsoring matters by Karin Fohlstedt, Charlotta Jörsäter, and Victoria Knopf as well as in organizing the physical Senior Program Committee meeting by Sandra Gustavsson Nylén.

The Conference Chairs and I took on the task of soliciting sponsors for CP 2013. We would like to thank our many sponsors for their extraordinarily generous support; they are prominently—and deservedly so—listed in alphabetical order on a following page.

Last but not least, I want to thank the ACP Executive Committee for honoring me with the invitation to serve as Program Chair of CP 2013.

# Conference Organization

## Conference Chairs

|                |                            |
|----------------|----------------------------|
| Mats Carlsson  | SICS, Sweden               |
| Pierre Flener  | Uppsala University, Sweden |
| Justin Pearson | Uppsala University, Sweden |

## Program Chair and Application Track Chair

|                   |                                           |
|-------------------|-------------------------------------------|
| Christian Schulte | KTH Royal Institute of Technology, Sweden |
|-------------------|-------------------------------------------|

## Workshop and Tutorial Chair

|                |                                |
|----------------|--------------------------------|
| Laurent Michel | University of Connecticut, USA |
|----------------|--------------------------------|

## Doctoral Program Chairs

|                   |                                                                                |
|-------------------|--------------------------------------------------------------------------------|
| Christopher Mears | Monash University, Australia                                                   |
| Nina Narodytska   | University of Toronto, Canada, and University<br>of New South Wales, Australia |

## Publicity Chair

|            |                              |
|------------|------------------------------|
| Guido Tack | Monash University, Australia |
|------------|------------------------------|

## Senior Program Committee

|                     |                                               |
|---------------------|-----------------------------------------------|
| Yves Deville        | UCLouvain, Belgium                            |
| Pierre Flener       | Uppsala University, Sweden                    |
| George Katsirelos   | INRA, Toulouse, France                        |
| Christophe Lecoutre | CRIL, University of Artois, France            |
| Jimmy Lee           | The Chinese University of Hong Kong           |
| Amnon Meisels       | Ben-Gurion University of the Negev, Israel    |
| Pedro Meseguer      | IIIA-CSIC, Spain                              |
| Laurent Michel      | University of Connecticut, USA                |
| Barry O'Sullivan    | 4C, University College Cork, Ireland          |
| Gilles Pesant       | École Polytechnique de Montreal, Canada       |
| Michel Rueher       | University of Nice - Sophia Antipolis, France |
| Stefan Szeider      | Vienna University of Technology, Austria      |

|                      |                                  |
|----------------------|----------------------------------|
| Michael Trick        | Carnegie Mellon University, USA  |
| Willem-Jan Van Hoeve | Carnegie Mellon University, USA  |
| Toby Walsh           | NICTA and UNSW, Australia        |
| Roland Yap           | National University of Singapore |

## Technical Program Committee

|                       |                                                     |
|-----------------------|-----------------------------------------------------|
| Fahiem Bacchus        | University of Toronto, Canada                       |
| Chris Beck            | University of Toronto, Canada                       |
| Nicolas Beldiceanu    | École des Mines de Nantes, France                   |
| Christian Bessiere    | Université Montpellier, France                      |
| Mats Carlsson         | SICS, Sweden                                        |
| Hubie Chen            | Universidad del País Vasco and Ikerbasque,<br>Spain |
| Geoffrey Chu          | University of Melbourne, Australia                  |
| Ivan Dotu             | Boston College, USA                                 |
| Thibaut Feydy         | University of Melbourne, Australia                  |
| Ian Gent              | University of St. Andrews, UK                       |
| Alexandre Goldsztejn  | CNRS, France                                        |
| Emmanuel Hebrard      | LAAS, CNRS, France                                  |
| John Hooker           | Carnegie Mellon University, USA                     |
| Said Jabbour          | CRIL, University of Artois, France                  |
| Peter Jeavons         | University of Oxford, UK                            |
| Christopher Jefferson | University of St. Andrews, UK                       |
| Narendra Jussien      | École des Mines de Nantes, France                   |
| Serdar Kadioglu       | Oracle Corporation, USA                             |
| Lars Kotthoff         | University College Cork, Ireland                    |
| Michele Lombardi      | University of Bologna, Italy                        |
| Ines Lynce            | Technical University of Lisbon, Portugal            |
| Christopher Mears     | Monash University, Australia                        |
| Ian Miguel            | University of St. Andrews, UK                       |
| Michela Milano        | University of Bologna, Italy                        |
| Jean-Noël Monette     | Uppsala University, Sweden                          |
| Peter Nightingale     | University of St. Andrews, UK                       |
| Justin Pearson        | Uppsala University, Sweden                          |
| Thierry Petit         | École des Mines de Nantes, France                   |
| Claude-Guy Quimper    | Université Laval, Canada                            |
| Louis-Martin Rousseau | École Polytechnique de Montreal, Canada             |
| Jean-Charles Régim    | Université Nice Sophia Antipolis, France            |
| Ashish Sabharwal      | IBM Research, USA                                   |
| Thomas Schiex         | INRA, France                                        |
| Meinolf Sellmann      | IBM Research, USA                                   |



|                       |                                              |
|-----------------------|----------------------------------------------|
| Paul Shaw             | IBM, France                                  |
| Helmut Simonis        | 4C, Ireland                                  |
| Christine Solnon      | INSA Lyon, France                            |
| Peter Stuckey         | NICTA and University of Melbourne, Australia |
| Guido Tack            | Monash University, Australia                 |
| Peter Van Beek        | University of Waterloo, Canada               |
| Pascal Van Hentenryck | NICTA and University of Melbourne, Australia |
| Brent Venable         | Tulane University, USA                       |
| G rard Verfaillie     | ONERA, France                                |
| Roie Zivan            | Ben-Gurion University of the Negev, Israel   |
| Stanislav  ivn y      | University of Warwick, UK                    |

## Application Track Program Committee

|                          |                                              |
|--------------------------|----------------------------------------------|
| Claire Bagley            | Oracle Corporation, USA                      |
| Pedro Barahona           | Universidade Nova de Lisboa, Portugal        |
| Ken Brown                | University College Cork, Ireland             |
| Hadrien Cambazard        | Grenoble University, France                  |
| Philippe Codognet        | CNRS / UPMC / University of Tokyo, Japan     |
| Sophie Demasse y         |  cole des Mines de Nantes, France            |
| Pierre Flener            | Uppsala University, Sweden                   |
| Maria Garcia de la Banda | Monash University, Australia                 |
| Arnaud Gotlieb           | SIMULA Research Laboratory, Norway           |
| Jimmy Lee                | The Chinese University of Hong Kong          |
| Michele Lombardi         | University of Bologna, Italy                 |
| Laurent Michel           | University of Connecticut, USA               |
| Tomas Eric Nordlander    | SINTEF ICT, Norway                           |
| Barry O’Sullivan         | 4C, University College Cork, Ireland         |
| Federico Pecora          |  rebro University, Sweden                    |
| Laurent Perron           | Google, France                               |
| Michel Rueher            | Universit  Nice Sophia Antipolis, France     |
| Jean-Charles R gin       | Universit  Nice Sophia Antipolis, France     |
| Martin Sachenbacher      | Technische Universit t M nchen, Germany      |
| Pierre Schaus            | UCLouvain, Belgium                           |
| Thomas Schiex            | INRA, France                                 |
| Paul Shaw                | IBM, France                                  |
| Helmut Simonis           | 4C, Ireland                                  |
| Peter Stuckey            | NICTA and University of Melbourne, Australia |
| Guido Tack               | Monash University, Australia                 |
| G rard Verfaillie        | ONERA, France                                |
| Mark Wallace             | Monash University, Australia                 |

## Additional Reviewers

Alejandro Arbelaez  
James Bailey  
Anton Belov  
David Bergman  
Christoph Berkholz  
Christian Bessiere  
Manuel Bodirsky  
Alessio Bonfietti  
Simone Bova  
Simon Brockbank  
Clément Carbonnel  
Gilles Chabert  
Jeff Choi  
Andre Cire  
David Cohen  
Remi Coletta  
Martin Cooper  
Jorge Cruz  
Veronica Dahl  
Alessandro Dal Palù  
Jessica Davies  
Simon de Givry  
Alban Derrien  
Gregory Duck  
Renaud Dumeur  
Uwe Egly  
Stefano Ermon  
Pierre Flener  
María Andreína Francisco Rodríguez  
Maurizio Gabbrielli  
Graeme Gange  
Marco Gavanelli  
Vibhav Gogate  
Arnaud Gotlieb  
Laurent Granvilliers  
Diarmuid Grimes  
Stefano Gualandi  
Evgeny Gurevsky  
Hossein Seyed Hashemi Doulabi  
Patrik Haslum  
Farshid Hassani Bijarbooneh  
Benoît Hoessen  
Marie-José Huguet

Barry Hurley  
Siddhartha Jain  
Mikoláš Janota  
Nicolas Jozefowicz  
Narendra Jussien  
George Katsirelos  
Zeynep Kiziltan  
William Klieber  
Arun Konagurthu  
Marco Kuhlmann  
Uwe Köckemann  
Arnaud Lallouet  
Javier Larrosa  
Yat Chiu Law  
Nadjib Lazaar  
Kevin Leo  
Olivier Lhomme  
Chu-Min Li  
Jerry Lonlac  
Florian Lonsing  
Xavier Lorca  
Jean-Baptiste Mairy  
Terrence W.K. Mak  
Arnaud Malapert  
Yuri Malitsky  
Vasco Manquinho  
Masoumeh Mansouri  
Joao Marques-Silva  
Barnaby Martin  
Nicholas Mattei  
Jacopo Mauro  
Christopher Mears  
Pedro Meseguer  
Claude Michel  
Andrea Micheli  
Michela Milano  
Thierry Moisan  
Eric Monfroy  
Jorge A. Navas  
Samba Ndojh Ndiaye  
Robert Nieuwenhuis  
Todd Niven  
Alexandre Papadopoulos

Federico Pecora  
Laurent Perron  
Justyna Petke  
Karen Petrie  
Nathalie Peyrard  
Cédric Piette  
Maria Silvia Pini  
Charles Prud'Homme  
Claude-Guy Quimper  
Raghuram Ramanujan  
Philippe Refalo  
Jean-Charles Régin  
Florian Richoux  
Roberto Rossi  
Olivier Roussel  
Lakhdar Sais  
András Salamon  
Horst Samulowitz  
Scott Sanner

Prateek Saxena  
Tom Schrijvers  
Andreas Schutt  
Joseph Scott  
Martina Seidl  
Sagar Sen  
Mohamed Siala  
Laurent Simon  
Friedrich Slivovsky  
Kostas Stergiou  
Sebastien Tabary  
Johan Thapper  
Evgenij Thorstensen  
Gilles Trombettoni  
Julien Vion  
Mohamed Wahbi  
Siert Wieringa  
Lebbah Yahia  
Alessandro Zanarini

## Conference Sponsors



We would like to thank our sponsors (in alphabetical order) for their generous support: Association for Constraint Programming, Association Française pour la Programmation par Contraintes, AIMMS, AMPL, Artificial Intelligence – An International Journal, Kjell och Märta Beijers Stiftelse, Cadence, Certus, COSYTEC, FICO, IBM Research, ICS, Jeppesen, KTH Royal Institute of Technology, Microsoft Research – INRIA Joint Centre, NICTA, ONERA, Quintiq, SICS, SINTEF, Trade Extensions, Uppsala University, Världsklass Uppsala, Örebro University.

# Table of Contents

## Invited Talks

|                                                                                                         |   |
|---------------------------------------------------------------------------------------------------------|---|
| Optimization for Policy Making: The Cornerstone for an Integrated Approach . . . . .                    | 1 |
| <i>Michela Milano</i>                                                                                   |   |
| Answer Set Programming: Boolean Constraint Solving for Knowledge Representation and Reasoning . . . . . | 3 |
| <i>Torsten Schaub</i>                                                                                   |   |
| Those Who Cannot Remember the Past Are Condemned to Repeat It . . . . .                                 | 5 |
| <i>Peter J. Stuckey</i>                                                                                 |   |

## Invited Public Lecture

|                              |   |
|------------------------------|---|
| Decide Different! . . . . .  | 7 |
| <i>Pascal Van Hentenryck</i> |   |

## Invited System Presentation

|                                                 |   |
|-------------------------------------------------|---|
| The Objective-CP Optimization System . . . . .  | 8 |
| <i>Pascal Van Hentenryck and Laurent Michel</i> |   |

## Best Technical Track Paper

|                                                                    |    |
|--------------------------------------------------------------------|----|
| Parallel Discrepancy-Based Search . . . . .                        | 30 |
| <i>Thierry Moisan, Jonathan Gaudreault, and Claude-Guy Quimper</i> |    |

## Best Application Track Paper

|                                                                                                     |    |
|-----------------------------------------------------------------------------------------------------|----|
| Bin Packing with Linear Usage Costs – An Application to Energy Management in Data Centres . . . . . | 47 |
| <i>Hadrien Cambazard, Deepak Mehta, Barry O’Sullivan, and Helmut Simonis</i>                        |    |

## Best Student Paper

|                                                                                                                                                    |    |
|----------------------------------------------------------------------------------------------------------------------------------------------------|----|
| Filtering <code>AtMostNValue</code> with Difference Constraints: Application to the Shift Minimisation Personnel Task Scheduling Problem . . . . . | 63 |
| <i>Jean-Guillaume Fages and Tanguy Lapègue</i>                                                                                                     |    |

# Technical Track Papers

|                                                                                                                                              |     |
|----------------------------------------------------------------------------------------------------------------------------------------------|-----|
| A Parametric Approach for Smaller and Better Encodings of Cardinality Constraints .....                                                      | 80  |
| <i>Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell</i>                                                       |     |
| To Encode or to Propagate? The Best Choice for Each Constraint in SAT .....                                                                  | 97  |
| <i>Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Peter J. Stuckey</i>                                     |     |
| Automated Symmetry Breaking and Model Selection in CONJURE.....                                                                              | 107 |
| <i>Ozgur Akgun, Alan M. Frisch, Ian P. Gent, Bilal Syed Hussain, Christopher Jefferson, Lars Kotthoff, Ian Miguel, and Peter Nightingale</i> |     |
| Improving WPM2 for (Weighted) Partial MaxSAT .....                                                                                           | 117 |
| <i>Carlos Ansótegui, Maria Luisa Bonet, Joel Gabàs, and Jordi Levy</i>                                                                       |     |
| MinSAT versus MaxSAT for Optimization Problems.....                                                                                          | 133 |
| <i>Josep Argelich, Chu-Min Li, Felip Manyà, and Zhu Zhu</i>                                                                                  |     |
| Adaptive Parameterized Consistency .....                                                                                                     | 143 |
| <i>Amine Balafrej, Christian Bessiere, Remi Coletta, and El Houssine Bouyakhf</i>                                                            |     |
| Global Inverse Consistency for Interactive Constraint Satisfaction .....                                                                     | 159 |
| <i>Christian Bessiere, Hélène Fargier, and Christophe Lecoutre</i>                                                                           |     |
| Counting Spanning Trees to Guide Search in Constrained Spanning Tree Problems .....                                                          | 175 |
| <i>Simon Brockbank, Gilles Pesant, and Louis-Martin Rousseau</i>                                                                             |     |
| On the Reduction of the CSP Dichotomy Conjecture to Digraphs .....                                                                           | 184 |
| <i>Jakub Bulín, Dejan Delić, Marcel Jackson, and Todd Niven</i>                                                                              |     |
| A Scalable Approximate Model Counter .....                                                                                                   | 200 |
| <i>Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi</i>                                                                             |     |
| Dominance Driven Search .....                                                                                                                | 217 |
| <i>Geoffrey Chu and Peter J. Stuckey</i>                                                                                                     |     |
| Tractable Combinations of Global Constraints .....                                                                                           | 230 |
| <i>David A. Cohen, Peter G. Jeavons, Evgenij Thorstensen, and Stanislav Živný</i>                                                            |     |
| Postponing Optimization to Speed Up MAXSAT Solving .....                                                                                     | 247 |
| <i>Jessica Davies and Fahiem Bacchus</i>                                                                                                     |     |

|                                                                                                                   |     |
|-------------------------------------------------------------------------------------------------------------------|-----|
| Dead-End Elimination for Weighted CSP .....                                                                       | 263 |
| <i>Simon de Givry, Steven D. Prestwich, and Barry O’Sullivan</i>                                                  |     |
| Solving Weighted CSPs by Successive Relaxations .....                                                             | 273 |
| <i>Erin Delisle and Fahiem Bacchus</i>                                                                            |     |
| Constraint-Based Program Reasoning with Heaps and Separation .....                                                | 282 |
| <i>Gregory J. Duck, Joxan Jaffar, and Nicolas C.H. Koh</i>                                                        |     |
| Model Combinators for Hybrid Optimization .....                                                                   | 299 |
| <i>Daniel Fontaine, Laurent Michel, and Pascal Van Hentenryck</i>                                                 |     |
| Modelling Destructive Assignments .....                                                                           | 315 |
| <i>Kathryn Francis, Jorge Navas, and Peter J. Stuckey</i>                                                         |     |
| An Improved Search Algorithm for Min-Perturbation .....                                                           | 331 |
| <i>Alex Fukunaga</i>                                                                                              |     |
| Explaining Propagators for Edge-Valued Decision Diagrams .....                                                    | 340 |
| <i>Graeme Gange, Peter J. Stuckey, and Pascal Van Hentenryck</i>                                                  |     |
| A Simple and Effective Decomposition for the Multidimensional<br>Binpacking Constraint .....                      | 356 |
| <i>Stefano Gualandi and Michele Lombardi</i>                                                                      |     |
| Maintaining Soft Arc Consistencies in BnB-ADOPT <sup>+</sup> during Search....                                    | 365 |
| <i>Patricia Gutierrez, Jimmy H.M. Lee, Ka Man Lei,<br/>Terrence W.K. Mak, and Pedro Mesequer</i>                  |     |
| Solving String Constraints: The Case for Constraint Programming .....                                             | 381 |
| <i>Jun He, Pierre Flener, Justin Pearson, and Wei Ming Zhang</i>                                                  |     |
| Blowing Holes in Various Aspects of Computational Problems, with<br>Applications to Constraint Satisfaction ..... | 398 |
| <i>Peter Jonsson, Victor Lagerkvist, and Gustav Nordh</i>                                                         |     |
| Solving QBF with Free Variables .....                                                                             | 415 |
| <i>William Klieber, Mikoláš Janota, Joao Marques-Silva, and<br/>Edmund Clarke</i>                                 |     |
| Globalizing Constraint Models .....                                                                               | 432 |
| <i>Kevin Leo, Christopher Mears, Guido Tack, and<br/>Maria Garcia de la Banda</i>                                 |     |
| A New Propagator for Two-Layer Neural Networks in Empirical Model<br>Learning .....                               | 448 |
| <i>Michele Lombardi and Stefano Gualandi</i>                                                                      |     |
| Bandit-Based Search for Constraint Programming .....                                                              | 464 |
| <i>Manuel Loth, Michèle Sebag, Youssef Hamadi, and Marc Schoenauer</i>                                            |     |

|                                                                                               |     |
|-----------------------------------------------------------------------------------------------|-----|
| Focused Random Walk with Configuration Checking and Break<br>Minimum for Satisfiability ..... | 481 |
| <i>Chuan Luo, Shaowei Cai, Wei Wu, and Kaile Su</i>                                           |     |
| Multi-Objective Constraint Optimization with Tradeoffs .....                                  | 497 |
| <i>Radu Marinescu, Abdul Razak, and Nic Wilson</i>                                            |     |
| Multidimensional Bin Packing Revisited .....                                                  | 513 |
| <i>Michael D. Moffitt</i>                                                                     |     |
| A Parametric Propagator for Discretely Convex Pairs of SUM<br>Constraints .....               | 529 |
| <i>Jean-Noël Monette, Nicolas Beldiceanu, Pierre Flener, and<br/>Justin Pearson</i>           |     |
| Breaking Symmetry with Different Orderings .....                                              | 545 |
| <i>Nina Narodytska and Toby Walsh</i>                                                         |     |
| Time-Table Extended-Edge-Finding for the Cumulative Constraint .....                          | 562 |
| <i>Pierre Ouellet and Claude-Guy Quimper</i>                                                  |     |
| Revisiting the Cardinality Reasoning for BinPacking Constraint.....                           | 578 |
| <i>François Pelsser, Pierre Schaus, and Jean-Charles Régin</i>                                |     |
| Value Interchangeability in Scenario Generation.....                                          | 587 |
| <i>Steven D. Prestwich, Marco Laumanns, and Ban Kawas</i>                                     |     |
| Embarrassingly Parallel Search .....                                                          | 596 |
| <i>Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert</i>                                |     |
| Multi-Objective Large Neighborhood Search.....                                                | 611 |
| <i>Pierre Schaus and Renaud Hartert</i>                                                       |     |
| Scheduling Optional Tasks with Explanation .....                                              | 628 |
| <i>Andreas Schutt, Thibaut Feydy, and Peter J. Stuckey</i>                                    |     |
| Residential Demand Response under Uncertainty .....                                           | 645 |
| <i>Paul Scott, Sylvie Thiébaux, Menkes van den Briel, and<br/>Pascal Van Hentenryck</i>       |     |
| Lifting Structural Tractability to CSP with Global Constraints .....                          | 661 |
| <i>Eugenij Thorstensen</i>                                                                    |     |
| Empirical Study of the Behavior of Conflict Analysis in CDCL<br>Solvers .....                 | 678 |
| <i>Djamal Habet and Donia Toumi</i>                                                           |     |
| Primal and Dual Encoding from Applications into Quantified Boolean<br>Formulas.....           | 694 |
| <i>Allen Van Gelder</i>                                                                       |     |



|                                                                |     |
|----------------------------------------------------------------|-----|
| Asynchronous Forward Bounding Revisited . . . . .              | 708 |
| <i>Mohamed Wahbi, Redouane Ezzahir, and Christian Bessiere</i> |     |
| Optimizing STR Algorithms with Tuple Compression . . . . .     | 724 |
| <i>Wei Xia and Roland H.C. Yap</i>                             |     |

## Application Track Papers

|                                                                                                        |     |
|--------------------------------------------------------------------------------------------------------|-----|
| Describing and Generating Solutions for the EDF Unit Commitment Problem with the ModelSeeker . . . . . | 733 |
| <i>Nicolas Beldiceanu, Georgiana Ifrim, Arnaud Lenoir, and Helmut Simonis</i>                          |     |
| Solving the Agricultural Land Allocation Problem by Constraint-Based Local Search . . . . .            | 749 |
| <i>Quoc Trung Bui, Quang Dung Pham, and Yves Deville</i>                                               |     |
| Constraint-Based Approaches for Balancing Bike Sharing Systems . . . . .                               | 758 |
| <i>Luca Di Gaspero, Andrea Rendl, and Tommaso Urli</i>                                                 |     |
| Constraint Based Computation of Periodic Orbits of Chaotic Dynamical Systems . . . . .                 | 774 |
| <i>Alexandre Goldsztejn, Laurent Granvilliers, and Christophe Jermann</i>                              |     |
| Laser Cutting Path Planning Using CP . . . . .                                                         | 790 |
| <i>Mikael Z. Lagerkvist, Martin Nordkvist, and Magnus Rattfeldt</i>                                    |     |
| Atom Mapping with Constraint Programming . . . . .                                                     | 805 |
| <i>Martin Mann, Feras Nahar, Heinz Ekker, Rolf Backofen, Peter F. Stadler, and Christoph Flamm</i>     |     |
| Beyond Feasibility: CP Usage in Constrained-Random Functional Hardware Verification . . . . .          | 823 |
| <i>Reuven Naveh and Amit Metodi</i>                                                                    |     |
| Stochastic Local Search Based Channel Assignment in Wireless Mesh Networks . . . . .                   | 832 |
| <i>M.A. Hakim Newton, Duc Nghia Pham, Wee Lum Tan, Marius Portmann, and Abdul Sattar</i>               |     |
| Automatic Generation and Delivery of Multiple-Choice Math Quizzes . . . . .                            | 848 |
| <i>Ana Paula Tomás and José Paulo Leal</i>                                                             |     |

Constrained Wine Blending ..... 864  
*Philippe Vismara, Remi Coletta, and Gilles Trombettoni*

The Berth Allocation and Quay Crane Assignment Problem Using  
a CP Approach ..... 880  
*Stéphane Zampelli, Yannis Vergados, Rowan Van Schaeren,  
Wout Dullaert, and Birger Raa*

**Author Index** ..... 897

# Optimization for Policy Making: The Cornerstone for an Integrated Approach

Michela Milano

DISI, University of Bologna  
V.le Risorgimento 2, 40136, Bologna, Italy

**Abstract.** Policy making is a very complex task taking into account several aspects related to sustainability, namely impact on the environments, health of productive sectors, economic implications and social acceptance. Optimization methods could be extremely useful for analysing alternative policy scenarios, but should be complemented with several other techniques such as machine learning, agent-based simulation, opinion mining and visualization to come up with an integrated system able to support decision making in the overall policy design life cycle. I will discuss how these techniques could be merged with optimization and I will identify some open research directions.

Policy making is the formulation of ideas or plans that are used by an organization or government as a basis for making decisions. Public policy issues cover a wide variety of fields such as economy, education, environment, health, social welfare, national and foreign affairs. They are extremely complex, occur in rapidly changing environments characterized by uncertainty, and involve conflicts among different interests and affect the three pillars of sustainable development, namely society, economy and the environment.

The government of a region or a nation should therefore take complex decisions on the basis of the available data (for example coming from the monitoring of previous policies), of the current economic situation, of the current level of environmental indicators, and on available resources. Basically the planning activity of a policy maker can be easily casted in a multi-criteria combinatorial optimization problem possibly under uncertainty, where Pareto optimal solutions are alternative political scenarios. Each scenario has its own cost and impact on environmental and economic indicators. Depending on the strategic political objectives, the policy maker might prefer one alternative among others. Thus, optimization can play a crucial role for improving the policy making process.

However, optimization is only one - yet important - cornerstone for the improvement of the overall policy making process. There are a number of techniques that could and should be merged with optimization to come up with integrated software tools aiding the policy maker in the overall policy design life cycle. One example is agent-based simulation [3] to mimic the social reaction to policy instruments. Another important technique is opinion mining [5] that basically extracts opinions and sentiments on specific policy topics from blogs and forums enabling e-participation in the policy design. Data mining and machine learning

in general would also be extremely important for processing the always increasing amount of data coming from sensors, extracting relations between these data and the political interventions and possibly insert this extracted model into the optimization model.

Finally, policy makers are not ICT experts and should be aided in the use of the above-mentioned technology. Advanced visualization techniques should play an important role in the human-machine interaction.

Despite a number of research papers have been published in each above mentioned area, what is totally missing at present is a comprehensive tool that assists the policy maker in all phases of the decision making process. The tool should compute alternative scenarios each comprising both a well assessed plan and the corresponding implementation strategies to achieve its objective, its cost and its social acceptance. We need a tool that is able to integrate and consider at the same time global objectives and individual/social reactions. These two aspects could be (and often are) in conflict and possibly game theory could be used to find an equilibrium between the two parts.

During the talk I will present some recent work developed under the EU FP7 project called ePolicyn - *Engineering the policy making life cycle* - aimed at developing decision support systems aiding the policy maker across all phases of the policy making process [1], [4], [2]. The case study will be on the regional energy plan of the Emilia Romagna region of Italy. We will show the different phases of the policy making process and explain where optimization could play a role and how other techniques should be integrated with it.

**Acknowledgment.** The author is partially supported by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n. 288147.

## References

1. Gavanelli, M., Riguzzi, F., Milano, M., Cagnoli, P.: Logic-Based Decision Support for Strategic Environmental Assessment. In: Theory and Practice of Logic Programming, 26th Int'l. Conference on Logic Programming, ICLP 2010, vol. 10(4-6), pp. 643–658 (July 2010), Special Issue
2. Gavanelli, M., Riguzzi, F., Milano, M., Cagnoli, P.: Constraint and optimization techniques for supporting policy making. In: Yu, T., Chawla, N., Simoff, S. (eds.) Computational Intelligent Data Analysis for Sustainable Development, Routledge (2013)
3. Gilbert, N.: Agent based models. Sage Publications Inc. (2007)
4. Milano, M.: Sustainable energy policies: research challenges and opportunities. In: Design, Automation and Test in Europe, DATE, pp. 1143–1148 (2013)
5. Pang, B., Lee, L.: Opinion mining and sentiment analysis. Foundations and Trends in Information Retrieval 2(1-2), 1–135 (2008)

# Answer Set Programming: Boolean Constraint Solving for Knowledge Representation and Reasoning

Torsten Schaub\*

University of Potsdam, Germany  
torsten@cs.uni-potsdam.de

Answer Set Programming (ASP; [1,2,3]) is a declarative problem solving approach, combining a rich yet simple modeling language with high-performance Boolean constraint solving capacities. ASP is particularly suited for modeling problems in the area of Knowledge Representation and Reasoning involving incomplete, inconsistent, and changing information. As such, it offers, in addition to satisfiability testing, various reasoning modes, including different forms of model enumeration, intersection or unioning, as well as multi-criteria and -objective optimization. From a formal perspective, ASP allows for solving all search problems in  $NP$  (and  $NP^{NP}$ ) in a uniform way. Hence, ASP is well-suited for solving hard combinatorial search problems, like system design and timetabling. Prestigious applications of ASP include composition of Renaissance music [4], decision support systems for NASA shuttle controllers [5], reasoning tools in systems biology [6,7,8] and robotics [9,10], industrial team-building [11], and many more. The versatility of ASP is nicely reflected by the ASP solver *clasp* [12], winning first places at various solver competitions, such as ASP, MISC, PB, and SAT competitions. The solver *clasp* is at the heart of the open source platform *Potassco* hosted at [potassco.sourceforge.net](http://potassco.sourceforge.net). Potassco stands for the “Potsdam Answer Set Solving Collection” [13] and has seen more than 30000 downloads world-wide since its inception at the end of 2008.

The talk will start with an introduction to ASP, its modeling language and solving methodology, and portray some distinguished ASP systems.

## References

1. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R., Bowen, K. (eds.) Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP 1988), pp. 1070–1080. MIT Press (1988)
2. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
3. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Answer Set Solving in Practice. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers (2012)

---

\* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

4. Boenn, G., Brain, M., De Vos, M., ffitch, J.: Automatic composition of melodic and harmonic music by answer set programming. In: Garcia de la Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 160–174. Springer, Heidelberg (2008)
5. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-prolog decision support system for the space shuttle. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 169–183. Springer, Heidelberg (2001)
6. Erdem, E., Türe, F.: Efficient haplotype inference with answer set programming. In: Fox, D., Gomes, C. (eds.) Proceedings of the Twenty-Third National Conference on Artificial Intelligence (AAAI 2008), pp. 436–441. AAAI Press (2008)
7. Gebser, M., Schaub, T., Thiele, S., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. *Theory and Practice of Logic Programming* 11(2-3), 323–360 (2011)
8. Gebser, M., Guziolowski, C., Ivanchev, M., Schaub, T., Siegel, A., Thiele, S., Veber, P.: Repair and prediction (under inconsistency) in large biological networks with answer set programming. In: Lin, F., Sattler, U. (eds.) Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR 2010), pp. 497–507. AAAI Press (2010)
9. Chen, X., Ji, J., Jiang, J., Jin, G., Wang, F., Xie, J.: Developing high-level cognitive functions for service robots. In: van der Hoek, W., Kaminka, G., Lespérance, Y., Luck, M., Sen, S. (eds.) Proceedings of the Ninth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010), pp. 989–996. IFAA-MAS (2010)
10. Erdem, E., Haspalamutgil, K., Palaz, C., Patoglu, V., Uras, T.: Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA 2011), pp. 4575–4581. IEEE (2011)
11. Grasso, G., Iiritano, S., Leone, N., Lio, V., Ricca, F., Scalise, F.: An ASP-based system for team-building in the Gioia-Tauro seaport. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 40–42. Springer, Heidelberg (2010)
12. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence* 187–188, 52–89 (2012)
13. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Schneider, M.: Potassco: The Potsdam answer set solving collection. *AI Communications* 24(2), 107–124 (2011)

# Those Who Cannot Remember the Past Are Condemned to Repeat It

Peter J. Stuckey

National ICT Australia, Victoria Laboratory  
Department of Computing and Information Systems,  
University of Melbourne, Australia  
`pstuckey@unimelb.edu.au`

**Abstract.** Constraint programming is a highly successful technology for tackling complex combinatorial optimization problems. Any form of combinatorial optimization involves some form of search, and CP is very well adapted to make use of programmed search and strong inference to solve some problems that are out of reach of competing technologies. But much of the search that happens during a CP execution is effectively repeated. This arises from the combinatorial nature of the problems we are tackling. Learning about past unsuccessful searches and remembering this in an effective way can exponentially reduce the size of the search space. In this talk I will explain lazy clause generation, which is a hybrid constraint solving technique that steals all the best learning ideas from Boolean satisfiability solvers, but retains all the advantages of constraint programming. Lazy clause generation provides the state of the art solutions to a wide range of problems, and consistently outperforms other solving approaches in the MiniZinc challenge.

## 1 Introduction

In the early days of constraint programming there was considerable interest in learning from failure via look-back methods [1] and intelligent backtracking [2]. But this research faded out as propagation approaches proved to be more successful at tackling complex problems [3].

The SAT community revitalized learning, which is now the most critical component in a modern Davis-Putnam-Logemann-Loveland SAT solver, essentially because they devised data structures to efficiently store and propagate hundreds of thousands of learnt nogoods [4]. This technology has been incorporated in constraint programming solvers, first by Katsirelos and Bacchus [5] who used literals of the form  $x = d$  and  $x \neq d$  to represent integer variables. This was extended in the *Lazy Clause Generation* (LCG) approach [6] by using literals of the form  $x \leq d$  and  $x \geq d$ . By storing nogoods that record the reason why a subtree search has failed, constraint programming solvers with learning can *exponentially* reduce the search required to find and prove optimal solutions.

In Lazy Clause Generation each constraint propagator is extended to be able to explain its propagation. Lazy Clause Generation has proved remarkably successful in tackling hard combinatorial optimization problems. It defines the state

of the art complete method in many well studied scheduling problems, such as resource constraint project scheduling (RCPSP) [7], and variations like RCPSP with generalized precedences [8]. LCG has lead to substantial benefits in real life packing problems, such as carpet cutting [9]. LCG solvers have dominated the MiniZinc challenge competition [www.minizinc.org](http://www.minizinc.org) since 2010 (although they are not eligible for prizes) illustrating the approach is applicable over a wide range of problem classes.

In this presentation, I will explain how lazy clause generation solvers work, some of the challenging algorithmic decisions that arise in creating explaining propagators, and some of the important emerging research directions such as *optimized Boolean encoding* [10], *lazy decomposition* [11], and *lifelong learning* [12].

## References

1. Dechter, R.: Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence* 41, 273–312 (1990)
2. Prosser, P.: MAC-CBJ: Maintaining arc consistency with conflict-directed backjumping. Technical Report Research Report 177, University of Strathclyde (1995)
3. Bessiere, C., Regin, 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)
4. 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* (2001)
5. Katsirelos, G., Bacchus, F.: Generalized nogoods in CSPs. In: *Proceedings of the 20th AAAI Conference on Artificial Intelligence, AAAI 2005*, pp. 390–396 (2005)
6. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. *Constraints* 14(3), 357–391 (2009)
7. Schutt, A., Feydy, T., Stuckey, P., Wallace, M.: Explaining the cumulative propagator. *Constraints* 16(3), 250–282 (2011)
8. Schutt, A., Feydy, T., Stuckey, P., Wallace, M.: Solving RCPSP/max by lazy clause generation. *Journal of Scheduling* (2012), online first: <http://dx.doi.org/10.1007/s10951-012-0285-x> (August 2012)
9. Schutt, A., Stuckey, P., Verden, A.: Optimal carpet cutting. In: Lee, J. (ed.) *CP 2011*. LNCS, vol. 6876, pp. 69–84. Springer, Heidelberg (2011)
10. Metodi, A., Codish, M., Stuckey, P.J.: Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. *Journal of Artificial Intelligence Research* 46, 303–341 (2013), <http://www.jair.org/papers/paper3809.html>
11. Abío, I., Stuckey, P.J.: Conflict directed lazy decomposition. In: Milano, M. (ed.) *CP 2012*. LNCS, vol. 7514, pp. 70–85. Springer, Heidelberg (2012)
12. Chu, G., Stuckey, P.J.: Inter-instance nogood learning in constraint programming. In: Milano, M. (ed.) *CP 2012*. LNCS, vol. 7514, pp. 238–247. Springer, Heidelberg (2012)



# Decide Different!

Pascal Van Hentenryck

National ICT Australia, Victoria Laboratory  
Department of Computing and Information Systems,  
University of Melbourne, Australia  
`pvh@nicta.com.au`

We live in a period where Information and Communication Technologies (ICT) has revolutionized the way we communicate, learn, work, and entertain ourselves. But we also live in challenging times, from climate change and natural disasters of increased intensity to rapid urbanization, pollution, economic stagnation, and a shrinking middle class in Western countries. In this lecture, we argue that ICT now has the opportunity to radically change the way we take decisions as a society, exploiting the wealth of data available to understand physical, biological, business, and human behaviors with unprecedented accuracy and speed. We illustrate this vision with challenging problems in disaster management, energy, medicine, and transportation.

**Acknowledgments.** NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

# The Objective-CP Optimization System

Pascal Van Hentenryck<sup>1</sup> and Laurent Michel<sup>2</sup>

<sup>1</sup> NICTA, Australia

<sup>2</sup> University of Connecticut, Storrs, CT 06269-2155

**Abstract.** OBJECTIVE-CP is an optimization system that views an optimization program as the combination of a model, a search, and a solver. Models in OBJECTIVE-CP follow the modeling style of constraint programming and are concretized into specific solvers. Search procedures are specified in terms of high-level nondeterministic constructs, search combinators, and node selection strategies. OBJECTIVE-CP supports fully transparent parallelization of multi-start and branch & bound algorithms. The implementation of OBJECTIVE-CP is based on a sequence of model transformations, followed by a concretization step. Moreover, OBJECTIVE-CP features a constraint-programming solver following a micro-kernel architecture for ease of maintenance and extensibility. Experimental results show the practicability of the approach.

## 1 Introduction

This paper presents an overview of OBJECTIVE-CP, an optimization system written in OBJECTIVE-C (an object-oriented layer on top of C). OBJECTIVE-CP builds on more than two decades of research on the design and implementation of constraint-programming systems, from CHIP to systems such as ILOG SOLVER, OPL, ILOG CONCERT, COMET, GECODE, and MINIZINC which have probably had the strongest influence on its design and implementation. The design of OBJECTIVE-CP takes the the view that

Optimization Program = Model + Search + Solver

or, in other words, that an optimization program consists of a model, a search, and an underlying constraint solver.

Models are first-class objects in OBJECTIVE-CP; they also follow the style of constraint programming and are solver-independent. This allows for easy experimentation with different technologies and smooth hybridizations [5,4]. Models can be concretized into a specific solver to obtain an optimization program, (e.g., a constraint programs or a mixed-integer program). The resulting optimization program can be solved using a black-box search or a dedicated search procedure expressed in terms of the model variables. Search procedures in OBJECTIVE-CP are specified in terms of high-level nondeterministic constructs, search combinators, and node selection strategies, merging the benefits of search controllers and continuations [22] on the one hand and compositional combinators (e.g., [17]) on the other hand. The search language is generic and independent of

the underlying solver, although obviously search procedures call the underlying solver for adding constraints, binding variables, and querying the search state. OBJECTIVE-CP transparently supports the parallelization of optimization programs, supporting parallel multi-start algorithms and parallel branch & bound.

The implementation of OBJECTIVE-CP performs a series of model transformations, including a flattening of the model, followed by a concretization of the final model into a specific solver. OBJECTIVE-CP also features a constraint-programming solver inspired by the micro-kernel approach to operating systems. It features small components, such as a propagation engine, a variable library, and a constraint library, that are separated and have minimal interfaces.

It is difficult to summarize the contributions of a large system. However, the following features of OBJECTIVE-CP are worth highlighting:

1. OBJECTIVE-CP enables the model and the search to be expressed in terms of the model variables, although the model can be concretized into different solvers;
2. OBJECTIVE-CP offers a rich, generic search language. The search language is independent of the underlying solver and merges the benefits of two high-level approaches to search: search controllers and search combinators. In particular, OBJECTIVE-CP provides a small set of abstractions that naturally combine to build complex search procedures.
3. OBJECTIVE-CP achieves a strong symbiosis with the underlying host language, i.e., OBJECTIVE-C. In particular, it allows for an iterative style in search procedures and makes heavy use of closures and first-order functions.
4. OBJECTIVE-CP provides first-class models which make it possible to offer model combinators and an implementation approach based on model transformations and concretizations.
5. OBJECTIVE-CP provides an automatic and transparent parallelization of optimization programs, even when the program feature a search procedure (i.e., not a black-box search).
6. OBJECTIVE-CP features a constraint-programming solver based on the concept of micro-kernel in operating systems, i.e., it strives to define small components with minimal interfaces.

This paper reviews the design and implementation of OBJECTIVE-CP. Section 2 gives a brief overview of OBJECTIVE-C. Section 3 then presents an overview of OBJECTIVE-CP, including models, search, and transparent parallelization. The implementation methodology and the experimental results are presented in Sections 4–5. Sections 6–7 discuss the related work and the conclusion.

## 2 The Host Language

OBJECTIVE-CP is written on top of OBJECTIVE-C, a high-level programming language that adds an object-oriented layer on top of C. OBJECTIVE-C marries the elegance of a fully dynamic object-oriented runtime based on dynamic message dispatching (a la SMALLTALK with the performance of C). OBJECTIVE-C

features syntactic extensions over  $C$  to model classes (called `@interface`), interfaces (called `@protocol`) and categories that provide the ability to extend the API of a class with new methods without requiring access to the source code. OBJECTIVE-C inherits the static typing of  $C$ , yet it offers the ability to be loosely typed for the object-oriented extensions. The SMALLTALK heritage is significant. For instance, OBJECTIVE-C separates the notion of message and response behavior. It also offers introspection, message interception, and rerouting. The syntax of OBJECTIVE-C may seem peculiar at first. A method call `cp.label(x)` is written as `[cp label: x]` where `label:` is the method name. When using multiple arguments, OBJECTIVE-C “names” each of them. For instance, a method call `cp.labelWith(x,v)` could become `[cp label: x with: v]`, where the method name is `label:with:`. OBJECTIVE-C also features closures and first-order functions. For instance,

---

```
1 [cp onSolution: ^{printf('found a solution \n')};];
```

---

uses a closure `^{printf('found a solution \n')}` which can be called subsequently. The snippet

---

```
1 [S enumerateWithBlock:^(int i) {printf('%d ',i); }];
```

---

depicts the use of a first-order function. The code enumerates the elements of set  $S$  and calls the first-order function passing each element to parameter  $i$ . The body of the function prints the value of the set elements. Overall, OBJECTIVE-C is a nice compromise between the flexibility of SMALLTALK and the efficiency of  $C$ . It is particularly well-adapted for developing complex systems.

### 3 The Design of Objective-CP

This section reviews the design of OBJECTIVE-CP and its main concepts: models, programs, search procedures, and transparent parallelization. The focus is on introducing the concepts informally and conveying a sense of the global design. Model composition is covered in detail in [4].

#### 3.1 Models

Figure 1 illustrates several features of OBJECTIVE-CP: It depicts a program which solves a capacitated warehouse location problem with a CP solver and a MIP solver. Lines 1–25 declare a model (line 1), its data (lines 2–7 where constants are omitted), its decision variables (lines 9–14), its constraints (lines 16–23), and its objective function (lines 24–25). The capacity constraints in line 17 feature reified constraints, while lines 21–22 feature element constraints to link the warehouse and store variables and to compute the transportation cost for each store. The objective function is stated in lines 24–25 and sums the fixed and transportation costs. This is a standard constraint-programming model for this problem, the only peculiarity being the syntax of OBJECTIVE-C.

---

```

1 id<ORModel> model = [ORFactory createModel];
2 ORInt fixed = ...;
3 ORInt maxCost = ...;
4 id<ORIntRange> Stores      = ...;
5 id<ORIntRange> Warehouses = ...;
6 ORInt* cap = ...;
7 ORInt** tcost = ...;
8
9 id<ORIntVarArray> cost =
10   [ORFactory intVarArray: model range:Stores domain: RANGE(model,0,maxCost)];
11 id<ORIntVarArray> supp =
12   [ORFactory intVarArray: model range:Stores domain: Warehouses];
13 id<ORIntVarArray> open =
14   [ORFactory intVarArray: model range:Warehouses domain: RANGE(model,0,1)];
15
16 for(ORInt i=Warehouses.low;i <= Warehouses.up;i++)
17   [model add: [Sum(model,s,Stores,[supp[s] eq: @(i)]) leq: @(cap[i])]];
18 for(ORInt i=Stores.low;i <= Stores.up;i++) {
19   id<ORIntArray> row = [ORFactory intArray: model range: Warehouses with:
20     ^ORInt(ORInt j) { return tcost[i][j];}];
21   [model add: [[open elt: supp[i] eq: @1]];
22   [model add: [cost[i] eq: [row elt: supp[i]]]];
23 }
24 [model minimize: Sum(model,s,Stores,cost[s]) plus:
25   Sum(model,w,Warehouses,[@(fixed) mul: open[w]])];
26
27 id<CPPProgram> cp = [ORFactory createCPPProgram: model];
28 id<MIPProgram> mip = [ORFactory createMIPProgram: model];
29 [cp solve];
30 [mip solve];

```

---

**Fig. 1.** Capacitated Warehouse Location in OBJECTIVE-CP

This model is a specification and cannot be executed. No data structures are allocated for the variables, the constraints, and the objectives. For instance, a variable contains its domains but not in a form that can be used for computation and a constraint only collects its variables or expressions. This is similar to models in modeling languages and in the ILOG CONCERT library.

Models are first-class objects in OBJECTIVE-CP: They can be cloned and transformed and it is possible to retrieve their variables, constraints, and objective. They also support the definition of model combinators, an abstraction to build hybrid optimization algorithms compositionally [4].

### 3.2 Programs

To execute a model in OBJECTIVE-CP, it is necessary to create an optimization program. Lines 27–28 from Figure 1

---

```

1 id<CPPProgram> cp  = [ORFactory createCPPProgram: model];
2 id<MIPProgram> mip = [ORFactory createMIPProgram: model];

```

---

create a CP and a MIP program. In other words, these lines concretize the warehouse location model into two executable programs. These two programs are then solved in lines 29–30 with both technologies.

The implementation of lines 27–28 involves a series of model transformations followed by an actual concretization which creates the solver variables, constraints, and objectives. Like in modern modeling languages such as MINIZINC,

---

```

1 id<ORModel> m = [ORFactory createModel];
2
3 // data declarations and reading
4
5 id<ORIntVarArray> slab =
6   [ORFactory intVarArray: m range: SetOrders domain: Slabs];
7 id<ORIntVarArray> load =
8   [ORFactory intVarArray: m range: Slabs domain: Capacities];
9
10 [m add: [ORFactory packing: slab itemSize: weight load: load]];
11 for(ORInt s = Slabs.low; s <= Slabs.up; s++)
12   [m add:[Sum(m,c,Colors,Or(m,o,coloredOrder[c],[slab[o] eq:@(s))]) leq:@2]];
13 [m minimize: Sum(m,s,Slabs,[loss elt: [load at: s]]]);
14
15 id<CPPProgram> cp = [ORFactory createCPPProgram: m];
16
17 [cp solve: ^{
18   for(ORInt i = SetOrders.low; i <= SetOrders.up; i++) {
19     ORInt ms = max(0,[cp maxBound: slab]);
20     [cp tryall: Slabs suchThat: ^bool(ORInt s) { return s <= ms+1; }
21       in: ^void(ORInt s) { [cp label: slab[i] with: s]; }
22   ];
23 }
24 }}

```

---

**Fig. 2.** The Steel Mill Slab Problem in OBJECTIVE-CP

models are flattened to avoid potential redundant normalizations of expressions, constraints and objectives by several solvers. Solvers can then be kept small and compact. The flattened model must also be linearized for MIP solvers that do not support reification and element constraints. Observe that a model can be concretized multiple times as Figure 1 demonstrates. This feature is useful when designing hybrid optimization models such as those implemented in CML [5], to implement multi-start solvers, or portfolios of algorithms.

### 3.3 Search

The warehouse location programs use black-box searches. However, OBJECTIVE-CP lets optimizers state their own search algorithms. Figure 2 depicts a program for the steel mill slab problem (the data declaration and reading are omitted for brevity). The model (lines 5–13) is the same as in [6,8]: Line 10 states a global packing constraint and line 12 imposes reified constraints for the color requirements. The objective function in line 13 features element constraints. The model is concretized into a CP program on line 15. The search procedure is specified in lines 18–24: It combines a C for-loop with a nondeterministic `tryall` instruction (first introduced in OPL [20]). The search procedure breaks value symmetries on the slabs as in [8].

There are three features of OBJECTIVE-CP that deserve to be highlighted here. First, an optimization program in OBJECTIVE-CP is the combination of a propagation engine and a (solver-independent) search explorer as shown in Figure 3. It follows that the search in OBJECTIVE-CP is generic: It only features generic abstractions such as search controllers, nondeterministic constructs, and combinators, which do not depend on a particular solver technology. Once again,

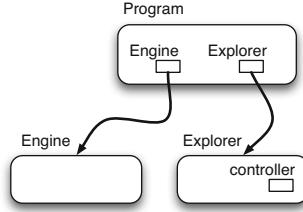


**Fig. 3.** The Structure of a CP Program in OBJECTIVE-CP

this is an interesting property from a software engineering standpoint since it provides reusability and separation of concerns.

Second, the search procedure operates on the model variables, not on concrete variables of the underlying solver (in contrast to ILOG CONCERT). Programmers in OBJECTIVE-CP do not need to know the existence of concrete variables: The optimization program is responsible for performing the mapping from model to concrete variables.

Third, OBJECTIVE-CP achieves a strong symbiosis with its host language, since native iterative control structures in OBJECTIVE-C (e.g., for-loops) and non-deterministic constructs in OBJECTIVE-CP (e.g., `tryall`) are interleaved to implement search procedures. It is useful to dive into a simple example to show this symbiosis. The excerpt

---

```

1 -(void) labelArray: (id<ORIntVarArray>) x {
2   for (int i=x.range.low; i <= x.range.up; i++)
3     while (![cp bound: x[i]]) {
4       int v = [cp min: x[i]];
5       [cp try: ^{ [cp label:x[i] with:v]; } or: ^{ [cp diff:x[i] with: v]; }];
6     }
7 }

```

---

depicts how to label an array of variables in OBJECTIVE-CP. Line 2 iterates over the indices of the array, while lines 3–6 label variable `x[i]`. The variable labeling is a while-loop that terminates when `x[i]` is bound. Each iteration takes the smallest value in the domain of `x[i]` and either assigns it to `x[i]` or remove it from its domain. This nondeterministic step uses the `try` nondeterministic construct ([20,3]). What is interesting in this snippet is the ability to use a for-loop over local variables (e.g., `i` and `v`) and to combine it with nondeterminism. This is possible because, when the OBJECTIVE-CP implementation selects an alternative to the `try` instruction, it restores the values of `i` and `v` to their states before the execution of the `try`. This symbiosis enables an iterative style for writing search procedures that avoid the need to use goals and the *and* combinator in [10,17]. This symbiosis provides significant software engineering benefits, including a small and modular implementation, and the ability to avoid decoupling the syntactic structure of the search and the control flow of the execution. Even more important perhaps are the benefits for debugging. Users can place breakpoints anywhere in the code and use native debugging tools to inspect the computation state. With goals or combinators for sequential composition, users,

when debugging, are in the interpreter code and have lost the connection to the original control flow.

The design of the search component in OBJECTIVE-CP is based on the motto

Search = Continuations + Controllers

or, in other words, the belief that a rich and extensible search language can be built bottom-up from continuations and search controllers. These abstractions make it possible to offer a rich collection of high-level nondeterministic constructs and search combinators. Moreover, controllers can also be used to specify node selection strategies. As a result, OBJECTIVE-CP unifies the search controllers of COMET [22] and the search combinators of [17] (and hence the goals of ILOG SOLVERS). We illustrate a few of these combinators and present a large neighborhood search based on them.

*Limits.* Limit combinators (e.g., [14]) impose conditions on a search procedure or, equivalently, on the exploration of a search subtree. These limits may concern a variety of measures including computation times and the number of solutions, as well as any condition provided by users. For instance,

---

```
1 [cp limitSolutions: k in: ^{ [cp labelArray: x]; }];
```

---

ensures that only  $k$  labelings of array  $x$  are returned. Once the  $k$ -th labeling has been obtained, the instruction fails. Limits can be composed naturally as in

---

```
1 [cp limitFailures: 200 in ^{
2   [cp limitSolutions: k in: ^{ [cp labelArray: x]; }];
3 }];
```

---

The resulting code also limits the number of failures in the labeling to 200.

*The Repeat Combinator.* The repeat combinator repeatedly executes a body until it succeeds, interleaving each iteration with the execution of a closure. In Prolog, it would be written as

---

```
1 repeat (Body, Restart) :- call(Body).
2 repeat (Body, Restart) :- call(Restart), repeat (Body, Restart).
```

---

It is particularly useful with various forms of randomization and local search algorithms. For instance, the fragment

---

```
1 [cp repeat: ^{ [cp limitFailures: nbFailures.value
2               in: ^{ [cp labelHeuristic: h]; } ] }
3   onRepeat: ^{ [nbFailures setValue: nbFailures.value * f]; }
4 }];
```

---

depicts a (randomized) search procedure with restarts. In this example, each iteration is allowed `nbFailures.value` failures and the number of failures is increased geometrically after each search. The same excerpt can be transformed in a generic restart combinator that repeatedly executes a body until a condition `c` is met and executes `restart` when the body fails, i.e.,



---

```

1 [cp solve: ^{
2   [cp limitTime: timeLimit in: ^{
3     [cp repeat: ^{
4       [cp limitFailures: failureLimit in: ^{
5         for(ORInt i = SetOrders.low; i <= SetOrders.up; i++) {
6           ORInt ms = max(0, [cp maxBound: slab]);
7           [cp tryall: Slabs suchThat: ^bool(ORInt s) { return s <= ms+1; }
8             in: ^void(ORInt s) {
9               [cp label: slab[o] with: s];
10            }];
11          }
12        }];
13      }
14      onRepeat: ^{
15        id<ORSolution> s = [[cp solutionPool] best];
16        if (s != nil)
17          [Orders enumerateWithBlock: ^ (ORInt i) {
18            if ([d next] <= Pr)
19              [cp label: x[i] with: [s intValue:x[i]]];
20          }];
21      }];
22    }];
23  }];

```

---

**Fig. 4.** A Large Neighborhood Search for the Steel Mill Slab Problem

---

```

1 -(void) restart: (void ^()) body when: (BOOL ^()) c onRestart: (void ^()) restart
2 {
3   [self repeat: [self limitCondition: c in: body()]; onRepeat: ^{restart()};];
4 }

```

---

The earlier restart search, together with a time limit, can be written

---

```

1 [cp limitTime: 300 in: ^ {
2   [cp restart: ^{ [cp labelHeuristic: h]; }
3     when: ^BOOL() { return cp.nbFailures > nbFailures.value; }
4     onRestart: ^{ [nbFailures setValue: nbFailures.value * f]; }
5   ];
6 }];

```

---

These code fragments demonstrate compositionality. Limit, repeat, and non-deterministic combinators are interleaved in arbitrary ways.

*Large Neighborhood Search.* Figure 4 illustrates how search combinators can be used to implement a large neighborhood search (LNS) in OBJECTIVE-CP. The LNS search uses two limit combinators and a repeat combinator to transform the core search depicted previously into a large neighborhood search. The outermost limit combinator ensures that the overall search does not exceed the time limit. In the search, the body of the repeat combinator is the basic search (lines 5–10) presented in Figure 2, enclosed in a limit combinator that limits the number of failures (line 4). The restart code in lines 15–20 defines the neighborhood structure: It takes the best solution found so far, and fixes the value of a variable in this solution with probability  $Pr$  ( $d$  is a uniform distribution in  $[0,1]$ ).

*Implementation of Search Combinators.* How does OBJECTIVE-CP implement combinators compositionally? *The key is to use controllers and to support the*

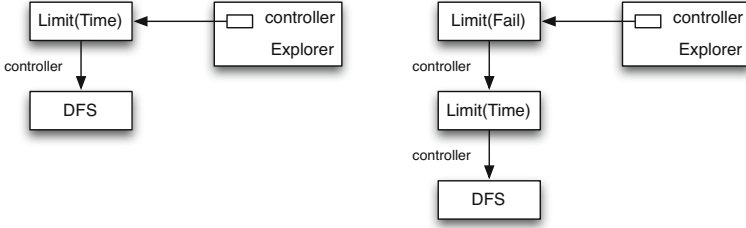


Fig. 5. Chaining Controllers

---

```

1 -(void) limitFailures: (int) max in: (void^()) body
2 {
3     id<ORSearchController> c = [[ORLimitFailures alloc] initWithLimit: max];
4     [self pushViewController: c];
5     body();
6     [c succeeds];
7     [self popController];
8 }

```

---

Fig. 6. The Implementation of a Limit Combinator

*ability to chain controllers.* Figure 6 depicts the implementation of the limit combinator for failures. Lines 3–4 create a limit controller (line 3) and push it as the top-level controller (line 4). Line 7 pops it once the body of the combinator is executed. Figure 5 depicts the chains of controllers for one and two limits, highlighting the compositionality of the approach. The effect of the `pushViewController` instruction is undone upon backtracking or, more generally, when the search jumps to other nodes in the search tree. The ability to chain controllers is the key for the modular and compositional design of search combinators. A controller for limiting the number of solutions is shown in Figure 7. It maintains the number of solutions produced so far (`_nbSol`), which is updated every time the body succeeds (line 11). Methods `startTryLeft` and `startTryRight` tests whether the limit is reached, in which case it fails. Otherwise, they delegate the call to the subcontroller, i.e., the next controller `_controller` in the chain. All other methods in the controller are inherited from the superclass and simply delegate the call to the subcontroller. Figure 8 depicts the implementation of the `repeat` combinator. Lines 2–3 create a continuation and a choice point. The first execution runs the body (line 6). Later executions creates a choice point, executes the `onRepeat` code, then the body.

### 3.4 Transparent Parallelization in Objective-CP

OBJECTIVE-CP supports fully transparent parallelizations of its programs. For instance, the concretization

---

```

1 id<CPPProgram> cp = [ORFactory createCPPProgram: model];

```

---

can be replaced by

---

```

1 id<CPPProgram> cp = [ORFactory createCPMultiStartProgram: model nb: 4];

```

---

---

```

1 @implementation ORLimitSolution {
2     int _max;
3     int _nbSol;
4 }
5 -(ORLimitSolution*) initWithLimit: (int) m
6 {
7     self = [super init];
8     _max = m; _nbSol = 0;
9     return self;
10 }
11 -(void) succeeds      { _nbSol++;}
12 -(BOOL) hasReachedLimit { return _nbSol >= _max; }
13 -(void) startTryLeft
14 {
15     if ([self hasReachedLimit]) [_controller fail];
16     else return [_controller startLeft];
17 }
18 -(void) startTryRight
19 {
20     if ([self hasReachedLimit]) [_controller fail];
21     else return [_controller startRight];
22 }
23 @end

```

---

**Fig. 7.** The Limit Solution Controller

---

```

1 -(void) repeat: (ORClosure) body onRepeat: (ORClosure) onRepeat {
2     NSCont* enter = [NSCont takeContinuation];
3     [_controller._val addChoice: enter];
4     if ([enter nbCalls]!=0)
5         if (onRepeat) onRepeat();
6     body();
7 }

```

---

**Fig. 8.** The Implementation of the `repeat` Combinator

to obtain a multistart search procedure capable of executing four searches on four threads with different random seeds: *The model and the search procedure are left unchanged.* At the implementation level, OBJECTIVE-CP concretizes the model four times to obtain four different CP programs. The search is executed on each CP program. Since the search is expressed in terms of the model objects, a particular thread executing the search will retrieve its concrete solver from the multistart program and then access the concrete objects in that solver.

OBJECTIVE-CP also supports a full transparent parallelization of a branch & bound search using a work-stealing model. For instance, the code

---

```

1 id<CPPProgram> cp = [ORFactory createParCPPProgram: model];

```

---

creates a parallel branch & bound that can execute search algorithms such as those described previously. These algorithms can be defined in terms of search combinators and nondeterministic constructs. The parallel implementation follows the computational model described in [11,12] which exploits search controllers (see Figure 9). A parallel constraint program consists of a set of workers, a problem pool, and a template to create the node selection strategy. Each worker is a solver with its own engine and search explorer. The explorer has its traditional chain of controllers and a parallel adapter that encapsulates two



$$\begin{aligned}
\tau_f(\langle X, \{C_1, \dots, C_k\}, O \rangle) &= \langle X \cup X', C', O' \rangle \quad \text{where} \\
\tau_c(C_i) &= \langle C'_i, X'_i \rangle \quad (1 \leq i \leq k), \\
\tau_e(O) &= \langle O', C_O, X_O \rangle, \\
X' &= \left( \bigcup_{i=1}^k X'_i \right) \cup X_O, \\
C' &= \left( \bigcup_{i=1}^k C'_i \right) \cup C_O. \\
\tau_e(e_l * e_r) &= \langle z, C, X \rangle \quad \text{where} \\
\tau_e(e_l) &= \langle z_l, C_l, X_l \rangle, \\
\tau_e(e_r) &= \langle z_r, C_r, X_r \rangle, \\
C &= \{\text{mult}(z_l, z_r, z)\} \cup C_l \cup C_r, \\
X &= \{z, z_l, z_r\} \cup X_l \cup X_r. \\
\tau_c(\text{alldifferent}(e_1, \dots, e_n)) &= \langle C, X \rangle \quad \text{where} \\
\tau_e(e_i) &= \langle z_i, C_i, X_i \rangle \quad (1 \leq i \leq n) \\
C &= \{\text{alldifferent}(z_1, \dots, z_n)\} \cup \bigcup_{i=1}^n C_i \\
X &= \{z_1, \dots, z_n\} \cup \bigcup_{i=1}^n X_i.
\end{aligned}$$

**Fig. 11.** Excerpts of The Flattening Transformation in OBJECTIVE-CP

---


$$\begin{array}{l}
1 \quad \tau_{MIP}(\text{alldifferent}(x_1, \dots, x_n)) = \{ \\
2 \quad \quad \llbracket x_1 = 1 \rrbracket + \dots + \llbracket x_n = 1 \rrbracket \leq 1, \\
3 \quad \quad \vdots \\
4 \quad \quad \llbracket x_1 = k \rrbracket + \dots + \llbracket x_n = k \rrbracket \leq 1, \\
5 \quad \}
\end{array}$$


---

**Fig. 12.** A Linearization of the Alldifferent Constraint

in OBJECTIVE-CP in terms of the function  $\tau_f$ ,  $\tau_c$ , and  $\tau_e$  to flatten a model, a constraint, and an expression. In these rules, the  $z_i$ 's are brand new variables not used anywhere in the model. For instance, the figure illustrates the flattening of the **alldifferent** constraint, which flattens the expressions  $e_1, \dots, e_n$  to obtain the variables  $z_1, \dots, z_n$ , the constraints  $C_1, \dots, C_n$ , and the new variables  $X_1, \dots, X_n$ . The resulting **alldifferent** constraint is solely expressed in terms of variables, not expressions. Finally,  $\tau_e$  illustrates the flattening of a multiplication expression to obtain very simple constraints in the solver.

Models can be linearized for use in a MIP solver and there is considerable literature on how to perform such transformations (e.g., [16,9]). Figure 12 describes the linearization of the **alldifferent** constraint assuming that the variables take values in  $1..k$ . The linearization defines a number of inequalities over the literals  $\llbracket x_i = j \rrbracket$ , i.e., 0/1 variables that denote whether  $x_i$  is assigned the value  $j$ . The linearization of the variables also generates constraints of the form

$$x_i = \sum_{j=1}^k j * \llbracket x_i = j \rrbracket.$$

As suggested in [16], these constraints may be enforced lazily when the model only uses the literals.

As indicated earlier, models are first-class objects in OBJECTIVE-CP and the implementation can include code of the form

---

```
id<ORModel> lfm = [[m copy] flatten] linearize];
```

---

Another benefit of the OBJECTIVE-CP architecture is the factorization of transformations (flattenings, normalizations, and linearizations) across multiple solvers. Solvers do not have to be concerned with these transformations which are now performed at the model level. Obviously it does not mean that solvers cannot manipulate constraint globally as is necessary, say in hull and box-consistency [1,23]. Rather it simply means that the transformations provide a normalization of the model expressions in a form appropriate for the solvers.

## 4.2 Concretization

The concretization  $\gamma$  takes a model  $m$  in a flattened form appropriate for a solver  $s$  and concretizes  $m$  into  $s$ , i.e.,  $s = \gamma(m)$ . The concretization associates a concrete variable with every model variable and a concrete object to every model object (e.g., a constraint). The concretization  $\gamma$  is used at various places. For instance, an instruction

---

```
[cp label: x with: v]
```

---

that labels variable  $x$  with value  $v$  is implemented by a call

---

```
[engine label:  $\gamma(x)$  with: v]
```

---

that concretizes variable  $x$  and calls the same method on the constraint engine. The literals of variable  $x$  can be accessed through

---

```
[cp literal: x for: v]
```

---

which returns  $\gamma(\llbracket x = v \rrbracket)$ . The instruction

---

```
[parcp label: x with: v]
```

---

in a parallel solver is implemented by the call

---

```
[cpk label: x with: v]
```

---

which itself becomes

---

```
[enginek label:  $\gamma_k(x)$  with: v]
```

---

where  $\text{cp}_k$  is the  $k^{\text{th}}$  solver,  $\text{engine}_k$  is its engine, and  $\gamma_k$  is its concretization.

Adding a constraint  $c$  during the search requires some care, since these constraints are expressed in terms of the original model. The optimization program must preserve the chain of transformations  $\tau_1, \dots, \tau_k$  and apply them to constraint  $c$  to obtain a tuple  $\langle X, C \rangle$ , where  $X$  is a set of new variables and  $C$  is a set of new constraints. Both  $X$  and  $C$  can now be concretized through  $\gamma$  and posted in the solver. In other words, the addition of a constraint  $c$  executes

$$\gamma(\tau_n(\dots(\tau_2(\tau_1(\{\{ \}, \{c\}\}))\dots)).$$

---

```

1 @protocol CPEngine
2 -(ORStatus)      add: (id<CPConstraint>) c;
3 -(void) setObjective: (id<ORObjective>) obj;
4 @end

```

---

**Fig. 13.** The CPEngine Interface

---

```

1 @protocol CPUSKernel
2 -(void) scheduleCtrEvt: (id<CPCtrEvent>) list;
3 -(void) scheduleValEvt: (id<CPValEvent>) list;
4 -(void) triggerLossEvt: (id<CPTriggerMap>) map;
5 -(void) triggerBindEvt: (id<CPTriggerMap>) map;
6 -(ORStatus) propagate;
7 @end

```

---

**Fig. 14.** The Micro-Kernel Interface

### 4.3 A Micro-Kernel Architecture

The constraint-programming solver of OBJECTIVE-CP is based on a micro-kernel architecture inspired by ideas from operating systems. Micro-kernel architectures have become popular in operating systems as they favor extensibility, maintenance, and easier proofs of correctness. This section briefly reviews the main ideas behind the micro-kernel architecture.

The CP solver in OBJECTIVE-CP is only concerned with constraint propagation, as the search in OBJECTIVE-CP is solver-independent and lies in a separate library. The solver itself consists of two objects: an *engine* that defines the API to add constraints and objectives and a *kernel* that implements the propagation and provide minimal functionalities to define new propagators. Their interfaces are sketched in Figures 13 and 14.

The engine interface is used to register native constraints and objectives. No decomposition or rewriting is necessary at this stage, since these transformations took place earlier in the pipeline, The constraint interface

---

```

1 @protocol CPConstraint
2 -(ORUInt) getId;
3 -(ORStatus) post;
4 @end

```

---

is minimalist and only requires each constraint to be uniquely identified and to support a `post` method. Objective function are similar and abstract away the nature of the objective function.

The micro-kernel provides the interface to schedule and propagate events. It supports four types of events: constraint, variable, value, and trigger events. Intuitively, constraint events are used to propagate a constraint, variable events to execute a closure (e.g., upon a variable modification), value events to apply a first-order function (e.g., when a value is removed from a domain), and trigger events to update triggers. For space reasons, we do not discuss triggers in the rest of the paper: It suffices to say that they provide functionalities related to the watched literals in MINION [7]. The interfaces `CPCtrEvent` and `CPValEvent` represent simple lists of constraints, closures, and first-order function applications (`CPCtrEvent` contains both constraints and variable events for efficiency

and simplicity reasons). Observe that the micro-kernel is agnostic with respect to the events themselves, which are in the realm of the variable definitions in other libraries. Hence, the micro-kernel architecture entirely separates the propagation from the variables and makes it possible to add new variable types compositionally without upgrading the kernel. In particular, the list of events are built and maintained outside the kernel.

The micro-kernel maintains an array of  $P + 1$  queues to track the scheduled closures and first-order function applications. By default, value events are at priority  $P$ , variable events at priority  $P - 1$ , and constraint events at priorities  $1..P - 2$ . Priority 0 has a special role to be discussed shortly. Conceptually, method `scheduleCtrEvt` receives a list of  $k$  pairs  $\langle c_i, p_i \rangle$  where  $c_i$  is a constraint and  $p_i$  is an integer priority in  $1..P - 2$  and updates the queues with

$$Q_{p_i} = \text{enqueue}(Q_{p_i}, c_i) \quad 0 \leq i \leq k - 1.$$

The other scheduling functions are similar.

Method `propagate` executes the propagation loop and its implementation is shown in Figure 15. The algorithm processes each non-empty queue in turn from the highest ( $P$ ) to the lowest (1) priority. Line 5 finds the index of the highest priority queue with some events. Lines 6–9 pick the first highest priority event, execute it (line 7) and carry on until  $p = 0$  which indicates that all queues in the  $1..P$  range are empty. Finally, lines 12 and 15 unconditionally execute the events held in  $Q_0$  even after a failure has been discovered. As is customary, the dispatching of messages may schedule additional events that will be handled during this cycle. Lines 4–13 are the body of a closure which is passed to the micro-kernel function `tryfail` together with a failure handler (line 15–16). Function `tryfail` executes the first closure and diverts the control flow to the second closure when the `fail` function is called during the first closure execution. The events in  $Q_0$  are always executed: They are typically used to collect and update monitoring information, which can then be used to implement heuristics or learning techniques. The micro-kernel does not have any reference to variable, domain, or even the nature of constraints: It only manipulates closures, first-order applications, and information about constraints (e.g., whether they have been propagated or not).

---

```

1  -(ORStatus)propagate {
2    BOOL done = NO;
3    return tryfail(ORStatus~{
4      while (!done) {
5        p = maxi=1P i · (Qi ≠ ∅)
6        while (p ≠ 0) {
7          execute(deQueue(Qp));
8          p = maxi=1P i · (Qi ≠ ∅);
9        }
10       done = Qp = ∅;
11     }
12     while (Q0 ≠ ∅) execute(deQueue(Q0));
13     return ORSuspend;
14   }, ~ {
15     while (Q0 ≠ ∅) execute(deQueue(Q0));
16     return ORFail;
17   });
18 }

```

---

**Fig. 15.** The Micro-Kernel Propagation

#### 4.4 A Finite-Domain Integer Variable

We now sketch the implementation of a finite-domain variable, whose class `CPIntVar` is outlined as follows:



---

```

1 @interface CPIntVar {
2   id<CPEngine> _engine;
3   id<CPDom> _dom;
4   id<CPCtrEvent> _min, _max, _bounds, _bind;
5   id<CPValEvent> _loss;
6 }
7 -(id)initVar: (CPEngine*) engine low: (ORInt) low up: (ORInt) up;
8 -(void) whenChangeMinDo: (void^()) f;
9 -(void) whenLoseValueDo: (void^(ORInt)) f;
10 -(void) whenChangePropagate: (id<CPConstraint>) c;
11 -(ORStatus) updateMin: (ORInt) newMin;
12 -(ORStatus) removeValue: (ORInt) value;
13 @end

```

---

Instance variable `_dom` points to a domain representation such as a range, a bit-vector, or a list of intervals. Methods `whenChangeMinDo` and `whenLoseValueDo` are used to register variable events and value events respectively. Their implementation is simple:

---

```

1 -(void) whenChangeMinDo: (void^()) f { [_min insert: f]; }
2 -(void) whenLoseValueDo: (void^(ORInt)) f { [_loss insert: f]; }

```

---

Method `whenChangePropagate` is slightly more involved: It creates a closure `c1` to propagate a constraint `ctr` and inserts a pair `(c1, ctr)` in the list.

The update methods of a variable are expected to schedule relevant events. Consider method `updateMin`:

---

```

1 -(ORStatus)updateMin:(ORInt)newMin {
2   BOOL changed = [_dom updateMin:newMin];
3   if (changed) {
4     [_engine scheduleCtrEvt:_min];
5     [_engine scheduleCtrEvt:_bounds];
6     if ([_dom size] == 1) [_engine scheduleCtrEvt:_bind];
7   }
8   return ORSuspend;
9 }

```

---

It updates the domain, and when the domain is modified, it schedules the execution of the events registered on the `_min` and `_bounds` lists. If the domain is a singleton, line 6 also schedules the `_bind` list. Note that method `updateMin` on the domain may raise a failure, which is captured in the `tryfail` construct.

## 4.5 Propagators

We now illustrate a few propagators. Figure 16 depicts a domain-consistent propagator for constraint  $x = y + c$  using value events. Lines 3–4 cover the cases where one of the variables is bound and the other variable is updated accordingly. Lines 6–7 initiate the filtering of  $x$  and  $y$  by tightening their respective bounds. Lines 8–11 prune the domain of the variables. Lines 13–14 associates first-order functions with  $x$  and  $y$  to respond to the loss of value  $v$  from their domains. Figure 17 sketches the implementation of a global constraint using variable and constraint events. The `post` method scans all the variables (lines 4–6) and registers a closure to update internal data structures when a variable

---

```

1 @implementation CPEqualDC //  $x = y + c$ 
2 -(ORStatus) post {
3     if (bound(x)) [y bind: x.min - c];
4     else if (bound(y)) [x bind: y.min + c];
5     else {
6         [x updateMin:y.min + c andMax:y.max + c];
7         [y updateMin:x.min - c andMax:x.max - c];
8         for(ORInt i = x.min; i <= x.max; i++)
9             if (![x member:i]) [y remove:i - c];
10        for(ORInt i = y.min; i <= y.max; i++)
11            if (![y member:i]) [x remove:i + c];
12
13        [x whenLoseValueDo:^(ORInt v) { [y remove: v - c];}];
14        [y whenLoseValueDo:^(ORInt v) { [x remove: v + c];}];
15    }
16    return ORSuspend;
17 }

```

---

**Fig. 16.** A Domain Consistency Propagator for  $x = y + c$ 


---

```

1 @implementation CPAllDifferent // on array  $\_x$ 
2 -(ORStatus) post {
3     [self initDataStructures];
4     for(ORInt i =  $\_x$ .low; i <=  $\_x$ .up; i++)
5         if (! $\_x$ [i].bound)
6             [ $\_var$ [i] whenBindDo: ^{ [self removeOnBind:i]; } ];
7     [self propagate];
8     for(ORInt k =  $\_x$ .low; k <=  $\_x$ .up; k++)
9         if (! $\_x$ [k].bound) [ $\_x$ [k] whenChangePropagate: self];
10    return ORSuspend;
11 }
12 -(void) propagate {
13     if (![self feasible]) fail();
14     [self prune];
15 }
16 @end

```

---

**Fig. 17.** The AllDifferent Skeleton

is bound. It propagates the constraint and registers itself with each variable to propagate whenever the domain of the source variable  $\_x[k]$  changes (lines 8–9). Observe that this implementation combines variable-based and constraint-based propagation.

## 5 Experimental Results

Every programming system strikes a trade-off between expressiveness and efficiency. This section examines the efficiency of OBJECTIVE-CP to quantify this trade-off more precisely. It describes the compilation, runtime, and search efficiency of OBJECTIVE-CP. All experiments were carried out on MacOS X 10.8.3 running on a Core i7 at 2.6Ghz. All the results are based on 10 runs of the program (given that tie-breaks are randomized). All the times are reported in milliseconds. Columns  $\mu(T)$  and  $\sigma(T)$  report the average and standard deviation for the total runtime (measured from the moment the program starts to the moment it terminates). Column  $\mu(|M|)$  reports the memory usage in kilobytes for the entire executable (it is measured at the `malloc` interface).

**Table 1.** Performance of OBJECTIVE-CP

| <i>Bench</i>  | $\mu(T)$ | $\sigma(T)$ | $\mu( M )$ | $ X $  | $ C $  | $G( X )$ | $G( C )$ | $\mu(T_c)$ | $\sigma(T_c)$ |
|---------------|----------|-------------|------------|--------|--------|----------|----------|------------|---------------|
| ais           | 1,444.3  | 111.8       | 1,336      | 59     | 33     | 1.49     | 1.88     | 1.9        | 0.6           |
| costas        | 5,298.3  | 7,210.7     | 1,544      | 240    | 424    | 1.65     | 1        | 7.7        | 1.8           |
| fdmul         | 427.3    | 64.2        | 674        | 29     | 15     | 1.83     | 1.6      | 1.4        | 0.5           |
| magicserie    | 9,808.4  | 86.1        | 122,050    | 300    | 301    | 302.99   | 1        | 961.8      | 23.9          |
| bibd          | 793.2    | 104.3       | 16,836     | 2,205  | 663    | 4        | 10.98    | 123        | 13.1          |
| debruijn      | 7,619.3  | 207         | 554,915    | 57,346 | 57,346 | 2        | 1        | 1,535.3    | 121.9         |
| golomb        | 28,068.2 | 238.8       | 904        | 132    | 134    | 1        | 1        | 2.2        | 0.4           |
| coloringModel | 14,872.7 | 248.2       | 1,818      | 81     | 1,309  | 1        | 1        | 10.9       | 2.4           |
| eq20          | 23.6     | 2.9         | 986        | 7      | 20     | 21       | 1        | 2.3        | 0.7           |
| latinSquare   | 498.7    | 55          | 1,660      | 147    | 182    | 1.33     | 1        | 4.5        | 1.6           |
| slab          | 2,937.6  | 154.2       | 52,313     | 223    | 113    | 179.2    | 26.54    | 465.3      | 7.1           |
| sport         | 5,987.8  | 380.3       | 2,018      | 287    | 113    | 1        | 1        | 4.2        | 1.2           |

**Table 2.** Comparison of OBJECTIVE-CP and COMET

| <i>System</i> | Perfect  |             | Sport    |             | Slab     |             | Slab-LNS |             |
|---------------|----------|-------------|----------|-------------|----------|-------------|----------|-------------|
|               | $\mu(T)$ | $\sigma(T)$ | $\mu(T)$ | $\sigma(T)$ | $\mu(T)$ | $\sigma(T)$ | $\mu(T)$ | $\sigma(T)$ |
| comet         | 6,145.64 | 180.8       | 4,655.7  | 90.9        | 5,205.5  | 102.6       | 17,241.7 | 10,644      |
| ocp           | 5,884.16 | 204.8       | 4,712.1  | 76.3        | 2,213.2  | 59.1        | 6,308.2  | 3,632.5     |

*Compilation Efficiency.* Table 1 discusses model compilation efficiency. Columns  $|X|$  and  $|C|$  report the size of the model (number of variables and constraints), while  $G(|X|)$  and  $G(|C|)$  give the growth rate when the model has been flattened and concretized.  $G(|X|) = 2.0$  states that the concretized model has twice as many variables as the original.  $\mu(T_c)$  and  $\sigma(T_c)$  report the average and standard deviation of the time (in milliseconds) needed to compile and concretize the high-level model. The compilation time is almost always insignificant rarely passing 10 milliseconds. For **Debruijn**, the compilation is the heaviest with about 1.5 seconds for a large instance with some 57,346 variables and constraints modeled as algebraic expressions. The next two most expensive are the **magicserie** and **slab** benchmarks that heavily rely on reifications.

*Runtime Efficiency.* Table 2 offers an insight on the performance of OBJECTIVE-CP relative to the COMET platform and shows that the OBJECTIVE-CP architecture competes with the COMET platform. The statistics for both systems were collected on the same (slightly faster) machine over 50 runs of each benchmark. **Slab-LNS** relies on a complex search heuristic with dynamic symmetry breaking and a large neighborhood search component (i.e., it uses several search combinators). Care was taken to implement the exact same search in COMET.

*Search Efficiency.* Table 3 provides an overview of several black-box search procedures embedded in combinators. Space limitations do not allow for a comprehensive suite of tests but, instead, we focus on demonstrating sound performance on a number of interesting search procedures. The magic square benchmark (size 12) was executed with both ABS and IBS within a restarting search with a failure limit doubling at each restart. The satisfaction knapsack (instance 4) results are

**Table 3.** Search with OBJECTIVE-CP

| <i>Bench</i>     | <i>Method</i> | $\mu(T_{cpu})$ | $\sigma(T_{cpu})$ | $\mu(T_{wc})$ | $\sigma(T_{wc})$ | $\mu(Choices)$ | $\sigma(Choices)$ | $\mu( M )$ |
|------------------|---------------|----------------|-------------------|---------------|------------------|----------------|-------------------|------------|
| magicsquareModel | ABS           | 7,871.8        | 3,251.6           | 8,043.9       | 3,335.7          | 57,952.5       | 37,181.1          | 4,850.1    |
| magicsquareModel | IBS           | 15,021.2       | 7,581.3           | 15,228.9      | 7,816.8          | 101,308.1      | 144,010.7         | 2,289.2    |
| knapsack         | ABS           | 401.9          | 115.3             | 409.4         | 116.5            | 8,975.5        | 2,493.2           | 1,413      |
| progressive      | ABS           | 2,763.8        | 695.4             | 2,797.3       | 703.5            | 1,226.1        | 822.1             | 17,303.5   |
| progressive      | IBS           | 958.1          | 281.1             | 984.9         | 287.9            | 2,988.4        | 2,139.8           | 15,511.8   |
| progressive      | WDeg          | 1,587.3        | 1,068.2           | 1,630.4       | 1,087.9          | 8,769.2        | 6,724.9           | 15,459.4   |

for ABS alone (the others being too long). For the progressive party problem, the configuration is (2,8) and three black-box searches were compared. The search were all embedded in the same failure-limited restarting search. The results show that OBJECTIVE-CP achieves strong results on these benchmarks using state-of-the-art black-box algorithms. Note that, while ABS is the slowest of the three on progressive, it does far fewer choices than the other two and the bulk of the running time is spent in the probing. (No attempts were made to tweak the parameters which were identical on all benchmarks.).

## 6 Related Work

Obviously, OBJECTIVE-CP builds on top of over two decades of research in constraint programming and modeling systems. This section reviews some of the most relevant work from an overall system design standpoint. OPL, ILOG CONCERT, COMET, the G12 project, and GECODE probably had the most influences on its design and implementation. At some level, OBJECTIVE-CP can be viewed as a synthesis of the salient features of each of these systems and delivers, what we believe, is a sweet spot in the trade-off between expressiveness and efficiency. It goes without saying that different users may prefer different abstraction levels (e.g., a modeling language or a low-level library) based on their experience and the nature of their applications.

OPL [20] is a modeling language implemented on top of ILOG SOLVER which features high-level modeling and search abstractions. OPL introduced the `try` and `tryall` constructs but did not support compositional combinators and its implementation of node selection strategies was somewhat ad-hoc. Its implementation was in terms of goals. Models were not first-class objects and a dedicated language OPL SCRIPT [21] was needed to compose models. OBJECTIVE-CP borrowed the nondeterministic constructs of OPL but boosted almost all other aspects of the system (which is expected given the progress in the last 15 years).

ILOG CONCERT [2] introduced models into C++ libraries. ILOG CONCERT makes it possible to define models that can be then extracted by algorithms. The extraction process associates modeling objects (prefixed by `Ilo`) with implementation objects (prefixed by `Ilc` for the CP solver). Search goals can be expressed on the implementation objects in a way similar to ILOG SOLVER. OBJECTIVE-CP does not expose implementation objects and the search is implemented

solely based on model variables. The search language of OBJECTIVE-CP is fundamentally different to those of ILOG CONCERT/ILOG SOLVER. At the functionality level, OBJECTIVE-CP enjoys a nice symbiosis with the underlying host languages, support an iterative style for search, and extensibility of the search languages. At the implementation level, it is based on continuations and controllers, not on recursive goals [15] which are close to the original implementation of constraint logic programming [19].

COMET [3] is a domain-specific language for hybrid optimization, featuring solvers for constraint programming, mathematical programming, and local search. Its search language for constraint programming is based on continuations and controllers [22] but it did not support compositional combinators as argued in [17]. COMET does not have the concept of models, which led to the development of CML [5]. OBJECTIVE-CP merge the controllers and the continuations of COMET with the concept of models and a more compositional search language into a low-level host language, achieving a strong symbiosis between the extensions and the host.

The G12 project at NICTA introduced the MINIZINC modeling language and its implementation through a series of transformations to FLATZINC [13]. MINIZINC has a lot of similarities with OPL but took a much more systematic implementation approach based on model transformations. OBJECTIVE-CP uses a similar strategy but exposes models as first-class objects and enables hybrid optimization through model combinators [4]. It also features a search language merging the benefits of search controllers [22] and search combinators [17].

## 7 Conclusion

This paper presented an overview of OBJECTIVE-CP based on the vision

Optimization Program = Model + Search + Solver

to strike a good balance between expressiveness, extensibility, and efficiency. From an expressiveness standpoint, OBJECTIVE-CP features high-level modeling and search languages which never refer to implementation objects. The search language of OBJECTIVE-CP is based on the motto

Search = Continuations + Controllers

or, in other words, the belief that a rich and extensible search language can be built bottom-up and compositionally from continuations and search controllers. OBJECTIVE-CP provides the ability to transparently parallelize multi-start and branch & bound algorithms. At the implementation level, OBJECTIVE-CP performs a number of model transformations of the user model before concretization into a specific solver. OBJECTIVE-CP features an efficient constraint-programming solver based on a micro-kernel architecture, separating propagation, variables, and constraints.

**Acknowledgments.** NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

## References

1. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.-F.: Revising hull and box consistency. In: ICLP, pp. 230–244 (1999)
2. Concert Technology. Reference Manual, version 12.1 (2009)
3. Inc. Dynadec. Comet v2.1 user manual. Technical report, Providence, RI (2009)
4. Fontaine, D., Michel, L., Van Hentenryck, P.: Model Combinators for Hybrid Optimization. In: Schulte, C. (ed.) CP 2013, vol. 8124, pp. 300–315. Springer, Heidelberg (2013)
5. Fontaine, D., Michel, L.: A High Level Language for Solver Independent Model Manipulation and Generation of Hybrid Solvers. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) CPAIOR 2012. LNCS, vol. 7298, pp. 180–194. Springer, Heidelberg (2012)
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)
7. 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)
8. Van Hentenryck, P., Michel, L.: The Steel Mill Slab Design Problem Revisited. In: Trick, M.A. (ed.) CPAIOR 2008. LNCS, vol. 5015, pp. 377–381. Springer, Heidelberg (2008)
9. Hooker, J.N.: Logic-Based Methods for Optimization: Combining Optimization and Constraint Satisfaction. John Wiley and Sons (2000)
10. Ilog Solver 4.4. Reference Manual. Ilog SA, Gentilly, France (1998)
11. Michel, L., See, A., Van Hentenryck, P.: Parallelizing constraint programs transparently. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 514–528. Springer, Heidelberg (2007)
12. Michel, L., See, A., Van Hentenryck, P.: Transparent Parallelization of Constraint Programming. *INFORMS Journal on Computing* 21(3), 363–382 (2009)
13. 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)
14. Perron, L.: Search Procedures and Parallelism in Constraint Programming. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 346–361. Springer, Heidelberg (1999)
15. Puget, J.-F.: A C++ Implementation of CLP. In: Proceedings of SPICIS 1994, Singapore (November 1994)
16. Refalo, P.: Linear Formulation of Constraint Programming Models and Hybrid Solvers. In: Dechter, R. (ed.) CP 2000. LNCS, vol. 1894, pp. 369–383. Springer, Heidelberg (2000)
17. Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., Stuckey, P.: Search Combinators. *Constraints* 18(2), 269–305 (2013)
18. Schulte, C., Tack, G.: View-Based Propagator Derivation. *Constraints* 18(1), 75–107 (2013)

19. Van Hentenryck, P.: Constraint Satisfaction in Logic Programming. The MIT Press, Cambridge (1989)
20. Van Hentenryck, P.: The OPL Optimization Programming Language. The MIT Press, Cambridge (1999)
21. Van Hentenryck, P., Michel, L.: OPL Script: Composing and Controlling Models. In: Apt, K.R., Kakas, A.C., Monfroy, E., Rossi, F. (eds.) *Compulog Net WS 1999*. LNCS (LNAI), vol. 1865, pp. 75–90. Springer, Heidelberg (2000)
22. Van Hentenryck, P., Michel, L.: Nondeterministic Control for Hybrid Search. In: Barták, R., Milano, M. (eds.) *CPAIOR 2005*. LNCS, vol. 3524, pp. 380–395. Springer, Heidelberg (2005)
23. Van Hentenryck, P., Michel, L., Deville, Y.: *Numerica: a Modeling Language for Global Optimization*. The MIT Press, Cambridge (1997)

# Parallel Discrepancy-Based Search

Thierry Moisan, Jonathan Gaudreault, and Claude-Guy Quimper

FORAC Research Consortium, Université Laval, Québec, Canada

Thierry.Moisan.1@ulaval.ca,

{Jonathan.Gaudreault,Claude-Guy.Quimper}@ift.ulaval.ca

**Abstract.** Backtracking strategies based on the computation of discrepancies have proved themselves successful at solving large problems. They show really good performance when provided with a high-quality domain-specific branching heuristic (variable and value ordering heuristic), which is the case for many industrial problems. We propose a novel approach (PDS) that allows parallelizing a strategy based on the computation of discrepancies (LDS). The pool of processors visits the leaves in exactly the same order as the centralized algorithm would do. The implementation allows for a natural/intrinsic load balancing to occur (filtering induced by constraint propagation would affect each processor pretty much in the same way), although there is no communication between processors. These properties make PDS a scalable algorithm that was used on a massively parallel supercomputer with thousands of cores. PDS improved the best known performance on an industrial problem.

## 1 Introduction

Constraint solvers have been used for decades and were successful at solving numerous operations research problems. For instance, these solvers are used for optimizing computer networks by better routing the traffic [1,2], and for planning and scheduling problems [3] in different industries, among them the forest products industry [4,5]. A solver accepts as input a combinatorial problem defined by a set of variables and a set of constraints posted on these variables. The solver usually explores the candidate solutions by doing a backtracking search in a tree.

With the rise of multi-core servers, there has been an increase in research for parallelizing constraint solvers. Parallelization is not trivial as there is need for a trade-off between the workload balance, the communication cost, and the duplication (redundancy) of work between the processors.

The choice of an efficient search strategy is instrumental in solving large industrial problems, even in a centralized environment (for performance reasons, it is essential to explore the most promising leaves first). Among others, backtracking strategies based on the analysis of discrepancies such as LDS [6], DDS [7], and DBDFS [8] have proved themselves successful at solving large problems. They show really good performance when provided with a high-quality branching heuristic (that is, variable and value ordering heuristic), which is the case for many industrial problems (e.g. [5]).



In this article, we propose a novel approach (PDS) that allows parallelizing a strategy based on the computation of discrepancies (i.e. LDS). The proposed approach shows the following characteristics:

- The pool of processors globally visits the leaves in exactly the same order as the centralized version of LDS would do.
- There is no need for communication between the processors.
- The implementation allows for a natural/intrinsic load balancing to occur (filtering induced by constraint propagation would affect each processor pretty much in the same way).
- The method provides robustness (if a processor dies, it can be replaced by a new one that must however restart the work allocated to this processor).
- It offers good scaling: adding additional processors can never slow down the global process, unlike approaches using communication.

These properties make PDS a scalable algorithm that we used to solve industrial problems from the forest products industry (see [4,5]) using a massively parallel supercomputer called Colosse deployed at Université Laval.

The remainder of this paper is organized as follows. Section 2 reviews basic concepts related to parallel tree search. Sections 3 and 4 describe the original algorithm and the parallel version. Section 5 reports theoretical results and evaluates statistically the performance of the algorithm in order to illustrate different characteristics. Section 6 describes the experiments run on industrial problems and their results. Section 7 concludes the paper.

## 2 Basic Concepts

This section provides an overview of the main approaches regarding parallel tree search. We then give an overview of previous attempts that were made in order to parallelize discrepancy-based strategies.

### 2.1 Search Space in Shared Memory

The simplest method for parallel tree search is implemented by having many cores share a list of open nodes (nodes for which there is at least one of the children that is still unvisited). Starved processors just pick up the most promising node in the list and expand it. By defining different node evaluation functions, one can implement different strategies (DFS, BFS and others). A comprehensive framework based on this idea was proposed in [9]. Good performance is often reported, as in [10] where a parallel Best First Search was implemented, and evaluated up to 64 processors.

Although this kind of mechanism intrinsically provides excellent load balancing, it is known not to scale beyond a certain number of processors; beyond that point, performance starts to decrease. For this reason, the approach cannot easily be adapted for massively parallel supercomputers with thousands of cores.

## 2.2 Search Space Splitting / Work Stealing

This family of approaches is often reported as the most frequently seen in the literature [11]. The main idea is to have the search tree split into different regions allocated to processors (e.g. one processor branches to the left, the other processor branches to the right). As it is unlikely those subtrees will be of equal size, a *work stealing* mechanism (see [12,13]) is needed. Because it uses both communication and computation time, this cannot easily be scaled up to thousands of processors. In practice, we observe a decrease in performance when reaching a certain number of processors. However, interesting work was reported in [14]; the authors allocated specific processors to coordination tasks, allowing more processors to be used before performance starts to decline.

Another promising approach is reported in [11]. The authors used a search space splitting mechanism allowing good load balancing without needing a work stealing approach. They use a hashing function allocating implicitly the leaves to the processors. Each processor applies the same search strategy in its allocated search space, which solves the load balancing problem. However, like previous approaches, leaves are globally visited in a different order than they would be on a single-processor system. This could be a pity in situations where we know a really good domain-oriented search strategy, a strategy that the parallel algorithm failed to exploit to its full potential.

## 2.3 Las Vegas Algorithms / Portfolios

This approach consists in allocating the same search space to each processor. Each processor explores it using a different strategy, leading to a different visiting order of the leaves. No communication is required and an excellent level of load balancing is achieved (they all search the same search space). Even if this approach causes a high level of redundancy between processors, it shows really good performance in practice. Shylo et al. [15] greatly improves the method using randomized restart [16,17,18] on each processor.

As there is no communication between processors, this approach is fully scalable, although on small multi-core computers some authors increase the efficiency of the method by allowing processors to share information learned during the search (e.g. nogoods, see [19]).

In general, the main advantage of the algorithm portfolio approach is that one does not need to know a good search strategy beforehand: many strategies will be automatically tried at the same time by the parallel system, thanks to randomization. This is very useful because, as mentioned by [20] and [21], defining good domain-specific labelling strategies (that is, variable and value ordering heuristic) is a difficult task.

However, for complex applications where general strategies are inefficient and where very good domain-specific strategies are known (e.g. [4,5]) one would like to have the parallel algorithm exploit the domain-specific strategy.

To the best of our knowledge, it is the first time that LDS is parallelized this way. In [14] LDS was used locally by processors to search in the trees allocated

to them (by a tree splitting / work stealing algorithm) but the global system did not replicate an LDS strategy. The original centralized LDS being an iterative algorithm, Boivin [22] tried running the first  $k$  iterations at the same time on  $k$  processors. The approach did not prove to be efficient for the following reason: when LDS is provided with a good labelling strategy, the  $k^{th}$  iteration of LDS visit leaves that have considerably less expected probability of success than those in the first iterations. For domain-specific problems where centralized LDS is known to be good, only the first few processors were really helpful in the parallel implementation. Moreover, they were experiencing load balancing problems.

Finally, LDS was adapted for distributed optimization in [23,24]. However, distributed problems (DisCSP [25], DCOP [26] and HDCOP [27]) refers to a different context than parallel computing. These are problems that are distributed by nature; different agents are responsible for establishing the value of distinct variables and communication/coordination are inherent to those approaches. Therefore, the algorithm called MacDS we proposed in [23,24,27] could not serve as a basis for a scalable parallel LDS algorithm.

The next section provides a comprehensive description of the centralized version of LDS that will be parallelized in Section 4.

### 3 LDS

Harvey and Ginsberg [6] describe LDS with binary search trees, i.e. trees where each non-leaf node has two children. We present a generalization of LDS to  $n$ -ary trees which includes a modification by Walsh [7] that prevents visiting a leaf more than once. The search space of a problem can be represented as a tree where each node corresponds to a partial assignment. The root is the empty partial assignment and the leaves are complete assignments (also called solutions). Each child has one more variable assigned than its parent.

The value ordering heuristic is a function that orders the children of a node from the most likely one to lead to a solution to the least likely one. When represented graphically, the left child is the most likely one to lead to a solution and the right child is the least likely one. A *discrepancy* is a deviation from the first choice of the heuristic. We say that the first choice of the heuristic has zero discrepancy, the second choice has one discrepancy, the third choice has two discrepancies and so on. The discrepancy of a node is the sum of the discrepancies associated to each choice on the path from the root of the tree to the node. Figure 1 shows a search tree where the number of discrepancies is shown for each node.

Harvey and Ginsberg demonstrated that, with a good value ordering heuristic, the expected quality of a leaf decreases as the number of discrepancies increases. For that reason, they proposed to visit the leaves with the fewest discrepancies first and to keep the leaves with the most discrepancies for the end. Algorithm 2 visits all the leaves that have exactly  $k$  discrepancies. Algorithm 1 launches the search to visit all leaves in increasing number of discrepancies.

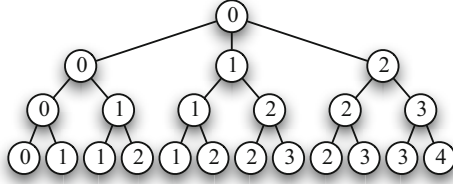


**Fig. 1.** Search tree. The discrepancy of each node is written inside the node.

## 4 PDS

We want to run an LDS search over multiple processors. Parallelization can be achieved in multiple ways but we set four goals that will influence our choices.

1. **Search Strategy Preservation.** We want the leaves of the search tree to be visited in the same order as they are on a single processor. Suppose that we mark each leaf of the tree with the time as it appears on a wall clock at the moment the leaf is visited. We assume that the clock is precise enough to break any ties. The ordering of the leaves by their visiting time should be the same regardless of the number of processors used.
2. **Workload Balancing.** We want the amount of work assigned to each processor to be evenly spread. This goal is particularly difficult to reach when the constraints filter the variable domains and make the search tree unbalanced.
3. **Robustness.** We aim at running the search on a large cluster of computers. It is frequent on those computers that a processor fails for different reasons and that the program must be restarted on another processor. It must be possible to identify which part of the search tree must be reassigned to another processor.
4. **Minimizing the Communication.** We aim at minimizing the communication between the processors. We actually want to avoid any communication. We make no assumptions about the geographical location of the processors and their ability to communicate. Communication should be limited to the broadcast of a solution.

---

**Algorithm 1.**  $\text{LDS}([\text{dom}(X_1), \dots, \text{dom}(X_n)])$

---

```

for  $k = 0..n$  do
   $s \leftarrow \text{LDS-Probe}([\text{dom}(X_1), \dots, \text{dom}(X_n)], k)$ 
  if  $s \neq \emptyset$  then return  $s$ 
return  $\emptyset$ 

```

---

**Algorithm 2.** LDS-Probe( $[\text{dom}(X_1), \dots, \text{dom}(X_n)], k$ )

---

```

Candidates  $\leftarrow \{X_i \mid |\text{dom}(X_i)| > 1\}$ 
if Candidates =  $\emptyset$  then
    if  $\text{dom}(X_1), \dots, \text{dom}(X_n)$  satisfies all the constraints then
        return  $\text{dom}(X_1), \dots, \text{dom}(X_n)$ 
    return  $\emptyset$ 
Choose a variable  $X_i \in \text{Candidates}$ 
Let  $v_0, \dots, v_{|\text{dom}(X_i)|-1}$  be the values in  $\text{dom}(X_i)$  sorted by the heuristic.
 $\underline{d} \leftarrow \max(0, k - \sum_{X_a \in \text{Candidates} \setminus \{X_i\}} (|\text{dom}(X_a)| - 1))$ 
 $\bar{d} \leftarrow \min(|\text{dom}(X_i)| - 1, k)$ 
for  $d = \underline{d}.. \bar{d}$  do
     $s \leftarrow \text{LDS-Probe}([\text{dom}(X_1), \dots, \text{dom}(X_{i-1}), \{v_d\},$ 
                         $\text{dom}(X_{i+1}), \dots, \text{dom}(X_n)], k - d)$ 
    if  $s \neq \emptyset$  then return  $s$ 
return  $\emptyset$ 

```

---

We define a variation of LDS that we call PDS. We label  $\rho$  processors with an integer between 0 and  $\rho - 1$  called the *processor id*. There is exactly one process running on each processor. The number of processors  $\rho$  and the processor id are given as input to each process. These two parameters are sufficient to identify which nodes of the search tree will be explored by each process.

We label each leaf  $s$  of the search tree by its visit time  $t(s)$  in a centralized LDS. The first leaf to be visited has a visit time of  $t(s_0) = 0$ , the second leaf has a visit time of 1 and so on. We assign each leaf to a processor in a round-robin way by assigning a leaf  $s$  to processor  $t(s) \bmod \rho$ . A processor  $j$  is only allowed to visit a leaf  $s$  that satisfies  $t(s) \bmod \rho = j$  or an ancestor of such a leaf. Consequently, before branching on a child node, a processor  $j$  has to check whether this child leads to a leaf it can visit. We show how to perform this test.

Let  $C(X_1, \dots, X_n, k)$  be the number of leaves with exactly  $k$  discrepancies in a search tree formed by the variables  $X_1, \dots, X_n$ . The function  $C(X_1, \dots, X_n, k)$  is recursively defined as follows.

$$C(X_1, \dots, X_n, k) = \begin{cases} 0 & \text{if } k < 0 \\ 1 & \text{if } k = 0 \\ \sum_{i=0}^{|\text{dom}(X_n)|-1} C(X_1, \dots, X_{n-1}, k - i) & \text{otherwise} \end{cases} \quad (1)$$

When all domains have cardinality two, the recursion becomes  $C(X_1, \dots, X_n, k) = C(X_1, \dots, X_{n-1}, k) + C(X_1, \dots, X_{n-1}, k - 1)$ . This recursion is the same that appears in Pascal's triangle to compute the binomial coefficients. We therefore have  $C(X_1, \dots, X_n, k) = \binom{n}{k}$  when  $|\text{dom}(X_i)| = 2$ . Intuitively, since each variable generates at most one discrepancy, the number of solutions with  $k$  discrepancies is the number of ways one can choose  $k$  variables among the  $n$  variables. When the domains have cardinalities greater than two, the recursion can be understood as follows: the variable  $X_n$  can generate a number of discrepancies  $i$  between 0 and  $|\text{dom}(X_n)| - 1$ . For each possible value of  $i$ , we count the number of solutions in the subtree of height  $n - 1$  that have exactly  $k - i$  discrepancies.

In equation (1), it seems that we consider a fixed ordering of the variables  $X_1, \dots, X_n$ . However, the variable ordering imposed by the heuristic does not need to be static, but is required to be deterministic.

Consider a node  $a$  where a value is going to be assigned to  $X_n$  and none of the variables  $X_1, \dots, X_n$  are assigned. The node  $a$  has for children the nodes  $c_0, \dots, c_{|\text{dom}(X_n)|-1}$ . Let  $l(a, k)$  be the processor assigned to the left-most leaf with  $k$  discrepancies in the subtree rooted at  $a$ . From this construction, we have  $l(a, k) = l(c_0, k)$  since branching from  $a$  to  $c_0$  adds no discrepancies to the partial assignment and that both expressions refer to the same leaf. There are  $C(X_1, \dots, X_{n-1}, k)$  leaves with  $k$  discrepancies in the subtree rooted at  $c_0$ . Since each of these leaves are assigned to the processors in a round-robin way, the processor assigned to the first leaf in the subtree rooted at  $c_1$  is therefore  $(l(c_0, k) + C(X_1, \dots, X_{n-1}, k)) \bmod \rho$ . The same reasoning applies for the other children leading to the following recursion.

$$l(c_i, k - i) = \begin{cases} l(a, k) & \text{if } i = 0 \\ (l(c_{i-1}, k - i + 1) + C(X_1, \dots, X_{n-1}, k - i + 1)) \bmod \rho & \text{otherwise} \end{cases}$$

We now have all the tools to present how the search strategy PDS proceeds. Each call to Algorithm 4 corresponds to the visit of a node in the search tree. The parameter  $k$  corresponds to the number of discrepancies that must lie on the path between this node and the leaves. Each processor visits only the nodes that lead to one of its assigned leaves. For each node  $a$  with children  $c_0, c_1, \dots$ , Algorithm 4 computes which processor will treat the left-most leaf of the subtree rooted at  $c_i$ . This allows computing a range of processors that will visit each child. If the current processor is among that range, then it branches to the child.

## 5 Analysis

This Section provides an analysis of PDS in order to illustrate different properties of the algorithm. Section 4 showed how parallel cores can globally visit the leaves in the same order as the centralized algorithm would do. We now demonstrate the quality of the intrinsic workload balance that is achieved. First, when exploring

---

### Algorithm 3. PDS( $[\text{dom}(X_1), \dots, \text{dom}(X_n)]$ )

---

```

 $l \leftarrow 0$ 
for  $k = 0..n$  do
   $Candidates \leftarrow \{X_i \mid |\text{dom}(X_i)| > 1\}$ 
   $z \leftarrow C(Candidates \setminus \{X_i\}, k)$ 
  if  $(currentProcessor - l) \bmod \rho < z$  then
     $s \leftarrow \text{PDS-Probe}([\text{dom}(X_1), \dots, \text{dom}(X_n)], k, l)$ 
     $l \leftarrow l + C(\{X_1, \dots, X_n\}, k) \bmod \rho$ 
    if  $s \neq \emptyset$  then return  $s$ 
return  $\emptyset$ 

```

---

**Algorithm 4.** PDS-Probe( $[\text{dom}(X_1), \dots, \text{dom}(X_n)], k, l$ )

---

```

Candidates  $\leftarrow \{X_i \mid |\text{dom}(X_i)| > 1\}$ 
if Candidates =  $\emptyset$  then
    if  $\text{dom}(X_1), \dots, \text{dom}(X_n)$  satisfies all the constraints then
        return  $\text{dom}(X_1), \dots, \text{dom}(X_n)$ 
    return  $\emptyset$ 
Choose a variable  $X_i \in \text{Candidates}$ 
Let  $v_0, \dots, v_{|\text{dom}(X_i)|-1}$  be the values in  $\text{dom}(X_i)$  sorted by the heuristic.
 $\underline{d} \leftarrow \max(0, k - \sum_{X_j \in \text{Candidates} \setminus \{X_i\}} (|\text{dom}(X_j)| - 1))$ 
 $\bar{d} \leftarrow \min(|\text{dom}(X_i)| - 1, k)$ 
for  $d = \underline{d}.. \bar{d}$  do
     $z \leftarrow C(\text{Candidates} \setminus \{X_i\}, k - d)$ 
    if  $(\text{currentProcessor} - l) \bmod \rho < z$  then
         $s \leftarrow \text{PDS-Probe}([\text{dom}(X_1), \dots, \text{dom}(X_{i-1}), \{v_d\},$ 
                            $\text{dom}(X_{i+1}), \dots, \text{dom}(X_n)], k - d, l)$ 
        if  $s \neq \emptyset$  then return  $s$ 
     $l \leftarrow l + z \bmod \rho$ 
return  $\emptyset$ 

```

---

the whole tree, the round-robin assignation of the processors ensures that the difference between the number of leaves visited by two processors is at most one. Workload balancing is easy to achieve when considering a complete search tree. However, it becomes harder to evenly divide the work among the processors when the tree is unbalanced. Search trees are often unbalanced when domain filtering and consistency technique are applied. We prove that when a value is filtered out of a variable domain and that a branch is cut from the tree, the workload is evenly reduced among all processors.

**Theorem 1.** *Let  $n$  be the number of variables in the problem. If a branch is cut from the search tree, the number of leaves removed from the workload of each processor differs by at most  $n + 1$ .*

*Proof.* The round-robin affection of the leaves with  $k$  discrepancies in a subtree guarantees that the number of leaves for each processor differs by at most one. Since we explore a subtree  $n + 1$  times for solutions with  $0, 1, \dots, n$  discrepancies, the difference of workload between the processors is at most  $n + 1$  leaves.  $\square$

## 5.1 Overhead

We do an overhead comparison of PDS, LDS, and DFS by counting the number of times a node is visited in a complete search tree associated to  $n$  binary variables.

A DFS in a tree with  $n$  variables visits the root and performs a DFS on two subtrees of  $n - 1$  variables. Let  $\text{DFS}(n)$  be the number of visited nodes.

$$\text{DFS}(n) = \begin{cases} 1 & \text{if } n = 0 \\ 2\text{DFS}(n) + 1 & \text{if } n > 0 \end{cases} \quad (2)$$

This non-homogeneous linear recurrence of first order solves to  $\text{DFS}(n) = 2^{n+1} - 1$ , i.e. the number of nodes in a complete binary tree of height  $n$ .

We consider a PDS with  $\rho$  processors. Let  $\text{PDSprobe}_\rho(n, k, j)$  be the number of nodes visited by processors  $j \in \{0, 1, \dots, \rho - 1\}$  on a tree with  $n$  binary variables for which we seek leaves of  $k$  discrepancies. We assume that the left-most leaf must be visited by processor 0. If the left-most leaf has to be visited by processor  $a$ , one can retrieve the number of visited nodes by relabeling the processors and computing  $\text{PDSprobe}_\rho(n, k, j - a \bmod \rho)$ . When  $k \in \{0, n\}$ , the tree has a unique leaf with  $k$  discrepancies and only the processor  $j = 0$  visits the  $n + 1$  nodes between the root and the leaf. If the number of leaves with  $k$  discrepancies,  $\binom{n}{k}$ , is smaller than or equal to  $j$ , then the processor  $j$  does not have to visit the tree. In all other cases, the number of visited nodes depends on the number of visited nodes in the left and right subtrees. We have the following recurrence.

$$\text{PDSprobe}_\rho(n, k, j) = \begin{cases} n + 1 & \text{if } j = 0 \wedge k \in \{0, n\} \\ 0 & \text{if } \binom{n}{k} \leq j \\ \text{PDSprobe}_\rho(n - 1, k, j) & \text{otherwise} \\ + \text{PDSprobe}_\rho(n - 1, k - 1, j - \binom{n-1}{k} \bmod \rho) + 1 & \end{cases}$$

Let  $\text{PDS}_\rho(n)$  be the total number of nodes visited by the  $\rho$  processors.

$$\begin{aligned} \text{PDS}_\rho(n) &= \sum_{k=0}^n \sum_{j=0}^{\rho-1} \text{PDSprobe}_\rho(n, k, j) = \sum_{k=1}^{n-1} \sum_{j=0}^{\rho-1} \text{PDSprobe}_\rho(n, k, j) + 2(n + 1) \\ &= \sum_{k=1}^{n-1} \sum_{j=0}^{\rho-1} \text{PDSprobe}_\rho(n - 1, k, j) \\ &\quad + \sum_{k=1}^{n-1} \sum_{j=0}^{\rho-1} \text{PDSprobe}_\rho(n - 1, k - 1, j - \binom{n-1}{k} \bmod \rho) \\ &\quad + \sum_{k=1}^{n-1} \sum_{j=0}^{\min(\rho, \binom{n}{k})-1} 1 + 2(n + 1) \end{aligned}$$

One can replace  $j - \binom{n-1}{k} \bmod \rho$  by  $j$  since we sum over  $j = 0.. \rho - 1$ . We also perform a change of indices for  $k$  in the same summation.

$$\begin{aligned} \text{PDS}_\rho(n) &= \sum_{k=1}^{n-1} \sum_{j=0}^{\rho-1} \text{PDSprobe}_\rho(n - 1, k, j) + \sum_{k=0}^{n-2} \sum_{j=0}^{\rho-1} \text{PDSprobe}_\rho(n - 1, k, j) \\ &\quad + \sum_{k=1}^{n-1} \min(\rho, \binom{n}{k}) + 2n + 2 \\ &= 2\text{PDS}_\rho(n - 1) + \sum_{k=1}^{n-1} \min(\rho, \binom{n}{k}) + 2 \end{aligned}$$



Using backward substitutions solves the recurrence.

$$\text{PDS}_\rho(n) = 2^n + 2^n \sum_{i=1}^n \sum_{k=0}^i \frac{1}{2^i} \min(\rho, \binom{i}{k})$$

When solved for  $\rho = 1$ , we retrieve the number of visited nodes with LDS. We also simplify for  $\rho \in \{2, 3\}$  assuming  $n \geq 3$ .

$$\text{LDS}(n) = 2^{n+2} - n - 3 \quad \text{PDS}_2(n) = 5 \cdot 2^n - 2n - 4 \quad \text{PDS}_3(n) = \frac{23}{4}2^n - 3n - 5$$

We observe that, as the number of variables grows, a LDS visits twice the number of nodes than a DFS. Therefore, when DFS finishes to visit the entire tree, LDS visited half of the leaves. However, these leaves have fewer than  $\frac{n}{2}$  discrepancies. So if the heuristic makes no mistakes at least half of the time, LDS finds a solution by the time DFS visits the entire tree. The overhead of LDS compared to DFS is therefore compensated by the search of more promising parts of the search tree.

As  $n$  grows, the ratios  $\frac{\text{PDS}_2(n)}{\text{LDS}(n)}$  and  $\frac{\text{PDS}_3(n)}{\text{LDS}(n)}$  tend to 1.25 and 1.43. These overheads of 25% and 43% grow slower than the number of processors and implies that 2 and 3 processors will visit the search tree in 62% and 48% of the time taken by one processor. Should the search visit the entire search tree, Figure 2 shows the speedup of PDS over LDS as the number of processors increases. We see that the speedup grows linearly except in the degenerate case where the number of leaves equal the number of processors.

To get a more accurate idea of the speedup, one needs to consider the quality of the solution (in an optimization problem) or the probability of finding a solution (in a satisfaction problem). This is done in the next section.

## 5.2 Statistical Analysis

We provide statistical results showing that the performance of the algorithm never declines, except in the degenerated case where there are more processors than leaves. It is therefore a worst-case analysis where the entire tree is explored.

Harvey and Ginsberg [6] showed, by analyzing binary SCP search trees from different problems, that the quality of a heuristic can be approximated/described by the probability  $p$  of finding a solution in the left subtree if no mistakes were made in the current partial assignment. Similarly, we say that the probability of finding a solution in the right subtree is  $q$ . If the solution is unique, we have  $p + q = 1$ . If there is more than one solution, we have  $p + q \geq 1$  since there is a probability of having a solution both in the left subtree and the right subtree.

The better a heuristic is, the greater the ratio  $\frac{p}{q}$  is. The extreme situation where  $\frac{p}{q} = 1$  corresponds to a heuristic that does no better than random variable/value selection (all leaves share the same probability of being a solution, and using an LDS would not be a logical choice). The probability that a leaf

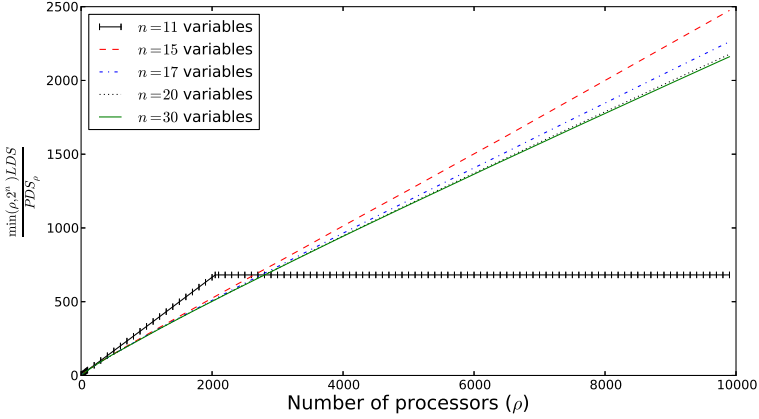


**Fig. 2.** Speedup for some number of processors

with  $k$  discrepancies is a solution is  $p^{n-k}q^k$  since it involves branching  $k$  times on the right and  $n - k$  times on the left.

Figure 3 shows the probability that a solution is found according to the number of visited nodes per processor. The probability that a leaf  $s_i$  with  $k$  deviations is a solution is  $P(s_i) = p^{n-k}q^k$ . The probability of finding a solution after visiting the leaves  $s_1, \dots, s_m$  is  $1 - \prod_{i=1}^m (1 - P(s_i))$ . For a given computation time, increasing the number of processors increases the probability that a solution has been found.

This clearly illustrates that increasing the number of processors increases the performance until  $\rho$  reaches the number of leaves in the search tree. From that point there is no more gain.

As the expected quality of a leaf decreases exponentially with its number of discrepancies (recall Section 3), adding more processors makes us visit additional leaves in the same computation time, but those leaves have smaller probability of success than the previous ones. This is a natural (and desired) consequence of using a good variable/value selection heuristics and a backtracking strategy visiting leaves in order of expected quality.

Figure 4 presents the speedup obtained for some probability that a solution is found. A lot of variation is present for low probability values. This is due to the use of the heuristic that points toward good solutions quickly. The speedup then converges toward a single value as the probability that a solution is found increases.

The next experiment studies the performance of the algorithm according to the quality of the variable/value selection heuristics used. We recall from Section 3 that the higher the  $\frac{p}{q}$  ratio is, the more likely the solutions will be concentrated in leaves having few discrepancies. In contrast, the extreme situation where  $p = q$  simulates the use of a heuristic that does no better than random variable/value selection (all leaves share the same probability of being a solution, and using an LDS would not be a logical choice).

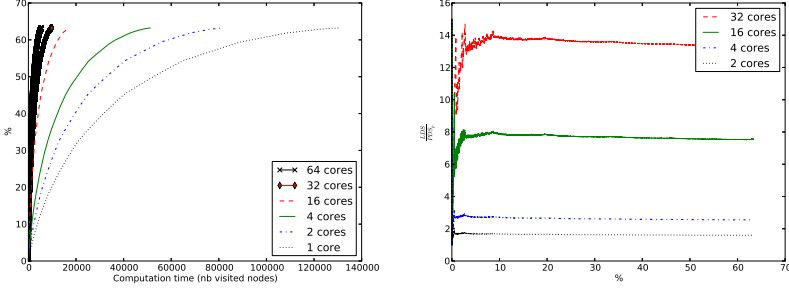


**Fig. 3.** Probability that a solution is found after some computation time [ $n = 15$  vars;  $p = 0.6$ ;  $q = 0.4$ ]

**Fig. 4.** Speedup for some probability that a solution is found [ $n = 15$  vars;  $p = 0.6$ ;  $q = 0.4$ ]

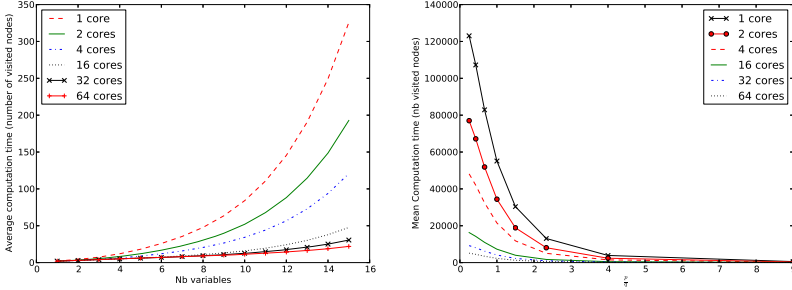


**Fig. 5.** Average computation time to find a solution according to the number of variables [ $p = 0.6$ ;  $q = 0.4$ ]

**Fig. 6.** Average computation time to find a solution according to the  $\frac{p}{q}$  ratio where  $p + q = 1$  with 15 variables

On Figure 6, the curve for 1 processor shows that computation time decreases exponentially when  $\frac{p}{q}$  increases. Other curves show that when we provide additional processors, the computation time still decreases exponentially, but much more quickly.

## 6 Experimentation with Industrial Data

In a lumber finishing facility, lumbers are planed and sorted according to their grade (i.e. quality). It may be trimmed in order to produce a shorter lumber of a higher grade and value. The operation that improves a piece of lumber only depends on the piece of lumber itself with no consideration for the actual customer demand. This causes the production of multiple finished products at the same time (co-production) from a single raw product (divergence). This makes the production very difficult to plan according to the customer demand. There is a finite set of processes that can be used to transform one raw product into many finished products. The plant can only process lumber of a single

category in a given production shift. Mills prefer long campaigns of a single category as it reduces costs: once the mill is configured for a given setup, they want to stay in this configuration for as many shifts as possible. The plant maintains an inventory of raw and finished products. For each customer order, a given quantity of a finished product has to be delivered at a specific time.

To sum up, the decisions that must be taken in order to plan the finishing operations are the following: (1) select a lumber category to process during a campaign, (2) decide when the campaign starts and for how long it lasts, and (3) for each campaign, decide the quantities of each compatible products to process. It is a single machine planning and scheduling problem. Each planning period corresponds to one "production shift" (approximately 4 hours). The objective is to minimize orders lateness (modelled as a penalty cost) and production costs. The problem is fully described in [4] which provides a good heuristic for this problem. In [5], the heuristic is used to guide the search using constraint programming (applying LDS) and it outperforms the DFS and the mathematical programming approach.

Industrial instances are huge and there is a need for good solutions in shorter computation time. The instances have 65,142 variables and 50,238 constraints. Among them, there are 42 discrete decision variables whose domains have cardinality 6 and 4200 continuous decision variables. As we have a really good branching heuristic for which LDS works really well, this problem is an ideal candidate for PDS. This search heuristic first branches on variable/values for the integer variables (decisions 1 and 2 in the previous paragraph). Once the values for these variables are known, the remaining continuous variables (3) define a linear program that can be easily solved to optimality using the simplex method. Therefore, each time we have a valid assignment of the integer variables, we consider we have reached a leaf and we solve a linear program to evaluate the value of this solution. This implies that the leaves have a heavier computation time than the inner nodes. This situation differs from Section 5.1 where all nodes have the same computation time.

We implemented PDS and ran it on Colosse, a supercomputer with more than 8000 cores (dual, quad-core Intel Nehalem CPUs, 2.8 GHz with 24 GB RAM). Two Canadian lumber companies involved in the project provided the industrial instances. The four datasets have from 30 to 42 production shifts, from 20 to 133 processes, from 60 to 308 customer orders, from 20 to 68 raw products, and from 60 to 222 finished products.

## 6.1 Results and Discussion

Figures 7 to 10 show the objective value according to the computation time (maximum of one hour) for different numbers of processors. We also computed the best solution for LDS on these instances and the curve is indistinguishable from the PDS(1) curve. In Figure 10, we can see that a solution of quality of  $1.1 \times 10^7$  is not found with 1 processor even after an hour but can be found in 10 minutes with 4096 processors. Furthermore, 1 processor obtained a solution of  $1.5 \times 10^7$  in one hour while the same solution is found in a few seconds with

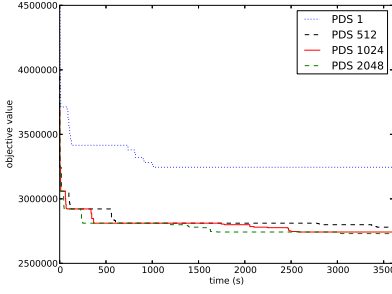


**Fig. 7.** Best solution found for K1 dataset

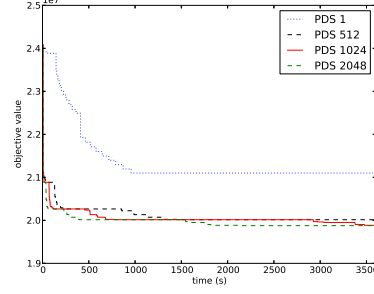


**Fig. 8.** Best solution found for M3 dataset

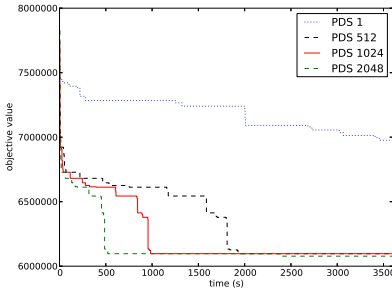


**Fig. 9.** Best solution found for M2 dataset

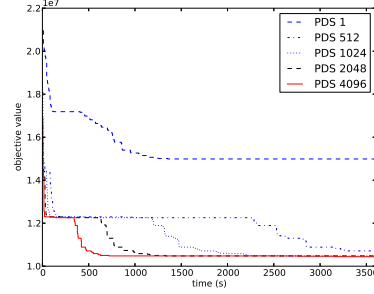


**Fig. 10.** Best solution found for M1 dataset

512 processors. This is a major improvement from an industrial point of view where computation time is the real constraint.

The harder instances are those where the heuristic has more difficulties and a good solution is obtained later in the search (Figure 7 is the easiest instance, Figure 10 is the hardest one). The absolute time saving is greater on harder instances when using PDS.

For each Figure from 7 to 10, the curves for 512, 1024, and 2048 processors have the same shape but get more compressed over time as the number of processors increases. This shows that the heuristic and the search strategies remains the same even in its parallelized version.

Table 1 lists statistics we computed during these experiments. The speedup is the ratio of the number of leaves visited by multiple processors over the number of leaves visited by one processor. PDS scales well: even with 4096 processors, the speedup is still increasing almost linearly.

One hour was insufficient to visit the entire search tree. With 4096 processors, we reached solutions with 6 discrepancies but did not visit all of them. Therefore, there is no idle time. However, we want to measure how the workload, in terms of visited leaves, differs between processors. Let  $\chi_j$  be the number of leaves

**Table 1.** Industrial datasets experiments statistics. The column  $\bar{\chi}$  is the average number of leaves visited by each processor. The column  $\max(\chi) - \min(\chi)$  is the maximum difference of leaves treated between processors. The column  $\frac{\bar{\chi}}{\min(\chi)}$  is the average percentage of leaves differences between each processor and the minimal number of leaves treated by one processor.

| dataSet | $\rho$ | speedup | $\bar{\chi}$ | $\max(\chi) - \min(\chi)$ | $\frac{\bar{\chi}}{\min(\chi)}(\%)$ |
|---------|--------|---------|--------------|---------------------------|-------------------------------------|
| K1      | 512    | 338.9   | 4668.6       | 550                       | 8.6                                 |
| K1      | 1024   | 585.9   | 4036.0       | 441                       | 7.97                                |
| K1      | 2048   | 941.9   | 3244.4       | 363                       | 7.79                                |
| M3      | 512    | 446.4   | 725.4        | 24                        | 1.89                                |
| M3      | 1024   | 863.1   | 701.3        | 57                        | 6.09                                |
| M3      | 2048   | 1601.1  | 650.4        | 25                        | 2.92                                |
| M2      | 512    | 432.7   | 920.3        | 53                        | 3.98                                |
| M2      | 1024   | 823.2   | 875.5        | 40                        | 3.49                                |
| M2      | 2048   | 1604.7  | 853.3        | 23                        | 1.94                                |
| M1      | 512    | 447.7   | 729.3        | 42                        | 4.03                                |
| M1      | 1024   | 869.1   | 707.9        | 42                        | 4.25                                |
| M1      | 2048   | 1656.1  | 674.4        | 79                        | 11.1                                |
| M1      | 4096   | 3152.3  | 641.9        | 57                        | 6.44                                |

processed by processor  $j$ . Let  $\min(\chi)$  be the minimum value of  $\chi_j$  for every  $j \in 0, 1, \dots, \rho-1$ . Let  $\bar{\chi}$  be the average number of leaves visited by each processor. The relative difference between  $\min(\chi)$  and  $\bar{\chi}$  is  $\frac{\bar{\chi}}{\min(\chi)}$ . This measure shows processors have visited roughly the same number of leaves.

We had hardware failures during the experiments and we have been able to restart a single processor while leaving the other ones running.

## 7 Conclusion

The contributions of this paper are twofold. First, we proposed a new parallelization scheme based on the LDS backtracking strategy. This parallelization does not alter the strategy since the visit order of the nodes remains unchanged. Moreover, PDS provides an intrinsic workload balancing, it scales on multiple processors, and it is robust to hardware failures. We provided a theoretical analysis that evaluated the performance of PDS based on the quality of the heuristic. This showed that adding more processors always provides a speedup.

Secondly, we experimented with a difficult industrial problem from the forest products industry for which an excellent problem-specific variable/value selection heuristics is known. This has been done by using as many as 4096 processors on a supercomputer. It shows the great potential of constraint programming in a massively parallel environment for which good search strategies are known.

## References

1. Chabrier, A., Danna, E., Le Pape, C., Perron, L.: Solving a network design problem. *Annals of Operations Research* 130, 217–239 (2004)
2. Le Pape, C., Régim, J.-C., Shaw, P.: Robust and parallel solving of a network design problem. In: Van Hentenryck, P. (ed.) *CP 2002. LNCS*, vol. 2470, pp. 633–648. Springer, Heidelberg (2002)
3. Le Pape, C., Baptiste, P.: Heuristic control of a constraint-based algorithm for the preemptive job-shop scheduling problem. *Journal of Heuristics* 5, 305–325 (1999)
4. Gaudreault, J., Forget, P., Frayret, J.M., Rousseau, A., Lemieux, S., D'Amours, S.: Distributed operations planning in the lumber supply chain: Models and coordination. *International Journal of Industrial Engineering: Theory, Applications and Practice* 17 (2010)
5. Gaudreault, J., Frayret, J.M., Rousseau, A., D'Amours, S.: Combined planning and scheduling in a divergent production system with co-production: A case study in the lumber industry. *Computers Operations Research* 38, 1238–1250 (2011)
6. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI 1995)*, pp. 607–613 (1995)
7. Walsh, T.: Depth-bounded discrepancy search. In: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI 1997)*, pp. 1388–1393 (1997)
8. Beck, J.C., Perron, L.: Discrepancy-bounded depth first search. In: *Proceedings of the Second International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR 2000)*, pp. 8–10 (2000)
9. Perron, L.: Search procedures and parallelism in constraint programming. In: Jaffar, J. (ed.) *CP 1999. LNCS*, vol. 1713, pp. 346–361. Springer, Heidelberg (1999)
10. Vidal, V., Bordeaux, L., Hamadi, Y.: Adaptive k-parallel best-first search: A simple but efficient algorithm for multi-core domain-independent planning. In: *Proceedings of the Third International Symposium on Combinatorial Search (SOCS 2010)* (2010)
11. Bordeaux, L., Hamadi, Y., Samulowitz, H.: Experiments with massively parallel constraint solving. In: *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI 2009)*, pp. 443–448 (2009)
12. Michel, L., See, A., Van Hentenryck, P.: Transparent parallelization of constraint programming. *INFORMS J. on Computing* 21, 363–382 (2009)
13. Chu, G., Schulte, C., Stuckey, P.J.: Confidence-based work stealing in parallel constraint programming. In: Gent, I.P. (ed.) *CP 2009. LNCS*, vol. 5732, pp. 226–241. Springer, Heidelberg (2009)
14. Xie, F., Davenport, A.: Massively parallel constraint programming for supercomputers: Challenges and initial results. In: Lodi, A., Milano, M., Toth, P. (eds.) *CPAIOR 2010. LNCS*, vol. 6140, pp. 334–338. Springer, Heidelberg (2010)
15. Shylo, O.V., Middelkoop, T., Pardalos, P.M.: Restart strategies in optimization: Parallel and serial cases. *Parallel Computing* 37, 60–68 (2010)
16. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of las vegas algorithms. *Information Processing Letters* 47, 173–180 (1993)
17. Gomes, C.P.: Boosting combinatorial search through randomization. In: *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence (AAAI 1998/IAAI 1998)*, pp. 431–437 (1998)

18. Gomes, C.P.: Complete randomized backtrack search. In: *Constraint and Integer Programming: Toward a Unified Methodology*, pp. 233–283 (2003)
19. Hamadi, Y., Sais, L.: Manysat: a parallel sat solver. *Journal on Satisfiability, Boolean Modeling and Computation* 6, 245–262 (2009)
20. Hamadi, Y., Ringwelski, G.: Boosting distributed constraint satisfaction. *Journal of Heuristics*, 251–279 (2010)
21. Puget, J.F.: Constraint programming next challenge: Simplicity of use. In: Wallace, M. (ed.) *CP 2004. LNCS*, vol. 3258, pp. 5–8. Springer, Heidelberg (2004)
22. Boivin, S., Gendron, B., Pesant, G.: Parallel constraint programming discrepancy-based search decomposition. *Optimization days*, Montréal, Canada (2007)
23. Gaudreault, J., Frayret, J.M., Pesant, G.: Discrepancy-based method for hierarchical distributed optimization. In: *Nineteenth International Conference on Tools with Artificial Intelligence (ICTAI 2007)*, pp. 75–81 (2007)
24. Gaudreault, J., Frayret, J.M., Pesant, G.: Distributed search for supply chain coordination. *Computers in Industry* 60, 441–451 (2009)
25. Yokoo, M.: *Distributed constraint satisfaction: foundations of cooperation in multi-agent systems*. Springer, London (2001)
26. Modi, P.J., Shen, W.M., Tambe, M., Yokoo, M.: Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence* 161, 149–180 (2006)
27. Gaudreault, J., Frayret, J.M., Pesant, G.: Discrepancy-based optimization for distributed supply chain operations planning. In: *Proceeding of the Ninth International Workshop on Distributed Constraint Reasoning (DCR 2007)* (2007)



# Bin Packing with Linear Usage Costs – An Application to Energy Management in Data Centres<sup>★</sup>

Hadrien Cambazard<sup>1</sup>, Deepak Mehta<sup>2</sup>, Barry O’Sullivan<sup>2</sup>, and Helmut Simonis<sup>2</sup>

<sup>1</sup> G-SCOP, Université de Grenoble, Grenoble INP; UJF Grenoble 1, CNRS, France  
hadrien.cambazard@grenoble-inp.fr

<sup>2</sup> Cork Constraint Computation Centre, University College Cork, Ireland  
{d.mehta,b.osullivan,h.simonis}@4c.ucc.ie

**Abstract.** EnergeTIC is a recent industrial research project carried out in Grenoble on optimizing energy consumption in data-centres. The efficient management of a data-centre involves minimizing energy costs while ensuring service quality. We study the problem formulation proposed by EnergeTIC. First, we focus on a key sub-problem: a bin packing problem with linear costs associated with the use of bins. We study lower bounds based on Linear Programming and extend the bin packing global constraint with cost information. Second, we present a column generation model for computing the lower bound on the original energy management problem where the pricing problem is essentially a cost-aware bin packing with side constraints. Third, we show that the industrial benchmark provided so far can be solved to near optimality using a large neighborhood search.

## 1 Introduction

Energy consumption is one of the most important sources of expense in data centers. The ongoing increase in energy prices (a 50% increase is forecasted by the French senate by 2020) and the growing market for cloud computing are the main incentives for the design of energy efficient centers. We study a problem associated with the EnergeTIC<sup>1</sup> project which was accredited by the French government (FUI) [2]. The objective is to control the energy consumption of a data center and ensure that it is consistent with application needs, economic constraints and service level agreements. We focus on how to reduce energy cost by taking variable CPU requirements of client applications, IT equipment and virtualization techniques into account.

There are a variety of approaches to energy management in data centres, the most well-studied of which is energy-aware workload consolidation. A Mixed Integer Programming (MIP) approach to dynamically configuring the consolidation of multiple services or applications in a virtualised server cluster has been proposed in [16]. That work focused on power efficiency and considered the costs of turning on/off the servers.

---

<sup>★</sup> The authors acknowledge their industrial partners (Bull, Schneider Electric, Business & Decision and UXP) as well as several public research institutions (G2Elab, G-SCOP and LIG). The authors from UCC are supported by Science Foundation Ireland Grant No. 10/IN.1/I3032.

<sup>1</sup> Minalogic EnergeTIC is a Global competitive cluster located in Grenoble France and fostering research-led innovation in intelligent miniaturized products and solutions for industry.

However, workloads were entirely homogeneous and there was little uncertainty around the duration of tasks. Constraint Programming is used in [8] with a different cost model.

A combinatorial optimisation model for the problem of loading servers to a desired utilisation level has, at its core, a bin packing (BP) problem [20]. In such a model each server is represented by a bin with a capacity equal to the amount of resource available. Bin packing is a very well studied NP-Hard problem. A significant amount of work has been conducted on lower bounds [13], approximation and exact algorithms. Although this research is still very active as demonstrated by the recent progress [17], researchers have started to look at variants involved in industrial applications.

In Section 2 we present an extension of bin packing which is a key sub-problem of the application domain and we show how to handle it efficiently with Constraint Programming (CP). In Section 3 we study the formulation of the EnergeTIC problem. In particular a lower bound computation technique is designed to assert the quality of the upper bounds found by a large neighborhood search. Section 4 reports the experiments on a real data-set followed by conclusions in Section 5.

## 2 Bin Packing with Linear Usage Costs

We consider a variant of the Bin Packing problem (BP) [20], which is the key sub-problem of the application investigated here. We denote by  $S = \{w_1, \dots, w_n\}$  the integer sizes of the  $n$  items such that  $w_1 \leq w_2 \leq \dots \leq w_n$ . A bin  $j$  is characterized by an integer capacity  $C_j$ , a non-negative fixed cost  $f_j$  and a non-negative cost  $c_j$  for each unit of used capacity. We denote by  $B = \{\{C_1, f_1, c_1\}, \dots, \{C_m, f_m, c_m\}\}$  the characteristics of the  $m$  bins. A bin is used when it contains at least one item. Its cost is a linear function  $f_j + c_j l_j$ , where  $l_j$  is the total size of the items in bin  $j$ . The total load is denoted by  $W = \sum_{i=1}^n w_i$  and the maximum capacity by  $C_{max} = \max_{1 \leq j \leq m} C_j$ . The problem is to assign each item to a bin subject to the capacity constraints so that we minimize the sum of the costs of all bins. We refer to this problem as the *Bin Packing with Usage Cost* problem (BPUC). BP is a special case of BPUC where all  $f_j$  are set to 1 and all  $c_j$  to 0. The following example shows that a good solution for BP might not yield a good solution for BPUC.

*Example 1.* In Figure 1, Scenario 1,  $B = \{(9,0,1), (3,0,2), (3,0,2), (3,0,2), (3,0,2)\}$  and  $S = \{2,2,2,2,3,3,3\}$ . Notice that  $\forall j, f_j = 0$ . The packing  $(P_1) : \{\{2,2,2,2\}, \{3\}, \{3\}, \{3\}, \{\}\}$  is using the minimum number of bins and has a cost of 26 ( $8*1 + 3*2 + 3*2 + 3*2$ ). The packing  $(P_2) : \{\{3,3,3\}, \{2\}, \{2\}, \{2\}, \{2\}\}$  is using one more bin but has a cost of 25 ( $9 + 2*2 + 2*2 + 2*2 + 2*2$ ). Here,  $(P_2)$  is better than  $(P_1)$  and using the minimum number of bins is not a good strategy. Now change the last unit cost to  $c_5 = 3$  (see Figure 1, Scenario 2). The cost of  $(P_1)$  remains unchanged since it does not use bin number 5 but the cost of  $(P_2)$  increases to 27, and thus  $(P_1)$  is now better than  $(P_2)$ .

**Literature Review.** A first relevant extension of BP for the current paper is called Variable Size Bin-Packing, where bins have different capacities and the problem is to minimize the sum of the wasted space over all used bins [15]. It can be seen as a special case of BPUC where all  $f_j = C_j$  and  $c_j = 0$ . Recent lower bounds and an exact approach are examined in [11]. A generalization to any kind of fixed cost is presented in [5], which can be seen as a special case of BPUC where all  $c_j = 0$ . Concave costs of bin utilization

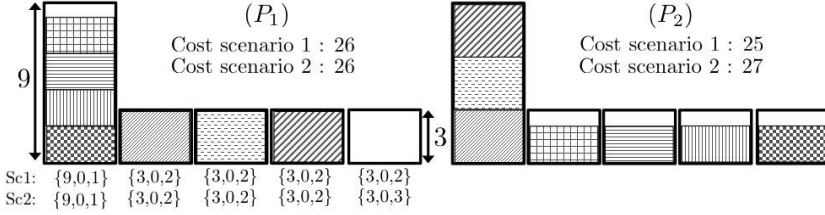


**Fig. 1.** Example of optimal solutions in two scenarios of costs. In Scenario 1, the best solution has no waste on the cheapest bin. In Scenario 2, it does not fill completely the cheapest bin.

studied in [12] are more general than the linear cost functions of BPUC. However [12] does not consider bins of different capacities and deals with the performance of classical BP heuristics whereas we are focusing on lower bounds and exact algorithms. Secondly, BP with general cost structures have been introduced [3] and studied [9]. The authors investigated BP with non-decreasing and concave cost functions of the number of items put in a bin. They extend it with profitable optional items in [4]. Their framework can model a fixed cost but does not relate to bin usage.

## 2.1 Basic Formulation and Lower Bounds

Numerous linear programming models have been proposed for BP [7]. We first present a formulation for BPUC. For each bin a binary variable  $y_j$  is set to 1 if bin  $j$  is used in the packing, and 0 otherwise. For each item  $i \in \{1, \dots, n\}$  and each bin  $j \in \{1, \dots, m\}$  a binary variable  $x_{ij}$  is set to 1 if item  $i$  is packed into bin  $j$ , and 0 otherwise. We add non-negative variables  $l_j$  representing the load of each bin  $j$ . The model is as follows:

$$\begin{aligned}
 &\text{Minimize} && z_1 = \sum_{j=1}^m (f_j y_j + c_j l_j) \\
 (1.1) &&& \sum_{j=1}^m x_{ij} = 1, && \forall i \in \{1, \dots, n\} \\
 (1.2) &&& \sum_{i=1}^n w_i x_{ij} = l_j, && \forall j \in \{1, \dots, m\} \\
 (1.3) &&& l_j \leq C_j y_j, && \forall j \in \{1, \dots, m\} \\
 (1.4) &&& x_{ij} \in \{0, 1\}, y_j \in \{0, 1\}, l_j \geq 0 && \forall j \in \{1, \dots, m\}, \forall i \in \{1, \dots, n\}
 \end{aligned} \tag{1}$$

Constraint (1.1) states that each item is assigned to one bin whereas (1.2) and (1.3) enforce the capacity of the bins. We now characterize the linear relaxation of the model. Let  $r_j = f_j/C_j + c_j$  be a real number associated with bin  $j$ . If bin  $j$  is filled completely,  $r_j$  is the cost of one unit of space in bin  $j$ . We sort the bins by non-decreasing  $r_j$ :  $r_{a_1} \leq r_{a_2} \leq \dots \leq r_{a_m}$ ;  $a_1, \dots, a_m$  is a permutation of the bin indices  $1, \dots, m$ . Let  $k$  be the minimum number of bins such that  $\sum_{j=1}^k C_{a_j} \geq W$ .

**Proposition 1.** Let  $z_1^*$  be the optimal value of the linear relaxation of the formulation (1). We have  $z_1^* \geq Lb_1$  with  $Lb_1 = \sum_{j=1}^{k-1} C_{a_j} r_{a_j} + (W - \sum_{j=1}^{k-1} C_{a_j}) r_{a_k}$ .

*Proof.*  $z_1^* = \sum_{j=1}^m (f_j y_j + c_j l_j) \geq \sum_{j=1}^m (f_j \frac{l_j}{C_j} + c_j l_j)$  because of the constraint  $l_j \leq C_j y_j$ , so  $z_1^* \geq \sum_{j=1}^m (\frac{f_j}{C_j} + c_j) l_j \geq \sum_{j=1}^m r_j l_j$ .  $Lb_1$  is the quantity minimizing  $\sum_{j=1}^m r_j l_j$  under the constraints  $\sum_j l_j = W$  where each  $l_j \leq C_j$ . To minimize the quantity we must split  $W$  over the  $l_j$  related to the smallest coefficients  $r_j$ . Hence,  $z_1^* \geq \sum_{j=1}^m r_j l_j \geq Lb_1$ .  $\square$

$Lb_1$  is a lower bound of BPUC that can be easily computed. Also notice that  $Lb_1$  is the bound that we get by solving the linear relaxation of formulation (1).

**Proposition 2.**  *$Lb_1$  is the optimal value of the linear relaxation of the formulation (1).*

*Proof.* For all  $j < k$ , we set each  $y_{a_j}$  to 1 and  $l_{a_j}$  to  $C_j$ . We fix  $l_{a_k}$  to  $(W - \sum_{j=1}^{k-1} C_{a_j})$  and  $y_{a_k}$  to  $l_{a_k}/C_{a_k}$ . For all  $j > k$  we set  $y_{a_j} = 0$  and  $l_{a_j} = 0$ . Constraints (1.3) are thus satisfied. Finally we fix  $x_{i,a_j} = \frac{l_{a_j}}{W}$  for all  $i, j$  so that constraints (1.2) and (1.1) are satisfied. This is a feasible solution of the linear relaxation of (1) achieving an objective value of  $Lb_1$ . We have, therefore,  $Lb_1 \geq z_1^*$  and consequently  $z_1^* = Lb_1$  from Proposition 1.  $\square$

Adding the constraint  $x_{ij} \leq y_j$  for each item  $i$  and bin  $j$ , strengthens the linear relaxation only if  $W < C_{a_k}$ . Indeed, the solution given in the proof is otherwise feasible for the constraint, ( $\forall j < k$ ,  $x_{i,a_j} = \frac{l_{a_j}}{W} \leq y_{a_j} = 1$  and for  $j = k$  we have  $\frac{l_{a_k}}{W} \leq \frac{l_{a_k}}{C_{a_k}}$  if  $W \geq C_{a_k}$ ).

## 2.2 Two Extended Formulations of BPUC

**The Cutting Stock Model.** The formulation of Gilmore and Gomory for the cutting stock problem [10] can be adapted for BPUC. The items of equal size are now grouped and for  $n' \leq n$  different sizes we denote the number of items of sizes  $w'_1, \dots, w'_{n'}$  by  $q_1, \dots, q_{n'}$  respectively. A cutting pattern for bin  $j$  is a combination of item sizes that fits into bin  $j$  using no more than  $q_d$  items of size  $w'_d$ . In the  $i$ -th pattern of bin  $j$ , the number of items of size  $w'_d$  that are in the pattern is denoted  $g_{dij}$ . Let  $I_j$  be the set of all patterns for bin  $j$ . The cost of the  $i$ -th pattern of bin  $j$  is therefore equal to  $co_{ij} = f_j + (\sum_{d=1}^{n'} g_{dij} w'_d) c_j$ . The cutting stock formulation is using a variable  $p_{ij}$  for the  $i$ -th pattern of bin  $j$ :

$$\begin{aligned}
 & \text{Minimize} && z_2 = \sum_{j=1}^m \sum_{i \in I_j} co_{ij} p_{ij} \\
 (2.1) & \sum_{j=1}^m \sum_{i \in I_j} g_{dij} p_{ij} = q_d && \forall d \in \{1, \dots, n'\} \\
 (2.2) & \sum_{i \in I_j} p_{ij} = 1 && \forall j \in \{1, \dots, m\} \\
 (2.3) & p_{ij} \in \{0, 1\} && \forall j \in \{1, \dots, m\}, i \in I_j
 \end{aligned} \tag{2}$$

Constraint (2.1) states that each item has to appear in a pattern (thus in a bin) and (2.2) enforces one pattern to be designed for each bin (convexity constraints). A pattern  $p_{ij}$  for bin  $j$  is valid if  $\sum_{d=1}^{n'} g_{dij} w'_d \leq C_j$  and all  $g_{dij}$  are integers such that  $q_d \geq g_{dij} \geq 0$ . The sets  $I_j$  have an exponential size so the linear relaxation of this model can be solved using column generation. The pricing step is a knapsack problem that can be solved efficiently by dynamic programming if the capacities are small enough.

**The ARC-FLOW Model.** Carvalho introduced an elegant Arc-Flow model for BP [6,7]. His model explicitly uses each unit of capacity of the bins. In the following we show how to adapt it for BPUC. Consider a multi-graph  $G(V, A)$ , where  $V = \{0, 1, \dots, C_{max}\} \cup \{F\}$  is the set of  $C_{max} + 2$  nodes labeled from 0 to  $C_{max}$  and a final node labeled  $F$ , and  $A = I \cup J$  is the set of two kinds of edges. An edge  $(a, b) \in I$  between two nodes labelled  $a \leq C_{max}$  and  $b \leq C_{max}$  represents the use of an item of size  $b - a$ . An edge of  $(a, F) \in J$  for each bin  $j$  represents the usage  $a$  of the bin  $j$ , and therefore  $a \leq C_j$ . An example of

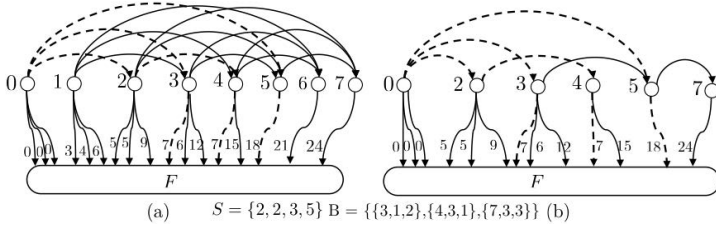


**Fig. 2.** (a) An example of the graph underlying the Arc-Flow model for  $S = \{2, 2, 3, 5\}$ ,  $B = \{\{3, 1, 2\}, \{4, 3, 1\}, \{7, 3, 3\}\}$  so that  $C_{max} = 7$ . A packing is shown using a dotted line:  $\{3\}$  is put in the first bin for a cost of 7,  $\{2, 2\}$  is in the second bin for a cost of 7 and  $\{5\}$  in the last bin for a cost of 18. (b) The graph underlying the Arc-Flow model after the elimination of symmetries.

such a graph is shown in Figure 2(a). Notice that this formulation has symmetries since a packing can be encoded by many different paths. Some reduction rules were given by Carvalho [6], which help in reducing such symmetries (see Figure 2(b)).

BPUC can be seen as a minimum cost flow between 0 and  $F$  with constraints enforcing the number of edges of a given length used by the flow to be equal to the number of items of the corresponding size. We have variables  $x_{ab}$  for each edge  $(a, b) \in I$  as well as variables  $y_{aj}$  for each pair of bin  $j \in \{1, \dots, m\}$  and  $a \in V$ . The cost of using an edge  $(a, F) \in J$  for bin  $j$  with  $a > 0$  is  $co_{aj} = f_j + a \cdot c_j$  and  $co_{0j} = 0$ . The model is as follows:

$$\begin{aligned}
 \text{Minimize} \quad & z_3 = \sum_{j=1}^m \sum_{k=0}^{C_{max}} co_{kj} y_{kj} \\
 (3.1) \quad & \sum_{(a,b) \in A} x_{ab} - \sum_{(b,c) \in A} x_{bc} - \sum_{j=1}^m y_{bj} = \begin{cases} 0 & \forall b \in \{1, 2, \dots, C_{max}\} \\ -m & \text{for } b = 0 \end{cases} \\
 (3.2) \quad & \sum_{a=0}^{C_j} y_{aj} = 1 & \forall j \in \{1, \dots, m\} \\
 (3.3) \quad & \sum_{(k,k+w'_d) \in A} x_{k,k+w'_d} = q_d & \forall d \in \{1, 2, \dots, n'\} \\
 (3.4) \quad & y_{aj} = 0 & \forall (j, a) \in \{1, \dots, m\} \times \{C_j + 1, \dots, C_{max}\} \\
 (3.5) \quad & x_{ab} \in \mathbb{N} & \forall (a, b) \in A \\
 (3.6) \quad & y_{aj} \in \{0, 1\} & \forall (j, a) \in \{1, \dots, m\} \times \{0, \dots, C_{max}\}
 \end{aligned} \tag{3}$$

Constraint (3.1) enforces the flow conservation at each node, and Constraint (3.2) states that each bin should be used exactly once. Constraint (3.3) ensures that all the items are packed, while Constraint (3.4) enforces that bin  $j$  is not used beyond its capacity  $C_j$ . A solution can be obtained again by decomposing the flow into paths. The number of variables in this model is in  $O((n' + m) \cdot C_{max})$  and the number of constraints is  $O(C_{max} + m + n')$ . Although its LP relaxation is stronger than that of Model (1), it remains dominated by that of Model (2).

**Proposition 3.**  $z_3^* \leq z_2^*$ . The optimal value of the linear relaxation of (3) is less than the optimal value of the linear relaxation of (2).

*Proof.* Let  $(p^*)$  be a solution of the linear relaxation of (2). Each pattern  $p_{ij}^*$  is mapped to a path of the Arc-Flow model. A fractional value  $p_{ij}^*$  is added on the arcs corresponding to the item sizes of the pattern (the value of the empty patterns for which all  $g_{dij} = 0$  is put on the arcs  $y_{0j}$ ). The flow conservation (3.1) is satisfied by construction, so is (3.2) because of (2.2) and so are the demand constraints (3.3) because of (2.1). Any solution of (2) is thus encoded as a solution of (3) for the same cost so  $z_3^* \leq z_2^*$ .  $\square$

**Proposition 4.**  $z_2^*$  can be stronger than  $z_3^*$  i.e there exist instances such that  $z_2^* > z_3^*$ .

*Proof.* Consider the following instance:  $S = \{1, 1, 2\}$  and  $B = \{\{3, 1, 1\}, \{3, 4, 4\}\}$ . Two items of size 1 occurs so that  $n' = 2$ ,  $q_1 = 2$ ,  $q_2 = 1$  corresponding to  $w'_1 = 1$ ,  $w'_2 = 2$ . The two bins have to be used and the first *dominates* the second (the maximum possible space is used in bin 1 in any optimal solution) so the optimal solution is the packing  $\{\{2, 1\}, \{1\}\}$  (cost of 12). Let's compute the value of  $z_2^*$ . It must fill the first bin with the pattern  $[g_{111}, g_{211}] = [1, 1]$  for a cost of 4. Only three possible patterns can be used to fill the second bin:  $[0, 0]$ ,  $[1, 0]$  and  $[2, 0]$  (a valid pattern  $p_{12}$  is such that  $g_{112} \leq 2$ ). The best solution is using  $[g_{112}, g_{212}] = [2, 0]$  and  $[g_{122}, g_{222}] = [0, 0]$  taking both a 0.5 value to get a total cost  $z_2^* = 4 + 6 = 10$ . The ARC-FLOW model uses a path to encode the same first pattern  $[1, 1]$  for bin 1. But it can build a path for bin 2 with a  $\frac{1}{3}$  unit of flow taking three consecutive arcs of size 1 to reach a better cost of  $\frac{1}{3} * 16 \approx 5,33$ . This path would be a pattern  $[3, 0]$  which is not valid for (2). So  $z_3^* \approx 9.33$  and  $z_2^* > z_3^*$ .  $\square$

The ARC-FLOW model may use a path containing more than  $q_d$  arcs of size  $w'_d$  with a positive flow whereas no such patterns exist in (3) **because** the sub-problem is subject to the constraint  $0 \leq g_{dij} \leq q_d$ . The cutting stock formulation used in [6] ignores this constraint and therefore the bounds are claimed to be equivalent.

### 2.3 Extending the Bin Packing Global Constraint

A bin packing global constraint was introduced in CP by [19]. We present an extension of this global constraint to handle BPUC. The scope and parameters are as follows:

$$\text{BINPACKINGUSAGECOST}([x_1, \dots, x_n], [l_1, \dots, l_m], [y_1, \dots, y_m], b, z, S, B)$$

Variables  $x_i \in \{1, \dots, m\}$ ,  $l_j \in [0, \dots, C_j]$  and  $b \in \{1, \dots, m\}$  denote the bin assigned to item  $i$ , the load of bin  $j$ , and the number of bins used, respectively. These variables are also used by the BINPACKING constraint. Variables  $y_i \in \{0, 1\}$  and  $z \in \mathbb{R}$  are due to the cost. They denote whether bin  $j$  is open, and the cost of the packing. The last two arguments refers to BPUC and give the size of the items as well as the costs (fixed and unit). In the following,  $\underline{x}$  (resp.  $\bar{x}$ ) denotes the lower (resp. upper) bound of variable  $x$ .

**Cost-Based Propagation Using  $Lb_1$ .** The characteristics of the bins of the restricted BPUC problem based on the current state of the domains of the variables is denoted by  $B'$ , and defined by  $B' = \{\{C'_1, f'_1, c_1\}, \dots, \{C'_m, f'_m, c_m\}\}$  where  $C'_j = \bar{l}_j - l_j$ , and  $f'_j = (1 - \underline{y}_j)f_j$ . The total load that remains to be allocated to the bins is denoted  $\bar{W}' = W - \sum_{j=1}^m \underline{l}_j$ . Notice that we use the lower bounds of the loads rather than the already packed items. We assume it is strictly better due to the reasonings of the bin packing constraint.

**Lower Bound of  $z$ .** The first propagation rule is the update of the lower bound  $\underline{z}$  of  $z$ . The bound is summing the cost due to open bins and minimum loads with the value of  $Lb_1$  on the remaining problem. It gives a maximum possible cost increase  $gap$ :

$$Lb'_1 = \sum_{j=1}^m (\underline{l}_j c_j + \underline{y}_j f_j) + Lb_1(W', B'); \quad \underline{z} \leftarrow \max(\underline{z}, Lb'_1); \quad gap = \bar{z} - Lb'_1 \quad (4)$$

*Bounds of the load variables.* We define the bin packing problem  $B''$  obtained by excluding the space supporting the lower bound  $Lb_1(W', B')$ .  $Lb_1$  is using  $L_j$  units of space on bin  $a_j$ . The bins  $a_1, \dots, a_{k-1}$  are fully used so  $\forall j < k, L_j = C'_{a_j}$ , for bin  $a_k$  we have  $L_k = W' - \sum_{j=1}^{k-1} C'_{a_j}$  and  $\forall j > k, L_j = 0$ . The resulting bins are defined as  $B'' = \{\{C''_1, f'_1, c_1\}, \dots, \{C''_m, f'_m, c_m\}\}$  where  $C''_{a_j} = 0$  for all  $j < k$ ,  $C''_{a_k} = C'_{a_k} - L_k$  and  $C''_{a_j} = C'_{a_j}$  for all  $j > k$ . Lower and upper bounds of loads are adjusted with rules (5).

Let  $q^1_{a_j}$  be the largest quantity that can be removed from a bin  $a_j$ , with  $j \leq k$ , and put at the other cheapest possible place without overloading  $\bar{z}$ . Consequently, when  $j < k$ ,  $q^1_{a_j}$  is the largest value in  $[1, L_j]$  such that  $(Lb_1(q^1_{a_j}, B'') - q^1_{a_j} r_{a_j}) \leq \text{gap}$ . When  $j = k$ , the same reasoning can be done by setting  $C''_{a_k} = 0$  in  $B''$ .

Similarly, let  $q^2_{a_j}$  be the largest value in  $[1, C'_{a_j}]$  that can be put on a bin  $a_j$ , with  $j \geq k$ , without triggering a contradiction with the remaining gap of cost.  $q^2_{a_j}$  is thus the largest value in  $[1, C'_{a_j}]$  such that  $(q^2_{a_j} r_{a_j} - (Lb_1(W', B') - Lb_1(W' - q^2_{a_j}, B'))) \leq \text{gap}$ .

$$\forall j \leq k, \quad \underline{l}_{a_j} \leftarrow \underline{l}_{a_j} + L_j - q^1_{a_j}; \quad \forall j \geq k, \quad \overline{l}_{a_j} \leftarrow \underline{l}_{a_j} + q^2_{a_j}. \quad (5)$$

*Channeling.* The constraint ensures two simple rules relating the load and open-close variables (a bin of zero load can be open):  $y_j = 0 \implies l_j = 0$  and  $l_j > 0 \implies y_j = 1$ .

*Bounds of the open-close variables.* The propagation rule for  $\underline{l}_j$  can derive  $\underline{l}_j > 0$  from (5), which in turn (because of the channeling between  $y$  and  $l$ ) will force a bin to open i.e.  $y_{a_j} \in \{0, 1\}$  will change to  $y_{a_j} = 1$ . To derive that a  $y_j$  has to be fixed to 0, we can use  $Lb_1$  similarly to the reasonings presented for the load variables (checking that the increase of cost for opening a bin remains within the gap).

Tightening the bounds of the load variables can trigger the existing filtering rules of the bin packing global constraint thus forbidding or committing items to bins. Notice that items are only increasing the cost indirectly by increasing the loads of the bins because the cost model is defined by the state of the bins (rather than the items). The cost-based propagation on  $x$  is thus performed by the bin packing global constraint solely as a consequence of the updates on the bin related variables, i.e.  $l$  and  $y$ .

**Algorithms and Complexity.** Assuming that  $B'$  and  $W'$  are available,  $Lb_1(W', B')$  can be computed in  $O(m \log(m))$  time. Firstly we compute the  $r_j$  values corresponding to  $B'$  for all bins. Secondly, we sort the bins in non-decreasing  $r_j$ . Finally, the bound is computed by iterating over the sorted bins and the complexity is dominated by the sorting step. After computing  $Lb_1(W', B')$ , the values  $a_j$  (the permutation of the bins) such that  $r_{a_1} \leq r_{a_2} \leq \dots \leq r_{a_m}$  are available as well as the critical  $k$  and  $L_k = W' - \sum_{j=1}^{k-1} C'$ . The propagation of  $\underline{l}_{a_j}$  and  $\overline{l}_{a_j}$  can then be done in  $O(m)$  as shown in Figure 3.

### 3 Application – Energy Optimization in a Data Centre

The system developed by EnergeTIC is based on a model of the energy consumption of the various components in a data centre, a prediction system to forecast the demand and an optimization component computing the placement of virtual machines onto servers.

**Algorithm 1: UpdateMinimumLoad****Input:**  $a_j$  with  $j \leq k$ ,  $B'$ ,  $gap$ **Output:** a lower bound of  $l_{a_j}$ 

```

1.  $costInc = 0$ ;  $q = 0$ ;  $b = k$ ;
2. If ( $j == k$ )  $\{b = k + 1\}$ ;
3. While ( $q < L_j$  &&  $b \leq m$ )
4.    $loadAdd = \min(L_j - q, C'_{a_b} - L_b)$ ;
5.    $costInc = loadAdd \times (r_{a_b} - r_{a_j})$ ;
6.   If  $((costInc + costInc) > gap)$ 
7.      $q = q + \lfloor \frac{gap - costInc}{r_{a_b} - r_{a_j}} \rfloor$ ;
8.   return  $L_j + l_{a_j} - q$ ;
9.    $costInc = costInc + costInc$ ;
10.   $q = q + loadAdd$ ;  $b = b + 1$ ;
11. return  $\underline{l_{a_j}}$ 

```

**Algorithm 2: UpdateMaximumLoad****Input:**  $a_j$  with  $j \geq k$ ,  $B'$ ,  $gap$ **Output:** an upper bound of  $l_{a_j}$ 

```

1.  $costInc = 0$ ;  $q = 0$ ;  $b = k$ ;
2. If ( $j == k$ )  $\{q = L_k$ ;  $b = k - 1\}$ ;
3. While ( $q < C'_{a_j}$  &&  $b \geq 0$ )
4.    $loadAdd = \min(L_b, C'_{a_j} - q)$ ;
5.    $costInc = loadAdd \times (r_{a_j} - r_{a_b})$ ;
6.   If  $((costInc + costInc) > gap)$ 
7.      $q = q + \lfloor \frac{gap - costInc}{r_{a_j} - r_{a_b}} \rfloor$ ;
8.   return  $\underline{l_{a_j}} + q$ ;
9.    $costInc = costInc + costInc$ ;
10.   $q = q + loadAdd$ ;  $b = b - 1$ ;
11. return  $\underline{l_{a_j}}$ 

```

**Fig. 3.** Propagation algorithms for updating the lower and upper bounds of the load variables

**Energy Model.** In the last decade green data centres have focused on limiting the amount of energy that is not used for running the client's applications. The Power Usage Effectiveness (PUE) is a key indicator introduced by the Green Grid consortium [1] which measures the ratio between the total energy consumption of the data centre and the energy used by its IT systems (e.g., servers, networks, etc.). A value of 1 is the perfect score. The current average in industry is around 1.7 and the most efficient data centres are reaching 1.4 or even 1.2. As not all electrical power consumed by the IT equipment is transformed into a useful work product, the need to refine such a metric arose quickly. Therefore, the Green Grid proposed a very fine grained indicator for that purpose [1]. This metric, although very accurate, is not really used in practice because of its complexity and no consensus has been reached for a practical and relevant indicator. The EnergeTIC project introduced a new energy indicator which is defined as the ratio between the total energy consumed and the energy really used to run clients' applications. This indicator however relies on a model of the energy consumption of each equipment. A system, based on three different servers (quad-, bi- and mono- processor) with different energy behaviors, was provided by Bull to perform the measurements. As an example, the energy cost of the power consumption of three different servers at different cpu loads taken from one of the problem instances is shown in Figure 4.

**Demand Model.** The demands of the real benchmarks used in the experimental section are coming from the Green Data Centre of Business & Decision Eolas located in Grenoble. It was used to study and validate the system operationally. It is instrumented with thousands of sensors spread over the site to monitor the energy consumption of the centre and claims a PUE between 1.28 and 1.34. It deals with an heterogeneous demand: web applications, e-commerce, e-business, e-administrations. An example showing variable requirements of CPU usage over 24 time-periods for multiple virtual machines taken from one of the problem instances is shown in Figure 5.



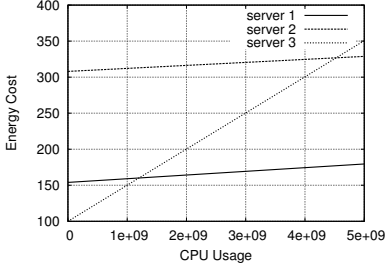


Fig. 4. Energy cost vs CPU Usage for 3 servers

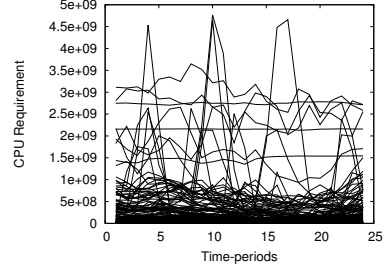


Fig. 5. Variable demands of virtual machines

### 3.1 Problem Description and Notation

The problem is to place a set of virtual machines on a set of servers over multiple time-periods to minimize the energy cost of the data center. The CPU usage of a VM is changing over time. At each period, we must ensure that the virtual machines have enough resources (CPU and memory). Let  $VM = \{v_1, \dots, v_n\}$  be the set of virtual machines,  $SE = \{s_1, \dots, s_m\}$  be the set of servers and  $T = \{p_1, \dots, p_h\}$  be the set of periods.

**Virtual Machines.** A virtual machine  $v_i$  is characterized by a memory consumption  $M_i$  independent of the time-period, a set  $SA_i \subseteq S$  of allowed servers where it can be hosted, and a potential initial server (for time-period  $p_0$ ) denoted by  $I_{serv_i}$  (which might be unknown). A virtual machine  $v_i$  has a CPU consumption  $U_{it}$  at time-period  $t$ .

**Servers.** A server  $s_j$  can be in two different states: ON=1 or STBY=0 (stand-by). It is characterized by: a CPU capacity  $Umax_j$ ; a memory capacity  $Mmax_j$ ; a fixed cost of usage  $Emin_j$  (in Watts); a unit cost  $\tau_j$  per unit of used capacity; a basic CPU consumption  $Ca_j$  when it is ON (to run the operating system and other permanent tasks); an energy consumption  $Estby_j$  when it is in state STBY; an energy consumption  $Estaj$  to change the state of the server from STBY to ON; an energy consumption  $Estoj$  to change the state of the server from ON to STBY; a maximum number  $Nmax_j$  of virtual machines that can be allocated to it at any time period; a set of periods  $P_j \subseteq T$  during which  $s_j$  is forced to be ON; and a potential initial state  $Istate_j \in \{0, 1\}$ .

If a server is ON, its minimum cost is  $Emin_j + \tau_j Ca_j$ , and if it is STBY, its cost is  $Estby_j$ . For the sake of simplicity, to compute the fixed energy cost of an active server we include the basic consumption  $Ca_j$  and the standby energy  $Estby_j$  in  $Emin_j$  so that  $Emin'_j = Emin_j - Estby_j + \tau_j Ca_j$ . This way we can state the BINPACKINGUSAGECOST directly with the semantic given earlier by adding the constant  $\sum_{s_j \in SE} Estby_j$  in the final objective value. We also shift the CPU capacity of the servers:  $Umax'_j = Umax_j - Ca_j$ .

**Migrations.** The maximum number of changes of servers among all virtual machines from one period to the next is denoted by  $N$  and the cost of a migration by  $Cmig$ .

The problem can be seen as a series of cost-aware bin packing problems (one per period) in two dimensions (CPU and memory) that are coupled by the migration constraints and the cost for changing the state of a server. Figure 6 gives an overview of the problem. This example has four servers, each shown by a rectangle whose dimensions

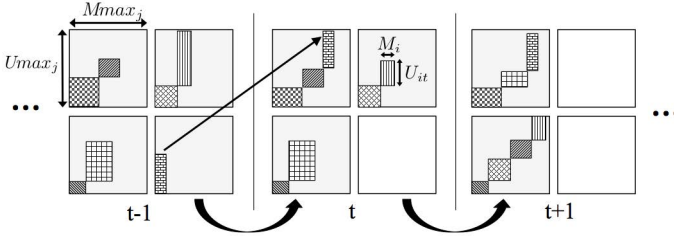


**Fig. 6.** A solution over three time periods. Virtual machines migrate to turn off two servers at  $t + 1$ .

$$\begin{aligned}
 &\text{Minimize } \sum_{s_j \in SE} \sum_{t \in T} (Estab_{jt} + Estob_{jt} + \tau_j cpu_{jt} + Enin'_j o_{jt}) + Cmig(\sum_{v_i \in VM} \sum_{t \in T} a_{it}) \\
 (6.1) \quad &\sum_{s_j \in SE} x_{ijt} = 1 && (\forall v_i \in VM, p_t \in T) \\
 (6.2) \quad &x_{ijt} = 0 && (\forall v_i \in VM, p_t \in T, s_j \notin SA_i) \\
 (6.3) \quad &x_{ijt} \leq o_{jt} && (\forall v_i \in VM, p_t \in T, s_j \in SE) \\
 (6.4) \quad &cpu_{jt} = \sum_{v_i \in VM} U_{it} x_{ijt} && (\forall s_j \in SE, p_t \in T) \\
 (6.5) \quad &cpu_{jt} \leq Umax'_j o_{jt} && (\forall s_j \in SE, p_t \in T) \\
 (6.6) \quad &\sum_{v_i \in VM} M_i x_{ijt} \leq Mmax_j o_{jt} && (\forall s_j \in SE, p_t \in T) \\
 (6.7) \quad &\sum_{v_i \in VM} x_{ijt} \leq Nmax_j o_{jt} && (\forall s_j \in SE, p_t \in T) \\
 (6.8) \quad &mig_{ijt} \geq x_{ijt} - x_{ijt-1} && (\forall v_i \in VM, s_j \in SE, p_t \in T) \\
 (6.8') \quad &a_{it} \geq \sum_{s_j \in SE} mig_{ijt} && (\forall v_i \in VM, p_t \in T) \\
 (6.9) \quad &\sum_{v_i \in VM} a_{it} \leq N && (\forall p_t \in T) \\
 (6.10) \quad &bto_{jt} \geq o_{jt} - o_{jt-1} && (\forall s_j \in SE, p_t \in T) \\
 (6.11) \quad &otb_{jt} \geq o_{jt-1} - o_{jt} && (\forall s_j \in SE, p_t \in T) \\
 (6.12) \quad &o_{jt} = 1 && (\forall s_j \in SE, p_t \in P_j) \\
 (6.13) \quad &x_{ij0} = 0 && (\forall v_i \in VM, s_j \in SE - \{Iserv_i\}) \\
 (6.14) \quad &x_{i,Iserv_i,0} = 1 && (\forall v_i \in VM) \\
 (6.15) \quad &o_{j0} = Istate_j && (\forall s_j \in SE)
 \end{aligned} \tag{6}$$

are representing the CPU and memory capacities. A virtual machine is a small rectangle whose height (its CPU) varies from one period to the next. In this scenario, the CPU needs of some virtual machines decrease allowing to find better packings and turn off two servers at  $t + 1$ .

### 3.2 An Integer Linear Model

We present the integer linear model of the problem in which the following variables are used:  $x_{ijt} \in \{0, 1\}$  indicates whether virtual machine  $v_i$  is placed on server  $s_j$  at time  $t$ .  $cpu_{jt} \in [0, Umax'_j]$  gives the CPU consumption of  $s_j$  at period  $t$ .  $o_{jt} \in \{0, 1\}$  is set to 1 if  $s_j$  is ON at time  $t$ , 0 otherwise.  $bto_{jt} \in \{0, 1\}$  is set to 1 if  $s_j$  was in STBY at  $t - 1$  and is turned ON at  $t$ .  $otb_{jt} \in \{0, 1\}$  is set to 1 if  $s_j$  was in ON at  $t - 1$  and is put STBY at  $t$ .  $mig_{ijt} \in \{0, 1\}$  is set to 1 if  $v_i$  is on  $s_j$  at time  $t$  and was on a different server at  $t - 1$ .  $a_{it} \in \{0, 1\}$  is set to 1 if  $v_i$  is on a different server at  $t$  than the one it was using at  $t - 1$ .

The initial state is denoted by  $t = 0$ . The model is summarized in Model (6). We omit the constant term  $\sum_{s_j \in SE} Estby_j$  from the objective function. Constraint (6.1) states that a virtual machine has to be on a server at any time; (6.2) enforces the forbidden servers

for each machine; (6.3) enforces a server to be ON if it is hosting at least one virtual machine; (6.4) links the CPU load of a server to the machines assigned to it. (6.5–6.7) are the resource constraints (CPU, memory and cardinality) of each server; (6.8, 6.8', 6.9) allow us to count the number of migrations and state the limit on  $N$  (6.8 and 6.8' together give a stronger linear relaxation than the single  $a_{it} \geq x_{ijt} - x_{ijt-1}$ ); (6.10–6.11) keeps track of the change of states of the servers; (6.12) states the periods where a server has to be ON; (6.13–6.15) enforce the initial state ( $t = 0$ ). The number of constraints of this model is dominated by the  $n \times m \times h$  number of (6.8) and (6.3).

### 3.3 Lower Bound – An Extended Formulation

Solving large-sized instances of this application domain within short time limits is beyond the capability of exact algorithms. Therefore, one is generally forced to use an incomplete approach. Although an incomplete approach like large neighborhood search can usually find feasible solutions quickly, their qualities are often not evaluated as no bounds or provable approximation ratio can be found in the literature. Hence, it is important to be able to compute tighter lower bounds. In this section we present a column generation-based approach for computing a lower bound. Although we focus on a lower bound for the particular formulation (6), we believe it is generic enough to be relevant to other closely related problems of the literature that have at their core a series of cost-aware bin packing problems coupled with cost/migration constraints.

Let  $b_{kt} \in \{0, 1\}$  be a variable for each bin packing of each time period to know whether the packing  $k$  is used for time period  $t$ . The set of all packings for period  $t$  is denoted by  $\Omega_t$ . The packing  $k$  of period  $t$  is characterized by its cost  $c_{kt}$ , the server where each virtual machine is run and the state of each server. We use  $x_{kijt} = 1$  if  $v_i$  is placed on  $s_j$  in the packing  $k$  at time period  $t$  and  $o_{kjt} = 1$  if server  $s_j$  is ON in the packing  $k$ . In addition to  $b_{kt}$ , the variables  $bto_{jt}$ ,  $otb_{jt}$ ,  $a_{it}$  and  $mig_{ijt}$  that we have already introduced for (6) are used in the column generation model (7). The restricted master problem is defined for a restricted number of packing variables ( $\forall t \leq m$ ,  $b_{kt} \in \Omega'_t \subset \Omega_t$ ):

$$\begin{aligned}
 \text{Minimize } z_4 &= \sum_{t \in T} (\sum_{s_j \in SE} (Est_{sj} bto_{jt} + Esto_{sj} otb_{jt}) + \sum_{k \in \Omega_t} c_{kt} b_{kt} + \sum_{v_i \in VM} Cmig_{it}) \\
 (7.1) \quad \sum_{k \in \Omega_t} b_{kt} &= 1 & (\forall p_t \in T) & (\lambda_t) \\
 (7.2) \quad mig_{ijt} &\geq \sum_{k \in \Omega_t} x_{kijt} b_{kt} - \sum_{k \in \Omega_t} x_{k,i,j,t-1} b_{k,t-1} & (\forall v_i \in VM, s_j \in SE, p_t \in T) & (\pi_{ijt}) \\
 (7.3) \quad a_{it} &\geq \sum_{s_j \in SE} mig_{ijt} & (\forall v_i \in VM, p_t \in T) & \\
 (7.4) \quad \sum_{v_i \in VM} a_{it} &\leq N & (\forall p_t \in T) & \\
 (7.5) \quad bto_{jt} &\geq \sum_{k \in \Omega_t} o_{kjt} b_{kt} - \sum_{k \in \Omega_t} o_{k,j,t-1} b_{k,t-1} & (\forall s_j \in SE, p_t \in T) & (\alpha_{jt}) \\
 (7.6) \quad otb_{jt} &\geq \sum_{k \in \Omega_t} o_{k,j,t-1} b_{k,t-1} - \sum_{k \in \Omega_t} o_{kjt} b_{kt} & (\forall s_j \in SE, p_t \in T) & (\beta_{jt})
 \end{aligned} \tag{7}$$

Let  $\lambda_t$ ,  $\pi_{ijt}$ ,  $\alpha_{jt}$  and  $\beta_{jt}$  be the dual variables of constraints (7.1), (7.2), (7.5) and (7.6) respectively. We have  $h$  independent pricing problems and for each time period  $t$  we are looking for a negative reduced cost packing. The number of constraints (7.2) can prevent us from solving the relaxation of the master problem alone. We therefore turned to a relaxation of the migration constraints. The rationale is that the migration cost is really dominated by the server costs. Let  $nmig_{jt} \in \mathbb{N}$  be the number of migrations occurring on server  $j$  and  $u_{kjt} = \sum_{i \in VM} x_{kijt}$  the number of virtual machines allocated

to server  $j$  in the  $k$ -th packing of time  $t$ . We suggest removing the  $a$  and  $mig$  variables from formulation (7), adding the  $nmig$  variables instead and replacing (7.2)–(7.4) by :

$$\begin{aligned} (7.2') \quad nmig_{jt} &\geq \sum_{k \in \Omega_t} u_{kjt} b_{kt} - \sum_k u_{k,j,t-1} b_{k,t-1} \quad (s_j \in SE, p_t \in T) & (\pi_{jt}) \\ (7.3') \quad \sum_{j \in SE} nmig_{jt} &\leq N & (\forall p_t \in T) & (\gamma_t) \end{aligned}$$

The last term in the objective is replaced by  $Cmig(\sum_{t \in T} \sum_{j \in SE} nmig_{jt})$ . The pricing problem for period  $t$  can now be seen as a cost-aware bin packing problem with an extra cost related to the number of items assigned to a bin and two side constraints: a cardinality and memory capacity constraint. The reduced cost  $r_{kt}$  of packing  $b_{kt}$  is equal to

$$r_{kt} = c_{kt} - \sum_{j \in SE} (o_{kjt}(-\alpha_{jt} + \alpha_{j,t+1} + \beta_{jt} - \beta_{j,t+1}) + u_{kjt}(-\pi_{jt} + \pi_{j,t+1})) - \lambda_t \quad (8)$$

For each bin  $j$ , the fixed and unit costs can be set to  $f_j = Emin'_j - (-\alpha_{jt} + \alpha_{j,t+1} + \beta_{jt} - \beta_{j,t+1})$  and  $c_j = \tau_j$  respectively. The cost depending on the number of items placed in bin  $j$  is denoted  $\tau c_j = -(-\pi_{jt} + \pi_{j,t+1})$ . Ignoring the constant term  $-\lambda_t$  of the objective function, we summarize the pricing problem of period  $t$  by a CP model:

$$\begin{aligned} &\textbf{Minimize } r_t = ccpu + ccard \\ (9.1) \quad &\text{BINPACKINGUSAGECOST}([x_1, \dots, x_n], [cpu_1, \dots, cpu_m], [y_1, \dots, y_m], nbb, ccpu, \\ &\quad [U_{1t}, \dots, U_{1n}], [(f_1, \tau_1), \dots, (f_m, \tau_m)]) \\ (9.2) \quad &\text{BINPACKINGUSAGECOST}([x_1, \dots, x_n], [oc_1, \dots, oc_m], [y_1, \dots, y_m], nbb, ccard, \\ &\quad [1, \dots, 1], [(0, \tau c_1), \dots, (0, \tau c_m)]) \\ (9.3) \quad &\text{BINPACKING}([x_1, \dots, x_n], [mem_{1t}, \dots, mem_{mt}], nbb', [M_{1t}, \dots, M_{mt}]) \\ (9.4) \quad &nbb \geq nbb' \\ (9.5) \quad &\text{GLOBALCARDINALITY}([x_1, \dots, x_n], [oc_1, \dots, oc_m]) \\ (9.6) \quad &y_j \begin{cases} = 1 & \text{if } p_t \in P_j \text{ or } f_j \leq 0 \\ \in \{0, 1\} & \text{otherwise} \end{cases} \quad (\forall s_j \in SE) \end{aligned} \quad (9)$$

Each variable  $x_i \in SA_i$  gives the bin where item  $v_i$  is placed.  $cpu_j \in [0, Umax'_j]$  and  $mem_j \in [0, Mmax_j]$  encode the cpu and memory load of bin  $j$ , respectively. The number of items placed in bin  $j$  is given by  $oc_j \in \{0, \dots, n\}$  and  $y_j \in \{0, 1\}$  indicates if bin  $j$  is ON or not. The number of bins used is  $nbb \in \{1, \dots, m\}$  ( $nbb'$  is an intermediate variable). Finally  $ccpu \geq 0$  and  $ccard \geq 0$  are real variables representing the costs related to CPU and cardinality. The costs are expressed by the state of the bins, thus matching the model of Section 2. A negative  $f_j$  is handled by pre-fixing  $y_j$  to 1 (constraint (9.6)).

**Dual Bound.** The bottleneck of this method is the hardness of the pricing step, as proving that no negative reduced cost packing exist is unlikely to be tractable. At any iteration, if the optimal reduced costs  $r^* = (r_1^*, \dots, r_h^*)$  are known, a well-known lower bound of the linear relaxation of the master is  $w_4 = z_4^* + \sum_{t \in P_t} r_t^*$  where  $z_4^*$  is the current optimal value of the **restricted** master at this iteration. Indeed, since  $r_t^*$  is the best reduced cost for period  $t$ ,  $\forall k \in \Omega_t$ ,  $r_{kt} \geq r_t^*$ , and using (8) we have the following:

$$\forall k \in \Omega_t, c_{kt} \geq r_t^* + \sum_{j \in SE} (o_{kjt}(-\alpha_{jt} + \alpha_{j,t+1} + \beta_{jt} - \beta_{j,t+1}) + u_{kjt}(-\pi_{jt} + \pi_{j,t+1})) + \lambda_t.$$

This shows that the solution  $(\gamma, \pi, \alpha, \beta, \lambda + r^*)$  is dual feasible for the master which explains  $w_4$ . Now this reasoning also holds for any value smaller than  $r_t^*$ . Therefore

we still get a valid lower bound  $\underline{w}_4$  if we use a lower bound  $\underline{r}_t^*$  of each  $r_t^*$  and  $\underline{w}_4 = z_4^* + \sum_{t \in P_t} \underline{r}_t^* \leq w_4$ . We note that this algorithm can therefore return a valid bound without succeeding in solving a single pricing problem to optimality. At the moment, the pricing problem is solved using a linear solver with a time-limit of three seconds so the best bound is used for  $\underline{r}_t^*$  if the time limit is reached. This is critical for scaling with sub-problem size. We can always return the best  $\underline{w}_4$  found over all iterations. In practice we terminate when the gap between  $\underline{w}_4$  and  $z_4^*$  is less than 0.1%.

### 3.4 Upper Bounds

The EnergeTIC team initially designed a MIP model that was embedded in their platform but it failed to scale. The details of this model are not reported here. We investigated three different approaches for computing upper bounds. The first approach is the MIP model (6) of Section 3.2 which is an improvement of the model designed by EnergeTIC. The second approach which we call Temporal Greedy (TG) is currently employed in their platform. It proceeds by decomposing time and is more scalable. It greedily solves the problem period by period using model (6) restricted to one period (enforcing the known assignment of the previous period). Each time-period is used as a starting period as long as there is time left, and therefore, if required, the assignment is extended in both directions (toward the beginning and toward the end). The last one is a large neighborhood search (LNS)[14], which was originally developed for the machine reassignment problem of 2012 ROADEF Challenge which had only 1 time-period. Therefore we extended it in order to handle multiple time-periods.

## 4 Experimental Results

**Cost-Aware Bin Packing Benchmarks.** We first compare on randomly generated instances the lower bounds  $z_1^*, z_2^*, z_3^*$  as well as exact algorithms: Model (1), ARC-FLOW Model (3) and a CP model using the `BINPACKINGUSAGECOST` constraint. Standard symmetry/dominance breaking techniques for BP are applied to the MIP [18] of Model (1) and CP [19]. A random instance is defined by  $(n, m, X)$ , where  $n$  is the number of items ( $n \in \{15, 25, 200, 250, 500\}$ ),  $m$  is the number of bins ( $m \in \{10, 15, 25, 30\}$ ), and parameter  $X \in \{1, 2, 3\}$  denotes that the item sizes are uniformly randomly generated in the intervals  $[1, 100]$ ,  $[20, 100]$ , and  $[50, 100]$  respectively. The capacities of the bins are picked randomly from the sets  $\{80, 100, 120, 150, 200, 250\}$  and  $\{800, 1000, 1200, 1500, 2000, 2500\}$  when  $n \in \{15, 25\}$  and  $n \in \{200, 250, 500\}$  respectively. The fixed cost of each bin is set to its capacity and the unit cost is randomly picked from the interval  $[0, 1]$ . For each combination of  $(n, m) \in \{(15, 10), (25, 15), (25, 25), (200, 10), (250, 15), (500, 30)\}$  and  $X \in \{1, 2, 3\}$  we generated 10 instances giving 180 instances in total.

The time-limit was 600 seconds. If an approach failed to solve an instance within the time-limit then 600 was recorded as its solution time. All the experiments were carried out on a Dual Quad Core Xeon CPU, running Linux 2.6.25 x64, with 11.76 GB of RAM, and 2.66 GHz processor speed. The LP solver used was CPLEX 12.5 (default parameters) and the CP solver was Choco 2.1.5. Table 1 reports results for some classes due to lack of space. We report the average cpu time (denoted `cpu`) and the average

**Table 1.** Comparison of bounds obtained using MIP, ARC-FLOW, CP, and Cutting-Stock approaches on random bin packing with usage cost problem instances with 600 seconds time-limit

| n   | m  | X | best ub | MIP     |         |       |             | CP      |       |             |         | ARC-FLOW |         |             |               | Cutting-Stock |
|-----|----|---|---------|---------|---------|-------|-------------|---------|-------|-------------|---------|----------|---------|-------------|---------------|---------------|
|     |    |   |         | $z_1^*$ | ub      | #nu   | cpu         | ub      | #nu   | cpu         | $z_1^*$ | ub       | #nu     | cpu         | $z_1^*$       | $z_2^*$       |
| 15  | 10 | 1 | 1005.2  | 956.8   | 1005.2  | 0 (0) | 1.2         | 1005.2  | 0 (0) | <b>0.5</b>  | 959.6   | 1005.2   | 0 (0)   | 2.1         | <b>960.3</b>  |               |
| 15  | 10 | 2 | 1267.4  | 1230.5  | 1267.4  | 0 (0) | 1.1         | 1267.4  | 0 (0) | <b>0.2</b>  | 1244.5  | 1267.4   | 0 (0)   | 0.7         | <b>1245.0</b> |               |
| 15  | 10 | 3 | 1574.5  | 1522.3  | 1574.5  | 0 (0) | 0.8         | 1574.5  | 0 (0) | 0.7         | 1553.0  | 1574.5   | 0 (0)   | <b>0.6</b>  | <b>1553.5</b> |               |
| 25  | 15 | 1 | 1665.6  | 1636.3  | 1665.6  | 0 (0) | 35.1        | 1665.6  | 0 (0) | <b>24.0</b> | 1638.9  | 1665.6   | 1 (0)   | 42.7        | <b>1639.0</b> |               |
| 25  | 15 | 2 | 2127.1  | 2086.4  | 2127.1  | 0 (0) | 74.2        | 2127.1  | 0 (0) | <b>12.9</b> | 2094.6  | 2127.1   | 0 (0)   | 61.2        | <b>2094.9</b> |               |
| 25  | 15 | 3 | 2682.8  | 2613.1  | 2682.8  | 0 (0) | 22.6        | 2685.6  | 2 (0) | 144.0       | 2657.9  | 2682.8   | 0 (0)   | <b>11.3</b> | 2657.9        |               |
| 500 | 30 | 1 | 32387.2 | 32187.0 | 32387.2 | 0 (0) | <b>18.1</b> | 32387.2 | 0 (0) | 57.6        | 32187.0 | -        | 10 (10) | 600         | 32187.0       |               |
| 500 | 30 | 2 | 40422.7 | 40235.8 | 40513.5 | 3 (0) | 301.2       | 40422.7 | 0 (0) | 34.2        | 40235.8 | -        | 10 (10) | 600         | 40235.8       |               |
| 500 | 30 | 3 | 53395.6 | 53236.3 | -       | 9 (2) | 558.5       | 53395.6 | 3 (0) | 201.3       | 53236.3 | -        | 10 (10) | 600         | 53236.3       |               |

value of upper/lower-bounds found (denoted  $ub / z_x^*$ ) (when a value is found for each instance of the class). Column #nu is a pair  $x(y)$  giving the number of instances  $x$  (resp.  $y$ ) for which an approach failed to prove optimality (resp. to find a feasible solution).

For the cutting-stock approach upper-bounds are not shown as the branch-and-price algorithm was not implemented. The CP approach shows better performance when scaling to larger size instances (and capacities) than the MIP and ARC-FLOW models.

**EnergeTIC Benchmarks.** The industry partners provided 74 instances, where the maximum number of virtual machines (items), servers (bins), and time-periods are 242, 20 and 287 respectively.<sup>2</sup> The time-limit is 600 seconds. As mentioned previously, we compared three approaches for computing upper bounds: the MIP model, the Temporal Greedy approach (TG) currently used by EnergeTIC, and large neighborhood search (LNS) [14]. We also analyzed the lower bounds provided by the linear relaxation of the MIP model (LP), the best lower bound established by MIP when reaching the time-limit (MIP LB), and the bound provided by the linear relaxation of formulation (7) (CG). Table 2 gives an overview of the results by reporting (over the 74 instances) the average/median/max values of the **gap** to the best known bound<sup>3</sup>, the **cpu** time, and the number of instances **#nu** when an approach fails to return any results within the time-limit. Table 2 also gives the results for a few hard instances.

*Upper Bounds.* Out of 74 instances, MIP was able to find solutions for 71 instances within the time-limit out of which 54 are proved optimal. It thus failed for 3 instances where the space requirement for CPLEX exceeded 11GB. Notice that the largest size instance has 1,389,080 decision variables. Clearly, MIP-based systematic search cannot scale in terms of time and memory. TG is able to find solutions for 73 instances (so it failed on one instance), out of which 26 are optimal. Its quality deteriorates severely when one should anticipate expensive peaks in demand by placing adequately virtual machines several time periods before the peak. This can be seen in Table 2 where the maximum gap is 119.35%. LNS succeeds to find feasible solutions for all instances within 2 seconds, on average, but it was terminated after 600 seconds and for 41

<sup>2</sup> The benchmarks are available from [http://www.4c.ucc.ie/\\$\sim\\$dm6/energetic.tar.gz](http://www.4c.ucc.ie/$\sim$dm6/energetic.tar.gz)

<sup>3</sup> The gaps for lower and upper bounds are computed as  $\frac{100 \times (best\_ub - lb)}{best\_ub}$  and  $\frac{100 \times (ub - best\_lb)}{best\_lb}$  respectively. To compute mean/average/max values of gaps or time of a given approach, we exclude the instances where it fails to return any value (no feasible solution or a zero lower bound).

**Table 2.** Comparison between lower and upper bounds of the various approaches with 600 seconds time-limit (over 74 instances in the first part of the table and on a few specific instances in the second part)

|     |        | Lower bounds |          |             |                 |        |          |          |          | Upper bounds |        |          |       |     |     |
|-----|--------|--------------|----------|-------------|-----------------|--------|----------|----------|----------|--------------|--------|----------|-------|-----|-----|
|     |        | LP           |          | CG          |                 | MIP LB |          | LNS      |          | MIP          |        | TG       |       |     |     |
|     |        | gap          | cpu      | gap         | cpu             | gap    | cpu      | gap      | cpu      | gap          | cpu    | gap      | cpu   | gap | cpu |
|     | Mean   | 9,64         | 3,13     | <b>0,32</b> | 23,31           | 0,90   | 191,92   | 0,51     | 0,03     | 191,92       | 7,00   | 42,50    |       |     |     |
|     | Median | 8,33         | 0,23     | <b>0,10</b> | 1,3             | 0      | 2,67     | 0        | 0        | 2,67         | 0,06   | 1,45     |       |     |     |
|     | Max    | 58,36        | 95,66    | <b>7,14</b> | 600             | 26,42  | 600      | 4,58     | 0,74     | 600          | 119,35 | 600      |       |     |     |
|     | #nu    | 3            |          | <b>0</b>    |                 | 3      |          | <b>0</b> | 3        |              | 1      |          |       |     |     |
| n   | m      | h            | value    | cpu         | value           | cpu    | value    | cpu      | value    | cpu          | value  | cpu      | value | cpu |     |
| 32  | 3      | 96           | 23492,8  | 9,5         | <b>25404,7</b>  | 15,2   | 25043,6  | 600      | 25586,7  | 25575,7      | 600    | 36049,7  | 112,3 |     |     |
| 36  | 3      | 287          | 122831,3 | 4,5         | <b>126716,8</b> | 132,2  | 126597,9 | 600      | 127018,6 | 127654,4     | 600    | 127036,6 | 600   |     |     |
| 242 | 20     | 24           | 0        | 600         | <b>37482,5</b>  | 600    | 0        | 600      | 40362,5  | -            | 600    | 43027,6  | 14,2  |     |     |
| 242 | 20     | 24           | 0        | 600         | <b>36890,8</b>  | 24,2   | 0        | 600      | 37701,6  | -            | 600    | 36897,4  | 600   |     |     |
| 242 | 20     | 287          | 0        | 600         | <b>431704,0</b> | 600    | 0        | 600      | 439926,2 | -            | 600    | -        | 600   |     |     |
| 90  | 7      | 8            | 10420,7  | 14,4        | <b>11431,9</b>  | 0,2    | 11236,3  | 600      | 11728,2  | 11435,3      | 600    | 11435,5  | 1,5   |     |     |

instances it found optimal solutions. Its average gap to the best known lower bound is less than 0.5% showing that LNS scales very well both in quality and problem size.

*Lower bounds.* The LP bound can be very far from the optimal value (its maximum gap is 58.36%) and does not scale since it fails on 3 instances even with 2 hour time-limit. The MIP obviously fails if the LP has failed. However, when solving the MIP, CPLEX automatically strengthens the formulation which allows us to solve many instances optimally where the LP bound was initially quite bad. Nevertheless, even after search there are cases where the gap can remain quite large (maximum of 26.42%). CG exhibits very good behaviour. Firstly, its gap clearly outperforms other bounds. Secondly, it can be stopped at any time and returns its current best master dual bound which is why #nu is 0 even though the time limit is reached on two cases (shown in Table 2). The first would improve to 38614.3 in 2000s whereas the second converges in 700 seconds without any improvement. Tables 2 shows that CG scales well both in terms of quality and size.

## 5 Conclusion and Future Work

Many optimisation problems in data centres can be described as a series of consecutive multi-dimensional Bin Packing with Usage Costs (BPUC) problems coupled by migration constraints and costs. First, we studied the lower bounds of a critical variant of bin packing for this domain that includes linear usage costs. We designed a CP approach that gives, so far, the best algorithm to solve BPUC exactly. Secondly, the usefulness of the exact algorithm and the efficient bounds for BPUC is shown within a column generation approach for the energy cost optimisation problem arising in data centres. These bounds are evaluated experimentally on real benchmarks and they assert the efficiency of the LNS approach [14] which was extended to handle consecutive BPUC problems.

The next step is to generalize the Martello and Toth bound  $L_2$  [13] to the linear cost function which should improve the `BINPACKINGUSAGECOST` global constraint. We also plan to evaluate both column generation and LNS approaches on even larger size instances. We intend to solve the pricing problem with CP as we believe it can scale better for larger size problems.

## References

1. A framework for data center energy productivity. Technical report, The green grid (2008)
2. Efficiency des Data Centers, les retombées du projet EnergeTIC. Technical report (2013), [http://www.vesta-system.cades-solutions.com/images/vestalis/4/energetic\\_white20paper.pdf](http://www.vesta-system.cades-solutions.com/images/vestalis/4/energetic_white20paper.pdf)
3. Anily, S., Bramel, J., Simchi-Levi, D.: Worst-case analysis of heuristics for the bin packing problem with general cost structures. *Operational Research* 42 (1994)
4. Baldi, M.M., Crainic, T.G., Perboli, G., Tadei, R.: The generalized bin packing problem. *Transportation Research Part E: Logistics and Transportation Review* 48(6), 1205–1220 (2012)
5. Crainic, T.G., Perboli, G., Rei, W., Tadei, R.: Efficient lower bounds and heuristics for the variable cost and size bin packing problem. *Comput. Oper. Res.* 38(11), 1474–1482 (2011)
6. de Carvalho, J.M.V.: Exact solution of bin packing problems using column generation and branch-and-bound. *Annals of Operations Research* 86(0), 629–659 (1999)
7. de Carvalho, J.M.V.: LP models for bin packing and cutting stock problems. *European Journal of Operational Research* 141(2), 253–273 (2002)
8. Dupont, C., Schulze, T., Giuliani, G., Somov, A., Hermenier, F.: An energy aware framework for virtual machine placement in cloud federated data centres. In: *Proceedings of the 3rd International Conference on Future Energy Systems: Where Energy, Computing and Communication Meet, e-Energy 2012*, pp. 4:1–4:10. ACM, New York (2012)
9. Epstein, L., Levin, A.: Bin packing with general cost structures. *Math. Program.* 132(1-2), 355–391 (2012)
10. Gilmore, P.C., Gomory, R.E.: A linear programming approach to the cutting-stock problem. *Operations research* 11, 863–888 (1963)
11. Haouari, M., Serairi, M.: Relaxations and exact solution of the variable sized bin packing problem. *Comput. Optim. Appl.* 48(2), 345–368 (2011)
12. Li, C.L., Chen, Z.L.: Bin-packing problem with concave costs of bin utilization. *Naval Research Logistics* 53, 298–308 (2006)
13. Martello, S., Toth, P.: Lower bounds and reduction procedures for the bin packing problem. *Discrete Applied Mathematics* 28(1), 59–70 (1990)
14. Mehta, D., O’Sullivan, B., Simonis, H.: Comparing solution methods for the machine reassignment problem. In: Milano, M. (ed.) *CP 2012. LNCS*, vol. 7514, pp. 782–797. Springer, Heidelberg (2012)
15. Monaci, M.: *Algorithms for Packing and Scheduling Problems*. PhD thesis, Universit di Bologna (2012)
16. Petrucci, V., Loques, O., Mosse, D.: A dynamic configuration model for power-efficient virtualized server clusters. In: *Proceedings of the 11th Brazilian Workshop on Real-Time and Embedded Systems* (2009)
17. Rothvoss, T.: Approximating bin packing within  $O(\log \text{OPT} \cdot \log \log \text{OPT})$  bins. Technical report, MIT (2013)
18. Salvagnin, D.: Orbital shrinking: A new tool for hybrid MIP/CP methods. In: Gomes, C., Sellmann, M. (eds.) *CPAIOR 2013. LNCS*, vol. 7874, pp. 204–215. Springer, Heidelberg (2013)
19. Shaw, P.: A constraint for bin packing. In: Wallace, M. (ed.) *CP 2004. LNCS*, vol. 3258, pp. 648–662. Springer, Heidelberg (2004)
20. Srikantaiah, S., Kansal, A., Zhao, F.: Energy aware consolidation for cloud computing. In: *Proceedings of HotPower* (2008)



# Filtering AtMostNValue with Difference Constraints: Application to the Shift Minimisation Personnel Task Scheduling Problem

Jean-Guillaume Fages<sup>1</sup> and Tanguy Lapègue<sup>2</sup>

<sup>1</sup> École des Mines de Nantes, LINA (UMR CNRS 6241), LUNAM Université,  
<sup>2</sup> École des Mines de Nantes, IRCCyN (UMR CNRS 6597), LUNAM Université,  
4 rue Alfred Kastler, La Chantrerie, BP20722, 44307 Nantes Cedex 3, France  
{jean-guillaume.fages,tanguy.lapegue}@mines-nantes.fr

**Abstract.** This paper introduces a propagator which filters a conjunction of difference constraints and an **AtMostNValue** constraint. This propagator is relevant in many applications such as the Shift Minimisation Personnel Task Scheduling Problem, which is used as a case study all along this paper. Extensive experiments show that it significantly improves a straightforward CP model, so that it competes with best known approaches from Operational Research.

**Keywords:** **AtMostNValue**, Constraints Conjunction, Global Constraints, Shift Minimisation Personnel Task Scheduling Problem.

## 1 Introduction

The problem of minimising the number of distinct values among a set of variables subject to difference constraints occurs in many real-life contexts, where an assignment of resources to tasks has to be optimised. For instance, in transports, crews have to be assigned to trips [25]. In schools, classes have to be assigned to rooms [6]. In airports, maintenance tasks have to be assigned to ground crew employees [9,10]. In some factories, fixed jobs have to be assigned to machines [12,13,22]. In a more theoretical context, one may need to color a graph, such that adjacent vertices have distinct colors and not every color can be taken by every node [15,16].

In order to illustrate our contribution, we consider the Shift Minimisation Personnel Task Scheduling Problem (SMPTSP). This problem belongs to the set of personnel scheduling problems (see [11,32] for an overview). It arises when a set of tasks, fixed in time, have to be assigned to a set of shifts so that overlapping tasks should not be assigned to the same shift. Each shift is associated with a given subset of assignable tasks. The objective is to minimise the number of used shifts. This problem typically occurs as the second step of decomposition methods which handle the creation of rosters in a first step and the assignment of tasks in a second one. With this kind of methods, side constraints, related to

personnel roster design, are considered in the first step only, hence the simplicity of the SMPTSP formulation. Nonetheless, current exact approaches from Operational Research fail to solve large scale instances. This is the main motivation for investigating a Constraint-Programming (CP) approach.

The core idea of CP is to design independent constraints that can be combined through common variables, in order to model constrained problems. However, in practice, it is often more interesting to design global constraints [3]. These constraints are able to consider a larger part of the problem, hence their filtering impact is increased. For instance, the `AllDifferent` global constraint is the conjunction over a clique of difference binary constraints, and it has been proved highly relevant within CP solvers [29]. However, developing effective global constraints is often difficult, and it also tends to make CP solver maintenance more expensive, which is one of the greatest concerns of the CP community [27]. Consequently, from a practical point of view, one would rather adapt existing constraints than to implement brand new ones, in order to capitalise over previous work. In this paper we investigate the interest of considering difference constraints when filtering the well known `AtMostNValue` constraint [2,4,28]. We introduce a new propagator whose implementation is based on the state-of-the-art `AtMostNValue` propagator [4]. A wide range of experiments show that our propagator significantly improves the CP model, so that it competes with the most recent SMPTSP dedicated approaches.

The remainder of the paper is organised as follows: Section 2 is devoted to the description of the SMPTSP, in Section 3 we show how the straightforward CP model of the SMPTSP can be improved with a new propagator. Our approach is validated by an extensive experimental study in Section 4, followed by our conclusions.

## 2 Description of the SMPTSP

In the following,  $\mathcal{T}$  and  $\mathcal{W}$  refer respectively to the set of tasks and workers (shifts may be seen as workers with specific skills). Given a task  $t \in \mathcal{T}$ , we refer to the set of workers that can be assigned to  $t$  as  $\mathcal{W}_t \subseteq \mathcal{W}$ . Since tasks are fixed, it is easy to find the set of maximal sets of overlapping tasks, which is referred to as  $\mathcal{C}$ . Actually, it amounts to finding the set of maximal cliques in an interval graph. The size of the largest clique, which provides a trivial lower bound on the required number of workers, is referred to as  $LB_{\neq}$ . For instance, if we consider the example given of Figure 1, we have  $\mathcal{C} = \{\mathcal{K}_1, \mathcal{K}_2, \mathcal{K}_3\}$  with  $\mathcal{K}_1 = \{t_1, t_2, t_3\}$ ,  $\mathcal{K}_2 = \{t_1, t_3, t_4\}$ ,  $\mathcal{K}_3 = \{t_4, t_5\}$  and  $LB_{\neq} = 3$ . This example will be used all along the article to illustrate our points.

The SMPTSP may be stated in Mathematical Programming, by using binary variables  $x_{t,w}$  and  $y_w$  which specify respectively if the task  $t$  is assigned to the worker  $w$  and if the worker  $w$  is used. Based on these variables, the number of used workers is given by (1) and the assignment of tasks to qualified workers

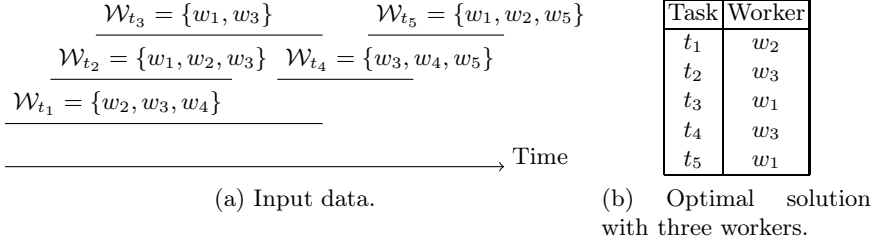


**Fig. 1.** A basic example with 5 workers ( $w_{1..5}$ ) and 5 tasks ( $t_{1..5}$ )

is ensured by (2). The purpose of the constraint (3) is twofold: first it prevents workers to work on overlapping tasks, then it ensures that workers assigned to at least one task are counted as used. We refer to this model as *MIP model*:

$$\begin{aligned} & \text{minimise} \quad \sum_{w \in \mathcal{W}} y_w & (1) \\ & \text{subject to:} \quad \sum_{w \in \mathcal{W}_t} x_{t,w} = 1, \forall t \in \mathcal{T} & (2) \\ & \quad \sum_{t \in \mathcal{K}} x_{t,w} \leq y_w, \forall w \in \mathcal{W}, \forall \mathcal{K} \in \mathcal{C} & (3) \\ & \quad x_{t,w} \in \{0, 1\}, \forall t \in \mathcal{T}, \forall w \in \mathcal{W}_t & (4) \\ & \quad y_w \in \{0, 1\}, \forall w \in \mathcal{W} & (5) \end{aligned}$$

### 3 A CP Model Based on the **AtMostNValue** Constraint

This section first introduces a straightforward CP formulation of the SMPTSP. Next, it recalls the former **AtMostNValue** propagator our approach is based on, and provides a new formalism to define propagators of the same family. Then, it introduces a new propagator which filters **AtMostNValue** while considering a set of difference constraints. We show how to improve and diversify its impact on variables. Finally, we discuss the case of dynamic difference constraints and provide some implementation guidelines.

#### 3.1 A Straightforward CP Formulation

The SMPTSP can be formulated within CP with a set of  $|\mathcal{T}|$  integer variables  $\mathcal{X}$  and one objective variable  $z$ . For each task  $t_i \in \mathcal{T}$ , the variable  $x_i$  gives the worker assigned to the task  $t_i$ . The objective variable  $z$  gives the number of workers assigned to at least one task. We refer to this model as *CP model*:

$$\begin{aligned} & \text{minimise} \quad z & (6) \\ & \text{subject to:} \quad \text{AllDifferent}(\{x_i \mid i \in \mathcal{K}\}), \forall \mathcal{K} \in \mathcal{C} & (7) \\ & \quad \text{AtMostNValue}(\mathcal{X}, z) & (8) \\ & \quad \text{Dom}(z) = [LB_{\neq}, |\mathcal{W}|] & (9) \\ & \quad \text{Dom}(x_i) = \mathcal{W}_{t_i}, \forall t_i \in \mathcal{T} & (10) \end{aligned}$$

The expressive language offered by CP enables to model the problem through two global constraints. In (7), **AllDifferent** constraints [29] are used to forbid workers

ubiquity, *i.e.*, tasks which overlap in time need to be assigned to different workers. In (8), the **AtMostNValue** constraint [4] is used to restrict the number of workers that are involved in the schedule. Then, variable initial domain definitions are given by (9) and (10). More precisely, trivial lower and upper bounds for  $z$  are respectively the maximum number of overlapping tasks and the number of available workers.

### 3.2 State-of-the-Art Filtering of the AtMostNValue Constraint

The **AtMostNValue** constraint belongs to the *Number of Distinct Values* constraint family [2]. It has been introduced in [28] to specify music programs but the first filtering algorithm was provided in [2]. Then, the **AtMostNValue** constraint has been widely investigated in [4], where the authors proved that ensuring the generalised arc consistency (GAC) of **AtMostNValue** is NP-hard, and provide various filtering algorithms. According to this study, the *greedy* propagator they introduced provides a good tradeoff between filtering and runtime. Thus, we use it as the reference propagator for filtering the **AtMostNValue** constraint. Before describing this propagator we need to recall a few definitions:

**Definition 1.** *The intersection graph of a set of variables  $\mathcal{X}$ ,  $G_{\mathcal{I}}(\mathcal{X}) = (V, E_{\mathcal{I}})$ , is defined by a vertex set  $V$  where each variable  $x_i \in \mathcal{X}$  is associated with a vertex  $i \in V$ , and an edge set  $E_{\mathcal{I}}$  representing domain intersections: for any  $(i, j) \in V^2$ , there is an edge  $(i, j) \in E_{\mathcal{I}}$  if and only if  $\text{Dom}(x_i) \cap \text{Dom}(x_j) \neq \emptyset$ .*

**Definition 2.** *An independent set of a graph  $G = (V, E)$  is a subset,  $A \subseteq V$ , of disjoint vertices, *i.e.*, for any  $(i, j) \in A^2$  such that  $i \neq j$ ,  $(i, j) \notin E$ .*

**Definition 3.** *A maximum independent set of a graph  $G$  is an independent set whose cardinality is maximal. The cardinality of a maximum independent set of a graph  $G$  is noted  $\alpha(G)$ .*

In the following, the set of all independent sets of a graph  $G$ , is referred to as  $\mathcal{IS}(G)$ . The filtering algorithm proposed in [4] stems from the search of a maximum independent set in  $G_{\mathcal{I}}(\mathcal{X})$ . Since this problem is NP-Hard [14], it actually computes an independent set  $A$  of  $G_{\mathcal{I}}(\mathcal{X})$ , in a greedy way, by selecting nodes of minimum degree first [17]. This heuristic is referred to as MD. Then, the propagator filters according to the following rules:

- $\mathcal{R}_1$ :  $\underline{z} \leftarrow \max(\underline{z}, |A|)$
- $\mathcal{R}_2$ :  $|A| = \bar{z} \Rightarrow \forall i \in V, \text{Dom}(x_i) \leftarrow \text{Dom}(x_i) \cap \bigcup_{a \in A} \text{Dom}(x_a)$

Where  $\underline{z}$  and  $\bar{z}$  respectively refer to the lower bound and the upper bound of the variable  $z$ .  $\mathcal{R}_1$  states that the cardinality of  $A$  is a valid lower bound for  $z$ .  $\mathcal{R}_2$  states that whenever the cardinality of the independent set  $A$  is equal to the upper bound of  $z$ , then variables in  $\mathcal{X}$  have to take their values among the subset of values induced by  $A$ . Indeed, variables associated with an independent set of an intersection graph take different values, by definition. Thus, using a value outside of this subset of values would lead to use at least  $|A| + 1$  values, which is a contradiction.

Thus, the greedy propagator of **AtMostNValue** takes a graph  $G$  as input, calls a function  $F$  to compute independent sets in  $G$  and then filters variable domains with a set of rules  $\mathcal{R}$ . Therefore, we introduce the notation  $\text{AMNV}\langle G|F|\mathcal{R} \rangle$  to define such a family of propagators. Consequently, the greedy propagator introduced in [4] is referred to as

$\text{AMNV}\langle G_{\mathcal{I}} | \text{MD} | \mathcal{R}_{1,2} \rangle$ . In the following, we suggest improvements for  $G$ ,  $F$  and  $\mathcal{R}$ , leading to a new propagator which filters **AtMostNValue** and a set of difference constraints.

To illustrate the state-of-the-art propagator, we now apply it to our example (cf. Figure 2). Because of variables domain definition, the intersection graph corresponding to our example, is a complete graph. Thus, **MD** select only one node,  $x_3$  for instance. Then  $\mathcal{R}_1$  states that the number of workers required is at least one. If we now assume that the number of workers required is at most one, then  $\mathcal{R}_2$  states that values  $w_2, w_4$  and  $w_5$  must be removed from the domain of the variables (cf. Figure 2a). Consequently, the edges  $(x_1, x_2)$ ,  $(x_1, x_5)$  and  $(x_5, x_4)$  have to be removed in order to obtain the new intersection graph. Based on this new graph, if we assume that  $x_1$  and  $x_5$  are then used as a new independent set (cf. Figure 2b) then  $\mathcal{R}_1$  states that the number of required workers is at least two, leading to fail, since  $z = 1$ .

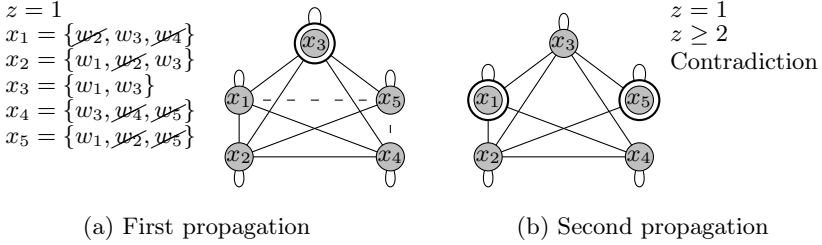


**Fig. 2.** Applying  $\text{AMNV}\langle G_{\mathcal{I}} | \text{MD} | \mathcal{R}_{1,2} \rangle$  to our example when  $z = 1$

### 3.3 Embedding Difference Constraints into **AtMostNValue**

As the **SMPTSP** only considers two kind of constraints (**AtMostNValue** and **AllDifferent**), filtering their conjunction may be very profitable. For that purpose, this section introduces an implied propagator, in the form  $\text{AMNV}\langle G | F | \mathcal{R} \rangle$ , which considers difference constraints. As a first step, we suggest to consider a new graph, referred to as a *constrained intersection graph* (Definition 4), instead of the intersection graph of variables.

**Definition 4.** Given a set of variables  $\mathcal{X}$  and a set of difference constraints  $\mathcal{D}$ , the *constrained intersection graph*,  $G_{\mathcal{CI}}(\mathcal{X}, \mathcal{D}) = (V, E_{\mathcal{CI}})$ , of  $\mathcal{X}$  and  $\mathcal{D}$  is defined by a vertex set  $V$  where each variable  $x_i \in \mathcal{X}$  is associated with a vertex  $i \in V$ , and an edge set  $E_{\mathcal{CI}}$  representing possible classes of equivalence: for any  $(i, j) \in V^2$ , there is an edge  $(i, j) \in E_{\mathcal{CI}}$  if and only if  $\text{Dom}(x_i) \cap \text{Dom}(x_j) \neq \emptyset$  and  $\text{neg}(x_i, x_j) \notin \mathcal{D}$ .

In this paper, we consider a single set of variables  $\mathcal{X}$  and a single set of difference constraints  $\mathcal{D}$ , thus, for the sake of clarity  $G_{\mathcal{CI}}(\mathcal{X}, \mathcal{D})$  and  $G_{\mathcal{I}}(\mathcal{X})$  will be respectively noted  $G_{\mathcal{CI}}$  and  $G_{\mathcal{I}}$ . It is worth noticing that  $G_{\mathcal{CI}} \subseteq G_{\mathcal{I}}$ .

**Proposition 1.**  $\mathcal{IS}(G_{\mathcal{I}}) \subseteq \mathcal{IS}(G_{\mathcal{CI}})$ , hence  $\alpha(G_{\mathcal{I}}) \leq \alpha(G_{\mathcal{CI}})$ .

*Proof.* Let  $A_{\mathcal{I}}$  be an independent set in  $G_{\mathcal{I}}$ . Since  $G_{\mathcal{CI}}$  and  $G_{\mathcal{I}}$  are based on the same variable set, they share the same vertex set, so  $A_{\mathcal{I}}$  is also a subset of vertices of  $G_{\mathcal{CI}}$ . Since vertices of  $A_{\mathcal{I}}$  are pairwise disjoint in  $G_{\mathcal{I}}$  (by assumption) and since all edges of  $G_{\mathcal{CI}}$  also belong to  $G_{\mathcal{I}}$ , then vertices of  $A_{\mathcal{I}}$  are also pairwise disjoint in  $G_{\mathcal{CI}}$ . Consequently,  $A_{\mathcal{I}}$  is an independent set of  $G_{\mathcal{CI}}$ . Thus, all independent sets of  $G_{\mathcal{I}}$  are independent sets of  $G_{\mathcal{CI}}$ , so  $\max_{I \in \mathcal{IS}(G_{\mathcal{I}})} |I| \leq \max_{I_c \in \mathcal{IS}(G_{\mathcal{CI}})} |I_c|$ , hence  $\alpha(G_{\mathcal{I}}) \leq \alpha(G_{\mathcal{CI}})$ .  $\square$

Note that a maximum independent set in  $G_I$ , is not necessarily maximal in  $G_{CI}$ . For instance, one may consider a non-empty set of variables with identical domains and with a difference constraint over each pair of distinct variables. Then  $G_I$  is a complete graph, whereas there are no edges, but loops, in  $G_{CI}$ . Consequently,  $\alpha(G_I) = 1$  whereas  $\alpha(G_{CI}) = |V|$ . It is worth noticing that in our context, the bigger the independent set, the higher the chance to filter variable domains. Thus, using  $G_{CI}$  is apriori better than using  $G_I$  to filter **AtMostNValue** when difference constraints figure in the model (Proposition 2).

To illustrate the interest of  $G_{CI}$ , we now use it on our example (cf. Figure 3). Because of difference constraints,  $G_{CI}$  is sparser than  $G_I$  (cf. 3a). Thus, MD is now able to compute a larger independent set, leading to find a better lower bound of  $z$ . For instance, if we consider the independent set  $\{x_1, x_2, x_3\}$  (cf. Figure 3b) then  $\text{AMNV}\langle G_{CI} | \text{MD} | \mathcal{R}_{1,2} \rangle$  states that the number of required workers is at least three. Moreover, if we assume that the number of required workers is at most three, then the value  $w_5$  has to be removed from the variable domains.

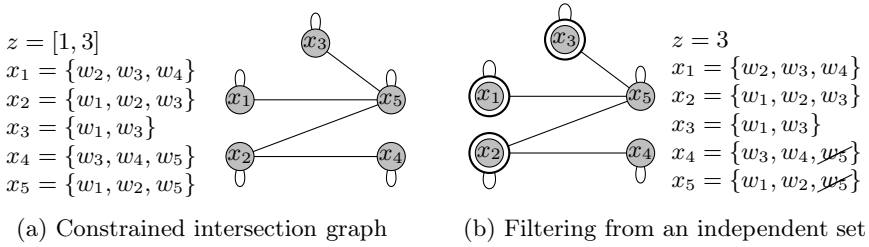


**Fig. 3.** Use of  $\text{AMNV}\langle G_{CI} | \text{MD} | \mathcal{R}_{1,2} \rangle$  on our example

**Proposition 2.** *Given an oracle  $\mathcal{O}$  which computes all maximum independent sets of any graph, then  $\text{AMNV}\langle G_{CI} | \mathcal{O} | \mathcal{R}_{1,2} \rangle$  dominates  $\text{AMNV}\langle G_I | \mathcal{O} | \mathcal{R}_{1,2} \rangle$*

*Proof.* First of all, since  $\mathcal{O}$  is able to compute all maximum independent sets of any graph, then the lower bound given by  $\mathcal{R}_1$  in  $G_{CI}$  is equal to  $\alpha(G_{CI})$  whereas the lower bound given by  $\mathcal{R}_1$  in  $G_I$  is equal to  $\alpha(G_I)$ . Since  $\alpha(G_I) \leq \alpha(G_{CI})$  (Proposition 1), then  $\text{AMNV}\langle G_{CI} | \mathcal{O} | \mathcal{R}_1 \rangle$  dominates  $\text{AMNV}\langle G_I | \mathcal{O} | \mathcal{R}_1 \rangle$ . Second, since  $\mathcal{O}$  is able to compute all maximum independent sets of any graph, and since  $\mathcal{IS}(G_I) \subseteq \mathcal{IS}(G_{CI})$  (Proposition 1), then values filtered by  $\text{AMNV}\langle G_I | \mathcal{O} | \mathcal{R}_2 \rangle$  are also filtered by  $\text{AMNV}\langle G_{CI} | \mathcal{O} | \mathcal{R}_2 \rangle$ . Consequently,  $\text{AMNV}\langle G_{CI} | \mathcal{O} | \mathcal{R}_2 \rangle$  dominates  $\text{AMNV}\langle G_I | \mathcal{O} | \mathcal{R}_2 \rangle$  and thus  $\text{AMNV}\langle G_{CI} | \mathcal{O} | \mathcal{R}_{1,2} \rangle$  dominates  $\text{AMNV}\langle G_I | \mathcal{O} | \mathcal{R}_{1,2} \rangle$ .  $\square$

**Proposition 3.** *Given an independent set  $A$  in  $G_{CI}$  such that  $|A| = \bar{z}$ , any solution of the conjunction of **AtMostNValue** and  $\mathcal{D}$  satisfies the following formula:  $\forall i \in V \setminus A, \exists a \in A_i$  s.t.  $x_i = x_a$ , where  $A_i$  denotes  $\{a \in A \mid (i, a) \in E_{CI}\}$ .*

*Proof.* Given an independent set  $A$  in  $G_{CI}$  such that  $|A| = \bar{z}$ . Let's assume that there exists a solution  $S$  to the conjunction of **AtMostNValue** and  $\mathcal{D}$  such that there exists a vertex  $i \in V \setminus A$  for which  $\forall a \in A_i, x_i \neq x_a$ . Thus,  $S$  is solution of the conjunction of **AtMostNValue** and  $\mathcal{D} \cup \{\text{neq}(x_i, x_a) \mid a \in A_i\}$ . Consequently,  $A \cup \{i\}$  is a valid independent set in  $G_{CI}$ . Then  $\mathcal{R}_1$  states that  $\bar{z} \leftarrow |A \cup \{i\}|$ , i.e.,  $\bar{z} \leftarrow \bar{z} + 1$  which is not possible. Consequently, such a solution  $S$  does not exist, hence Proposition 3 holds.  $\square$

From a filtering perspective, Proposition 3 leads to consider the following rule:

$$- \mathcal{R}_3: \quad |A| = \bar{z} \Rightarrow \forall i \in V \setminus A \left\{ \begin{array}{l} \text{Dom}(x_i) \leftarrow \text{Dom}(x_i) \cap \bigcup_{a \in A_i} \text{Dom}(x_a) \\ A_i = \{a\} \Rightarrow \text{Dom}(x_a) \leftarrow \text{Dom}(x_a) \cap \text{Dom}(x_i) \end{array} \right.$$

This rule is actually a refined variant of  $\mathcal{R}_2$ . While this change is quite simple, it may have a significant impact in practice, especially on large scale problems were, presumably,  $\forall i \in V \setminus A, |A_i| \ll |A|$ . Note the particular case that occurs when, for some node  $i \in V \setminus A$ ,  $A_i = \{a\}$  enables to learn the valid equality  $x_i = x_a$ . This way, it is possible to filter the domain of the variable  $x_a$  of the independent set as well. From a theoretical point of view  $\mathcal{R}_3$  is also stronger than  $\mathcal{R}_2$  (cf Proposition 4).

**Proposition 4.** *Given a deterministic heuristic  $H$  which computes a set of independent sets in  $G_{C\mathcal{I}}$ , then  $\text{AMNV}\langle G_{C\mathcal{I}} | H | \mathcal{R}_3 \rangle$  dominates  $\text{AMNV}\langle G_{C\mathcal{I}} | H | \mathcal{R}_2 \rangle$*

*Proof.* Since  $\text{AMNV}\langle G_{C\mathcal{I}} | H | \mathcal{R}_3 \rangle$  and  $\text{AMNV}\langle G_{C\mathcal{I}} | H | \mathcal{R}_2 \rangle$  use the same deterministic heuristic  $H$  in the same graph  $G_{C\mathcal{I}}$ , then they filter with the same independent set  $A$ . Since, for any node  $i \in V$ ,  $A_i = \{a \in A | (i, a) \in E_{C\mathcal{I}}\}$ , then  $A_i \subseteq A$ . Consequently, any value filtered by  $\mathcal{R}_2$  is also filtered by  $\mathcal{R}_3$ .  $\square$

To illustrate this rule, we now apply it on our example (cf. Figure 4). If we consider the independent set  $\{x_1, x_3, x_4\}$  (cf. Figure 4a) and an objective  $z = [1, 3]$ , then  $\text{AMNV}\langle G_{C\mathcal{I}} | \text{MD} | \mathcal{R}_{1,2} \rangle$  deduces that  $z = 3$  but cannot filter  $\mathcal{X}$  domains. However,  $\text{AMNV}\langle G_{C\mathcal{I}} | \text{MD} | \mathcal{R}_{1,3} \rangle$  enables to filter the value  $w_5$ , which is not included in  $\text{Dom}(x_1) \cup \text{Dom}(x_3) = \{w_1, w_2, w_3, w_4\}$ , from the domain of  $x_5$  and values  $w_1$  and  $w_2$  for variable  $x_2$  because they do not figure in  $\text{Dom}(x_4) = \{w_3, w_4, w_5\}$  (cf. Figure 4b). Moreover,  $\text{AMNV}\langle G_{C\mathcal{I}} | \text{MD} | \mathcal{R}_{1,3} \rangle$  also enables to learn the valid equality  $x_2 = x_4$ , which enables to remove values  $w_4$  and  $w_5$  from the domain of  $x_4$ .

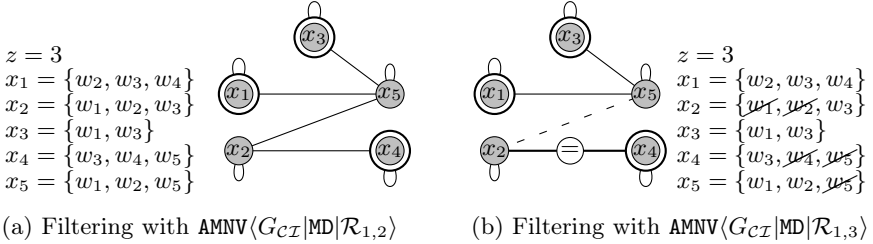


**Fig. 4.** Using  $\text{AMNV}\langle G_{C\mathcal{I}} | \text{MD} | \mathcal{R}_{1,3} \rangle$  on our example

### 3.4 Diversifying Filtering

CP frameworks traditionally perform a fix point over constraints at each branching node [30]. This implies that our model may compute thousands of independent sets during the search process. Thus, it is advised to use a greedy algorithm, such as MD, to filter the **AtMostNValue** constraint [4]. This heuristic is quite efficient but it lacks diversification for both its bound and the resulting filtering, which depend on the computed independent set. This may lead to unfortunate results when the considered graph does not suit this deterministic heuristic.

**Proposition 5.** *Given two functions  $F_1$  and  $F_2$  which compute a set of independent sets in  $G_{CZ}$ , respectively noted  $\mathcal{A}_1$  and  $\mathcal{A}_2$ . Let a set  $\mathcal{A}_{22}$  denote  $\mathcal{A}_2 \setminus \mathcal{A}_1$ .*

*If  $\max_{I_2 \in \mathcal{A}_{22}} |I_2| < \max_{I_1 \in \mathcal{A}_1} |I_1|$ , then  $\text{AMNV}\langle G_{CZ} | F_1 | \mathcal{R}_{1,3} \rangle$  dominates  $\text{AMNV}\langle G_{CZ} | F_2 | \mathcal{R}_{1,3} \rangle$ .*

*Proof.* First,  $\max_{I_2 \in \mathcal{A}_{22}} |I_2| < \max_{I_1 \in \mathcal{A}_1} |I_1|$  implies that  $\max_{I_2 \in \mathcal{A}_2} |I_2| \leq \max_{I_1 \in \mathcal{A}_1} |I_1|$ , hence  $\text{AMNV}\langle G_{CZ} | F_1 | \mathcal{R}_1 \rangle$  dominates  $\text{AMNV}\langle G_{CZ} | F_2 | \mathcal{R}_1 \rangle$ . Second, let's now assume that a value is removed from a domain of a variable in  $\mathcal{X}$ , by  $\text{AMNV}\langle G_{CZ} | F_2 | \mathcal{R}_3 \rangle$ . If this filtering occurs while considering an independent set of  $\mathcal{A}_1 \cap \mathcal{A}_2$ , then the same filtering is performed by  $\text{AMNV}\langle G_{CZ} | F_1 | \mathcal{R}_{1,3} \rangle$ . Else, this filtering occurs when considering an independent set  $I_2$  of  $\mathcal{A}_{22}$ . A necessary condition so that  $\mathcal{R}_3$  triggers filtering is that,  $|I_2| = \bar{z}$ . As  $|I_2| < \max_{I_1 \in \mathcal{A}_1} |I_1|$ , then the problem is actually infeasible, which is captured by  $\text{AMNV}\langle G_{CZ} | F_1 | \mathcal{R}_1 \rangle$ . Thus,  $\text{AMNV}\langle G_{CZ} | F_1 | \mathcal{R}_{1,3} \rangle$  dominates  $\text{AMNV}\langle G_{CZ} | F_2 | \mathcal{R}_{1,3} \rangle$ .  $\square$

As highlighted by the Proposition 5, it may be interesting to obtain several independent sets to apply filtering rules over a greater set of variables. The simplest way to get diversification would be to randomly break ties of MD. However, it turns out that this does not bring enough diversification to improve results. Thus, we introduce the  $\mathbf{R}^k$  algorithm (cf. Algorithm 1), which keeps the philosophy of a fast and simple greedy algorithm, while providing diversification. This heuristic performs  $k$  iterations, each one computes an independent set by selecting nodes randomly, with a uniform probability distribution. It thus provides a set of  $k$  independent sets. Note that we also tried to use a weighted random that favors nodes with a small degree, but it does not outperform  $\mathbf{R}^k$ , while being more complicated, and thus it is not presented in this paper. From a theoretical point of view, this approach presents several interesting properties. First, it offers control over its runtime complexity and its expected quality. Second, computing independent sets randomly tends to impact variable domains homogeneously. Note that, when  $k \rightarrow \infty$ , then the method tends to enumerate and filter with all maximum independent sets, which is optimal with the given filtering rules. However, from a practical point of view, one has to bound the number of iterations in order to get a reasonable runtime. We suggest the default setting  $k = 30$ , which seems to perform better than MD, on average, without introducing a significant overhead. However, MD offers a good approximation ratio [17], hence we recommend to use both.

### 3.5 The Case of Dynamic Difference Constraints

Our approach focuses on the SMPTSP which considers a set of tasks that are already fixed in time. Thus, difference constraints are given as input through a set of **AllDifferent** constraints. However, one may be interested in filtering **AtMostNValue** when difference constraints appear dynamically during the search. For instance, in disjunctive scheduling problems where tasks have to be fixed in time, difference constraints implicitly appear as variable domains are reduced. It is worth noticing that in this situation, difference constraints would no longer be well propagated because of the absence of **AllDifferent** constraints in the model. Fortunately, the **AtMostNValue** propagator can help to get back a global view of the problem and thus a powerful filtering thanks to Proposition 6.

**Proposition 6.** *If  $A$  is an independent set of  $G_{CZ}$ , let  $\mathcal{X}_A$  denote  $\{x_i \in \mathcal{X} | i \in A\}$ , then the constraint **AllDifferent**( $\mathcal{X}_A$ ) holds.*



---

**Algorithm 1.**  $R^k$  algorithm to compute independent sets

---

**Global variables:**

$G_{C\mathcal{I}}$ , the constrained intersection graph of which independent sets have to be computed  
 $k$ , the number of iterations (and thus the number of independent sets to compute)

```

1:  $\mathcal{A} \leftarrow \emptyset$  // Creates a set of independent sets, initially empty
2:  $count \leftarrow k$ 
3: while ( $count \geq 0$ ) do
4:    $count \leftarrow count - 1$ 
5:    $G \leftarrow G_{C\mathcal{I}}.copy()$  // Copies the constrained intersection graph
6:    $A \leftarrow \emptyset$  // Creates an independent set  $A$ , initially empty
7:   while ( $G \neq \emptyset$ ) do
8:      $x \leftarrow randomNode(G)$  // Randomly selects a node  $x$  in  $G$ 
9:      $A \leftarrow A \cup \{x\}$  // Adds  $x$  to the independent set  $A$ 
10:     $G \leftarrow G \setminus \{y \mid (x, y) \in G\}$  // Removes  $x$ 's neighbors from  $G$  (including  $x$  itself)
11:  end while
12:   $\mathcal{A} \leftarrow \mathcal{A} \cup \{A\}$  // adds the independent set  $A$  to  $\mathcal{A}$ 
13: end while
14: return  $\mathcal{A}$  // returns a set of independent sets of  $G_{C\mathcal{I}}$ 

```

---

*Proof.* By definition of a constrained intersection graph, an independent set represents variables that are either already different or constrained to be.  $\square$

From Proposition 6, we derive a new filtering rule for **AtMostNValue** which calls a filtering algorithm of **AllDifferent** over variables corresponding to the independent set  $A$  it has computed:

–  $\mathcal{R}_4$ : **AllDifferent**( $\{x_i \in \mathcal{X} \mid i \in A\}$ )

Note that this rule can either directly call a filtering algorithm of **AllDifferent** over the appropriate subset of variables or post dynamically an **AllDifferent** constraint into the solver. The first option is the simplest one, but cannot filter incrementally, so a *bound-consistent* filtering algorithm of **AllDifferent**, such as the one presented in [26], would presumably be more relevant than the GAC one [29]. Moreover, the subset of variables to be different is lost right after filtering and the next propagation may involve a worse independent set. Instead, the second option can involve incremental GAC **AllDifferent** constraints which would remain in the current search branch. However, since each independent set can post an **AllDifferent** constraint, it has the drawback of leading to a potential explosion over the number of such constraints. Thus, one needs to set up a constraint pool to manage those constraints, by retaining only the most interesting constraints and removing dominated ones. One can see a parallel with *Cut pools* of MIP solvers [1]. For solver maintenance simplicity purposes, we recommend the first option.

### 3.6 Implementation

So far, we have seen that it was possible to tune the original greedy **AtMostNValue** propagator. We now suggest a simple and non-intrusive implementation (see Algorithm 2). In this implementation, no assumption is made on the set of difference constraints  $\mathcal{D}$  which can thus grow during the resolution process. For instance,  $\mathcal{D}$  can be modified by the user or other constraints. Note also that no assumption is made about the kind of variables in  $\mathcal{X}$ , which can then be multidimensional continuous variables, as in [7]. The constrained intersection graph is stored as a backtrackable structure which is updated at the beginning of the filtering algorithm, *i.e.*, after potentially many domain

modifications. This turns out to be much faster than updating  $G_{CI}$  incrementally on each domain modification or rebuilding it from scratch. Once the graph has been updated, the propagator computes a set of independent sets with a function  $F$ . For each independent set, the propagator filters the lower bound of  $z$  and filters  $\mathcal{X}$  with a set of rules  $\mathcal{R}$ . Thus, one can see that this propagator is flexible and not intrusive.

---

**Algorithm 2.**  $\text{AMNV}\langle G_{CI} | F | \mathcal{R} \rangle$  - filtering algorithm

---

**Global variables:**

$z$ , an integer variable  
 $\mathcal{X}$ , a set of variables  
 $\mathcal{D}$ , a set of difference constraints  
 $G_{CI} = (V, E_{CI})$ , a backtrackable graph  
 $F$ , a function that computes a set of independent sets in a given graph  
 $\mathcal{R}$ , a set of filtering rules

```

1: if (first call of the propagator) then
2:   // graph generation
3:    $V \leftarrow [1, |\mathcal{X}|]$ 
4:    $E_{CI} \leftarrow \emptyset$ 
5:   for  $((i, j) \in V^2)$  do
6:     if  $(\text{Dom}(x_i) \cap \text{Dom}(x_j) \neq \emptyset \wedge \text{neq}(x_i, x_j) \notin \mathcal{D})$  then
7:        $E_{CI} \leftarrow E_{CI} \cup \{(i, j)\}$ 
8:     end if
9:   end for
10:   $G_{CI} \leftarrow (V, E_{CI})$ 
11: else
12:  // graph lazy update
13:  for  $((i, j) \in E_{CI})$  do
14:    if  $(\text{Dom}(x_i) \cap \text{Dom}(x_j) = \emptyset \vee \text{neq}(x_i, x_j) \in \mathcal{D})$  then
15:       $E_{CI} \leftarrow E_{CI} \setminus \{(i, j)\}$ 
16:    end if
17:  end for
18: end if
19:  $\mathcal{A} \leftarrow F(G_{CI})$  // computes a set  $\mathcal{A}$  of independent sets
20: for  $(A \in \mathcal{A})$  do
21:    $\text{filter}(\mathcal{X}, z, G_{CI}, A, \mathcal{R})$  // rule-based filtering
22: end for

```

---

## 4 Experimental Study

In order to evaluate the interest of our contribution, we have performed extensive tests. First, Section 4.1 introduces and motivates a new benchmark data set for the SMPTSP. Second, Section 4.2 describes the search strategies that were used in this study. Next, in Section 4.3, we focus on the objective lower bound quality at root node. We show that it changes completely depending on the model that is used. Section 4.4 highlights the potential benefit of strengthening the filtering with  $\mathbf{R}^k$ . Section 4.5 focuses on the scalability issue, *i.e.*, it investigates the ability of our approach to compute tight bounds, even on large instances. Finally, in Section 4.6 we compare our approach to the best known results on the state-of-the-art SMPTSP instances.

Our algorithms have been implemented in JAVA, we have used Cplex 12.4 with default settings to test *MIP model* and Choco 13-03 [8] to perform every test related to *CP model*. All our implementations are available online at [23]. In the following line-graphs, instances are sorted in order to ease graphs reading. Moreover, in order to simplify result analysis, we note respectively  $z^*$  and  $\underline{z}_r$  the optimal objective

and the objective lower bound at root node. Finally, a CP model using a propagator  $\text{AMNV}\langle G|\mathbf{F}|R\rangle$ , is noted  $\downarrow\text{AMNV}\langle G|\mathbf{F}|R\rangle$  when used within a top-down minimisation strategy and  $\uparrow\text{AMNV}\langle G|\mathbf{F}|R\rangle$  when used within a bottom-up minimisation strategy.

#### 4.1 A New Set of Challenging SMPTSP Instances

Very recently, Smet *et. al.* have shown in [31] that the 137 instances provided by Krishnamoorthy *et. al.* in [20] admit a feasible solution with an objective value equal to  $LB_{\neq}$ . Since worker skills are taken into account in [20], this result is somehow surprising: one may expect that considering worker skills would have an impact on the optimum, but actually, it only makes it harder to find a feasible solution. Consequently, these instances do not provide much challenge regarding to the search of interesting lower bounds. Therefore, we propose a new set of challenging instances whose maximal number of overlapping tasks does not provide a good lower bound.

In order to generate this new benchmark we used a dedicated procedure based on some of the empirical results presented in [20] and [31]. First of all, it is specified in [20] that the average tightness, defined as the sum of processing times over the sum of shift lengths should be close to 90% in order to obtain challenging instances. Another hardness analysis [31] shows that the smaller the average task processing time, the more difficult instances are to solve. Based on these two conclusions, we designed a dedicated procedure able to generate instances for which the maximum number of overlapping tasks does not provide a good lower bound.

The procedure which is given in details in [23] generates randomly a set of tasks ranging from 15 minutes to 2 hours. We consider six different kinds of shift. The first three aim at splitting a working day into shifts of 8 hours which is very common in personnel scheduling [32]. The other three are obtained from the previous ones by introducing an offset of 2 hours to their starting time, so that each task is entirely contained in at least one shift. Based on this simple procedure, we provide 100 new instances with a number of tasks ranging from 70 to 1600 and a number of available workers ranging from 60 to 950. These instances along with our generator are available online [23]. In the following, we refer to the instances of Krishnamoorthy *et. al.* as `Data_137`, and our generated instances as `Data_100`.

#### 4.2 Search Strategies

Our contribution mainly concerns the lower bounding of the problem, and the lower the objective upper bound, the stronger our filtering. Hence, we naturally employed a bottom-up minimisation approach. It tries to compute a solution involving  $k$  workers, where  $k$  is initialised to the root lower bound of  $z$  and incremented by one each time the solver proved unsatisfiability. Thus, the first solution found is optimal.

It is worth noticing that when using a bottom-up approach, most part of the search concerns infeasible regions of the search space. In order to get out of them as quickly as possible, it seems wiser to use intensification processes within the branching heuristic, following the *Fail First* principle [18]. Thus, our bottom-up approach uses a sequential heuristic: a first heuristic randomly determines which subset of workers will be used in the solution, then a **first-fail** heuristic is used to assign tasks to workers: the variable associated with the smallest domain is selected first, and then fixed to its lower bound. The overall branching scheme is reinforced by the **Last Conflict** heuristic of [24] which aims at identifying critical variables.

### 4.3 Impact of Suggested Improvements on the Root Lower Bound

In order to evaluate the practical interest of our contribution, we have compared the value of  $\underline{z}_r$  for several settings, on Data\_100. Results are reported on Figure 5. Using  $\text{AMNV}\langle G_{\mathcal{I}}|\text{MD}|\mathcal{R}_{1,3}\rangle$  instead of  $\text{AMNV}\langle G_{\mathcal{I}}|\text{MD}|\mathcal{R}_{1,2}\rangle$  dramatically improves the value of  $\underline{z}_r$ : on average, our approach is able to increase  $LB_{\neq}$  by a factor two, whereas the former approach is not able to reach this trivial lower bound. Moreover, our approach scales much better than the classical one which is not able to increase  $\underline{z}_r$  whatever the size of the instance. Actually, this is not really surprising since  $\text{AMNV}\langle G_{\mathcal{I}}|\text{MD}|\mathcal{R}_{1,2}\rangle$  is blind to **AllDifferent** constraints, which represent a big part of the problem. Using  $G_{\mathcal{C}\mathcal{I}}$  instead of  $G_{\mathcal{I}}$  makes these **AllDifferent** constraints visible to **AtMostNValue**, hence the increased of  $\underline{z}_r$ . Finally, to a lesser extent, combining  $\mathbf{R}^{30}$  and MD also contributes to improve  $\underline{z}_r$  by 7%. Given the high quality of the lower bound reached by  $\text{AMNV}\langle G_{\mathcal{C}\mathcal{I}}|\text{MD}|\mathcal{R}_{1,3}\rangle$ , an increase of 7% is already significant. Moreover, considering the simplicity of this modification, it is very interesting.

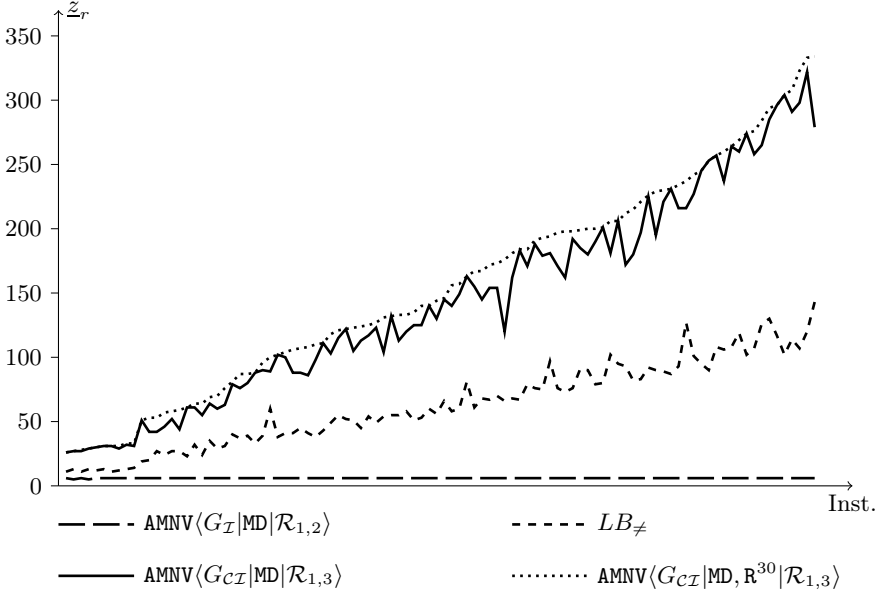


Fig. 5. Value of  $\underline{z}_r$  depending on the propagator of **AtMostNValue**.

One may wonder if the straightforward CP model is able to catch up with our approach after a few minutes. Hence, we have evaluated the evolution of  $\underline{z}$  after five minutes of bottom-up resolution. It turned out that its best improvement was by 7 units only, which is far too small to catch up with  $LB_{\neq}$ .

### 4.4 Managing the Tradeoff between Filtering and Runtime

As explained in Section 3.4, using the heuristic  $\mathbf{R}^k$  is a simple and effective way to obtain diversification in filtering. The parameter  $k$  enables to manage the tradeoff between expected filtering power and runtime. Since  $\mathbf{R}^k$  is suggested as a complement to

**MD**, it must be seen as an improvement opportunity. Back to our case, many values of  $k$  have been tested, within a time limit of 5 minutes (Table 1). Interestingly, the optimal setting changes quite a lot from one data set to another, hence using an automatic algorithm configuration program [19] seems relevant to get the best results. More precisely, it can be seen that for **Data\_100** instances, the more filtering the better. The time spent in  $R^k$  is compensated by the search space reduction it provides. Thus, it may be worth strengthening even more the filtering, by filtering conjunctions of **AllDifferent** constraints for instance [5]. Instead, **Data\_137** instances are well solved by our model, hence the faster  $R^k$  the better. We report that using  $R^{800}$  with  $\mathcal{R}_{1,2}$  instead of  $\mathcal{R}_{1,3}$ , leads to finding 7 optima only. Thus, the interest of  $R^k$  lies not only in finding large independent sets ( $\mathcal{R}_1$ ), but mostly in diversifying the filtering ( $\mathcal{R}_3$ ).

**Table 1.** Number of optima found with  $\uparrow\text{AMNV}\langle G_{CZ}|\text{MD}, R^k|\mathcal{R}_{1,3}\rangle$ . For reference, *MIP model* finds respectively 65 and 46 optima on **Data\_137** and **Data\_100**.

| Benchmark | Data_137 |            |     |    |    | Data_100 |     |     |           |      |
|-----------|----------|------------|-----|----|----|----------|-----|-----|-----------|------|
| $k$       | 10       | 30         | 50  | 70 | 90 | 100      | 200 | 400 | 800       | 1600 |
| Nb Opt.   | 101      | <b>109</b> | 106 | 99 | 95 | 19       | 22  | 28  | <b>35</b> | 31   |

#### 4.5 Scalability Study

We now evaluate the ability of our approach to provide good bounds on large scale instances. For that, we compare the values of  $\underline{z}$  and  $\bar{z}$  obtained with *MIP model*, with those obtained with  $\uparrow\text{AMNV}\langle G_{CZ}|\text{MD}, R^{800}|\mathcal{R}_{1,3}\rangle$  and  $\downarrow\text{AMNV}\langle G_{CZ}|\text{MD}, R^{30}|\mathcal{R}_{1,3}\rangle$ , after 5 minutes of resolution. For this last one, we did not use the branching heuristic described in section 4.2 which is no longer relevant in a top-down mode. Instead we used **first-fail** to select a task and then we assign it to any worker, preferably already used in the schedule. Moreover, our propagator is unlikely to filter significantly in a top-down approach, since the upper bound is presumably too high to enable filtering. Consequently, we set  $k$  to 30 so that it improves **MD** at least expense.

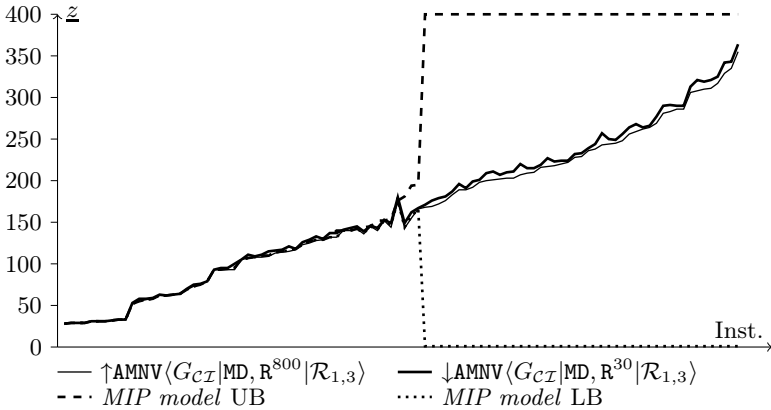


**Fig. 6.** Objective value after a 5 minutes resolution, on **Data\_100**

Figure 6 gives general trends regarding the ability to scale of *MIP model* and *CP model*. It shows that *MIP model* fails to solve about half of instances: it is not able to provide either lower bounds or upper bounds. Regarding this scalability issue, the top-down and bottom-up CP approaches perform very well. Their relative gap varies around 2%. Moreover, it does not increase much with the instance size. Consequently, using  $\uparrow\text{AMNV}\langle G_{CI}|\text{MD}, \mathbf{R}^{800}|\mathcal{R}_{1,3}\rangle$  and  $\downarrow\text{AMNV}\langle G_{CI}|\text{MD}, \mathbf{R}^{30}|\mathcal{R}_{1,3}\rangle$  leads to finding tight bounds for  $z$ , even on large instances.

#### 4.6 A Competitive Approach

In order to estimate the quality of our approach, we consider state-of-the-art SMPTSP instances [21]. We compare the performances of  $\uparrow\text{AMNV}\langle G_{CI}|\text{MD}, \mathbf{R}^{30}|\mathcal{R}_{1,3}\rangle$  and *MIP model*, but we also compare our results to those of Krishnamoorthy *et. al.* and Smet *et. al.* who use two different metaheuristics. As illustrated on Figure 7, the CP model clearly dominates *MIP model* on this data set, since it is able to reach the optimum of all but one instance within the time limit. It is important to notice that the only other method that is able to solve these instances to optimality is the metaheuristic given in [31]. This metaheuristic is actually faster than our approach, since it is able to reach all optima in 5 minutes, whereas our approach, after 5 minutes, has proved 80% of optima. However, it provides no lower bound, hence it does not prove optimality, except when  $LB_{\neq}$  equals  $z^*$ .

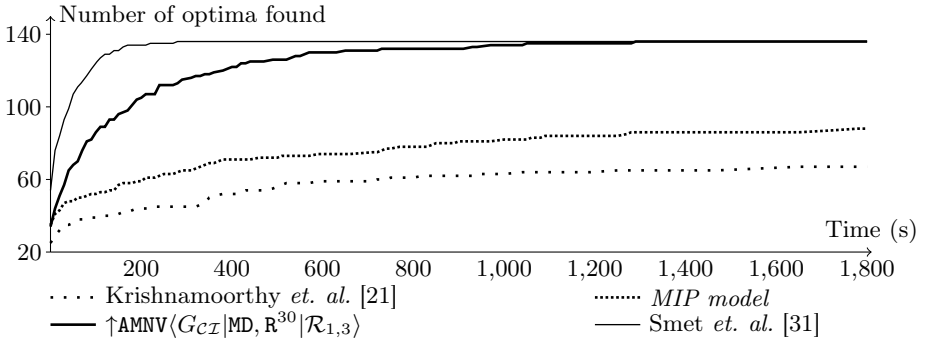


Fig. 7. Number of optima found, depending on runtime, on Data\_137

As the SMPTSP often occurs as a subproblem of a more general method, it seems critical to provide a good lower bound in a short time. To evaluate the quality of  $\underline{z}_r$ , we compare its value and its required runtime with those of the lower bounding procedure given in [21]. This procedure is based on a Lagrangean relaxation, therefore, we refer to its lower bound as  $\underline{z}_L$ . To have a fair comparison, we do not use  $LB_{\neq}$  in our model, since it gives the optimal value on the state-of-the-art instances. As illustrated by Figure 8a, the absolute gap between  $\underline{z}_r$  and  $z^*$  is on average smaller than the gap between  $\underline{z}_L$  and  $z^*$ . More precisely  $\underline{z}_L$  is on average seven times larger than  $\underline{z}_r$ . Moreover the empirical worst-case gap of  $\underline{z}_r$  is also much better than the one of  $\underline{z}_L$ , since there is a factor 6 between these two values. Finally, as illustrated by Figure 8b the runtime of  $\underline{z}_r$  is much

smaller than the one of  $\underline{z}_\mathcal{L}$ : the average runtime of our lower bounding procedure is thirty times quicker than the one presented in [21].

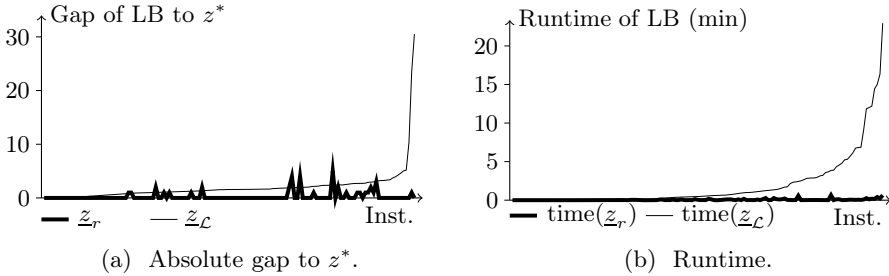


**Fig. 8.** Overall quality of  $\underline{z}_r$  and  $\underline{z}_\mathcal{L}$  on Data\_137

## 5 Conclusions

In this paper, we have presented a new propagator to filter the conjunction of an **AtMostNValue** constraint and difference constraints. This propagator relies on a more appropriate graph structure, as well as refined filtering rules. We provide a simple way to diversify filtering, in order to improve the overall approach. Moreover, this propagator gives control over the tradeoff between filtering and runtime. Since it is simple to implement, effective and relevant for many applications, we believe that this propagator would benefit the CP community.

Furthermore, we have introduced a CP approach to solve the SMPTSP. This model outperforms previous exact approaches on state-of-the-art benchmarks. Furthermore, it provides very good lower bounds, even on large instances, in a short time. Thus, the CP approach competes with both exact and heuristic state-of-the-art approaches.

Future work may focus on the application of this propagator in the context of dynamic difference constraints, which occurs in many disjunctive scheduling problems. Another research perspective would be to investigate how to adapt the parameter  $k$  of  $\mathbb{R}^k$  during the search, to improve even more the tradeoff between filtering and runtime.

**Acknowledgements.** The authors thank the anonymous referees as well as Xavier Lorca, Damien Prot and Thierry Petit for their useful comments.

## References

1. Andreello, G., Caprara, A., Fischetti, M.: Embedding  $\{0, 1/2\}$ -cuts in a branch-and-cut framework: A computational study. *INFORMS J. on Computing* 19(2), 229–238 (2007)
2. Beldiceanu, N.: Pruning for the *minimum* constraint family and for the *numberofdistinctvalues* constraint family. In: Walsh, T. (ed.) *CP 2001*. LNCS, vol. 2239, pp. 211–224. Springer, Heidelberg (2001)
3. Beldiceanu, N., Carlsson, M., Demasse, S., Petit, T.: Global constraint catalogue: Past, present and future. *Constraints* 12(1), 21–62 (2007)

4. Bessière, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: Filtering Algorithms for the NValue Constraint. *Constraints* 11(4), 271–293 (2006)
5. Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C.G., Walsh, T.: Propagating conjunctions of alldifferent constraints. *CoRR* abs/1004.2626 (2010)
6. Carter, M.W., Tovey, C.A.: When is the classroom assignment problem hard? *Operations Research* 40(1), 28–39 (1992)
7. Chabert, G., Jaulin, L., Lorca, X.: A constraint on the number of distinct vectors with application to localization. In: Gent, I.P. (ed.) *CP 2009*. LNCS, vol. 5732, pp. 196–210. Springer, Heidelberg (2009)
8. CHOCO Team: choco: an Open Source Java Constraint Programming Library. Tech. rep., Ecole des Mines de Nantes (2010), <http://www.emn.fr/z-info/choco-solver/>
9. Dijkstra, M.C., Kroon, L.G., Salomon, M., Van Nunen, J.A.E.E., van Wassenhove, L.N.: Planning the Size and Organization of KLM's Aircraft Maintenance Personnel. *Interfaces* 24(6), 47–58 (1994)
10. Dowling, D., Krishnamoorthy, M., Mackenzie, H., Sier, D.: Staff rostering at a large international airport. *Annals of Operations Research* 72, 125–147 (1997)
11. Ernst, A.T., Jiang, H., Krishnamoorthy, M., Sier, D.: Staff scheduling and rostering: A review of applications, methods and models. *European Journal of Operational Research* 153(1), 3–27 (2004)
12. Fischetti, M., Martello, S., Toth, P.: The fixed job schedule problem with spread-time constraints. *Operations Research* 35(6), 849–858 (1987)
13. Fischetti, M., Martello, S., Toth, P.: The fixed job schedule problem with working-time constraints. *Operations Research* 37(3), 395–403 (1989)
14. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*, 1st edn. W. H. Freeman (January 1979)
15. Golumbic, M.C.: *Algorithmic Graph Theory and Perfect Graphs*, 2nd edn. *Annals of Discrete Mathematics*, vol. 57. Elsevier (2004)
16. Gualandi, S., Malucelli, F.: Exact solution of graph coloring problems via constraint programming and column generation. *INFORMS J. Comput. Sc.* 24, 81–100 (2012)
17. Halldórsson, M.M., Radhakrishnan, J.: Greed is good: Approximating independent sets in sparse and bounded-degree graphs. *Algorithmica* 18(1), 145–163 (1997)
18. Haralick, R.M., Elliott, G.L.: Increasing tree search efficiency for constraint satisfaction problems. In: *Proceedings of the 6th International Joint Conference on Artificial Intelligence, IJCAI 1979*, vol. 1, pp. 356–364. Morgan Kaufmann Publishers Inc. (1979)
19. Kadioglu, S., Malitsky, Y., Sellmann, M., Tierney, K.: Isac - instance-specific algorithm configuration. In: *ECAI. Frontiers in Artificial Intelligence and Applications*, vol. 215, pp. 751–756. IOS Press (2010)
20. Krishnamoorthy, M., Ernst, A.T.: The personnel task scheduling problem. In: *Optimization Methods and Applications*, pp. 343–368. Springer, US (2001)
21. Krishnamoorthy, M., Ernst, A., Baatar, D.: Algorithms for large scale shift minimisation personnel task scheduling problems. *European Journal of Operational Research* 219, 34–48 (2012)
22. Kroon, L.G., Salomon, M., Wassenhove, L.N.V.: Exact and approximation algorithms for the tactical fixed interval scheduling problem. *Operations Research* 45(4) (1997)
23. Lapègue, T., Fages, J.G., Prot, D., Bellenguez-Morineau, O.: *Personnel Task Scheduling Problem Library* (2013), <https://sites.google.com/site/ptsplib/smptsp/home>



24. Lecoutre, C., Sais, L., Tabary, S., Vidal, V.: Reasoning from last conflict(s) in constraint programming. *Artif. Intell.* 173(18), 1592–1614 (2009)
25. Leone, R., Festa, P., Marchitto, E.: A Bus Driver Scheduling Problem: a new mathematical model and a GRASP approximate solution. *Journal of Heuristics* 17(4), 441–466 (2010)
26. López-Ortiz, A., Quimper, C.G., Tromp, J., van Beek, P.: A fast and simple algorithm for bounds consistency of the alldifferent constraint. In: *IJCAI*, pp. 245–250. Morgan Kaufmann (2003)
27. Monette, J.N., Flener, P., Pearson, J.: Towards solver-independent propagators. In: Milano, M. (ed.) *CP 2012*. LNCS, vol. 7514, pp. 544–560. Springer, Heidelberg (2012)
28. Pachet, F., Roy, P.: Automatic generation of music programs. In: Jaffar, J. (ed.) *CP 1999*. LNCS, vol. 1713, pp. 331–345. Springer, Heidelberg (1999)
29. Régis, J.C.: A Filtering Algorithm for Constraints of Difference in CSPs. In: *National Conference on Artificial Intelligence*, pp. 362–367. AAAI (1994)
30. Schulte, C., Stuckey, P.J.: Efficient constraint propagation engines. *CoRR* abs/cs/0611009 (2006)
31. Smet, P., Wauters, T., Mihaylow, M., Vanden Berghe, G.: The shift minimisation personnel task scheduling problem: a new hybrid approach and computational insights. Technical report (2013)
32. Van den Bergh, J., Beliën, J., De Bruecker, P., Demeulemeester, E., De Boeck, L.: Personnel scheduling: A literature review. *European Journal of Operational Research* 226(3), 367–385 (2013)

# A Parametric Approach for Smaller and Better Encodings of Cardinality Constraints

Ignasi Abío<sup>1</sup>, Robert Nieuwenhuis<sup>2</sup>, Albert Oliveras<sup>2</sup>, and Enric Rodríguez-Carbonell<sup>2</sup>

<sup>1</sup> Theoretical Computer Science, TU Dresden, Germany

<sup>2</sup> Technical University of Catalonia, Barcelona

**Abstract.** Adequate encodings for high-level constraints are a key ingredient for the application of SAT technology. In particular, *cardinality constraints* state that at most (at least, or exactly)  $k$  out of  $n$  propositional variables can be true. They are crucial in many applications. Although sophisticated encodings for cardinality constraints exist, it is well known that for small  $n$  and  $k$  straightforward encodings without auxiliary variables sometimes behave better, and that the choice of the right trade-off between minimizing either the number of variables or the number of clauses is highly application-dependent. Here we build upon previous work on Cardinality Networks to get the best of several worlds: we develop an arc-consistent encoding that, by recursively decomposing the constraint into smaller ones, allows one to decide which encoding to apply to each sub-constraint. This process minimizes a function  $\lambda \cdot \text{num\_vars} + \text{num\_clauses}$ , where  $\lambda$  is a parameter that can be tuned by the user. Our careful experimental evaluation shows that (e.g., for  $\lambda = 5$ ) this new technique produces much smaller encodings in variables and clauses, and indeed strongly improves SAT solvers' performance.

## 1 Introduction

This paper presents a new encoding into SAT of *cardinality constraints*, that is, constraints of the form  $x_1 + \dots + x_n \# k$ , where  $k$  is a natural number, the  $x_i$  are propositional variables, and the relation operator  $\#$  belongs to  $\{<, >, \leq, \geq, =\}$ . Cardinality constraints are present in many practical SAT applications, such as cumulative scheduling [17] or timetabling [4]. They also arise as components of some SAT-based techniques, e.g., for MaxSAT [11].

Here we are interested in encoding a cardinality constraint  $C$  with a clause set  $S$  (possibly with auxiliary variables) that is not only equisatisfiable, but also *arc-consistent*: given a partial assignment  $A$ , if  $x_i$  is true (false) in every extension of  $A$  satisfying  $C$ , then unit propagating  $A$  on  $S$  sets  $x_i$  to true (false)<sup>1</sup>. Enforcing arc-consistency by unit propagation in this way has of course an important positive impact on the practical efficiency of SAT solvers.

A straightforward encoding of a cardinality constraint  $x_1 + \dots + x_n \leq k$  is to state, for each subset  $Y$  of  $\{x_1, \dots, x_n\}$  with  $|Y| = k + 1$ , that at least one variable of  $Y$  must be false. This can be done by asserting  $\binom{n}{k+1}$  clauses of the form  $\overline{x_{i_1}} \vee \dots \vee \overline{x_{i_{k+1}}}$ . This kind of construction frequently works well, although it is of course not reasonable for

---

<sup>1</sup> Sometimes this notion is called *generalized arc-consistency*.

large  $n$  and  $k$ , which is our aim in this work. Successively more sophisticated encodings using auxiliary variables have been defined that require fewer clauses (see Section 2). But still, for small  $n$  and  $k$  the straightforward encodings may behave better in practice. An additional issue is that, for the efficiency of the SAT solver, the choice of the right trade-off between minimizing either the number of auxiliary variables or the number of clauses is highly application-dependent.

Here we build upon and improve previous work on encoding cardinality constraints with Cardinality Networks [2,3], which use  $O(n \log^2 k)$  variables and clauses (see Section 3). The idea is to get the best of several worlds: we develop a hybrid arc-consistent encoding that, by recursively decomposing the constraint into smaller ones, allows one to decide whether to apply a recursive (see Section 4) or a direct (see Section 5) encoding to each sub-constraint. This process minimizes a function  $\lambda \cdot \text{num\_vars} + \text{num\_clauses}$ , where  $\lambda$  is a parameter that can be tuned by the user (see Section 6). Our experimental evaluation shows that (e.g., for  $\lambda = 5$ ) this new technique produces much smaller encodings in variables *and* clauses, and indeed strongly improves the performance of SAT solvers (see Section 7).

## 2 Related Work

Because of their practical importance, encodings of cardinality constraints into SAT have been thoroughly studied over the last few years. In this section we review some of the most important works in the literature.

In [20], Warners considered the more general pseudo-Boolean case, where constraints are of the form  $a_1x_1 + \dots + a_nx_n \leq k$ , being the  $a_i$ 's and the  $k$  integer coefficients and the  $x_i$ 's Boolean variables. The encoding is based on using adders for numbers represented in binary. For cardinality constraints the encoding uses  $O(n)$  clauses and variables, but does not preserve arc consistency.

Bailleux and Boufkhad presented in [5] an arc-consistent encoding of cardinality constraints that uses  $O(n \log n)$  variables and  $O(n^2)$  clauses. The encoding consists of a *totalizer* and a *comparator*. The totalizer can be seen as a binary tree, where the leaves are the  $x_i$ 's variables. Each inner node is labeled with a number  $s$  and uses  $s$  auxiliary variables to represent, in unary, the sum of the leaves of the corresponding subtree. As for the comparator, it is easily encoded thanks to the unary representation, which also allows handling constraints of the form  $k_1 \leq x_1 + \dots + x_n \leq k_2$  without splitting.

A more applied work is the one of Büttner and Rintanen [19]. Although their main interest was in planning, they suggested two encodings of cardinality constraints. The first one is based on encoding an injective mapping between the true  $x_i$ 's variables and  $k$  elements. It uses  $O(nk)$  clauses and variables and is not arc-consistent. The other encoding is a small modification of [5]. Based on the observation that counting up to  $k + 1$  suffices, they can reduce the number of variables and clauses used in each node. The resulting encoding requires  $O(nk)$  variables and  $O(nk^2)$  clauses, which improves on [5] if  $k$  is small enough.

In [18], Sinz proposed two different encodings, both based on counters. The first encoding uses a sequential counter where numbers are represented in unary. It needs  $O(nk)$  clauses and variables and is arc-consistent. The second one is based on a parallel counter, where numbers are represented in binary. It requires  $O(n)$  clauses and variables, but is not arc-consistent.

Another kind of encoding was used in [6], where a BDD-like technique was proposed for pseudo-Boolean constraints. The encoding is arc-consistent, and uses  $O(n^2)$  clauses and variables when applied to cardinality constraints. The idea is as follows: given a pseudo-Boolean constraint  $a_1x_1 + \dots + a_nx_n \leq k$ , the root of the BDD is labeled with variable  $D_{n,k}$ , expressing that the sum of the first  $n$  terms is at most  $k$ . The two corresponding children are  $D_{n-1,k}$  and  $D_{n-1,k-a_n}$ , indicating the two cases that correspond to setting  $x_n$  to false and true, respectively. Then the necessary clauses are added to express the relationship between the variables, and trivial cases are treated accordingly.

The same authors presented in [7] a polynomial and arc-consistent encoding of pseudo-Boolean constraints. When restricted to cardinality constraints it is similar to [5], but the latter is better in terms of size.

Yet another approach for encoding cardinality constraints was suggested in [1]. The authors revisit the idea of using totalizers, and realize that totalizers require two parameters: the encoding used (unary or binary) and the way the totalizers are grouped (e.g.  $(a + b) + (c + d)$  or  $((a + b) + c) + d$ ). A thorough experimentation is performed, to which they add two extra aspects: (i) how to order the variables; and (ii), the use of encodings in parallel, hoping the SAT solver will focus on the most appropriate one for each problem.

Finally, Eén and Sörensson [10] presented three encodings for pseudo-Boolean constraints. The first encoding is BDD-based, similar to [6]. The second one, based on adder networks, improves that of [20] in that it uses less adders, but is still linear and does not preserve arc consistency. Finally, their third encoding uses *Sorting Networks* [8]. A Sorting Network takes input variables  $(x_1 \dots x_n)$  and returns as outputs  $(y_1 \dots y_n)$  the sorted input values in decreasing order. Hence, an output variable  $y_k$  will become true iff there are at least  $k$  true input variables, and false iff there are at least  $n - k + 1$  false ones. Now, to express  $x_1 + \dots + x_n \geq k$ , it suffices to add a unit clause  $y_k$ ; similarly, for  $x_1 + \dots + x_n \leq k$  one adds  $\overline{y_{k+1}}$ , and both are added if the relation is  $=$ . This encoding, when restricted to cardinality constraints, preserves arc consistency and requires  $O(n \log^2 n)$  clauses and variables. The Cardinality Networks of [2,3] reduce this to  $O(n \log^2 k)$ , which is important as often  $n \gg k$ .

A similar approach uses so-called Pairwise Cardinality Networks [9], which are based on Pairwise Sorting Networks [15] instead of Sorting Networks. By means of partial evaluation, this method also achieves  $O(n \log^2 k)$  variables and clauses. Finally, we were recently informed that a hybrid approach based on Pairwise Cardinality Networks similar to that presented here was implemented in the BEE system [13]. However, no detailed description or experimental evaluation is available. Moreover, our proposal in this paper is more general, in the sense that it allows the user to tune the parameter  $\lambda$  when minimizing the objective function  $\lambda \cdot \text{num\_vars} + \text{num\_clauses}$ .

### 3 Preliminaries

In this work we describe a method for producing cardinality networks that generalizes the construction of [3]. The core idea of these approaches, which dates back to [8], consists in encoding a circuit that implements mergesort by means of a set of clauses. The most basic components of these circuits are 2-comparators.

A *2-comparator* is a sorting network of size 2, i.e., it has 2 input variables ( $x_1$  and  $x_2$ ) and 2 output variables ( $y_1$  and  $y_2$ ) such that  $y_1$  is true if and only if at least one of the input variables is true, and  $y_2$  is true if and only if both two input variables are true. In the following, 2-comparators are denoted by  $(y_1, y_2) = 2\text{-Comp}(x_1, x_2)$ . As pointed out in [3], for encoding  $\leq$ -constraints, only the three clauses on the first row of Fig. 1 are needed to guarantee arc-consistency. The three clauses on the second row suffice for  $\geq$ -constraints and all six must be present when encoding  $=$ -constraints. Note that the usual polarity argument [16] cannot be directly applied here, as we are interested not only in preserving satisfiability, but also arc-consistency under unit propagation.

$$\begin{array}{l} x_1 \rightarrow y_1, \quad x_2 \rightarrow y_1, \quad x_1 \wedge x_2 \rightarrow y_2, \\ \overline{x_1} \rightarrow \overline{y_2}, \quad \overline{x_2} \rightarrow \overline{y_2}, \quad \overline{x_1} \wedge \overline{x_2} \rightarrow \overline{y_1} \end{array} \qquad \begin{array}{c} x_1 \text{ --- } \bullet \text{ --- } y_1 \\ | \\ x_2 \text{ --- } \bullet \text{ --- } y_2 \end{array}$$

Fig. 1. A 2-comparator: clauses (left) and graphical representation (right)

## 4 Arbitrary-Sized Recursive Cardinality Networks

In this section we generalize the recursive construction of cardinality networks given in [3] by allowing inputs and outputs of any size, not necessarily a power of two. Not only does this avoid adding dummy variables that are not actually needed (which, as will be seen in Section 7, has an impact on performance on its own), but also becomes useful when combining with the direct (non-recursive) constructions of Section 5.

In what follows, we denote by  $\lfloor r \rfloor$  and  $\lceil r \rceil$  the floor and ceiling functions respectively. Moreover, for simplicity, we will assume that the constraint to be encoded is a  $\leq$ -constraint. However, similar constructions for the other constraints can be devised.

### 4.1 Merge Networks

A *merge network* takes as input two (decreasingly) ordered sets of sizes  $a$  and  $b$  and produces a (decreasingly) ordered set of size  $a + b$ . We can build a merge network with inputs  $(x_1, \dots, x_a)$  and  $(x'_1, \dots, x'_b)$  in a recursive way as follows<sup>2</sup>:

- If  $a = b = 1$ , a merge network is a 2-comparator:

$$\text{Merge}(x_1; x'_1) := 2\text{-Comp}(x_1, x'_1).$$

- If  $a = 0$ , a merge network returns the second input:

$$\text{Merge}(); x'_1, x'_2, \dots, x'_b := (x'_1, x'_2, \dots, x'_b).$$

<sup>2</sup> Notice we use the notation  $\text{Merge}(X; X')$  instead of  $\text{Merge}((X), (X'))$  for simplicity.

- If  $a$  and  $b$  are even,  $a > 0$ ,  $b > 0$  and either  $a > 1$  or  $b > 1$ , let us define

$$\begin{aligned}
 (z_1, z_3, \dots, z_{a-3}, z_{a-1}, &= \text{Merge}(x_1, x_3, \dots, x_{a-1}; \\
 z_{a+1}, z_{a+3}, \dots, z_{a+b-1}) &= x'_1, x'_3, \dots, x'_{b-1}), \\
 (z_2, z_4, \dots, z_{a-2}, z_a, &= \text{Merge}(x_2, x_4, \dots, x_a; \\
 z_{a+2}, z_{a+4}, \dots, z_{a+b}) &= x'_2, x'_4, \dots, x'_b), \\
 (y_2, y_3) &= 2\text{-Comp}(z_2, z_3), \\
 &\dots \\
 (y_{a+b-2}, y_{a+b-1}) &= 2\text{-Comp}(z_{a+b-2}, z_{a+b-1}).
 \end{aligned}$$

Then,

$$\text{Merge}(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b) := (z_1, y_2, y_3, \dots, y_{a+b-1}, z_{a+b}).$$

- If  $a$  is even,  $b$  is odd,  $a > 0$ ,  $b > 0$  and either  $a > 1$  or  $b > 1$ , let us define

$$\begin{aligned}
 (z_1, z_3, \dots, z_{a-1}, &= \text{Merge}(x_1, x_3, \dots, x_{a-1}; \\
 z_{a+1}, z_{a+3}, \dots, z_{a+b}) &= x'_1, x'_3, \dots, x'_b), \\
 (z_2, z_4, \dots, z_a, z_{a+2}, &= \text{Merge}(x_2, x_4, \dots, x_a; \\
 z_{a+4}, \dots, z_{a+b-1}) &= x'_2, x'_4, \dots, x'_{b-1}), \\
 (y_2, y_3) &= 2\text{-Comp}(z_2, z_3), \\
 &\dots \\
 (y_{a+b-1}, y_{a+b}) &= 2\text{-Comp}(z_{a+b-1}, z_{a+b}).
 \end{aligned}$$

Then,

$$\text{Merge}(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b) := (z_1, y_2, y_3, \dots, y_{a+b-1}, y_{a+b}).$$

- If  $a$  and  $b$  are odd,  $a > 0$ ,  $b > 0$  and either  $a > 1$  or  $b > 1$ , let us define

$$\begin{aligned}
 (z_1, z_3, \dots, z_{a-2}, z_a, &= \text{Merge}(x_1, x_3, \dots, x_a; \\
 z_{a+1}, z_{a+3}, \dots, z_{a+b}) &= x'_1, x'_3, \dots, x'_b), \\
 (z_2, z_4, \dots, z_{a-3}, z_{a-1}, &= \text{Merge}(x_2, x_4, \dots, x_{a-3}, x_{a-1}; \\
 z_{a+2}, z_{a+4}, \dots, z_{a+b-1}) &= x'_2, x'_4, \dots, x'_{b-1}), \\
 (y_2, y_3) &= 2\text{-Comp}(z_2, z_3), \\
 &\dots \\
 (y_{a+b-2}, y_{a+b-1}) &= 2\text{-Comp}(z_{a+b-2}, z_{a+b-1}).
 \end{aligned}$$

Then,

$$\text{Merge}(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b) := (z_1, y_2, y_3, \dots, y_{a+b-1}, z_{a+b}).$$

- The remaining cases are defined thanks to the symmetry of the merge function, i.e., due to  $\text{Merge}(X, X') = \text{Merge}(X', X)$ .

The base cases do not require any explanation. As regards the recursive ones, first notice that the set of values  $x_1, x_2, \dots, x_a, x'_1, x'_2, \dots, x'_b$  is always preserved. Further, the

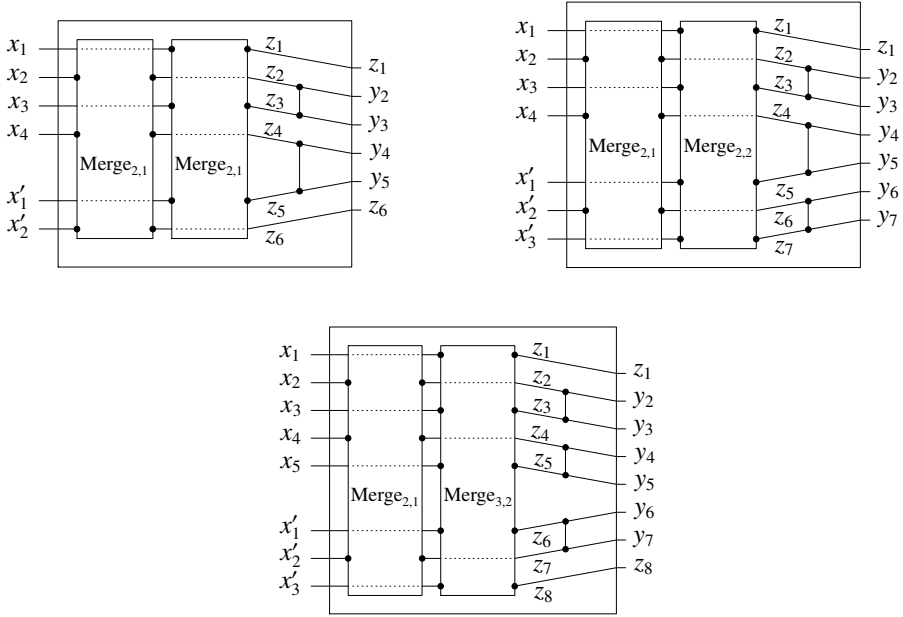


**Fig. 2.** Different examples of merge networks

output bits are sorted, as  $z_{2i} \geq z_{2(i+1)}$ ,  $z_{2i} \geq z_{2(i+1)+1}$ ,  $z_{2i+1} \geq z_{2(i+1)}$  and  $z_{2i+1} \geq z_{2(i+1)+1}$  imply that  $\min(z_{2i}, z_{2i+1}) \geq \max(z_{2(i+1)}, z_{2(i+1)+1})$ . Figure 2 shows examples of some of these recursive cases.

The number of auxiliary variables and clauses of a merge network defined in this way can be recursively computed. A merge network with inputs of size  $(1, 1)$  needs 2 variables and 3 clauses. A merge network with inputs of size  $(0, b)$  needs no variables and clauses. A merge network with inputs of size  $(a, b)$  with  $a > 1$  or  $b > 1$  needs  $V_1 + V_2 + 2 \left\lfloor \frac{a+b-1}{2} \right\rfloor$  variables and  $C_1 + C_2 + 3 \left\lfloor \frac{a+b-1}{2} \right\rfloor$  clauses, where  $V_1$  and  $C_1$  are the number of variables and clauses in a merge network with inputs of size  $\left(\left\lceil \frac{a}{2} \right\rceil, \left\lceil \frac{b}{2} \right\rceil\right)$ , and  $V_2, C_2$  are idem in a merge network with inputs of size  $\left(\left\lfloor \frac{a}{2} \right\rfloor, \left\lfloor \frac{b}{2} \right\rfloor\right)$ .

In comparison to [3], in that work it was assumed that  $a = b = 2^m$  for some  $m \geq 0$ . Thanks to this, only one base case ( $a = b = 1$ ) and one recursive case ( $a, b$  even) were considered there. All the other cases introduced here are needed for arbitrary  $a$  and  $b$ .

## 4.2 Sorting Networks

A *sorting network* takes an input of size  $n$  and sorts it. It can be built in a recursive way as follows, using the same strategy as in mergesort:

- If  $n = 1$ , the output of the sorting network is its input:

$$\text{Sorting}(x_1) := x_1$$

- If  $n = 2$ , a sorting network is a single merge (i.e., a 2-comparator):

$$\text{Sorting}(x_1, x_2) := \text{Merge}(x_1; x_2).$$

- For  $n > 2$ , take  $l$  with  $1 \leq l < n$ : Let us define

$$\begin{aligned} (z_1, z_2, \dots, z_l) &= \text{Sorting}(x_1, x_2, \dots, x_l), \\ (z_{l+1}, z_{l+2}, \dots, z_n) &= \text{Sorting}(x_{l+1}, x_{l+2}, \dots, x_n), \\ (y_1, y_2, \dots, y_n) &= \text{Merge}(z_1, z_2, z_l; z_{l+1}, \dots, z_n). \end{aligned}$$

Then,

$$\text{Sorting}(x_1, x_2, \dots, x_n) := (y_1, y_2, \dots, y_n).$$

Again, the number of auxiliary variables and clauses needed in these networks can be recursively computed. A sorting network of input size 1 needs no variables and clauses. A sorting network of input size 2 needs 2 variables and 3 clauses. A sorting network of input size  $n$  composed by a sorting network of size  $l$  and a sorting network of size  $n - l$  needs  $V_1 + V_2 + V_3$  variables and  $C_1 + C_2 + C_3$  clauses, where  $(V_1, C_1)$ ,  $(V_2, C_2)$  are the number of variables and clauses used in the sorting networks of sizes  $l$  and  $n - l$ , and  $(V_3, C_3)$  are the number of variables and clauses needed in the merge network with inputs of sizes  $(l, n - l)$ .

In comparison to [3], in that work  $n$  is assumed to be a power of two. Moreover, in the recursive case  $l$  is always chosen to be  $n/2$ , while here we can build sorting networks of any size, and have the additional freedom of choosing the sizes of the two sorting network components.

### 4.3 Simplified Merge Networks

A *simplified merge* is a reduced version of a merge, used when we are only interested in some of the outputs, but not all. Recall that we want to encode a constraint of the form  $x_1 + \dots + x_n \leq k$ , and hence we are only interested in the first  $k + 1$  bits of the sorted output. Thus, in a  $c$ -simplified merge network, the inputs are two sorted sequences of variables  $(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b)$ , and the network produces a sorted output of the desired size,  $c$ ,  $(y_1, y_2, \dots, y_c)$ . The network satisfies that  $y_r$  is true if there are at least  $r$  true inputs. We can build a recursive simplified merge as follows:

- If  $a = b = c = 1$ , let us add the clauses  $x_1 \rightarrow y$ ,  $x'_1 \rightarrow y^3$ . Then:

$$\text{SMerge}_1(x_1; x'_1) := y.$$

- If  $a > c$ , we can ignore the last  $a - c$  bits of the first input (similarly if  $b > c$ ):

$$\text{SMerge}_c(x_1, x_2, \dots, x_a; x'_1, \dots, x'_b) = \text{SMerge}_c(x_1, x_2, \dots, x_c; x'_1, \dots, x'_b).$$

- If  $a + b \leq c$ , the simplified merge is a merge:

$$\text{SMerge}_c(x_1, \dots, x_a; x'_1, \dots, x'_b) = \text{Merge}(x_1, \dots, x_a; x'_1, \dots, x'_b).$$

---

<sup>3</sup> Notice that these clauses correspond to the bit of the 2-comparator with lower index. Clause  $\overline{x_1} \wedge \overline{x_2} \rightarrow \overline{y}$  does not need to be included here following the reasoning given in Section 3.



- If  $a, b \leq c$ ,  $a + b > c$  and  $c$  is even: Let us define

$$\begin{aligned} (z_1, z_3, \dots, z_{c+1}) &= \text{SMerge}_{c/2+1}(x_1, x_3, \dots; x'_1, x'_3, \dots), \\ (z_2, z_4, \dots, z_c) &= \text{SMerge}_{c/2}(x_2, x_4, \dots; x'_2, x'_4, \dots), \\ (y_2, y_3) &= \text{2-Comp}(z_2, z_3), \\ &\dots \\ (y_{c-2}, y_{c-1}) &= \text{2-Comp}(z_{c-2}, z_{c-1}). \end{aligned}$$

and add the clauses  $z_c \rightarrow y_c$ ,  $z_{c+1} \rightarrow y_c$ . Then,

$$\text{SMerge}_c(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b) := (z_1, y_2, y_3, \dots, y_c),$$

- If  $a, b \leq c$ ,  $a + b > c$  and  $c > 1$  is odd: Let us define

$$\begin{aligned} (z_1, z_3, \dots, z_c) &= \text{SMerge}_{\frac{c+1}{2}}(x_1, x_3, \dots; x'_1, x'_3, \dots), \\ (z_2, z_4, \dots, z_{c-1}) &= \text{SMerge}_{\frac{c-1}{2}}(x_2, x_4, \dots; x'_2, x'_4, \dots), \\ (y_2, y_3) &= \text{2-Comp}(z_2, z_3), \\ &\dots \\ (y_{c-1}, y_c) &= \text{2-Comp}(z_{c-1}, z_c). \end{aligned}$$

Then,

$$\text{SMerge}_c(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b) := (z_1, y_2, y_3, \dots, y_c).$$

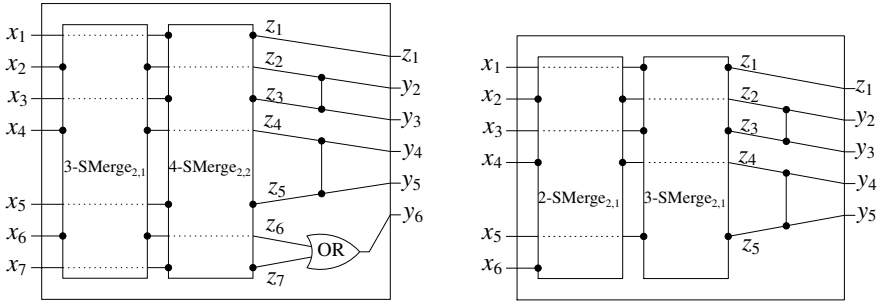


**Fig. 3.** Two examples of simplified merge networks

Figure 3 shows two examples of simplified merges: The first one shows a 6-simplified merge with inputs of sizes 3 and 4. The second one corresponds to a 5-simplified merge with inputs of sizes 2 and 4.

We can recursively compute the auxiliary variables and clauses needed in simplified merge networks. In the recursive case, we need  $V_1 + V_2 + c - 1$  variables and  $C_1 + C_2 + C_3$  clauses, where  $(V_1, C_1), (V_2, C_2)$  are the number of clauses and variables needed in simplified merge networks of sizes  $(\lceil \frac{a}{2} \rceil, \lceil \frac{b}{2} \rceil, \lfloor \frac{c}{2} \rfloor + 1)$ ,  $(\lfloor \frac{a}{2} \rfloor, \lfloor \frac{b}{2} \rfloor, \lfloor \frac{c}{2} \rfloor)$ , and

$$C_3 = \begin{cases} \frac{3c-3}{2} & \text{if } c \text{ is odd,} \\ \frac{3c-2}{2} + 2 & \text{if } c \text{ is even.} \end{cases}$$

Compared to [3], there it was assumed that  $a = b = 2^m$  for some  $m \geq 0$ , and  $c = 2^m + 1$ . Similarly to merge networks, only one base case and one recursive case were considered. All the other cases introduced here are needed for arbitrary  $a, b$  and  $c$ .

#### 4.4 $m$ -Cardinality Networks

An  $m$ -cardinality network takes an input of size  $n$  and outputs the first  $m$  sorted bits. Recursively, an  $m$ -cardinality network with input  $x_1, x_2, \dots, x_n$  can be defined as follows:

- If  $n \leq m$ , a cardinality network is a sorting network:

$$\text{Card}_m(x_1, x_2, \dots, x_n) := \text{Sorting}(x_1, x_2, \dots, x_n).$$

- If  $n > m$ , take  $l$  with  $1 \leq l < n$ . Let us define

$$\begin{aligned} (z_1, z_2, \dots, z_A) &= \text{Card}_m(x_1, x_2, \dots, x_l), \\ (z'_1, z'_2, \dots, z'_B) &= \text{Card}_m(x_{l+1}, x_{l+2}, \dots, x_n), \\ (y_1, y_2, \dots, y_m) &= \text{SMerge}_m(z_1, z_2, \dots, z_A; z'_1, z'_2, \dots, z'_B), \end{aligned}$$

where  $A = \min\{l, m\}$  and  $B = \min\{n - l, m\}$ . Then,

$$\text{Card}_m(x_1, x_2, \dots, x_n) := (y_1, y_2, \dots, y_m).$$

Again, the number of auxiliary variables and clauses needed in these networks can be recursively computed. An  $m$ -cardinality network of size  $n$  composed by an  $m$ -cardinality network of size  $l$  and an  $m$ -cardinality network of size  $n - l$  needs  $V_1 + V_2 + V_3$  variables and  $C_1 + C_2 + C_3$  clauses, where  $(V_1, C_1)$ ,  $(V_2, C_2)$  are the number of variables and clauses used in the  $m$ -cardinality networks of sizes  $l$  and  $n - l$ , and  $(V_3, C_3)$  are idem in the  $m$ -simplified merge network with inputs of sizes  $(\min\{l, m\}, \min\{n - l, m\})$ .

Compared to [3], in that work  $m$  is assumed to be a power of two, and  $n$  a multiple of  $m$ . Moreover, similarly to sorting networks, in the recursive case  $l$  is always chosen to be  $m$ , while here we have an additional degree of freedom.

Using the same techniques in [3] one can easily prove the arc-consistency of the encoding.

**Theorem 1.** *The Recursive Cardinality Network encoding is arc-consistent: consider a cardinality constraint  $x_1 + \dots + x_n \leq k$ , its corresponding cardinality network  $(y_1, y_2, \dots, y_{k+1}) = \text{Card}_{k+1}(x_1, x_2, \dots, x_n)$ , and the unit clause  $\neg y_{k+1}$ . If we now set to true  $k$  input variables, then unit propagation sets to false the remaining  $n - k$  input variables.*

*Proof (sketch).* The proof relies on the following lemmas, which formalize the propagation properties of the building blocks of cardinality networks:

**Lemma 1 (Merge Networks).** *Let  $S$  be the set of clauses of*

$$(y_1, y_2, \dots, y_{a+b}) = \text{Merge}(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b).$$

*Let  $p, q \in \mathbb{N}$  with  $0 \leq p \leq a$  and  $0 \leq q \leq b$ . Then:*

1.  $S \cup \{x_1, \dots, x_p, x'_1, \dots, x'_q\} \models_{\text{UP}} y_1, \dots, y_{p+q}$ .
2. If  $p < a$  and  $q < b$  then  $S \cup \{x_1, \dots, x_p, x'_1, \dots, x'_q, \overline{y_{p+q+1}}\} \models_{\text{UP}} \overline{x_{p+1}}, \overline{x'_{q+1}}$ .
3. If  $p = a$  and  $q < b$  then  $S \cup \{x_1, \dots, x_p, x'_1, \dots, x'_q, \overline{y_{p+q+1}}\} \models_{\text{UP}} \overline{x'_{q+1}}$ .
4. If  $p < a$  and  $q = b$  then  $S \cup \{x_1, \dots, x_p, x'_1, \dots, x'_q, \overline{y_{p+q+1}}\} \models_{\text{UP}} \overline{x_{p+1}}$ .

**Lemma 2 (Sorting Networks).** Let  $X = (x_1, x_2, \dots, x_n)$ ,  $X' \subseteq X$  and  $S$  be the set of clauses of  $(y_1, y_2, \dots, y_n) = \text{Sorting}(X)$ . Let  $p = |X'|$ . Then:

1.  $S \cup X' \models_{\text{UP}} y_1, \dots, y_p$ .
2. If  $p = |X'| < n$ , then  $S \cup X' \cup \{\overline{y_{p+1}}\} \models_{\text{UP}} \overline{x_i}$  for all  $x_i \notin X'$ .

**Lemma 3 (Simplified Merge Networks).** Let  $S$  be the set of clauses of

$$(y_1, y_2, \dots, y_c) = \text{SMerge}_c(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b).$$

Let  $p, q \in \mathbb{N}$  be such that  $0 \leq p \leq a$ ,  $0 \leq q \leq b$ . Then:

1. If  $p + q \leq c$ , then  $S \cup \{x_1, \dots, x_p, x'_1, \dots, x'_q\} \models_{\text{UP}} y_1, \dots, y_{p+q}$ .
2. If  $p < a$ ,  $q < b$  and  $p + q < c$ , then  $S \cup \{x_1, \dots, x_p, x'_1, \dots, x'_q, \overline{y_{p+q+1}}\} \models_{\text{UP}} \overline{x_{p+1}}, \overline{x'_{q+1}}$ .
3. If  $p = a$ ,  $q < b$  and  $p + q < c$ , then  $S \cup \{x_1, \dots, x_p, x'_1, \dots, x'_q, \overline{y_{p+q+1}}\} \models_{\text{UP}} \overline{x'_{q+1}}$ .
4. If  $p < a$ ,  $q = b$  and  $p + q < c$ , then  $S \cup \{x_1, \dots, x_p, x'_1, \dots, x'_q, \overline{y_{p+q+1}}\} \models_{\text{UP}} \overline{x_{p+1}}$ .

**Lemma 4 (Cardinality Networks).** Let  $X = (x_1, x_2, \dots, x_n)$ ,  $X' \subseteq X$  and  $S$  be the set of clauses of  $(y_1, y_2, \dots, y_m) = \text{Card}_m(X)$ . Let  $p = |X'|$ . Then:

1. If  $p \leq m$ , then  $S \cup X' \models_{\text{UP}} y_1, \dots, y_p$ .
2. If  $p < m$ , then  $S \cup X' \cup \{\overline{y_{p+1}}\} \models_{\text{UP}} \overline{x_i}$  for all  $x_i \notin X'$ .

Each lemma is proved by induction and using the corresponding lemmas of the inner building blocks. The proofs of Lemmas 1 and 3 require considering four cases according to the parities of  $p$  and  $q$ . Finally, the theorem follows as a corollary of Lemma 4.

For the sake of illustration, let us prove the case  $a, b \leq c$ ,  $a + b > c$ , with  $c$  even, of the inductive case of property 1 in Lemma 3. So, let us consider the set of clauses of

$$(z_1, y_2, y_3, \dots, y_c) = \text{SMerge}_c(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b)$$

consisting of the clauses  $z_c \rightarrow y_c$ ,  $z_{c+1} \rightarrow y_c$  and those in

$$\begin{aligned} (z_1, z_3, \dots, z_{c+1}) &= \text{SMerge}_{c/2+1}(x_1, x_3, \dots; x'_1, x'_3, \dots), \\ (z_2, z_4, \dots, z_c) &= \text{SMerge}_{c/2}(x_2, x_4, \dots; x'_2, x'_4, \dots), \\ (y_2, y_3) &= 2\text{-Comp}(z_2, z_3), \\ &\dots \\ (y_{c-2}, y_{c-1}) &= 2\text{-Comp}(z_{c-2}, z_{c-1}). \end{aligned}$$

Let  $p, q \in \mathbb{N}$  such that  $0 \leq p \leq a$ ,  $0 \leq q \leq b$  and  $p + q \leq c$ . If  $p = q = 0$  there is nothing to prove. Otherwise let us show  $S \cup \{x_1, \dots, x_p, x'_1, \dots, x'_q\} \models_{\text{UP}} z_1, y_i$  for all  $2 \leq i \leq p + q$ .

Here we focus on the subcase  $p$  and  $q$  even, being the other three cases analogous. Hence, let  $p = 2p'$  and  $q = 2q'$ . In  $x_1, x_2, \dots, x_p$  there are  $p'$  odd indices and  $p'$  even indices. Similarly, in  $x'_1, x'_2, \dots, x'_q$  there are  $q'$  odd indices and  $q'$  even indices. Thus, using the IH (note  $p' + q' \leq c/2 < c/2 + 1$ ), we have that the clauses of the subnetwork  $(z_1, z_3, \dots, z_{c+1}) = \text{SMerge}_{c/2+1}(x_1, x_3, \dots; x'_1, x'_3, \dots)$  propagate by unit propagation the literals  $z_1, \dots, z_{2(p'+q')-1}$ ; and that the clauses of  $(z_2, z_4, \dots, z_c) = \text{SMerge}_{c/2}(x_2, x_4, \dots; x'_2, x'_4, \dots)$  propagate by unit propagation the literals  $z_2, \dots, z_{2(p'+q')}$ . Altogether, all literals  $z_j$  with  $1 \leq j \leq p + q$  can be propagated by unit propagation.

Let us take  $2 \leq i \leq p + q$ . If  $i$  is odd then, thanks to literals  $z_{i-1}$  and  $z_i$  and clause  $z_{i-1} \wedge z_i \rightarrow y_i$  of the 2-comparator  $(y_{i-1}, y_i) = 2\text{-Comp}(z_{i-1}, z_i)$ , literal  $y_i$  is propagated. If  $i$  is even, then thanks to literal  $z_i$  and clause  $z_i \rightarrow y_i$ , literal  $y_i$  is propagated too.

## 5 Direct Cardinality Networks

In this section we introduce an alternative technique for building cardinality networks which we call *direct*, as it is non-recursive. This method uses many fewer auxiliary variables than the recursive approach explained in Section 4. On the other hand, the number of clauses of this construction makes it competitive only for small sizes. However, this is not a problem as we will see in Section 6, as the two techniques can be combined.

As in the recursive construction described in Section 4, the building blocks of direct cardinality networks are merge, sorting and simplified merge networks:

- **Merge Networks.** They are defined as follows<sup>4</sup>:

$$\text{Merge}(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b) := (y_1, y_2, y_3, \dots, y_{a+b-1}, y_{a+b}),$$

with clauses  $\{x_i \rightarrow y_i, x'_j \rightarrow y_j, x_i \wedge x'_j \rightarrow y_{i+j} : 1 \leq i \leq a, 1 \leq j \leq b\}$ . Notice we need  $a + b$  variables and  $ab + a + b$  clauses.

- **Sorting Networks.** A sorting network can be built as follows:

$$\text{Sorting}(x_1, x_2, \dots, x_n) := (y_1, y_2, \dots, y_n),$$

with clauses  $\{x_{i_1} \wedge x_{i_2} \wedge \dots \wedge x_{i_k} \rightarrow y_k : 1 \leq k \leq n, 1 \leq i_1 < i_2 < \dots < i_k \leq n\}$ . Therefore, we need  $n$  auxiliary variables and  $2^n - 1$  clauses.

- **Simplified Merge Networks.** The definition of  $c$ -simplified merge is the same as in Section 4, except for the cases in which  $a, b \leq c$  and  $a + b > c$ , where:

$$\text{SMerge}_c(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b) := (y_1, y_2, \dots, y_c),$$

with clauses  $\{x_i \rightarrow y_i, x'_j \rightarrow y_j, x_i \wedge x'_j \rightarrow y_{i+j} : 1 \leq i \leq a, 1 \leq j \leq b, i + j \leq c\}$ .

This approach needs  $c$  variables and  $(a + b)c - \frac{c(c-1)}{2} - \frac{a(a-1)}{2} - \frac{b(b-1)}{2}$  clauses.

- **$m$ -Cardinality Networks.** As in Section 4, except for the case  $n > m$ , where:

$$\text{Card}_m(x_1, x_2, \dots, x_n) := (y_1, y_2, \dots, y_m)$$

with clauses  $\{x_{i_1} \wedge x_{i_2} \wedge \dots \wedge x_{i_k} \rightarrow y_k : 1 \leq k \leq m, 1 \leq i_1 < i_2 < \dots < i_k \leq n\}$ .

This approach needs  $m$  variables and  $\binom{n}{1} + \binom{n}{2} + \dots + \binom{n}{m}$  clauses.

As regards the arc-consistency of the encoding, the following can be easily proved:

**Theorem 2.** *The Direct Cardinality Network encoding is arc-consistent.*

*Proof (sketch).* The proof uses lemmas analogous to Lemmas 1, 2, 3 and 4. For illustration purposes, let us show property 1 in Lemma 3. Let us consider the clause set of  $(y_1, y_2, \dots, y_c) = \text{SMerge}_c(x_1, x_2, \dots, x_a; x'_1, x'_2, \dots, x'_b)$ , i.e.,

$$\{x_i \rightarrow y_i, x'_j \rightarrow y_j, x_i \wedge x'_j \rightarrow y_{i+j} : 1 \leq i \leq a, 1 \leq j \leq b, i + j \leq c\}.$$

Let  $p, q \in \mathbb{N}$  be such that  $0 \leq p \leq a, 0 \leq q \leq b$  and  $p + q \leq c$ . If  $p = q = 0$  there is nothing to prove. Otherwise let us consider  $1 \leq k \leq p + q$ . Let  $0 \leq i \leq p$  and  $0 \leq j \leq q$  be such that  $i + j = k$ . If  $i = 0$  then  $j = k$  and the clause  $x'_j \rightarrow y_j$  propagates  $y_k$ . Similarly, if  $j = 0$  then  $i = k$  and the clause  $x_i \rightarrow y_i$  propagates  $y_k$ . Finally, if  $i \geq 1$  and  $j \geq 1$  the clause  $x_i \wedge x'_j \rightarrow y_{i+j}$  propagates  $y_k$ .

<sup>4</sup> Direct merge networks are similar to the totalizers of [7].

## 6 Combining Recursive and Direct Cardinality Networks

The recursive approach produces shorter networks than the direct approach when the input is middle-sized. Still, the recursive method for building a network needs to inductively produce networks for smaller and smaller input sizes. At some point, the networks we need have a sufficiently small number of inputs such that the direct method can build them using fewer clauses and variables than the recursive approach. Here a *mixed encoding* is presented: large cardinality networks are built with the recursive approach but their components are produced with the direct approach if their size is small enough.

In more detail, assume a merge of input sizes  $a$  and  $b$  is needed. We can use the direct approach, which needs  $V_D = a + b$  auxiliary variables and  $C_D = ab + a + b$  clauses; or we could use the recursive approach. With the recursive approach, we have to build two merge networks of sizes  $(\lceil \frac{a}{2} \rceil, \lceil \frac{b}{2} \rceil)$  and  $(\lfloor \frac{a}{2} \rfloor, \lfloor \frac{b}{2} \rfloor)$ . These networks are also built with this mixed approach. Then, we compute the clauses and variables needed in the recursive approach,  $V_R$  and  $C_R$ , with the formula of Section 4.1:  $V_R = V_1 + V_2 + 2 \lfloor \frac{a+b-1}{2} \rfloor$ ,  $C_R = C_1 + C_2 + 3 \lfloor \frac{a+b-1}{2} \rfloor$ , where  $(V_1, C_1)$  and  $(V_2, C_2)$  are, respectively, the number of variables and clauses needed in the recursive merge networks.

Finally, we compare the values of  $V_R$ ,  $V_D$ ,  $C_R$  and  $C_D$ , and decide which method is better for building the merge network. Notice that we cannot minimize both the number of variables and clauses; therefore, here we try to minimize the function  $\lambda \cdot V + C$ , for some fixed value  $\lambda > 0$ .<sup>5</sup> The parameter  $\lambda$  allows us to adjust the relative importance of the number of variables with respect to the number of clauses of the encoding. Notice that this algorithm for building merge networks (and similarly, sorting, simplified merge and cardinality networks) can easily be implemented with dynamic programming. See Section 7 for an experimental evaluation of the numbers of variables and clauses in cardinality networks built with this mixed approach.

The arc-consistency of the mixed encoding easily follows from the arc-consistency of the two encodings it is based on.

**Theorem 3.** *The Mixed Cardinality Network encoding is arc-consistent.*

*Proof (sketch).* The proof uses lemmas analogous to Lemmas 1, 2, 3 and 4. In turn, these lemmas are proved by combining the proofs outlined in Theorems 1 and 2.

## 7 Experimental Evaluation

In previous work [3], it was shown that power-of-two (Recursive) Cardinality Networks have overall better performance than other well-known methods such as Sorting Networks [10], Adders [10] and the BDD-based encoding of [6]. In what follows we will show that the generalization of Cardinality Networks to arbitrary size and their combination with Direct Encodings, yielding what we have called the **Mixed** approach, makes them significantly better, both in the size of the encoding and the SAT solver runtime.

We start the evaluation focusing on the size of the resulting encoding. In Figure 4 we show a representative graph, which indicates the size, in terms of variables and clauses, of the encoding of a cardinality network with input size 100 and varying output size  $m$ .

<sup>5</sup> This function can be replaced by any other monotone function that can be efficiently evaluated.

It can be seen that, since we minimize the function  $\lambda \cdot V + C$ , where  $V$  is the number of variables and  $C$  the number of clauses, the bigger  $\lambda$  is, the fewer variables we obtain, at the expense of a slight increase in the number of clauses. Also, it can be seen that using power-of-two Cardinality Networks as in [3] is particularly harmful when  $m$  is slightly larger than a power of two.

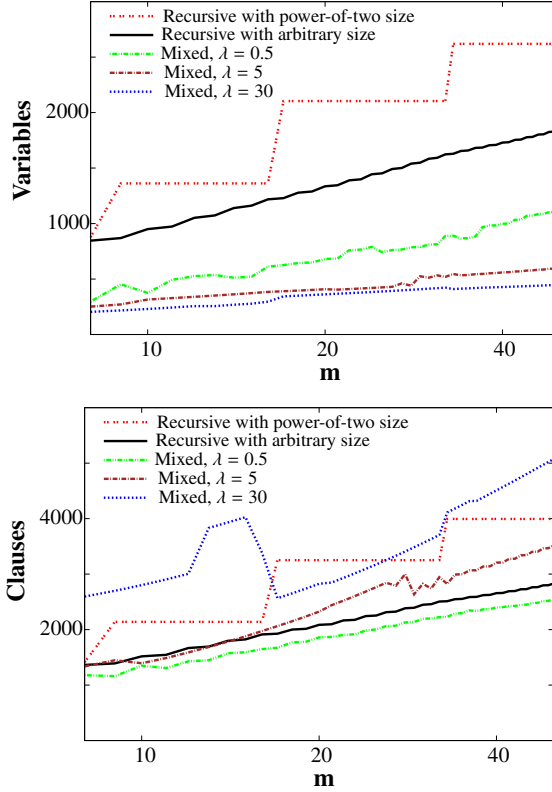


**Fig. 4.** Number of variables and clauses generated by **Mixed** and the Recursive Cardinality Networks approaches with input size 100 and different output sizes  $m$

Although having a smaller encoding is beneficial, this should be accompanied with a reduction in SAT solver runtime. Hence, let us now move to assess how our new encoding affects the performance of SAT solvers. In this evaluation, in addition to considering the power-of-two Recursive Cardinality Networks in [3] (**Power-of-two CN**), the (arbitrary-size) Recursive Cardinality Networks presented in Section 4 (**Arbitrary-sized CN**) and the **Mixed** approach of Section 6, we have also included other well-known encodings in the literature: the adder-based encoding (**Adder**) of [10] and the BDD-based encoding (**BDD**) of [6]. We believe these encodings are representative of all different approaches that have been used to deal with cardinality constraints. Other works, like the adder-based encoding of [20], the BDD-based one of [10] or the work by

Anbulagan and Grastien [1], are small variations or combinations of the encodings we have chosen. Moreover, we have implemented an SMT-based approach (**SMT**) to Cardinality Constraints. In a nutshell, we have coupled a SAT solver with a theory solver that handles all cardinality constraints. As soon as a cardinality constraint is violated by the current partial assignment, the SAT solver is forced to backtrack and, when the value of a variable can be propagated thanks to a cardinality constraint, this information is passed to the SAT solver. In other words, cardinality constraints are not translated into SAT, but rather tackled by a dedicated algorithm, similar in nature to what some pseudo-Boolean solvers do. See [14] for more information about SMT.

The SAT solver we have used in this evaluation is Lingeling version *ala*, a state-of-the-art CDCL (Conflict-Driven Clause Learning) SAT solver that implements several in/preprocessing techniques. All experiments were conducted on a 2Ghz Linux Quad-Core AMD with the three following sets of benchmarks:

**1.-MSU4 Suite.** These benchmarks are intermediate problems generated by an implementation of the *msu4* algorithm [12], which reduces a Max-SAT problem to a series of SAT problems with cardinality constraints. The *msu4* implementation was run of a variety of problems (filter design, logic synthesis, minimum-size test pattern generation, haplotype inference and maximum-quartet consistency) from the Partial Max-SAT division of the Third Max-SAT evaluation<sup>6</sup>. The suite consists of about 14000 benchmarks, each of which contains multiple  $\leq$ -cardinality constraints.

**2.-Discrete-Event System Diagnosis Suite.** The second set of benchmarks we have used is the one introduced in [1]. These problems come from discrete-event system (DES) diagnosis. As it happened with the Max-SAT problems, a single DES problem produced a family of “SAT + cardinality constraints” problems. This way, out of the roughly 600 DES problems, we obtained a set of around 6000 benchmarks, each of which contained a single very large  $\leq$ -cardinality constraint.

**3.-Tomography Suite.** The last set of benchmarks we have used is the one introduced in [5]. The idea is to first generate an  $N \times N$  grid in which some cells are filled and some others are not. The problem consists in finding out which are the filled cells using only the information of how many filled cells there are in each row, column and diagonal. For that purpose, variables  $x_{ij}$  are used to indicate whether cell  $(i, j)$  is filled and several  $=$ -cardinality constraints impose how many filled cells there are in each row, column and diagonal. We generated 2600 benchmarks (100 instances for each size  $N = 15 \dots 40$ ).

Results are summarized in Table 1, which compares the **Mixed** (with  $\lambda = 5$ ) encoding with the aforementioned encodings. The time limit was set to 600 seconds per benchmark and we only considered benchmarks for which at least one of the methods took more than 5 seconds. There are three tables, one for each benchmark suite. In each table, columns indicate in how many benchmarks the **Mixed** encoding exhibits the corresponding speed-up or slow-down factor with respect to the method indicated in each row. For example, in the table for the **MSU4 suite**, the first row indicates that in 43 benchmarks, Power-of-two Cardinality Networks timed out (**TO**) whereas our new encoding did not. The columns next to it indicate that in 732 benchmarks the novel encoding was at least 4 times faster, in 2957 between 2 and 4 times faster, etc.

<sup>6</sup> See <http://www.maxsat.udl.cat/08/index.php?disp=submitted-benchmarks>

**Table 1.** Comparison of SAT solver runtime. Figures show number of benchmarks in which **Mixed** shows the corresponding speed-up/slow-down factor w.r.t. other methods.

|                           | Speed-up factor of Mixed |      |      |      |      | Slow-down factor of Mixed |      |     |     |      |
|---------------------------|--------------------------|------|------|------|------|---------------------------|------|-----|-----|------|
|                           | TO                       | 4    | 2    | 1.5  | TOT. | 1.5                       | 2    | 4   | TO  | TOT. |
| <b>MSU4 suite</b>         |                          |      |      |      |      |                           |      |     |     |      |
| <b>Power-of-two CN</b>    | 43                       | 732  | 2957 | 1278 | 5010 | 1                         | 23   | 13  | 11  | 48   |
| <b>Arbitrary-sized CN</b> | 10                       | 149  | 544  | 726  | 1429 | 3                         | 106  | 43  | 80  | 232  |
| <b>Adder</b>              | 985                      | 1207 | 1038 | 1250 | 4480 | 0                         | 13   | 36  | 40  | 89   |
| <b>BDD</b>                | 187                      | 1139 | 1795 | 1292 | 4413 | 4                         | 10   | 31  | 36  | 81   |
| <b>SMT</b>                | 1143                     | 323  | 102  | 53   | 1621 | 0                         | 1417 | 211 | 63  | 1691 |
| <b>DES suite</b>          |                          |      |      |      |      |                           |      |     |     |      |
| <b>Power-of-two CN</b>    | 13                       | 21   | 265  | 638  | 937  | 6                         | 12   | 7   | 46  | 71   |
| <b>Arbitrary-sized CN</b> | 19                       | 21   | 75   | 404  | 519  | 5                         | 12   | 11  | 45  | 73   |
| <b>Adder</b>              | 218                      | 235  | 611  | 1283 | 2347 | 0                         | 5    | 3   | 42  | 50   |
| <b>BDD</b>                | 705                      | 3944 | 759  | 51   | 5459 | 0                         | 0    | 0   | 0   | 0    |
| <b>SMT</b>                | 3003                     | 1134 | 262  | 73   | 4472 | 0                         | 15   | 19  | 15  | 49   |
| <b>Tomography suite</b>   |                          |      |      |      |      |                           |      |     |     |      |
| <b>Power-of-two CN</b>    | 118                      | 388  | 408  | 175  | 1089 | 64                        | 82   | 159 | 121 | 426  |
| <b>Arbitrary-sized CN</b> | 104                      | 430  | 432  | 169  | 1135 | 67                        | 81   | 158 | 11  | 417  |
| <b>Adder</b>              | 492                      | 591  | 371  | 143  | 1597 | 14                        | 20   | 39  | 35  | 108  |
| <b>BDD</b>                | 0                        | 0    | 0    | 0    | 0    | 112                       | 1367 | 184 | 51  | 1714 |
| <b>SMT</b>                | 0                        | 10   | 25   | 11   | 46   | 112                       | 1250 | 155 | 68  | 1585 |

We can see from the table that in the **MSU4** and **DES** suites, which contain benchmarks coming from real-world applications, our new encoding in general outperforms the other methods (except for some instances in which **Mixed** times out and the other cardinality network-based encodings do not; also, in **MSU4**, **SMT** and **Mixed** obtain comparable results). We want to remark that the gain comes both from using arbitrary-sized networks as well as from combining them with direct encodings, as can be seen from the second row of each table. In particular, this shows the negative impact of the dummy variables of [3], which hinder the performance in spite of the unit propagation of the SAT solver. Finally, in the **Tomography** suite, the **BDD**-based encoding and the **SMT** system outperform all other methods, but among the rest of the approaches the **Mixed** encoding exhibits the best performance. Altogether, the **Mixed** encoding is the most robust technique according to the results of this evaluation.

## 8 Conclusion and Future Work

The contributions of this paper are: (i) an extension of the recursive cardinality networks of [3] to arbitrary input and output sizes; (ii) a non-recursive construction of cardinality networks that is competitive for small sizes; (iii) a parametric combination of these two approaches for producing cardinality networks that not only improves on the size of the encoding, but also yields significant speedups in SAT solver performance.

As regards future work, we plan to develop encoding techniques for cardinality constraints that do not process constraints one-at-a-time but simultaneously, in order to exploit their similarities. We foresee that the flexibility of the approach presented here with respect to the original construction in [3], will open the door to sharing the internal networks among the cardinality constraints present in a SAT problem.



**Acknowledgments.** Abío is supported by DFG Graduiertenkolleg 1763 (QuantLA). All other authors are partially supported by Spanish MEC/MICINN under SweetLogics project (TIN 2010-21062-C02-01). We also thank the reviewers for their comments.

## References

1. Anbulagan, A.G.: Importance of Variables Semantic in CNF Encoding of Cardinality Constraints. In: Bulitko, V., Beck, J.C. (eds.) Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA 2009. AAAI (2009)
2. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks and their applications. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 167–180. Springer, Heidelberg (2009)
3. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality Networks: a theoretical and empirical study. *Constraints* 16(2), 195–221 (2011)
4. Achá, R.A., Nieuwenhuis, R.: Curriculum-based course timetabling with SAT and MaxSAT. *Annals of Operations Research*, 1–21 (February 2012)
5. Bailleux, O., Boufkhad, Y.: Efficient CNF Encoding of Boolean Cardinality Constraints. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 108–122. Springer, Heidelberg (2003)
6. Bailleux, O., Boufkhad, Y., Roussel, O.: A translation of pseudo boolean constraints to sat. *JSAT* 2(1-4), 191–200 (2006)
7. 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)
8. Batchner, K.E.: Sorting Networks and their Applications. In: AFIPS Spring Joint Computing Conference, pp. 307–314 (1968)
9. Codish, M., Zazon-Ivry, M.: Pairwise cardinality networks. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 154–172. Springer, Heidelberg (2010)
10. Eén, N., Sörensson, N.: Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation* 2, 1–26 (2006)
11. Fu, Z., Malik, S.: Solving the minimum-cost satisfiability problem using SAT based branch-and-bound search. In: Proceedings of the 2006 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2006, pp. 852–859. ACM, New York (2006)
12. Marques-Silva, J., Planes, J.: Algorithms for Maximum Satisfiability using Unsatisfiable Cores. In: 2008 Conference on Design, Automation and Test in Europe Conference, DATE 2008, pp. 408–413. IEEE Computer Society (2008)
13. Metodi, A., Codish, M., Stuckey, P.J.: Boolean equi-propagation for concise and efficient sat encodings of combinatorial problems. *J. Artif. Intell. Res., JAIR* 46, 303–341 (2013)
14. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM, JACM* 53(6), 937–977 (2006)
15. Parberry, I.: The pairwise sorting network. *Parallel Processing Letters* 2, 205–211 (1992)
16. David, A.: Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.* 2(3), 293–304 (1986)
17. 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)

18. Sinz, C.: Towards an optimal CNF encoding of boolean cardinality constraints. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005)
19. Büttner, M., Rintanen, J.: Satisfiability planning with constraints on the number of actions. In: Biundo, S., Myers, K.L., Rajan, K. (eds.) 15th International Conference on Automated Planning and Scheduling, ICAPS 2005, pp. 292–299. AAAI (2005)
20. Warners, J.P.: A Linear-Time Transformation of Linear Inequalities into Conjunctive Normal Form. *Information Processing Letters* 68(2), 63–69 (1998)

# To Encode or to Propagate? The Best Choice for Each Constraint in SAT

Ignasi Abío<sup>1</sup>, Robert Nieuwenhuis<sup>2</sup>,  
Albert Oliveras<sup>2</sup>, Enric Rodríguez-Carbonell<sup>2</sup>, and Peter J. Stuckey<sup>3</sup>

<sup>1</sup> Theoretical Computer Science, TU Dresden, Germany

<sup>2</sup> Technical University of Catalonia, Barcelona

<sup>3</sup> National ICT Australia and the University of Melbourne

**Abstract.** Sophisticated compact SAT encodings exist for many types of constraints. Alternatively, for instances with many (or large) constraints, the SAT solver can also be extended with built-in propagators (the SAT Modulo Theories approach, SMT). For example, given a cardinality constraint  $x_1 + \dots + x_n \leq k$ , as soon as  $k$  variables become true, such a propagator can set the remaining variables to false, generating a so-called explanation clause of the form  $x_1 \wedge \dots \wedge x_k \rightarrow \overline{x_i}$ . But certain “bottle-neck” constraints end up generating an exponential number of explanations, equivalent to a naive SAT encoding, much worse than using a compact encoding with auxiliary variables from the beginning. Therefore, Abío and Stuckey proposed starting off with a full SMT approach and *partially* encoding, on the fly, only certain “active” parts of constraints. Here we build upon their work. Equipping our solvers with some additional bookkeeping to monitor constraint activity has allowed us to shed light on the effectiveness of SMT: many constraints generate very few, or few different, explanations. We also give strong experimental evidence showing that it is typically unnecessary to consider partial encodings: it is competitive to encode the few really active constraints entirely. This makes the approach amenable to any kind of constraint, not just the ones for which *partial* encodings are known.

## 1 Introduction

The “SAT revolution” [Var09] has made SAT solvers a very appealing tool for solving constraint satisfaction and optimization problems. Apart from their efficiency, SAT tools are push-button technology, with a single fully automatic variable selection heuristic. For many types of constraints, sophisticated compact SAT encodings exist. Such encodings usually introduce auxiliary variables, which allows one to obtain succinct formulations. Auxiliary variables frequently also have a positive impact on the size and reusability of the *learned* clauses (lemmas), and, in combination with the possibility of deciding (splitting) on them, on the quality of the search.

**Building in Constraints: SAT Modulo Theories (SMT).** On problem instances with many (or very large) constraints, where encodings lead to huge numbers of clauses and variables, it may be preferable to follow an alternative approach: in SMT [NOT06,BHvMW09], the SAT solver is extended with a built-in

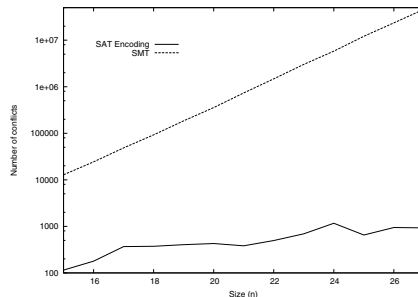
propagator for each constraint, making it amenable to sophisticated constraint-specific reasoning (as in Constraint Programming). For example, given a cardinality constraint  $x_1 + \dots + x_n \leq k$ , as soon as  $k$  of its variables become true, such a propagator can set any other variable  $x_j$  to false. If at some later point this propagated literal  $\overline{x_j}$  takes part in a conflict, a so-called *explanation* clause of the form  $x_{i_1} \wedge \dots \wedge x_{i_k} \rightarrow \overline{x_j}$  is used, thus fully integrating such propagators in the SAT solver's conflict analysis and backjumping mechanisms. As usual in SMT, here we consider that such explanations are (i) only produced when needed during conflict analysis and (ii) are not learned (only the resulting lemma is).

**The Remarkable Effectiveness of SMT.** SMT is remarkably effective. The intuitive reason is that, while searching for a solution for a given problem instance, some constraints only block the current solution candidate very few times, and moreover they do this almost always in the same way. In this paper we shed some more light on this intuitive idea. We perform experiments with a number of notions of *constraint activity* in this sense, that is, the (recent or total) number of (different or all) explanations that each constraint generates. Indeed, as we will see: A) many constraints generate very few, or few different, explanations, and B) generating only these explanations can be much more effective than dealing with a full encoding of the constraint.

**The Dark Side of SMT.** Frequently, there are also certain “bottle-neck” constraints that end up generating an exponential number of explanations, equivalent to a naive SAT encoding. A theoretical but illustrative example is:

**Lemma 1.** *An SMT solver will generate an exponential number of explanations when proving the unsatisfiability of the input problem consisting of only the two cardinality constraints  $x_1 + \dots + x_n \leq n/2$  and  $x_1 + \dots + x_n > n/2$ .*

This lemma holds because any SMT solver, when proving unsatisfiability, generates a propositionally unsatisfiable set of clauses (the input ones plus the lemmas), and if a single one of the all  $\binom{n}{k+1} + \binom{n}{n-k}$  explanations (where  $k = n/2$ ) has not been generated, say, the explanation  $\overline{x_1} \vee \dots \vee \overline{x_{k+1}}$ , then the assignment that sets  $x_1, \dots, x_{k+1}$  to true and the remaining  $n - k - 1$  variables to false is a model. Such situations indeed happen in practice: for some constraints SMT ends up generating a full or close to full encoding, which is moreover a very naive exponential one, with no auxiliary variables. If a polynomial-size encoding for such a constraint exists (possibly with auxiliary variables), using it right from the beginning is a much better alternative. This is shown in the following figure:



It gives the number of conflicts needed to prove unsatisfiability of the previous example, varying  $n$ , with our Barcellogic solver in SMT mode and with a SAT encoding based on Cardinality Networks. SMT exhibits exponential behavior (note the logarithmic scale). The encoding-based version scales up much better; in fact, a polynomial-size refutation for it exists, although it is not clear from the figure whether the solver always finds it or not.

**Getting the Best of Both.** In their *conflict-directed lazy decomposition (LD)* approach [AS12], Abío and Stuckey proposed starting off the solver using an SMT approach for all constraints of the problem instance, and *partially* encoding (or *decomposing*), on the fly, only the “active” *parts of* some constraints. The decision of when and which auxiliary variables to introduce during the solving process is taken with a particular concrete encoding in mind: if, according to the explanations that are being generated, it is observed that an auxiliary variable of the encoding and its corresponding part of the encoding would have been “active”, then it is added to the formula, together with all of the involved clauses of the encoding. In this way, fully active constraints end up being completely encoded using the compact encoding with auxiliary variables, and less active ones are handled by SMT. In [AS12] it is shown that this can be done for the Cardinality/Sorting Network encoding of cardinality constraints, and, although in a complicated way, for BDD-encodings of pseudo-Boolean constraints, performing essentially always at least as well as the best of SMT and encoding.

**Going Beyond.** A shortcoming of [AS12] is that it is highly dependent on the constraint to be dealt with and the chosen encoding, making it unlikely to be applicable to other more complex constraints, and in any case equipping the theory solver with the required features is a highly non-trivial task. Here we propose another technique that is much simpler. It does not depend on the concrete constraint under consideration and can in fact be applied to any class of constraints that can be either encoded or built in. As mentioned previously, we have devised and analyzed bookkeeping methods for different notions of constraint activity that are cheap enough not to slow down solving appreciably. As a result, here we show, giving strong experimental evidence, that it is typically unnecessary to consider partial encodings: the few really active constraints can usually be encoded –on the fly– entirely. This makes the approach amenable to any kind of constraint, not just the ones for which *partial* encodings are known. Results on problems containing cardinality and pseudo-Boolean constraints are comparable, and frequently outperform all three of its competitors: SMT, encoding, and the partial lazy decomposition method of [AS12].

## 2 SAT and SAT Encoding

Let  $\mathcal{X} = \{x_1, x_2, \dots, x_n\}$  be a finite set of propositional *variables*. If  $x \in \mathcal{X}$  then  $x$  and  $\bar{x}$  are *literals*. The *negation* of a literal  $l$ , written  $\bar{l}$ , denotes  $\bar{x}$  if  $l$  is  $x$ , and  $x$  if  $l$  is  $\bar{x}$ . A *clause* is a disjunction of literals  $l_1 \vee \dots \vee l_n$ . A (*CNF*) *formula* is a conjunction of clauses.

An *assignment*  $A$  is a set of literals such that  $\{x, \bar{x}\} \subseteq A$  for no  $x$ . A literal  $l$  is *true* in  $A$  if  $l \in A$ , is *false* if  $\bar{l} \in A$ , and is *undefined* otherwise. A clause  $C$  is true in  $A$  if at least one of its literals is true in  $A$ . A formula  $F$  is true in  $A$  if all its clauses are true in  $A$ , and then  $A$  is a *model* of  $F$ . The *satisfiability (SAT) problem* consists in, given a formula  $F$ , to decide if it has a model. Systems that decide the SAT problem are called *SAT solvers*.

A function  $C : \{0, 1\}^n \rightarrow \{0, 1\}$  is called a *constraint*. Given a constraint  $C$ , a *(SAT) encoding* for it is a formula  $F$  (possibly with auxiliary variables) that is equisatisfiable. An important class of constraints are *cardinality constraints*, which state that at most (or at least, or exactly)  $k$  out of  $n$  variables can be true. Common encodings for it are based on networks of adders [BB03,uR05,AG09,Sin05], or Sorting Networks [ES06,CZI10,ANORC09,ANORC11]. Cardinality constraints are generalized by *pseudo-Boolean constraints*, of the form  $a_1x_1 + \dots + a_nx_n \# k$ , where the  $a_i$  and  $k$  are integer coefficients, and  $\#$  belongs to  $\{\leq, \geq, =\}$ . Again, several encodings exist, based on performing arithmetic [War98,BBR09,ES06] or computing BDD's [BBR06,ES06,ANO<sup>+</sup>12]. Most convenient encodings are the ones for which the SAT solver's unit propagation mechanism preserves domain-consistency.

### 3 To Encode or Not to Encode?

In this section we will discuss situations where encoding a constraint is better than using a propagator for it or vice versa, and how to detect them. The reasoning will consist of both theoretical insights and experimental evaluation. For the latter, 5 benchmarks suites will be used, in which all benchmarks solvable in less than 5 seconds by both methods have been removed.

**1.-MSU4:** 5729 problems generated in the execution of the *msu4* algorithm [MSP08] for Max-SAT. Each benchmark contains very few  $\leq$ -cardinality constraints.

**2.-Discrete-Event System Diagnosis:** 4526 discrete-event system (DES) diagnosis [AG09] problems. Each benchmark contains a single very large  $\leq$ -cardinality constraint.

**3.-Tomography:** 2021 tomography problems introduced in [BB03]. Each problem contains many  $=$ -cardinality constraints.

**4.-PB Evaluation:** 669 benchmarks from the pseudo-Boolean Competition<sup>1</sup> 2011 (category DEC-SMALLINT-LIN), with multiple cardinality and pseudo-Boolean constraints.

**5.-RCPSP:** 577 benchmarks coming from the PSP-Lib<sup>2</sup>. These are scheduling problems with a fixed makespan. Several pseudo-Boolean constraints are present.

To start with, let us experimentally confirm that SMT and encoding-based methods are complementary, and so a hybrid method getting the best of both is worth pursuing. For this purpose, we implemented an SMT-based system

<sup>1</sup> <http://www.cril.univ-artois.fr/PB11/>

<sup>2</sup> <http://webserver.wi.tum.de/psplib>

and encodings into SAT. For cardinality constraints, we used the Cardinality Networks encoding of [ANORC11], whereas for pseudo-Boolean constraints, the BDD-based one of [ANO<sup>+</sup>12]. The reason for this choice is that, according to the experimental results of [ANORC11, ANO<sup>+</sup>12], these two encodings are the globally most robust ones in practice. However, any other choice would have been possible, since the approach we will present is encoding-independent. A time limit of 600 seconds was set per benchmark and, in order to have a fair comparison, in both systems the same underlying SAT solver was used (Barcelogic). Results can be seen in Table 1, where one can observe that the encoding performs very well in the MSU4 and DES suite, and is significantly worse in the other three.<sup>3</sup>

**Table 1.** Comparison between encoding and SMT. Table on the left indicate the percentage of benchmarks where each method outperforms (is at least 1.5 times faster than) the other. On the right, the geometric mean (in seconds) of the instances solves by both methods.

| Benchmark suite | Encoding $\geq 1.5\times$ faster | SMT $\geq 1.5\times$ faster | Geometric mean |       |
|-----------------|----------------------------------|-----------------------------|----------------|-------|
|                 |                                  |                             | Encoding       | SMT   |
| MSU4            | 39.37%                           | 15.39%                      | 1.71           | 23.53 |
| DES             | 92.06%                           | 0.28%                       | 2.3            | 56.02 |
| Tomography      | 5.93%                            | 86.49%                      | 46.95          | 4.37  |
| PB evaluation   | 7.02%                            | 43.49%                      | 25.53          | 3.79  |
| RCPSP           | 0.69%                            | 46.62%                      | 106.65         | 5.8   |

Lemma 1 explains why SMT is worse in some suites, but not why it is better in some others. The latter happens on benchmarks with many constraints. A possible explanation could be that many of these constraints are not very active, i.e. they produce very few, if any, explanations. If this is the case, SMT has an advantage over an encoding: only active constraints will generate explanations, whereas an encoding approach would also have to encode all low-activity constraints right from the beginning. This notion of constraint activity, counting the number of times the propagator generates an explanation, is very similar to earlier activity-based lemma deletion policies in SAT solvers [GN02]. In order to evaluate how often this situation happens, we ran our SMT system computing the number of explanations each constraint generates. Results can be seen in Table 2, where we considered a constraint to have low activity if it generates less than 100 (possibly repeated) explanations.

Each row contains the data for each suite: e.g., in 74.6% of the MSU4 benchmarks between 0 and 5% of the constraints had low activity. In the PB evaluation and in the RCPSP benchmarks, the number of low-activity constraints is high and hence, this might explain why SMT behaves better than the encoding on these suites. However, in the Tomography suite, constraints tend to be very active, which refutes our conjecture of why SMT performs so well on these benchmarks.

<sup>3</sup> Note that rows do not add up to 100 % as benchmarks in which the two methods are comparable are not shown.

**Table 2.** Number of low-activity constraints in distinct benchmarks suites

| Suite         | Perc. of benches with this perc. of low-act. constr. |       |        |        |        |        |        |         |
|---------------|------------------------------------------------------|-------|--------|--------|--------|--------|--------|---------|
|               | 0-5%                                                 | 5-10% | 10-20% | 20-40% | 40-60% | 60-80% | 80-95% | 95-100% |
| MSU4          | 74.6                                                 | 0     | 0      | 0      | 24.9   | 0.5    | 0      | 0       |
| DES           | 99.9                                                 | 0     | 0      | 0      | 0      | 0      | 0      | 0.1     |
| Tomography    | 100                                                  | 0     | 0      | 0      | 0      | 0      | 0      | 0       |
| PB evaluation | 54                                                   | 21.6  | 20.5   | 0.6    | 1.1    | 0.6    | 1.7    | 20.5    |
| RCPSP         | 0                                                    | 0     | 2.2    | 13.2   | 51.1   | 31.3   | 2.2    | 0       |

What happens in the Tomography suite is that although constraints are very active, the SMT solver does not end up generating the whole naive encoding because many explanations are repeated. Hence, a sophisticated encoding would probably generate many more clauses, as the whole constraint would be decomposed, even irrelevant parts.

To confirm this hypothesis we ran our SMT solver counting repeated explanations. Results can be seen in Table 3. Each row<sup>4</sup> corresponds to a different suite: e.g., the 100 in the third row indicates that all benchmarks in the Tomography suite had at least half of its constraints producing between 80 and 95% of repeated explanations. In general, if a constraint produces many repeated explanations, it is unlikely that it might end up generating its whole naive encoding. This explains why SMT has good results in this suite, as well as in PB evaluation and RCPSP. Hence, the number of repeated explanations seems to be a robust indicator of whether we should encode a constraint or use a propagator.

**Table 3.** The percentage of benchmark instances where more than half the constraints have a given percentage of repeated explanations

| Suite         | Benches with >50% of the ctrs. w./ this perc. of rep. expl. |       |        |        |        |        |        |         |
|---------------|-------------------------------------------------------------|-------|--------|--------|--------|--------|--------|---------|
|               | 0-5%                                                        | 5-10% | 10-20% | 20-40% | 40-60% | 60-80% | 80-95% | 95-100% |
| MSU4          | 53.8                                                        | 9.1   | 11.6   | 8.5    | 2      | 0.8    | 0.2    | 0       |
| DES           | 21.4                                                        | 29.8  | 35.2   | 13.6   | 0      | 0      | 0      | 0       |
| Tomography    | 0                                                           | 0     | 0      | 0      | 0      | 0      | 100    | 0       |
| PB evaluation | 6.2                                                         | 0     | 0      | 0      | 0      | 0.6    | 14.2   | 51.7    |
| RCPSP         | 0                                                           | 0     | 0      | 0      | 0      | 5.5    | 52.7   | 1.1     |

## 4 Implementation and Experimental Evaluation

Taking into account Section 3, we implemented a system that processes SAT problems augmented with cardinality and pseudo-Boolean constraints. Although our approach is easily applicable much more generally, here we focus on these two types of constraints in order to be able to compare with [AS12]. Our aim is to show that a very simple approach gets the best of SMT and encoding methods. The starting point for our implementation is an SMT solver equipped with the ability of encoding cardinality constraints via Cardinality Networks and pseudo-Boolean constraints via BDDs.

In order to know which constraints to encode we need to keep track of the percentage of different explanations that the constraints generate. To do this we

<sup>4</sup> Note that the percentages in each row do not need to add 100.



attach to each constraint all the different explanations it produces. When an explanation is generated, we traverse the list of previous explanations, checking if it already exists. To speed up comparison, we first compare the size and only if they are equal, we compare the explanations, which are sorted to make comparison faster. This would be very expensive if constraints with many different explanations existed, but those constraints end up being encoded and after that do not cause any further bookkeeping overhead. Hence, more complex data structures would not help here. In our implementation, we only collect information during the first 2 minutes, since, according to our experiments, after that the information stabilizes.

Another important source of information to consider is how large the ad-hoc encoding of each constraint would be. If the number of generated explanations becomes close to the number of clauses the encoding requires, according to our experiments then it is advantageous to encode the constraint. Besides, if a constraint is producing many different explanations, we found that it is likely to end up generating the full (or a large part of the) naive encoding. Discovering and avoiding this situation is highly beneficial.

We also experimented with different ways of counting the number of *recent* occurrences of a given explanation in conflicts, without any significant findings.

Finally, following all previous observations, we encode a constraint if at least one of two conditions holds: (i) the number of different explanations is more than half the number of clauses of the compact, sophisticated encoding, (ii) more than 70% of the explanations are new and more than 5000 explanations have already been generated.

We compared the resulting system (**New** in the tables) with an SMT system, another one which encodes all constraints from the start (**Enc.**) and Lazy Decomposition [AS12] (**LD**). Results can be seen in Table 4. Each cell contains the

**Table 4.** Comparison among different methods on all benchmarks suites

| MSU4 |             |             |             |             |             |             |
|------|-------------|-------------|-------------|-------------|-------------|-------------|
|      | <10s        | <30s        | <60s        | <120s       | <300s       | <600s       |
| Enc. | <b>5374</b> | <b>5525</b> | 5578        | 5621        | 5659        | 5677        |
| SMT  | 4322        | 4530        | 4603        | 4667        | 4737        | 4767        |
| LD   | 5196        | 5414        | 5528        | 5598        | 5655        | 5674        |
| New  | 5222        | 5479        | <b>5585</b> | <b>5636</b> | <b>5666</b> | <b>5679</b> |

| DES  |             |             |             |             |             |             |
|------|-------------|-------------|-------------|-------------|-------------|-------------|
|      | <10s        | <30s        | <60s        | <120s       | <300s       | <600s       |
| Enc. | <b>2521</b> | <b>3333</b> | <b>3692</b> | <b>3903</b> | <b>4102</b> | <b>4228</b> |
| SMT  | 362         | 654         | 850         | 1023        | 1256        | 1452        |
| LD   | 570         | 1230        | 1761        | 2525        | 3558        | 4019        |
| New  | 836         | 2156        | 3293        | 3800        | 4053        | 4166        |

| Tomography |             |             |             |             |             |             |
|------------|-------------|-------------|-------------|-------------|-------------|-------------|
|            | <10s        | <30s        | <60s        | <120s       | <300s       | <600s       |
| Enc.       | 773         | 1112        | 1314        | 1501        | 1759        | 1932        |
| SMT        | 1457        | 1748        | 1858        | 1962        | <b>2014</b> | <b>2021</b> |
| LD         | 1027        | 1239        | 1399        | 1561        | 1763        | 1918        |
| New        | <b>1556</b> | <b>1818</b> | <b>1935</b> | <b>1971</b> | 2012        | <b>2021</b> |

| PB evaluation |            |            |            |            |            |            |
|---------------|------------|------------|------------|------------|------------|------------|
|               | <10s       | <30s       | <60s       | <120s      | <300s      | <600s      |
| Enc.          | 268        | 337        | 358        | 376        | 399        | 414        |
| SMT           | <b>364</b> | <b>377</b> | <b>386</b> | <b>392</b> | <b>409</b> | 414        |
| LD            | 352        | 371        | 379        | 388        | 403        | <b>416</b> |
| New           | 269        | 341        | 360        | 381        | 404        | 415        |

| RCPSP |            |            |            |            |            |            |
|-------|------------|------------|------------|------------|------------|------------|
|       | <10s       | <30s       | <60s       | <120s      | <300s      | <600s      |
| Enc.  | 7          | 22         | 52         | 91         | 139        | 175        |
| SMT   | <b>132</b> | <b>179</b> | <b>206</b> | 224        | <b>249</b> | <b>272</b> |
| LD    | 114        | 160        | 178        | 189        | 216        | 228        |
| New   | 111        | 169        | 202        | <b>225</b> | <b>249</b> | 271        |

number of problems that could be solved in less than the number of seconds of the corresponding column.

The first important conclusion is that we can obtain comparable, in some cases better, results than the LD method. This is worth mentioning since our approach is much simpler to implement and does not pose any requirement on the encodings to be used. Secondly, our approach always solves a very similar number of problems to the best option for each suite. Only in the DES suite, there is some difference that can be explained by the fact that SMT has extraordinarily poor performance on those benchmarks. Thus, just running the system in SMT mode for few seconds before encoding the constraints, as our new system does on these instances, has a strong negative impact because the many explanations generated in this early stage hinder the search later on. This could be mitigated by using more aggressive lemma deletion policies.

## 5 Conclusions and Other Related Work

This work is part of a project with the aim of deepening our understanding of what choices between SMT and encodings are optimal in practical problems. Here we have seen that the use of adaptive strategies is clearly advantageous. Moreover, we have given a simpler approach for which it becomes possible to handle many other types of constraints, as we plan to investigate next.

Another possibility for future work concerns the version of SMT in which explanation clauses are generated and learned immediately when a constraint propagates, as in the initial version of Lazy Clause Generation [OSC07], which worked remarkably well on resource-constrained project scheduling problems (RCPSPs). Indeed, we have now discovered that in these problems the number of different explanations is specially low. It may turn out to be advantageous to handle these constraints with clauses, which are prioritized in the solver with respect to constraint propagators.

**Related Work.** Apart from [AS12], another related proposal is [MP11]: to solve a propositional formula  $F$  plus additional pseudo-Boolean constraints, the SAT solver first finds a model  $M$  for  $F$  (“unsat” if there is none); then a few of the constraints that are false in  $M$  are picked (“sat” if there is none), simplified using the unit clauses found so far, encoded, and added to  $F$ ; and the process is iterated. A drawback of this method is that it may fully encode low-activity constraints just because they happen to be false in  $M$ , whereas we really monitor activity. Also, we only need one run of the solver. Finally, it is clear that any method that encodes on the fly (including ours) can simplify constraints with the unit clauses available at that point.

**Acknowledgments.** First four authors partially supported by MEC/MICINN under SweetLogics project (TIN 2010-21062-C02-01). Abío is also supported by DFG Graduiertenkolleg 1763 (QuantLA). NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

## References

- AG09. Anbulagan, Grastien, A.: Importance of Variables Semantic in CNF Encoding of Cardinality Constraints. In: Bulitko, V., Beck, J.C. (eds.) Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA 2009. AAAI (2009)
- ANO<sup>+</sup>12. 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)
- ANORC09. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality networks and their applications. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 167–180. Springer, Heidelberg (2009)
- ANORC11. Asín, R., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: Cardinality Networks: a theoretical and empirical study. *Constraints* 16(2), 195–221 (2011)
- AS12. Abío, I., Stuckey, P.J.: Conflict directed lazy decomposition. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 70–85. Springer, Heidelberg (2012)
- BB03. Bailleux, O., Boufkhad, Y.: Efficient CNF Encoding of Boolean Cardinality Constraints. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 108–122. Springer, Heidelberg (2003)
- BBR06. Bailleux, O., Boufkhad, Y., Roussel, O.: A translation of pseudo boolean constraints to sat. *JSAT* 2(1-4), 191–200 (2006)
- BBR09. 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)
- BHvMW09. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (February 2009)
- CZI10. Codish, M., Zazon-Ivry, M.: Pairwise cardinality networks. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 154–172. Springer, Heidelberg (2010)
- ES06. Eén, N., Sörensson, N.: Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation* 2, 1–26 (2006)
- GN02. Goldberg, E., Novikov, Y.: BerkMin: A Fast and Robust SAT-Solver. In: 2002 Conference on Design, Automation, and Test in Europe, DATE 2002, pp. 142–149. IEEE Computer Society (2002)
- MP11. Manolios, P., Papavasileiou, V.: Pseudo-Boolean solving by incremental translation to SAT. In: Formal Methods in Computer-Aided Design, FMCAD 2011 (2011)
- MSP08. Marques-Silva, J., Planes, J.: Algorithms for Maximum Satisfiability using Unsatisfiable Cores. In: 2008 Conference on Design, Automation and Test in Europe Conference, DATE 2008, pp. 408–413. IEEE Computer Society (2008)
- NOT06. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM, JACM* 53(6), 937–977 (2006)
- OSC07. 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)

- Sin05. Sinz, C.: Towards an optimal CNF encoding of boolean cardinality constraints. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 827–831. Springer, Heidelberg (2005)
- uR05. Büttner, M., Rintanen, J.: uttner and J. Rintanen. Satisfiability planning with constraints on the number of actions. In: Biundo, S., Myers, K.L., Rajan, K. (eds.) 15th International Conference on Automated Planning and Scheduling, ICAPS 2005, pp. 292–299. AAAI (2005)
- Var09. Vardi, M.Y.: Symbolic techniques in propositional satisfiability solving. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 2–3. Springer, Heidelberg (2009)
- War98. Warners, J.P.: A Linear-Time Transformation of Linear Inequalities into Conjunctive Normal Form. *Information Processing Letters* 68(2), 63–69 (1998)

# Automated Symmetry Breaking and Model Selection in CONJURE

Ozgur Akgun<sup>1</sup>, Alan M. Frisch<sup>2</sup>, Ian P. Gent<sup>1</sup>, Bilal Syed Hussain<sup>1</sup>,  
Christopher Jefferson<sup>1</sup>, Lars Kotthoff<sup>3</sup>, Ian Miguel<sup>1</sup>, and Peter Nightingale<sup>1</sup>

<sup>1</sup> University of St. Andrews

<sup>2</sup> University of York

<sup>3</sup> Cork Constraint Computation Centre

**Abstract.** Constraint modelling is widely recognised as a key bottleneck in applying constraint solving to a problem of interest. The CONJURE automated constraint modelling system addresses this problem by automatically refining constraint models from problem specifications written in the ESSENCE language. ESSENCE provides familiar mathematical concepts like sets, functions and relations nested to any depth. To date, CONJURE has been able to produce a set of alternative model kernels (i.e. without advanced features such as symmetry breaking or implied constraints) for a given specification. The first contribution of this paper is a method by which CONJURE can break symmetry in a model as it is introduced by the modelling process. This works at the problem class level, rather than just individual instances, and does not require an expensive detection step after the model has been formulated. This allows CONJURE to produce a higher quality set of models. A further limitation of CONJURE has been the lack of a mechanism to select among the models it produces. The second contribution of this paper is to present two such mechanisms, allowing effective models to be chosen automatically.

## 1 Introduction and Background

For constraint programming to achieve its potential widespread industrial and academic use, reducing the *modelling bottleneck* [29] is of central importance. This is the problem of formulating a problem of interest as a constraint model suitable for input to a constraint solver. There are typically many possible models for a given problem, and the model chosen can dramatically affect the efficiency of constraint solving. This presents a serious obstacle for non-expert users, who have difficulty in formulating a good (or even correct) model from among the many possible alternatives. Therefore, automating constraint modelling is a desirable goal. Numerous approaches have been taken to automate aspects of constraint modelling, including: case-based reasoning [23]; theorem proving [6]; automated transformation of medium-level solver-independent constraint models [27, 28, 30, 33]; and refinement of abstract constraint specifications [9] in languages such as ESRA [8], ESSENCE [10],  $\mathcal{F}$  [18] or Zinc [21, 25]. Some systems [2–4, 7, 22] aim to learn constraint models from positive or negative examples.

This paper focuses on the refinement-based approach, in which a user writes abstract constraint specifications to describe a problem at a higher level than that where

modelling decisions are normally made. Abstract constraint specification languages, e.g. ESSENCE and Zinc, support abstract variables with types for common mathematical structures such as sets, multisets, functions, and relations, as well as nested types, such as set of sets and multiset of functions. Problems can often be specified very concisely in this way. For example, the Social Golfers Problem [17], which is to find a set of partitions of golfers subject to some constraints, can be specified directly (see Fig. 1) without the need to model the sets or partitions as matrices of Integer variables.

We use ESSENCE in this paper [10]. An ESSENCE specification, such as that in Fig. 1, identifies: the input parameters of the problem class (*given*), whose values define an instance; the combinatorial objects to be found (*find*); and the constraints the objects must satisfy (*such that*). An objective function may also be specified (*min/maximising*) and identifiers may be declared (*letting*). Abstract constraint specifications must be *refined* into concrete constraint models for existing constraint solvers. Our CONJURE system<sup>1</sup> [1] uses refinement rules to convert an ESSENCE specification into the solver-independent constraint modelling language ESSENCE' [30]. From ESSENCE' we use SAVILE ROW<sup>2</sup> to translate the model into input for a particular constraint solver while performing solver-specific model optimisations.

CONJURE has been able to produce the *kernels* of constraint models, without advanced features like symmetry breaking often used by experts to improve model performance. The first contribution of this paper is to automate the generation of symmetry-breaking constraints. Much symmetry enters constraint models through the process of constraint modelling [11]. CONJURE exploits this by breaking symmetry as it enters the model. This obviates the need for an expensive symmetry detection step following model formulation, as used by other approaches [24, 26]. The added symmetry breaking constraints hold for the entire parameterised problem class — not just a single problem instance — without the need to identify graph automorphisms.

The second contribution of this paper is to automate model selection. Previously, CONJURE has been able to produce a (typically large) set of alternative models through the application of alternative refinement rules, but not to select among these models. CONJURE can now automatically select the best models for a problem class.

## 2 Automated Symmetry Breaking

Symmetry enters constraint models in two ways. Some problems have inherent symmetries, which if not broken get reflected in the model. Other symmetries are introduced by the modelling process; in this case a single solution to the problem corresponds to multiple assignments to the variables of the model. We call these *model* symmetries. As an example, consider the Social Golfers Problem (Fig. 1), which requires finding a set of  $w$  partitions. If this set is modelled as an array indexed by  $1..w$  then all  $w!$  permutations of the array correspond to the same set. This symmetry is introduced when an arbitrary decision is made about which set element goes in which cell of the array. Similarly, if the  $g*s$  *Golfers* are modelled by the Integers  $1..g*s$  then  $g*s$  symmetries are introduced because of the arbitrary decision of which golfer corresponds to which

<sup>1</sup> [https://bitbucket.org/stacs\\_cp/conjure-public/](https://bitbucket.org/stacs_cp/conjure-public/)

<sup>2</sup> <http://savilerow.cs.st-andrews.ac.uk>

```

given    w, g, s : int(1..)
letting Golfers be new type of size g * s
find    sched : set (size w) of partition (regular, size g)
           from Golfers

such that
  forall week1, week2 in sched, week1 != week2 .
    forall group1 in parts(week1) .
      forall group2 in parts(week2) .
        |group1 intersect group2| < 2

```

Fig. 1. ESSENCE specification of the Social Golfers Problem

Integer. The problem-specification language ESSENCE has been designed such that, unlike other modelling languages, problems can be specified without having to make the arbitrary decisions that introduce model symmetries.

Frisch et al. [11] show how each modelling rule of CONJURE can be extended to generate a description of the symmetries it introduces and how the generated descriptions can be composed to form a description of the symmetries introduced into the model. The intention was that this could then be used to generate symmetry-breaking constraints, though these descriptions were never fully developed into a method for automatically generating symmetry-breaking constraints.

The current version of CONJURE takes a different approach to generating symmetry breaking constraints: every rule that introduces symmetries also generates a constraint to break those symmetries. CONJURE has 28 such rules. There is only one rule which does not break all symmetries it introduces – the rule that refines an unnamed type, such as *Golfers*, to a range of Integers. This is because each unnamed type can be used in multiple places, and the symmetry of an unnamed type must be broken in a globally consistent way. All the other symmetries we introduce are independent, so we can add constraints which immediately break each introduced group of symmetries in a valid and complete manner. This leads to globally valid and complete symmetry breaking. We plan to handle unnamed types in the future.

To illustrate how CONJURE rules can be extended to generate symmetry-breaking constraints, consider the rule given below to build the explicit representation of a set.

```

Representation: Set~Explicit~Sym
Matches:       set (size &n, ..) of &tau
Produces:      refn : matrix indexed by [int(1..&n)] of &tau
Constraint:    allDiff(refn)

```

This rule transforms a set of a size  $n$  into a matrix with  $n$  index values, where each value in the matrix is a member of the set. A constraint is imposed to ensure that the cells of the matrix are all different. For any  $\tau$  other than Integers or Booleans, CONJURE has to further decompose the `allDiff` constraint into  $O(n^2)$  not-equal constraints.

Now consider extending this rule to generate a constraint to break the symmetry it introduces, that the index values of the matrix can be permuted in any way. The simplest way to break this symmetry is to impose a total order on the elements of the matrix. As the elements of the matrix can be any type  $\tau$  we introduce two new operators,  $\dot{\leq}$  and  $\dot{<}$ . These operators provide a total ordering (and a strict version of the same total ordering) for all types in CONJURE. These orderings are not intended to be

“natural” and are not available to ESSENCE users. They are used only in refinement rules to generate effective symmetry-breaking constraints. Using these orderings, the `Set~Explicit~Sym` rule is modified to a rule that breaks all the symmetries it introduces:

```
Representation: Set~Explicit
Matches:       set (size &n, ..) of &tau
Produces:      refn : matrix indexed by [int(1..&n)] of &tau
Constraint:    forall i : int(1..&n-1) . refn[i] .< refn[i+1]
```

Rather than introducing a chain of  $\leq$  constraints, this rule exploits the fact that the elements of the set are required to be all different and strengthens the ordering to  $<$  constraint. This replaces  $O(n^2)$  not-equal constraints with only  $O(n)$   $<$  constraints.

Other refinement rules can exploit the fact that symmetry breaking is performed immediately to produce more efficient refinements. Consider refining the constraint  $S = T$  by representing the sets  $S$  and  $T$  as matrices  $S'$  and  $T'$  with the `Set~Explicit~Sym` representation. To find if  $S'$  and  $T'$  represent the same set we must check if each element of  $S'$  is equal to *any* element of  $T'$  and whether the two sets have the same cardinality, since the order of elements in the matrices can be different. However, when the `Set~Explicit` representation is used we can refine  $S = T$  to the constraint  $S' = T'$ , because each assignment of  $S$  is represented by exactly one assignment to  $S'$ , which satisfies the symmetry breaking constraint. This gives a much smaller constraint, which propagates much more efficiently.

We illustrate the new approach to symmetry-breaking by showing how the SGP specification (Fig. 1) is refined into a model with symmetry-breaking constraints. We consider generating only one model. We will consider only how the decision variables are refined, ignoring all constraints other than symmetry-breaking constraints. First, CONJURE replaces type of size  $g*s$  with `int(1..g*s)`:

```
given w, g, s : int(1..)
find sched' : set (size w) of partition (regular, size g) from int(1..g*s)
```

After this, CONJURE refines the type of the decision variable by rewriting the outer set constructor using the `Set~Explicit` rule given in the previous section. This generates the following refinement.

```
given w, g, s : int(1..)
find sched' : matrix indexed by [int(1..w)] of
               partition (regular, size g) from int(1..g*s)
such that forall i : int(1..w-1). sched'[i] .< sched'[i+1]
```

This refinement step shows all of the important features of our method. CONJURE has introduced a new, compact constraint which both breaks symmetry, and ensures all members of the matrix are distinct. Next, it transforms the partition into a set of sets:

```
given w, g, s : int(1..)
find sched'' : matrix indexed by [int(1..w)] of
               set (size g) of set (size (g*s)/g) of int(1..g*s)
such that forall i : int(1..w-1). sched''[i] .< sched''[i+1],
               forall j : int(1..w).
                   forall k1,k2 in sched''[j], k1 != k2. | k1 intersect k2 | = 0
```

This refinement does not appear to have changed the symmetry-breaking constraint but it has in fact been refined from a partition to a set of sets. CONJURE has also added a constraint to impose that the cells of the partition are distinct. This structural constraint constrains the sets to be disjoint. CONJURE now applies `Set~Explicit` again.



```

given      w, g, s : int(1..)
find      sched''' : matrix indexed by [int(1..w), int(1..g)]
              of set (size (g*s)/g) of int(1..g*s)
such that forall i : int(1..w-1). sched'''[i,...] .< sched'''[i+1,...],
              forall j : int(1..w). forall k : int(1..g-1).
              sched'''[j,k] .< sched'''[j,k+1]

```

The first constraint here is the refined version of the already existing symmetry-breaking constraint. Once again by design the  $\leq$  constraint maps naturally to the matrices used in refinement. The second constraint is the symmetry breaking on matrix of sets, now transformed into a matrix of matrices. CONJURE uses the same refinement rule, even though we are now refining a set inside a matrix. CONJURE automatically deals with the array indices and inserts the outer **forall**  $j : \text{int}(1..w)$  in a process called *lifting*. We finally apply *Set~Explicit* once more and change the  $\leq$  and  $\leq$  constraints to their final form – lexicographic ordering constraints on matrices and ordering on Integers.

```

given      w, g, s : int(1..)
find      sched''' : matrix indexed by [int(1..w), int(1..g),int(1..(g*s/g))]
              of int(1..g*s)
such that forall i : int(1..w-1). sched'''[i,...] <lex sched'''[i+1,...],
              forall j : int(1..w). forall k : int(1..(g*s)/g-1).
              sched'''[j,k,...] <lex sched'''[j,k+1,...]
              forall j : int(1..w). forall k : int(1..g)
              forall l : int(1..(g*s)/g-1). sched'''[j,k,l] < sched'''[j,k,l+1]

```

If CONJURE had not the broken symmetries immediately, but instead used the *Set~Explicit~Sym* representation, the constraints requiring each partition in the outermost set to be different would now be very complex. This shows the benefit of breaking symmetries as soon as they are introduced, rather than delaying and using a general technique for symmetry breaking after model generation is finished.

### 3 Automated Model Selection

Our previous work [1] shows that CONJURE can successfully refine a set of model kernels (i.e. excluding symmetry breaking and implied constraints) from a given specification, and that this set contains the kernels of *effective* models. However, without symmetry breaking the performance of these model kernels is poor since refinement of abstract types naturally introduces a great deal of symmetry. Therefore, the symmetry breaking approach described above is a necessary step in producing practically useful models. Having thus enhanced CONJURE the natural next step is to provide a means to select an effective model automatically. We propose and evaluate two such approaches: a lightweight heuristic based purely on an analysis of model structure and an approach that uses a set of training instances to perform model selection by means of a race.

#### 3.1 The Compact Heuristic

If time is limited it is sensible to provide a rapid model selection method, avoiding both generating all models and training using instance data. Our solution is a heuristic employed during refinement to commit greedily to promising modelling choices at each point where an abstract type or a constraint expression may be refined in multiple ways.

It is named Compact since it favours transformations that produce smaller expressions. For an abstract type, we define an ordering as follows: concrete domains (such as `bool`, `matrix`) are smaller than abstract domains; within concrete domains, `bool` is smaller than `int` and `int` is smaller than `matrix`. These rules are applied recursively, so that a one-dimensional matrix of `int` is smaller than any two-dimensional matrix. Abstract type constructors have the ordering `set` < `mset` < `function` < `relation` < `partition`, which is also applied recursively. Compact will select the smallest domain according to this order. For a constraint expression (and the objective), Compact chooses the refinement with the most shallow abstract syntax tree.

### 3.2 Racing

Our second selection method takes as input a set of instances representative of the distribution of instances a user wishes to solve. Our measure of quality of a model with respect to an instance is the time taken for SAVILE ROW to instantiate the model and translate for input to the MINION constraint solver [13] plus the time taken for MINION to solve the instance. We include the time taken by SAVILE ROW since it adds desirable instance-specific optimisations to the model, such as common subexpression elimination [14]. Given a parameter  $\rho \geq 1$ , a model is  $\rho$ -dominated on an instance by another model if the measure for the second model is at least  $\rho$  times faster than the first.

We iterate over the set of instances and conduct a *race* [5] for each. The set of models entered into the race for instance  $i$  are the winners of the race for instance  $i - 1$ , with all models entered in the first race. The ‘winners’ of an instance race are the models not  $\rho$ -dominated by any other model. After we have iterated over all of the supplied instances, the subset of models remaining is selected for the specified class. This naturally suggests the notion of a *model portfolio*, analogous to algorithm portfolios [16, 19].

A set of instances is  $\rho$ -fractured if every model is  $\rho$ -dominated on at least one instance. If the supplied set of instances is fractured, races run with different instance orderings can produce disjoint sets of models. We observe this experimentally in Section 3.4 and discuss its consequences.

### 3.3 Case Study: Equidistant Frequency Permutation Arrays

We illustrate the model selection process using the Equidistant Frequency Permutation Array (EFPA) problem [20]: ‘The problem has parameters  $v, q, \lambda, d$  and it is to find a set  $E$  of size  $v$ , of sequences of length  $q\lambda$ , such that each sequence contains  $\lambda$  of each symbol in the set  $\{1, \dots, q\}$ . For each pair of sequences in  $E$ , the pair are Hamming distance  $d$  apart (i.e. there are  $d$  places where the sequences disagree)’.

This problem is specified in ESSENCE (see Fig. 2) with a single abstract decision variable  $E$  and two constraints. The first ensures that each codeword must contain each symbol  $\lambda$  times, the second that each pair of codewords must differ in exactly  $d$  places. CONJURE refines this specification into 45 models. The type of  $E$  is a fixed size set of total functions. The outer set is always modelled using the explicit representation (as a vector of the inner type) and the symmetry is broken by constraining the elements of the vector to be in increasing order according to  $\prec$ . The total function is refined in two ways: to a vector, or to a relation. In the latter case the relation is refined in four different

```

given      d, lambda, q, v : int(1..)
letting    Character      be domain int(1..q)
letting    Index         be domain int(1..lambda * q)
letting    String        be domain function (total)
                                Index --> Character
find      E              : set (size v) of String
such that forall s in E . forall a : Character .
                                (sum i : Index . toInt(s(i) = a)) = lambda,
                                forall s1, s2 in E, s1 != s2 .
                                (sum i : Index . toInt(s1(i) != s2(i))) = d

```

**Fig. 2.** ESSENCE specification of the EFPA Problem

ways, giving five representations of  $E$  in total. Subsets of these five are channelled and constraints are stated on different representations to create 45 models.

For EFPA we use 24 instances from Huczynska et al. [20], and 12 easier instances that were created by taking the satisfiable instances from Huczynska et al. and reducing  $v$  by one. Identifying instances by the tuple  $\langle d, \lambda, q, v \rangle$ , the first instance we race is  $\langle 3, 7, 7, 5 \rangle$ . This instance is exceptionally discriminating. The number of winners is 4, so we have eliminated 41 models at this stage. We will see in Section 3.4 that not all problems converge so quickly. Second, the remaining models are raced on the instance  $\langle 3, 8, 8, 6 \rangle$ . This does not eliminate any models, although they are ranked in a different order. This process is continued for another 30 instances that eliminate no models. Instance  $\langle 6, 4, 3, 12 \rangle$  eliminates one model, leaving three. Finally, the last three instances eliminate no more models so the final winning set has three models.

All of the final set of models contain the vector representation of the total function. Two of the models refine the function to a relation, then to a two-dimensional matrix of Boolean variables (which is channelled with the vector). These two models differ in one constraint. The relative similarity of these three models shows that on this problem there is a clear cluster of similar winners among a more diverse set of models.

For this problem, Compact generates the model which uses the vector representation for the function variable without any channelling. Although it uses far less information and is very quick in comparison to racing, it manages to find one of the ‘winner’ models.

### 3.4 Experimental Evaluation

In this section, we present the results of model selection for the five problem classes presented in Table 1: EFPA [20], Social Golfers Problem (SGP) [15], Progressive Party Problem (PPP) [32], the SONET network design problem [31], and Error Correcting Codes (ECC) [12]. Although not generally feasible in practice, for the purpose of this experiment we ran a race for every model on every instance with no pruning of models between races. We set  $\rho = 2$  and a timeout of one hour. Furthermore, a model that solves an instance within ten seconds is considered to be non-dominated on that instance. The results presented in the *Winner set size* column of Table 1 show the number of non-dominated models in each case. For the second problem class, SGP, the set of instances is fractured; every winner set contains either model 2 or model 3 but not both.

Now consider the performance of the racing scheme. Notice that the winner set of a race must contain all the non-dominated models. It may contain further models that

**Table 1.** Experimental results

| Problem | Inputs |           | Steps to convergence |           | Results         |         |
|---------|--------|-----------|----------------------|-----------|-----------------|---------|
|         | Models | Instances | Mean                 | Std. Dev. | Winner set size | Compact |
| EFPA    | 45     | 36        | 9.64                 | 4.65      | 1               | Yes     |
| SGP     | 4      | 37        | 5.14                 | 1.78      | Fractured       | Yes     |
| PPP     | 81     | 11        | 5.67                 | 1.47      | 4               | No      |
| ECC     | 108    | 26        | 2.75                 | 0.43      | 4               | No      |
| SONET   | 27     | 47        | 4.30                 | 1.93      | 1               | Yes     |

were not eliminated because of the eager pruning policy. Such models are dominated on some instance by models that were eliminated earlier in the racing process. The number and identity of these extra models is dependent on the order that the instances are considered. Also dependent on this order is the rate of convergence.

In order to test the importance of instance order, we ran 50 races with randomly-selected instance orders. The racing scheme does not know if the problem instances are fractured, though in some cases it may detect that it is. We make the distinction solely for the sake of this study. For the four non-fractured problem classes, all 50 sample races yielded a winner set comprising exactly the non-dominated models. In contrast, the SGP does exhibit fracturing. On the 50 runs every winner set is a singleton comprising either model 1, 2 or 3. The mean and standard deviation of the number of instances raced before reaching the final model set are given in Table 1 under the *Steps to convergence* heading.

Table 1 also presents whether Compact manages to generate a model that is in one of the winner sets found by racing. It finds a winner model for two out of four non-fractured problem classes. Moreover, it finds a winner model for one of the subdivisions in the fractured class, SGP. This is a promising result, considering that Compact works with far less information than racing and is very cheap.

## 4 Conclusions

This paper has demonstrated significant progress towards the goal of automated constraint modelling. We have shown how symmetry can be broken cheaply and automatically as it enters the model through the modelling process, increasing the quality of the models that CONJURE can produce beyond model kernels. Furthermore, we have shown how CONJURE can select *effective* models using a racing process and the Compact heuristic.

**Acknowledgements.** We thank the anonymous reviewers for their comments. This research is supported by UK EPSRC grants no EP/H004092/1 and EP/K015745/1 and EU FP7 grant 284715.

## References

1. Akgun, O., Miguel, I., Jefferson, C., Frisch, A.M., Hnich, B.: Extensible automated constraint modelling. In: AAI 2011: Twenty-Fifth Conference on Artificial Intelligence (2011)
2. Beldiceanu, N., Simonis, H.: A model seeker: Extracting global constraint models from positive examples. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 141–157. Springer, Heidelberg (2012)
3. Bessière, 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)
4. Bessiere, C., Coletta, R., Koriche, F., O’Sullivan, B.: Acquiring constraint networks using a SAT-based version space algorithm. In: AAI 2006, pp. 1565–1568 (2006)
5. Birattari, M., Stützle, T., Paquete, L., Varrentapp, K.: A racing algorithm for configuring metaheuristics. In: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 11–18. Morgan Kaufmann (2002)
6. Charnley, J., Colton, S., Miguel, I.: Automatic generation of implied constraints. In: Proc. of ECAI 2006, pp. 73–77. IOS Press (2006)
7. Coletta, R., Bessière, C., O’Sullivan, B., Freuder, E.C., O’Connell, S., Quinqueton, J.: Semi-automatic modeling by constraint acquisition. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 812–816. Springer, Heidelberg (2003)
8. Flener, P., Pearson, J., Ågren, M.: Introducing ESRA, a relational language for modelling combinatorial problems (Abstract). In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 971–971. Springer, Heidelberg (2003)
9. Frisch, A.M., Jefferson, C., Hernandez, B.M., Miguel, I.: The rules of constraint modelling. In: Proc. of the IJCAI 2005, pp. 109–116 (2005)
10. 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), <http://dx.doi.org/10.1007/s10601-008-9047-y>
11. Frisch, A.M., Jefferson, C., Martínez-Hernandez, B., Miguel, I.: Symmetry in the generation of constraint models. In: Proceedings of the International Symmetry Conference (2007)
12. Frisch, A., Jefferson, C., Miguel, I.: Constraints for breaking more row and column symmetries. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 318–332. Springer, Heidelberg (2003)
13. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: Proceedings ECAI 2006, pp. 98–102 (2006)
14. Gent, I.P., Miguel, I., Rendl, A.: Common subexpression elimination in automated constraint modelling. In: Workshop on Modeling and Solving Problems with Constraints, pp. 24–30 (2008)
15. Gent, I.P., Walsh, T.: CSPLib: a benchmark library for constraints. Tech. rep. (April 16, 1999), <http://citeseer.ist.psu.edu/5054.html>, <http://www.cs.strath.ac.uk/~apes/reports/apes-09-1999.ps.gz>
16. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence 126(1-2), 43–62 (2001)
17. Harvey, W.: Symmetry breaking and the social golfer problem. In: Proc. SymCon 2001: Symmetry in Constraints, co-located with CP 2001, pp. 9–16 (2001)
18. Hnich, B.: Thesis: Function variables for constraint programming. AI Commun. 16(2), 131–132 (2003)
19. Huberman, B.A., Lukose, R.M., Hogg, T.: An economics approach to hard computational problems. Science 275(5296), 51–54 (1997)

20. 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)
21. Koninck, L.D., Brand, S., Stuckey, P.J.: Data independent type reduction for zinc. In: ModRef 2010 (2010)
22. Lallouet, A., Lopez, M., Martin, L., Vrain, C.: On learning constraint problems. In: 22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI), vol. 1, pp. 45–52 (2010)
23. Little, J., Gebruers, C., Bridge, D.G., Freuder, E.C.: Using case-based reasoning to write constraint programs. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 983–983. Springer, Heidelberg (2003)
24. Mancini, T., Cadoli, M.: Detecting and breaking symmetries by reasoning on problem specifications. In: Zucker, J.-D., Saitta, L. (eds.) SARA 2005. LNCS (LNAI), vol. 3607, pp. 165–181. Springer, Heidelberg (2005)
25. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P.J., de la Banda, M.G., Wallace, M.: The design of the zinc modelling language. *Constraints* 13(3) (2008), <http://dx.doi.org/10.1007/s10601-008-9041-4>
26. Mears, C., Niven, T., Jackson, M., Wallace, M.: Proving symmetries by model transformation. In: Lee, J. (ed.) CP 2011. LNCS, vol. 6876, pp. 591–605. Springer, Heidelberg (2011)
27. Mills, P., Tsang, E., Williams, R., Ford, J., Borrett, J.: EaCL 1.5: An easy abstract constraint optimisation programming language. Tech. rep., University of Essex, Colchester, UK (December 1999)
28. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.R.: MiniZinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007)
29. Puget, J.F.: Constraint programming next challenge: Simplicity of use. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 5–8. Springer, Heidelberg (2004)
30. Rendl, A.: Thesis: Effective Compilation of Constraint Models. Ph.D. thesis, University of St. Andrews (2010)
31. Smith, B.M.: Search strategies for optimization: Modelling the sonet problem. Tech. Rep. Research Report APES-70-2003. presented at 2nd International Workshop on Reformulating Constraint Satisfaction Problems (2003)
32. Smith, B.M., Brailsford, S.C., Hubbard, P.M., Williams, H.P.: The progressive party problem: Integer linear programming and constraint programming compared. In: Montanari, U., Rossi, F. (eds.) CP 1995. LNCS, vol. 976, pp. 36–52. Springer, Heidelberg (1995)
33. Van Hentenryck, P.: The OPL Optimization Programming Language. MIT Press, Cambridge (1999)

# Improving WPM2 for (Weighted) Partial MaxSAT\*

Carlos Ansótegui<sup>1</sup>, Maria Luisa Bonet<sup>2</sup>, Joel Gabàs<sup>1</sup>, and Jordi Levy<sup>3</sup>

<sup>1</sup> DIEI, Univ. de Lleida  
{carlos,joel.gabas}@diei.udl.cat  
<sup>2</sup> LSI, UPC  
bonet@lsi.upc.edu  
<sup>3</sup> IIIA-CSIC  
levy@iiia.csic.es

**Abstract.** Weighted Partial MaxSAT (WPMS) is an optimization variant of the Satisfiability (SAT) problem. Several combinatorial optimization problems can be translated into WPMS. In this paper we extend the state-of-the-art WPM2 algorithm by adding several improvements, and implement it on top of an SMT solver. In particular, we show that by focusing search on solving to optimality subformulas of the original WPM2 instance we increase the efficiency of WPM2. From the experimental evaluation we conducted on the PMS and WPMS instances at the 2012 MaxSAT Evaluation, we can conclude that the new approach is both the best performing for industrial instances, and for the union of industrial and crafted instances.

## 1 Introduction

In the last decade Satisfiability (SAT) solvers have progressed dramatically in performance due to new algorithms, such as, conflict directed clause learning [36], and better implementation techniques. Thanks to these advances, nowadays the best SAT solvers can tackle hard decision problems. Our aim is to push this technology forward to deal with optimization problems.

The Maximum Satisfiability (MaxSAT) problem is the optimization version of SAT. The idea behind this formalism is that sometimes not all constraints of a problem can be satisfied, and we try to satisfy the maximum number of them. The MaxSAT problem can be further generalized to the Weighted Partial MaxSAT (WPMS) problem.

In the MaxSAT community, we find two main classes of algorithms: branch and bound [17, 22, 24, 26, 27] and SAT-based [2, 14, 19–21, 31–33]. The latter clearly dominate on industrial and some crafted instances, as we can see in the results of the last 2012 MaxSAT Evaluation. SAT-based MaxSAT algorithms basically reformulate a MaxSAT instance into a sequence of SAT instances. By solving these SAT instances the MaxSAT problem can be solved [6].

---

\* This research has been partially founded by the CICYT research projects TASSAT (TIN2010-20967-C04-01/03/04) and ARINF (TIN2009-14704-C03-01).

In this paper we revisit the SAT-based MaxSAT algorithm WPM2 [5] which belongs to a family of algorithms that exploit the information from the unsatisfiable cores the underlying SAT solver provides. This algorithm is the natural extension to the weighted case of the Partial MaxSAT algorithm PM2 [3, 4]. In our experimental investigation the original WPM2 algorithm solves 796 out of 1474 from the whole benchmark of PMS and WPMS industrial and crafted instances at the 2012 MaxSAT Evaluation. We have extended WPM2 with several complementary improvements. First of all, we apply the stratification approach described in [2], what results in solving 74 additional instances. Secondly, we introduce a new criteria to decide when soft clauses can be hardened, that provides 66 additional solved instances. The hardening of soft clauses in MaxSAT SAT-based solvers has been previously studied in [2, 33]. Finally, our most effective contribution is to introduce a new strategy that focuses search on solving to optimality subformulas of the original MaxSAT instance. Actually, the new WPM2 algorithm is parametric on the approach we use to optimize these subformulas. This allows to combine the strength of exploiting the information extracted from unsatisfiable cores and other optimization approaches. By solving these smaller optimization problems we get the most significant boost in our new WPM2 algorithm. In particular, we experiment with three approaches: (i) refine the lower bound on these subformulas with the subsetsum function [5, 13], (ii) refine the upper bound with the strategy applied in minisat+ [15], SAT4J [10], qmaxsat [21] or ShinMaxSat [20], and (iii) a binary search scheme where the lower bound and upper bound are refined as in the previous approaches. The best performing approach in our experimental analysis is the second one and it allows to solve up to 238 additional instances. As a summary, the overall speed-up we achieved on the original WPM2 solver is about 378 additional solved instances, a 47% more.

As we mentioned, SAT-based MaxSAT algorithms reformulate a MaxSAT instances into a sequence of SAT instances. Obviously, it is important to use an efficient SAT solver. Also, most SAT-based MaxSAT algorithms require the addition of Pseudo-Boolean (PB) linear constraints as a result of the reformulation process. These PB constraints are used to bound the cost of the optimal assignment. Currently, in most state-of-the-art SAT-based MaxSAT solvers, PB constraints are translated into SAT. However, there is no known SAT encoding which can guarantee the original propagation power of the constraint, i.e, what we call arc-consistency, while keeping the translation low in size. The best approach so far, has a cubic complexity [8]. This can be a bottleneck for WPM2 [5] and also for other algorithms such as, BINCD [19] or SAT4J [10].

In order to treat PB constraints with specialized inference mechanisms and a moderate cost in size, while preserving the strength of SAT techniques for the rest of the formula, we use the Satisfiability Modulo Theories (SMT) technology [35]. Related work in this sense can be found in [34]. Also, in [1] a Weighted Constraint Satisfaction Problems (WCSP) solver implementing the original WPM1 [4] algorithm is presented.



An SMT instance is a generalization of a Boolean formula in which some propositional variables have been replaced by predicates with predefined interpretations from background theories such as, e.g., linear integer arithmetic. Most modern SMT solvers integrate a SAT solver with decision procedures (theory solvers) for sets of literals belonging to each theory. This way, we can hopefully get the best of both worlds: in particular, the efficiency of the SAT solver for the Boolean reasoning and the efficiency of special-purpose algorithms for the theory reasoning.

Another reasonable choice would be to use a PB solver, which can be seen as a particular case of an SMT solver specialized on the theory of PB constraints [28, 29]. However, if we also want to solve problems modeled with richer formalisms like WCSP, the SMT approach seems a better choice since we can take advantage of a wide range of theories [1].

In this work, we implemented both the last version of the WPM1 algorithm [2] and the revisited version of the WPM2 algorithm on top the of the SMT solver Yices. Then, we performed an extensive experimental evaluation comparing them with the best two solvers for PMS and WPMS categories at the 2012 MaxSAT Evaluation and with three additional solvers that did not take part but have been reported to exhibit good performance: *bincd2*, which is the new version of the BINCD algorithm [19] described in [33], with the best configuration reported by authors, *maxhs* from [14], which consists in an hybrid SAT and Integer Linear Programming (ILP) approach, and *ilp* which performs a translation of WPMS into ILP solved with IBM-CPLEX studio124 [7].

We observe that the implementation on SMT of our new WPM2 algorithm with the second approach for optimizing the subformulas is the best performing solver for both PMS and WPMS industrial instances. We also observe that it is the best performing for the union of PMS and WPMS industrial and crafted instances, what shows this is a robust approach. These results make us conjecture that by improving the interaction of our new WPM2 algorithm with diverse optimization techniques applied on the subformulas we can get additional speed-ups.

This paper proceeds as follows. Section 2 presents some preliminary concepts. Section 3 describes WPM2 [5] and the new improvements. Section 4 describes the SMT problem and discuss some implementation details of the SMT-based MaxSAT algorithms. Section 5 presents the experimental evaluation. Finally, Section 6 shows the conclusions and the future work.

## 2 Preliminaries

A *literal* is either a Boolean variable  $x$  or its negation  $\bar{x}$ . A *clause*  $C$  is a disjunction of literals. A *weighted clause* is a pair  $(C, w)$ , where  $C$  is a clause and  $w$  is a natural number or infinity, indicating the penalty for falsifying the clause  $C$ . A *Weighted Partial MaxSAT formula* is a multiset of weighted clauses

$$\varphi = \{(C_1, w_1), \dots, (C_m, w_m), (C_{m+1}, \infty), \dots, (C_{m+m'}, \infty)\}$$

where the first  $m$  clauses are soft and the last  $m'$  clauses are hard. The set of variables occurring in a formula  $\varphi$  is noted as  $\text{var}(\varphi)$ .

A *(total) truth assignment* for a formula  $\varphi$  is a function  $I : \text{var}(\varphi) \rightarrow \{0, 1\}$ , that can be extended to literals, clauses and SAT formulas. For MaxSAT formulas is defined as  $I(\{(C_1, w_1), \dots, (C_m, w_m)\}) = \sum_{i=1}^m w_i (1 - I(C_i))$ . The *optimal cost* of a formula is  $\text{cost}(\varphi) = \min\{I(\varphi) \mid I : \text{var}(\varphi) \rightarrow \{0, 1\}\}$  and an *optimal assignment* is an assignment  $I$  such that  $I(\varphi) = \text{cost}(\varphi)$ .

The *Weighted Partial MaxSAT problem* for a Weighted Partial MaxSAT formula  $\varphi$  is the problem of finding an *optimal assignment*.

### 3 WPM2 Algorithm

The WPM2 algorithm [5] is described in Algorithm 1. The fragments in gray (lines 4, 10, 11, 13- 18 and 20) correspond to the new improvements we have incorporated.

In the WPM2 algorithm, we extend soft clauses  $C_i$  with a unique fresh auxiliary blocking variable  $b_i$  obtaining  $\varphi_w = \{C_i \vee b_i\}_{i=1\dots m} \cup \{C_{m+i}\}_{i=1\dots m'}$ . Notice that  $b_i$  will be set to true by a SAT solver on  $\varphi_w$  if  $C_i$  is false. We also work with a set  $AL$  of at-least PB constraints of the form  $\sum_{i \in A} w_i b_i \geq k$  on the variables  $b_i$ , and a similar set  $AM$  of at-most constraints of the form  $\sum_{i \in A} w_i b_i \leq k$ , that are modified at every iteration of the algorithm.

Intuitively, the WPM2 algorithm refines at every iteration the lower bound on  $\varphi$  till it reaches the optimum  $\text{cost}(\varphi)$ . The  $AM$  constraints are used to bound the cost of the falsified clauses. The  $AL$  constraints are used to impose that subsets of soft clauses have a minimum cost and to compute the  $AM$  constraints, as we will see later. The algorithm ends when  $\varphi_w \cup \text{CNF}(AL \cup AM)$  becomes satisfiable<sup>1</sup>, where CNF is the translation to SAT of the PB constraints.

Technically speaking, the  $AL$  constraints give lower bounds on  $\text{cost}(\varphi)$ . The  $AM$  constraints enforce that all solutions of the set of constraints  $AL \cup AM$  are the solutions of  $AL$  of minimal cost. This ensures that any solution of the formula sent to the solver,  $\varphi_w \cup \text{CNF}(AL \cup AM)$ , if there is any, is an optimal assignment of  $\varphi$ . Therefore, given a set of at-least constraints  $AL$  we compute a *corresponding* set of at-most constraints  $AM$  as follows. First, we need to introduce the notion of *core* and *cover*. A core is a set of indexes  $A$  such that  $\sum_{i \in A} w_i b_i \geq k \in AL$ . Function  $\text{core}(\sum_{i \in A} w_i b_i \geq k)$  returns the core  $A$  and function  $\text{cores}(AL)$  returns  $\{\text{core}(al) \mid al \in AL\}$ . Covers are defined from cores as follows.

**Definition 1.** *Given a set of cores  $L$ , we say that the set of indexes  $A$  is a cover of  $L$ , if it is a minimal non-empty set such that, for every  $A' \in L$ , if  $A' \cap A \neq \emptyset$ , then  $A' \subseteq A$ . Given a set of cores  $L$ , we denote the set of covers of  $L$  as  $\text{SC}(L)$ .*

<sup>1</sup> The  $AL$  constraints are redundant, i.e., not required to be sent to the SAT solver for the soundness of the algorithm but help to speed up the search.

**Algorithm 1.** Revisited WPM2 algorithm.

---

**Input:**  $\varphi = \{(C_1, w_1), \dots, (C_m, w_m), (C_{m+1}, \infty), \dots, (C_{m+m'}, \infty)\}$

1: **if**  $\text{sat}(\{C_i \in \varphi \mid w_i = \infty\}) = (\text{UNSAT}, \neg, \neg)$  **then return**  $(\infty, \emptyset)$

2:  $\varphi_w := \{C_1 \vee b_1, \dots, C_m \vee b_m, C_{m+1}, \dots, C_{m+m'}\}$  ▷Extend all soft clauses

3:  $AL := \{w_1 b_1 \geq 0, \dots, w_m b_m \geq 0\}$  ▷Set of at-least constraints

4:  $w_{max} := \infty$

5: **while true do**

6:      $AM := \emptyset$  ▷Set of at-most constraints

7:     **foreach**  $(\sum_{i \in A} w_i b_i \geq k) \in AL$  **do**

8:         **if**  $A \in SC(\text{cores}(AL))$  **then**

9:              $AM := AM \cup \{\sum_{i \in A} w_i b_i \leq k\}$

10:      $(st, \varphi_c, \mathcal{I}) := \text{sat}(\varphi_w \setminus \{C_i \vee b_i \mid (C_i, w_i) \in \varphi \wedge w_i < w_{max}\}) \cup \text{CNF}(AL \cup AM)$

11:     **if**  $st = \text{SAT}$  **and**  $w_{max} = 0$  **then return**  $(\mathcal{I}(\varphi), \mathcal{I})$

12:     **else**

13:         **if**  $st = \text{SAT}$  **then**

14:              $W := \sum \{w_i \mid (C_i, w_i) \in \varphi \wedge w_i < w_{max}\}$

15:              $\varphi_h := \text{harden}(\varphi, AM, W)$

16:              $w_{max} := \text{decrease}(w_{max}, \varphi)$

17:         **else**

18:              $A := \{i \mid (C_i \vee b_i) \in (\varphi_c \setminus \varphi_h)\}$  ▷New core

19:              $A := \bigcup_{\substack{A' \in \text{cores}(AL) \\ A' \cap A \neq \emptyset}} A'$  ▷New cover

20:              $k := \text{newbound}(AL \cup \varphi_w, A)$

21:              $AL := \{al \in AL \mid \text{core}(al) \neq A\} \cup \{\sum_{i \in A} w_i b_i \geq k\}$

---

Given a set  $AL$ , the set  $AM$  is the set of at-most constraints  $\sum_{i \in A} w_i b_i \leq k$  such that  $A \in SC(\text{cores}(AL))$  and  $k$  is the solution of minimizing  $\sum_{i \in A} w_i b_i$  subject to  $AL$  and  $b_i \in \{0, 1\}$ .

The algorithm starts with  $AL = \{w_1 b_1 \geq 0, \dots, w_m b_m \geq 0\}$  and the corresponding  $AM := \{w_1 b_1 \leq 0, \dots, w_m b_m \leq 0\}$  that ensures that the unique solution of  $AL \cup AM$  is  $b_1 = \dots = b_m = 0$  with cost 0<sup>2</sup>. At every iteration, the algorithm calls a SAT solver with  $\varphi_w \cup \text{CNF}(AL \cup AM)$ . If it returns SAT, then the interpretation  $\mathcal{I}$  is a MaxSAT solution of  $\varphi$  and we return the optimal cost  $\mathcal{I}(\varphi)$ . If it returns UNSAT, then we use the information of the unsatisfiable core  $\varphi_c$  obtained by the SAT solver to enlarge the set  $AL$ , excluding more interpretations on the  $b_i$ 's that are not partial solutions of  $\varphi_w$ . Before calling again the SAT solver, we update  $AM$  conveniently, to ensure that solutions to the new constraints  $AL \cup AM$  are still minimal solutions of the new  $AL$  constraint set. Notice that in every iteration the set of solutions of  $\{b_1, \dots, b_m\}$  defined by  $AL$  is decreased, whereas the set of solutions of  $AM$  is increased.

---

<sup>2</sup> In the implementation, we do not add a blocking variable to a soft clause till it appears into a core.

One key point in WPM2 is to compute the *newbound*( $AL, A$ ) (line 20) which corresponds to the following optimization problem:

$$\text{minimize } \sum_{i \in A} w_i \cdot b_i \quad \text{subject to } \left\{ \sum_{i \in A} w_i \cdot b_i \geq k \right\} \cup AL \quad (1)$$

where  $k = 1 + \sum \{k' \mid \sum_{i \in A'} w_i b_i \leq k' \in AM \wedge A' \subseteq A\}$ .

Notice that by removing the  $AL$  constraints in (1), we get the subsetsum problem [13]. In the original WPM2 algorithm [5], the subsetsum problem is progressively solved until we get a solution that also satisfies the  $AL$  constraints. This satisfiability check in the original WPM2 is performed with a SAT solver.

In what follows, we present how we have modified the original WPM2 algorithm (fragments in gray in Algorithm 1) by incorporating several improvements: the application of a stratified approach, the hardening of soft clauses and the optimization of the subformulas defined by the covers.

### 3.1 Stratified Approach

As in [4] for WPM1, we apply a stratified approach. The stratified approach (lines 4, 10, 11 and 16) consists in sending to the SAT solver only those soft clauses with weight  $w_i \geq w_{max}$ . Then, when the SAT solver returns SAT, if there are still unsent clauses, we decrease  $w_{max}$  to include additional clauses to the formula. From [4], we also apply the diversity heuristic (line 16) which supplies us with an efficient method to calculate how we have to reduce the value of  $w_{max}$  in the stratified approach, so that, when there is a big variety of distinct weights,  $w_{max}$  decreases faster, and, when there is a low diversity,  $w_{max}$  is decreased to the following value of  $w_i$ . Similar approach with an alternative heuristic for grouping clauses can be found in [32].

### 3.2 Clause Hardening

The hardening of soft clauses in MaxSAT SAT-based solvers has been previously studied in [2, 11, 18, 23, 25, 30, 33]. Inspired by these works we study a hardening scheme for WPM2. While clause hardening was reported to have no positive effect in WPM1 [2], we will see that it boosts efficiency in WPM2.

The clause hardening (lines 14, 15 and 18) consists in considering hard those soft clauses whose satisfiability we know does not need to be reconsidered. We need some lemma ensuring that falsifying those soft clauses would lead us to suboptimal solutions. In the case of WPM1, all soft clauses satisfying  $w_i > W$ , where  $W = \sum \{w_i \mid (C_i, w_i) \in \varphi \wedge w_i < w_{max}\}$  is the sum of weights of clauses not sent to the SAT solver, can be hardened. The correctness of this transformation is ensured by the following lemma:

**Lemma 1 (Lemma 24 in [6])**

Let  $\varphi_1 = \{(C_1, w_1), \dots, (C_m, w_m), (C_{m+1}, \infty), \dots, (C_{m+m'}, \infty)\}$  be a MaxSAT formula with cost zero, let  $\varphi_2 = \{(C'_1, w'_1), \dots, (C'_r, w'_r)\}$  be a MaxSAT formula without hard clauses and  $W = \sum_{j=1}^r w'_j$ . Let

$$\text{harden}(w) = \begin{cases} w & \text{if } w \leq W \\ \infty & \text{if } w > W \end{cases}$$

and  $\varphi'_1 = \{(C_i, \text{harden}(w_i)) \mid (C_i, w_i) \in \varphi_1\}$ . Then,  $\text{cost}(\varphi_1 \cup \varphi_2) = \text{cost}(\varphi'_1 \cup \varphi_2)$ , and any optimal assignment for  $\varphi'_1 \cup \varphi_2$  is an optimal assignment of  $\varphi_1 \cup \varphi_2$ .

However, this lemma is not useful in the case of WPM2 because we do not proceed by transforming the formula, like in WPM1. Therefore, we generalize this lemma. For this, we need to introduce the notion of *optimal* of a formula.

**Definition 2.** Given a MaxSAT formula  $\varphi = \{(C_1, w_1), \dots, (C_m, w_m), (C_{m+1}, \infty), \dots, (C_{m+m'}, \infty)\}$ , we say that  $k$  is a (possible) optimal of  $\varphi$  if there exists a subset  $A \subseteq \{1, \dots, m\}$  such that  $\sum_{i \in A} w_i = k$ .

Notice that, for any interpretation  $I$  of the variables of  $\varphi$ , we have that  $I(\varphi)$  is an optimal of  $\varphi$ . However, if  $k$  is an optimal, there does not exist necessarily an interpretation  $I$  satisfying  $I(\varphi) = k$ . Notice also that, given  $\varphi$  and  $k$ , finding the *next optimal*, i.e. finding the smallest  $k' > k$  such that  $k'$  is an optimal of  $\varphi$  is equivalent to the subset sum problem.

**Lemma 2.** Let  $\varphi_1 \cup \varphi_2$  be a MaxSAT formula and  $k_1$  and  $k_2$  values such that:  $\text{cost}(\varphi_1 \cup \varphi_2) = k_1 + k_2$  and any assignment  $I$  satisfies  $I(\varphi_1) \geq k_1$  and  $I(\varphi_2) \geq k_2$ . Let  $k'$  be the smallest possible optimal of  $\varphi_2$  such that  $k' > k_2$ . Let  $\varphi_3$  be a set of soft clauses with  $W = \sum \{w_i \mid (C_i, w_i) \in \varphi_3\}$ .

Then, if  $W < k' - k_2$ , then any optimal assignment  $I'$  of  $\varphi_1 \cup \varphi_2 \cup \varphi_3$  assigns  $I'(\varphi_2) = k_2$

*Proof.* Let  $I'$  be any optimal assignment of  $\varphi_1 \cup \varphi_2 \cup \varphi_3$ . On the one hand, as for any other assignment, we have  $I'(\varphi_2) \geq k_2$ .

On the other hand, any of the optimal assignments  $I$  of  $\varphi_1 \cup \varphi_2$  can be extended (does not matter how) to the variables of  $\text{var}(\varphi_3) \setminus \text{var}(\varphi_1 \cup \varphi_2)$ , such that

$$I(\varphi_1 \cup \varphi_2 \cup \varphi_3) = I(\varphi_1) + I(\varphi_2) + I(\varphi_3) \leq k_1 + k_2 + W < k_1 + k' \quad (2)$$

Now, assume that  $I'(\varphi_2) \neq k_2$ , then  $I'(\varphi_2) \geq k'$ . As any other assignment,  $I'(\varphi_1) \geq k_1$ . Hence,  $I'(\varphi_1 \cup \varphi_2 \cup \varphi_3) \geq k_1 + k' > I(\varphi_1 \cup \varphi_2 \cup \varphi_3)$ , but this contradicts the optimality of  $I'$ . Therefore,  $I'(\varphi_2) = k_2$ . □

In order to apply this lemma we have to consider partitions of the formula  $\varphi_1 \cup \varphi_2$  ensuring  $\text{cost}(\varphi_1 \cup \varphi_2) = k_1 + k_2$  and  $I(\varphi_1) \geq k_1$  and  $I(\varphi_2) \geq k_2$ , for any assignment  $I$ . This can be easily ensured, in the case of WPM2, if both  $\varphi_1$  and  $\varphi_2$  are unions of covers. Then, we only have to check if the next possible optimal  $k'$  of  $\varphi_2$  exceeds the previous one  $k_2$  more than the sum  $W$  of the weights of the clauses not sent to the SAT solver. In such a case, we can consider all soft clauses of  $\varphi_2$  and their corresponding AM constraint with  $k_2$  as hard clauses. In other words, we do not need to recompute the partial optimal  $k_2$  of  $\varphi_2$ .

Finally, in line 15 of Algorithm 1, function *harden*( $\varphi, AM, W$ ) returns the set of soft clauses  $\varphi_h$  that needs to be considered hard based on the previous analysis according to: the current set of covers  $AM$ , the next optimals of these covers and the sum of the weights  $W$  of soft clauses beyond the current  $w_{max}$ , i.e., not yet sent to the SAT solver.

### 3.3 Cover Optimization

As we have mentioned earlier, one key point in WPM2 is how to compute the *newbound*( $AL, A$ ) (line 20). Actually, we can solve to optimality the subformulas defined by the union of the soft clauses related to the cover  $A$  and the hard clauses.

**Definition 3.** *Given a MaxSAT formula  $\varphi = \{(C_1, w_1), \dots, (C_m, w_m), (C_{m+1}, \infty), \dots, (C_{m+m'}, \infty)\}$  and a set of indexes  $A$ , we define the subformula,  $\varphi[A]$ , as follows:  $\varphi[A] = \{(C_i, w_i) \in \varphi \mid i \in A \vee w_i = \infty\}$*

Solving to optimality  $\varphi[A]$  give us the optimal value  $k = cost(\varphi[A])$  for the  $AM$  constraint related to cover  $A$ . In order to do this, while taking advantage of the  $AL$  constraints generated so far, we only have to extend the minimization problem corresponding to the *newbound* (1) function, by adding  $\varphi_w$  to the constraints, i.e., *newbound*( $AL \cup \varphi_w, A$ )<sup>3</sup>. Notice that *newbound*( $AL \cup \varphi_w, A$ )  $\geq$  *newbound*( $AL, A$ ).

In order to optimize  $\varphi[A]$ , we can use any exact approach related to MaxSAT, such as, MaxSAT branch and bound algorithms, MaxSAT SAT-based algorithms, saturation under the MaxSAT resolution rule, or we can use other solving techniques such as PB solvers or ILP techniques, etc. Our new WPM2 algorithm is parametric on any suitable optimization solving approach. In this work, we present three approaches.

The first and natural approach consists in iteratively refining (increasing) the lower bound on the optimal  $k$  for  $\varphi[A]$  by applying the subsetsum function as in the original WPM2. The procedure stops when we satisfy the constraints  $AL \cup \varphi_w$ . Notice that since we have included  $\varphi_w$  into the set of constraints, the solution we will eventually get has to be optimal for  $\varphi[A]$ .

The second approach consists in iteratively refining (decreasing) the upper bound following the strategy applied in minisat+ [15], SAT4J [10], qmaxsat [21] or ShinMaxSat [20]. The upper bound  $ub$  is initially set to the sum of the weights  $w_i$  of the soft clauses in  $\varphi[A]$ . Then, we iteratively test whether  $k = ub - 1$  is feasible or not. Whenever we get a satisfying assignment, we update  $ub$  to the sum of the weights  $w_i$  of those soft clauses where  $b_i$  evaluates to true under the satisfying assignment. If we get an unsatisfiable answer, the previous  $ub$  is the optimal value for  $\varphi[A]$ .

The third approach applies a binary search scheme [12, 16, 19]. We additionally refine the lower bound as in our first approach and the upper bound as in the second approach.

<sup>3</sup> We can actually exclude from  $\varphi_w$  all the soft clauses not in  $\varphi[A]$ .

The worst case complexity, in terms of the number of calls to the SAT solver, of the new WPM2 algorithm is the number of times that the newbound function is called (bounded by the number of clauses) multiplied by the number of SAT calls needed in each call to the newbound function. This latter number is logarithmic on the sum of the weights of the clauses of the core if we use a binary search, hence essentially the number of clauses. Therefore, the worst case complexity, when using a binary search to solve to optimality the subformulas, is quadratic on the number of soft clauses.

In order to see that the number of calls to the newbound function is bounded by the number of clauses we just need to recall that WPM2 merges the covers. Consider a binary tree where the soft clauses are the leaves, and the internal nodes represent the merges (calls to the newbound function). A binary tree of  $n$  leaves has  $n-1$  internal nodes.

Solving to optimality all the covers can be very costly since these are NP-hard problems. Depending on the unsatisfiable cores we get in the general loop of the WPM2 algorithm some covers have to be merged. Therefore, we may argue that part of the work we did in order to optimize these covers can be useless<sup>4</sup>. For example, a reasonable strategy is to optimize the current cover only if it was not the result of merging other covers, i.e., when the last unsatisfiable core is contained into a cover. In the experimental evaluation, we will see that although the number of solved instances does not vary too much, the mean time for solving some families can be decreased.

## 4 Engineering Efficient SMT-Based MaxSAT Solvers

We have implemented both the last version of the WPM1 algorithm [2] and the revisited version of the WPM2 algorithm on top the of the SMT solver Yices.

As we have said, an SMT instance is a generalization of SAT where some propositional variables are replaced by predicates with predefined interpretations from background theories. Among the theories considered in the SMT library [9] we are interested in QF\_LIA (Quantifier-Free *Linear Integer Arithmetic*). With the QF\_LIA theory we can model the PB constraints that SAT-based MaxSAT algorithms generate during their execution. Therefore, for the SMT-based MaxSAT algorithm, we just need to replace the conversion to CNF (line 10 in Algorithm 1) by the proper linear integer arithmetic predicates.

As suggested in [16, 31], we can preserve some learned lemmas from previous iterations that may help to reduce the search space. In order to do that, we execute the SMT solver in incremental mode. Within this mode, we can call the solve routine and add new clauses (assertions) on demand, while preserving learned lemmas. However, notice that our algorithms delete parts of the formula between iterations. For example, in lines 7 to 9 of Algorithm 1 we recompute the set AM, possibly erasing some of the at-most constraints. Therefore, we have to take care also of any learned lemma depending on them.

---

<sup>4</sup> The related *AL* constraints can still be kept.

The SMT solver Yices gives the option of marking assertions as *retractable*. If the SMT solver does not support the deletion of assertions but supports the usage of assumptions, we can replace every retractable assertion  $C$ , with  $a \rightarrow C$ , where  $a$  is an assumption. Before each call, we activate the assumptions of assertions that have not been retracted by the algorithm. Notice that assertions that do have been retracted will have a pure literal ( $\bar{a}$ ) such that  $a$  has not been activated. Therefore, the solver can safely set to false  $a$  deactivating the clause. Moreover, any learned lemma on those assertions will also include  $\bar{a}$ . For example, Z3 and Mathsat SMT solvers do not allow to delete clauses, but they allow the use of assumptions.

## 5 Experimental Results

In this section we present an intensive experimental investigation on the PMS and WPMS industrial and crafted instances from the 2012 MaxSAT Evaluation. We provide results for our new WPM2 SMT-based MaxSAT solver, for a WPM1 [2] SMT-based MaxSAT solver, the best two solvers for each category of the 2012 MaxSAT Evaluation, and three solvers which did not participate but the authors have reported to exhibit good performance. We run our experiments on a cluster featured with 2.27 GHz processors, memory limit of 3.9 GB and a timeout of 7200 seconds per instance.

The experimental results are presented in Tables 1 and 2 following the same classification criteria as in the 2012 MaxSAT Evaluation. For each solver and family of instances, we present the number of solved instances in parenthesis and the mean solving time. Solvers are ordered from left to right according to the total number of solved instances. The results for the best performing solver in each family are presented in bold. The number of instances of every family is specified in the column under the sign '#'. Since different families may have different number of instances, we also include for each solver the mean ratio of solved instances.

Our new WPM2 algorithm is implemented on top of the Yices SMT solver (version 1.0.29). The different versions of WPM2 and corresponding implementations are named *wpm2* where subindexes can be <sub>*s*</sub> that stands for stratified approach with diversity heuristic and <sub>*h*</sub> for hardening. Regarding to how we perform the cover optimization, <sub>*l*</sub> stands for lower bound refinement based on subsetsum, <sub>*u*</sub> for upper bound refinement based on satisfying truth assignment, and <sub>*b*</sub> for binary search. Finally, <sub>*a*</sub> stands for optimizing all the covers and <sub>*c*</sub> for optimizing only covers that contain the last unsatisfiable core.

Table 1 shows our first experiment, where we evaluate the impact of each variation on the original *wpm2*. By using a stratified approach with the diversity heuristic (*wpm2<sub>s</sub>*) we solve some additional instances in all categories having the best improvement in WPMS crafted. Overall, we solve 74 more instances. By adding hardening (*wpm2<sub>sh</sub>*) we solve 66 more instances, mainly in WPMS industrial family *haplotyping-pedigrees*.



Table 1. Experimental results of different versions of *wpm2*

| Instance set            | #    | <i>wpm2</i>     | <i>wpm2s</i> | <i>wpm2sh</i>     | <i>wpm2shlc</i> | <i>wpm2shla</i>   | <i>wpm2shbc</i> | <i>wpm2shba</i>    | <i>wpm2shuc</i>    | <i>wpm2shua</i>   |
|-------------------------|------|-----------------|--------------|-------------------|-----------------|-------------------|-----------------|--------------------|--------------------|-------------------|
| <b>PMS-Industrial</b>   |      |                 |              |                   |                 |                   |                 |                    |                    |                   |
| aes                     | 7    | 0.00(0)         | 0.00(0)      | 0.00(0)           | 0.00(0)         | 0.00(0)           | 0.00(0)         | <b>10.34(1)</b>    | 1836.54(1)         | 514.14(1)         |
| bcp-fir                 | 59   | 83.29(57)       | 23.44(57)    | 23.44(57)         | 40.87(57)       | 104.75(57)        | 113.13(57)      | 146.03(57)         | 60.91(58)          | <b>57.36(58)</b>  |
| bcp-hipp-yRa1_simp      | 17   | 504.81(13)      | 155.95(12)   | 155.95(12)        | 95.16(12)       | 239.17(13)        | 151.30(13)      | 813.36(16)         | <b>107.24(16)</b>  | 160.55(16)        |
| bcp-hipp-yRa1_su        | 38   | 380.47(19)      | 164.70(16)   | 164.70(16)        | 181.49(18)      | 667.59(19)        | 226.66(25)      | 585.82(28)         | 555.92(33)         | <b>315.87(34)</b> |
| bcp-msp                 | 64   | 821.88(25)      | 756.59(26)   | 756.59(26)        | 283.50(28)      | 606.35(28)        | 283.78(31)      | 464.47(30)         | <b>711.08(36)</b>  | 912.70(34)        |
| bcp-mtg                 | 40   | 1363.97(18)     | 852.58(23)   | 852.58(23)        | 1320.80(28)     | 786.09(34)        | 1296.44(32)     | 940.62(37)         | <b>997.38(35)</b>  | <b>578.75(39)</b> |
| bcp-syn                 | 74   | 60.29(41)       | 302.54(41)   | 302.54(41)        | 296.12(41)      | 83.15(39)         | 242.78(42)      | 69.23(42)          | <b>103.21(43)</b>  | 78.22(42)         |
| circuit-trace-compact   | 4    | 285.74(3)       | 835.72(4)    | 835.72(4)         | 230.85(3)       | 134.79(4)         | 145.72(4)       | 151.41(4)          | <b>118.24(4)</b>   | 129.10(4)         |
| haploype-assembly       | 6    | <b>2.87(5)</b>  | 7.17(5)      | 7.17(5)           | 9.80(5)         | 42.86(5)          | 15.53(5)        | 51.44(5)           | 18.28(4)           | 65.94(5)          |
| pbo-mqc-nencdr          | 84   | 866.38(84)      | 821.16(84)   | 821.16(84)        | 142.43(84)      | <b>107.70(84)</b> | 130.90(84)      | 127.99(84)         | 245.77(84)         | 257.06(84)        |
| pbo-mqc-nlogencdr       | 84   | 362.20(84)      | 353.14(84)   | 353.14(84)        | 24.48(84)       | <b>17.19(84)</b>  | 58.42(84)       | 124.43(84)         | 140.44(84)         | 160.73(84)        |
| pbo-routing             | 15   | <b>0.46(15)</b> | 2.14(15)     | 2.14(15)          | 3.89(15)        | 6.59(15)          | 4.88(15)        | 6.72(15)           | 5.42(15)           | 6.73(15)          |
| protein-ins             | 12   | 2626.34(10)     | 2162.13(9)   | 2162.13(9)        | 476.03(12)      | 552.46(12)        | 360.76(12)      | 284.63(12)         | <b>234.31(12)</b>  | 333.85(12)        |
| Total                   | 504  | 374             | 376          | 376               | 387             | 394               | 404             | 415                | 425                | <b>428</b>        |
|                         |      | 69.6%           | 70.9%        | 70.9%             | 72.5%           | 76.0%             | 77.5%           | 81.4%              | 81.7%              | <b>83.6%</b>      |
| <b>WPMS-Industrial</b>  |      |                 |              |                   |                 |                   |                 |                    |                    |                   |
| haplotyping-pedigrees   | 100  | 16.01(22)       | 292.29(25)   | 242.08(90)        | 154.12(92)      | 142.59(92)        | 199.61(96)      | 86.67(95)          | 202.70(98)         | <b>176.02(98)</b> |
| timetabling             | 26   | 1017.27(8)      | 720.89(8)    | 651.84(8)         | 1008.59(9)      | <b>430.67(9)</b>  | 1544.43(9)      | 931.17(8)          | 932.77(8)          | 1438.59(9)        |
| upgradeability-problem  | 100  | 19.19(100)      | 19.67(100)   | <b>15.27(100)</b> | 96.88(100)      | 375.70(100)       | 99.51(100)      | 365.70(100)        | 97.83(100)         | 371.96(100)       |
| Total                   | 226  | 130             | 133          | 198               | 201             | 201               | 205             | 203                | 206                | <b>207</b>        |
|                         |      | 50.9%           | 51.9%        | 73.6%             | 75.5%           | 75.5%             | 76.9%           | 75.3%              | 76.3%              | <b>77.5%</b>      |
| <b>Total Industrial</b> |      |                 |              |                   |                 |                   |                 |                    |                    |                   |
|                         | 730  | 504             | 509          | 574               | 588             | 595               | 609             | 618                | 631                | <b>635</b>        |
|                         |      | 66.1%           | 67.3%        | 71.4%             | 73.0%           | 75.9%             | 77.4%           | 80.2%              | 80.7%              | <b>82.5%</b>      |
| <b>PMS-Crafted</b>      |      |                 |              |                   |                 |                   |                 |                    |                    |                   |
| frb                     | 25   | 0.00(0)         | 0.00(0)      | 0.00(0)           | 0.00(0)         | 0.00(0)           | 0.00(0)         | 0.00(0)            | 0.00(0)            | 0.00(0)           |
| job-shop                | 3    | 68.31(3)        | 63.37(3)     | 63.37(3)          | 50.84(3)        | 53.11(3)          | 51.29(3)        | <b>49.87(3)</b>    | 88.41(3)           | 58.75(3)          |
| maxclicque-random       | 96   | 478.65(76)      | 471.97(76)   | 471.97(76)        | 435.07(77)      | 533.97(78)        | 423.89(83)      | 392.04(82)         | 565.44(87)         | <b>512.17(89)</b> |
| maxclicque-structured   | 62   | 779.71(21)      | 592.07(21)   | 592.07(21)        | 511.52(22)      | 783.37(24)        | 650.87(25)      | 838.38(26)         | 802.09(26)         | <b>876.90(27)</b> |
| maxone-3sat             | 80   | 16.42(80)       | 18.11(80)    | 18.11(80)         | 10.90(80)       | 9.39(80)          | 5.34(80)        | <b>5.29(80)</b>    | 6.65(80)           | 6.47(80)          |
| maxone-structured       | 60   | 156.20(58)      | 89.15(59)    | 89.15(59)         | 14.42(60)       | 32.46(60)         | 13.11(60)       | 46.68(60)          | <b>9.24(60)</b>    | 50.43(60)         |
| min-enc-kbree           | 42   | 1607.47(4)      | 707.46(4)    | 707.46(4)         | 2057.16(5)      | 2086.38(5)        | 2254.64(6)      | 1777.41(5)         | <b>921.09(6)</b>   | 1906.23(5)        |
| pseudo-miplib           | 4    | 122.23(4)       | 93.95(4)     | 93.95(4)          | 109.21(4)       | 64.35(4)          | <b>27.35(4)</b> | 37.81(4)           | 54.30(4)           | 34.23(4)          |
| Total                   | 372  | 246             | 247          | 247               | 251             | 254               | 261             | 260                | 266                | <b>268</b>        |
|                         |      | 64.9%           | 65.1%        | 65.1%             | 65.9%           | 66.5%             | 67.6%           | 67.4%              | 68.4%              | <b>68.5%</b>      |
| <b>WPMS-Crafted</b>     |      |                 |              |                   |                 |                   |                 |                    |                    |                   |
| auc-paths               | 86   | 0.00(0)         | 376.67(1)    | 370.25(1)         | 706.38(33)      | 546.57(33)        | 1019.93(74)     | 948.64(75)         | 435.71(82)         | <b>271.16(82)</b> |
| auc-scheduling          | 84   | 0.00(0)         | 133.39(51)   | 136.80(51)        | 71.37(84)       | 69.59(84)         | 2.73(84)        | 1.66(84)           | 1.25(84)           | <b>1.07(84)</b>   |
| min-enc-planning        | 56   | 491.39(25)      | 141.80(38)   | 138.29(38)        | 3.24(56)        | 2.47(56)          | 0.75(56)        | <b>0.74(56)</b>    | 0.80(56)           | 0.05(1)           |
| min-enc-warehouses      | 18   | 11.36(1)        | 3.36(1)      | 3.43(1)           | 0.08(1)         | 0.08(1)           | 0.04(1)         | 0.05(1)            | <b>1.227.86(2)</b> | 0.05(1)           |
| pseudo-miplib           | 12   | 2936.02(3)      | 1533.63(3)   | 1487.40(3)        | 13.72(3)        | 11.81(3)          | 471.93(4)       | 990.65(5)          | 299.53(4)          | <b>695.31(5)</b>  |
| random-net              | 74   | 0.00(0)         | 0.00(0)      | 0.00(0)           | 1041.56(15)     | 1943.41(17)       | 1633.69(33)     | <b>1529.94(33)</b> | 3369.18(17)        | 3151.03(15)       |
| wcap-spot5-dir          | 21   | 187.40(9)       | 310.45(10)   | 248.53(12)        | 420.10(14)      | 498.10(14)        | 495.17(14)      | 470.92(14)         | <b>8.27(14)</b>    | 113.90(14)        |
| wcap-spot5-log          | 21   | 1067.69(8)      | 469.38(10)   | 120.60(6)         | 100.07(13)      | 59.16(13)         | 133.75(14)      | 48.91(13)          | 23.84(14)          | <b>16.50(14)</b>  |
| Total                   | 372  | 46              | 114          | 115               | 219             | 221               | 280             | <b>281</b>         | 273                | 271               |
|                         |      | 19.5%           | 31.9%        | 32.5%             | 52.2%           | 52.6%             | 62.9%           | <b>63.4%</b>       | 62.0%              | 62.0%             |
| <b>Total Crafted</b>    |      |                 |              |                   |                 |                   |                 |                    |                    |                   |
|                         | 744  | 292             | 361          | 362               | 470             | 475               | 541             | <b>541</b>         | 539                | 539               |
|                         |      | 42.2%           | 48.5%        | 48.8%             | 59.1%           | 59.5%             | 65.2%           | <b>65.4%</b>       | 65.2%              | 65.3%             |
| <b>Total (W)PMS</b>     |      |                 |              |                   |                 |                   |                 |                    |                    |                   |
|                         | 1474 | 796             | 870          | 936               | 1058            | 1070              | 1150            | 1159               | 1170               | 1174              |
|                         |      | 54.1%           | 57.9%        | 60.1%             | 66.1%           | 67.7%             | 71.3%           | 72.8%              | 72.9%              | <b>73.9%</b>      |

Table 2. Experimental results of best *wpm2* version compared with other solvers

(a) Partial Industrial

| Instance set             | #   | <i>wpm2_shua</i> | <i>bincd2</i>     | <i>qms0.2lg2</i>  | <i>pubo2.1</i> | <i>shinms</i> | <i>wpm1</i>    | <i>tlp</i>        |
|--------------------------|-----|------------------|-------------------|-------------------|----------------|---------------|----------------|-------------------|
| ses                      | 7   | 514.14(1)        | 453.22(1)         | 3154.99(1)        | 0.00(0)        | 0.00(0)       | 3073.19(1)     | <b>1310.95(3)</b> |
| bcp-fir                  | 59  | 57.36(58)        | 44.09(58)         | 108.17(56)        | 68.10(56)      | 13.54(22)     | 10.87(57)      | <b>62.86(59)</b>  |
| bcp-lpp-yrRa_simp        | 17  | 100.95(16)       | 170.49(16)        | <b>366.14(17)</b> | 174.98(16)     | 40.66(16)     | 70.27(16)      | 666.94(6)         |
| bcp-lpp-yrRa_ls          | 3   | 143.57(16)       | 143.57(16)        | 143.57(16)        | 143.57(16)     | 28.54(58)     | 21.54(58)      | 28.54(58)         |
| bcp-mg                   | 64  | 912.70(34)       | <b>213.47(38)</b> | 451.30(30)        | 96.14(26)      | 281.37(22)    | 1053.47(7)     | 855.96(37)        |
| bcp-mg                   | 40  | 578.75(39)       | 1.15(40)          | <b>0.15(40)</b>   | 0.57(40)       | 0.60(40)      | 8.54(40)       | 769.27(29)        |
| bcp-syn                  | 74  | 78.22(42)        | 28.56(43)         | 283.64(35)        | 21.82(39)      | 86.98(33)     | 59.30(45)      | <b>18.95(71)</b>  |
| circuit-trace-compaction | 4   | 129.10(4)        | 109.31(4)         | <b>45.01(4)</b>   | 200.11(2)      | 52.22(4)      | 118.43(4)      | 6921.80(1)        |
| haplotype-assembly       | 6   | 65.94(5)         | 728.30(5)         | 153.22(5)         | 9.09(5)        | 0.00(0)       | <b>2.63(5)</b> | 2124.72(5)        |
| pbo-mqc-nencdr           | 84  | 257.06(84)       | 278.45(84)        | <b>58.78(84)</b>  | 222.19(68)     | 145.71(84)    | 804.25(54)     | 1109.89(6)        |
| pbo-mqc-nlogencdr        | 84  | 140.44(84)       | 78.58(84)         | <b>23.69(84)</b>  | 71.86(82)      | 180.37(79)    | 403.38(55)     | 508.21(6)         |
| pbo-routing              | 15  | 6.73(15)         | <b>1.14(15)</b>   | 3.61(15)          | 27.67(15)      | 4.80(15)      | 1.75(15)       | 19.68(15)         |
| protein-ins              | 12  | 333.85(12)       | 314.09(3)         | <b>128.58(12)</b> | 0.11(1)        | 206.51(4)     | 1812.03(3)     | 2.72(1)           |
| Total                    | 504 | <b>428</b>       | 423               | 418               | 374            | 353           | 330            | 239               |
| Mean ratio               |     | <b>83.6%</b>     | 78.2%             | 83.0%             | 66.3%          | 63.6%         | 68.3%          | 48.9%             |

(b) Weighted Partial Industrial

| Instance set           | #   | <i>wpm2_shua</i>  | <i>wpm1</i>        | <i>pubo2.1</i> | <i>bincd2</i> | <i>marhs</i> | <i>tlp</i>  | <i>shinms</i> |
|------------------------|-----|-------------------|--------------------|----------------|---------------|--------------|-------------|---------------|
| haplotype-pedigrees    | 100 | <b>176.02(98)</b> | 212.75(93)         | 123.00(87)     | 544.80(73)    | 1089.24(39)  | 1892.16(18) | 1203.99(47)   |
| timetabling            | 26  | 1438.59(0)        | <b>1347.39(11)</b> | 671.45(7)      | 168.55(8)     | 1249.85(6)   | 0.00(0)     | 2261.00(5)    |
| upgradeability-problem | 100 | 371.96((100)      | <b>4.57(100)</b>   | 32.67(100)     | 76.40(100)    | 13.41(100)   | 19.26(100)  | 0.00(0)       |
| Total                  | 226 | <b>207</b>        | 204                | 194            | 181           | 145          | 118         | 52            |
| Mean ratio             |     | 77.5%             | <b>78.4%</b>       | 71.3%          | 67.9%         | 54.0%        | 39.3%       | 22.1%         |

(c) Partial Crafted

| Instance set         | #   | <i>tlp</i>        | <i>akmsJs</i>   | <i>qms0.21</i>    | <i>shinms</i> | <i>wpm2_shua</i> | <i>bincd2</i> | <i>pubo2.1</i> | <i>wpm1</i> |
|----------------------|-----|-------------------|-----------------|-------------------|---------------|------------------|---------------|----------------|-------------|
| frb                  | 25  | 1152.97(13)       | 159.47(5)       | <b>346.73(25)</b> | 43.52(23)     | 0.00(0)          | 0.00(0)       | 151.16(15)     | 0.00(0)     |
| job-shop             | 3   | 0.00(0)           | 0.00(0)         | <b>41.51(3)</b>   | 36.44(3)      | 58.75(3)         | 100.43(3)     | 93.17(1)       | 835.23(3)   |
| maxclique-random     | 92  | 45.13(96)         | 269.93(85)      | 339.94(76)        | 512.30(81)    | 85.20(89)        | 19.45(64)     | 662.30(39)     | 37.50(39)   |
| maxclique-structured | 62  | <b>281.00(60)</b> | 281.00(60)      | 281.00(60)        | 281.00(60)    | 87.50(80)        | 10.00(80)     | 37.50(80)      | 37.50(80)   |
| maxcut-3sat          | 80  | 13.12(80)         | <b>0.47(80)</b> | 198.39(80)        | 694.52(78)    | 6.47(80)         | 8.28(80)      | 36.97(63)      | 97.24(80)   |
| maxcut-structured    | 60  | 337.60(59)        | 482.20(38)      | <b>6.35(60)</b>   | 3.53(59)      | 50.43(60)        | 7.71(60)      | 718.41(42)     | 718.41(42)  |
| min-enc-kbtree       | 42  | <b>162.89(42)</b> | 3199.18(34)     | 248.02(6)         | 513.97(5)     | 1906.23(5)       | 274.79(6)     | 306.58(2)      | 2469.35(6)  |
| pseudo-miplib        | 4   | 34.12(4)          | 258.91(3)       | <b>1.84(4)</b>    | 3.44(4)       | 34.23(4)         | 48.74(4)      | 93.19(4)       | 163.61(4)   |
| Total                | 372 | <b>332</b>        | 297             | 291               | 271           | 268              | 245           | 228            | 207         |
| Mean ratio           |     | 76.5%             | 63.2%           | <b>81.1%</b>      | 77.0%         | 68.5%            | 65.3%         | 59.3%          | 58.3%       |

(d) Weighted Partial Crafted

| Instance set      | #   | <i>tlp</i>      | <i>wpm1</i>     | <i>wpm2_shua</i> | <i>shinms</i>     | <i>akmsJs</i> | <i>pubo2.1</i>  | <i>marhs</i> | <i>bincd2</i> |
|-------------------|-----|-----------------|-----------------|------------------|-------------------|---------------|-----------------|--------------|---------------|
| auc-paths         | 86  | <b>0.49(86)</b> | 108.14(63)      | 271.15(82)       | 317.62(84)        | 2.57(86)      | 110.67(19)      | 35.41(86)    | 1414.73(12)   |
| auc-scheduling    | 84  | <b>0.38(84)</b> | 1.56(84)        | 1.07(84)         | 5.81(84)          | 68.27(84)     | 7.65(81)        | 965.10(78)   | 141.77(81)    |
| mini-enc-planning | 36  | 296.32(56)      | 2.61(55)        | 0.80(56)         | 8.13(52)          | 143.21(40)    | <b>0.46(56)</b> | 439.18(31)   | 32.74(34)     |
| maxclique-random  | 92  | 45.13(96)       | 269.93(85)      | 339.94(76)       | 512.30(81)        | 85.20(89)     | 19.45(64)       | 662.30(39)   | 37.50(39)     |
| pseudo-miplib     | 12  | 82.82(3)        | 157.33(4)       | 695.31(5)        | <b>127.99(5)</b>  | 0.26(2)       | 3.97(3)         | 0.03(1)      | 1072.69(4)    |
| random-net        | 74  | 532.74(59)      | <b>4.03(70)</b> | 3151.03(15)      | 0.00(0)           | 4060.60(8)    | 42.15(35)       | 2770.76(10)  | 0.00(0)       |
| wcsp-spot5-dir    | 21  | 42.88(18)       | 37.70(13)       | 113.90(14)       | <b>743.63(21)</b> | 1555.46(6)    | 61.89(8)        | 101.11(6)    | 127.73(12)    |
| wcsp-spot5-log    | 21  | 322.93(8)       | 399.59(14)      | 16.50(14)        | <b>200.73(17)</b> | 108.70(5)     | 357.32(6)       | 299.32(13)   | 299.32(13)    |
| Total             | 372 | <b>332</b>      | 318             | 271              | 264               | 233           | 222             | 219          | 177           |
| Mean ratio        |     | <b>78.6%</b>    | 77.4%           | 62.0%            | 64.8%             | 45.3%         | 54.4%           | 41.6%        | 45.6%         |

**Table 3.** Summary of solved instances and mean ratio % for best solvers

| solvers                    | pms           | wpms          | Ind.          | pms           | wpms          | Cra.          | Total         |
|----------------------------|---------------|---------------|---------------|---------------|---------------|---------------|---------------|
|                            | <b>428</b>    | <b>207</b>    | <b>635</b>    | 268           | 271           | 539           | <b>1174</b>   |
| <i>wpm2<sub>shua</sub></i> | <b>83.6 %</b> | 77.5 %        | <b>82.5 %</b> | 68.5 %        | 62.0 %        | 65.3 %        | <b>73.9 %</b> |
|                            | 330           | 204           | 534           | 207           | 318           | 525           | 1059          |
| <i>wpm1</i>                | 68.3 %        | <b>78.4 %</b> | 70.2 %        | 58.3 %        | 77.4 %        | 67.9 %        | 69.0 %        |
|                            | 423           | 181           | 604           | 245           | 177           | 422           | 1026          |
| <i>bincd2</i>              | 78.2 %        | 67.9 %        | 76.3 %        | 65.3 %        | 45.6 %        | 55.5 %        | 65.9 %        |
|                            | 239           | 118           | 357           | <b>332</b>    | <b>332</b>    | <b>664</b>    | 1021          |
| <i>ilp</i>                 | 48.9 %        | 39.3 %        | 47.1 %        | 76.5 %        | <b>78.6 %</b> | <b>77.6 %</b> | 62.3 %        |
|                            | 374           | 194           | 568           | 228           | 222           | 450           | 1018          |
| <i>pwbo2.1</i>             | 66.3 %        | 71.3 %        | 67.2 %        | 59.3 %        | 54.5 %        | 56.9 %        | 62.1 %        |
|                            | 353           | 52            | 405           | 271           | 264           | 535           | 940           |
| <i>shinms</i>              | 63.6 %        | 22.1 %        | 55.8 %        | 77.0 %        | 64.8 %        | 70.9 %        | 63.4 %        |
|                            | 418           |               |               | 291           |               |               |               |
| <i>qms</i>                 | 83.0 %        |               |               | <b>81.1 %</b> |               |               |               |

Regarding our three approaches for optimizing the covers, we can see that by optimizing with subsetsum (*wpm2<sub>shla</sub>*) we solve some additional instances in all categories having the best improvement in WPMS industrial with 18 more and in WPMS crafted with 106 more. It is important to highlight that optimizing covers with subsetsum, instead of applying the subsetsum as in the original WPM2 algorithm, leads to a total improvement of 134 additional solved instances, with respect to *wpm2<sub>sh</sub>*.

Optimizing all covers by refining the upper bound (*wpm2<sub>shua</sub>*), we get an additional boost with respect to *wpm2<sub>shla</sub>*. We can see that we solve some additional instances in all categories. We get the best improvement for PMS industrial, solving 34 additional instances, and for WPMS crafted, 50 more. Notice that the overall increase with respect to *wpm2<sub>sh</sub>* is of 238 additional solved instances.

Binary search (*wpm2<sub>shba</sub>*) improves 10 instances in WPMS crafted with respect to *wpm2<sub>shua</sub>*. But the global performance with respect to *wpm2<sub>sh</sub>*, 223, is not as good as only refining the upper bound (*wpm2<sub>shua</sub>*).

Optimizing only covers that contain the last unsatisfiable core solves almost the same instances as optimizing all covers but improves the average running time in the WPMS industrial family *upgradeability-problem* by a factor of 4.

Table 2 shows the results of our second experiment where we compare the best variation and implementation of our new WPM2 algorithm (*wpm2<sub>shua</sub>*) with several solvers. In particular, we compare with the best two solvers for the PMS and WPMS industrial and crafted instances of the 2012 MaxSAT Evaluation: PMS industrial (*qms0.21g2*, *pwbo2.1*), WPMS industrial (*pwbo2.1* [31, 32], *wpm1* [2]<sup>5</sup>), PMS crafted (*qms0.21* [21], *akms<sub>ls</sub>* [22] and WPMS crafted (*wpm1*, *shinms* [20]). We also compare with three additional MaxSAT solvers: *bincd2*, which is the new version of the BINCD algorithm [19] described in [33], with the best configuration reported by authors, *maxhs* from [14], which consists in an hybrid SAT-ILP approach, and *ilp*, which translates WPMS into ILP and applies the MIP solver IBM-CPLEX studio124 [7].

<sup>5</sup> We present in this paper a version implemented on top of the Yices SMT solver.

Table 2(a) presents the results for the PMS industrial instances. Our *wpm2<sub>shua</sub>* is the first one in solved instances with 428 and mean ratio with 93.6%, closely followed by *bincd2* and *qms0.21g2*.

Table 2(b) presents the results for the WPMS industrial instances. As we can see, our *wpm2<sub>shua</sub>* and *wpm1* dominate this category with 207 and 204 solved instances and 77.5% and 78.4% mean ratio, resp.

As a summary of industrial instances, we can conclude that our *wpm2<sub>shua</sub>* is the best performing solver with a total of 635 solved instances, followed by *bincd2* with a total of 604. We do not have results for any version of *qms* since it only works for PMS instances. The closest solver to the search scheme of *qms* would be *shinms* but it does not perform well for WPMS industrial.

Table 2(c) presents the results for the PMS crafted instances. The *ilp* approach solves 332 of 372 instances, 35 more than *akms<sub>ls</sub>*. This is remarkable since branch and bound solvers, like *akms<sub>ls</sub>*, have always dominated this category since 2006. PMS solver *qms0.21* is the third in solved instances but the first in mean ratio with 81.1%. Our *wpm2<sub>shua</sub>* is the fifth in solved instances with 268 and the fourth in mean ratio with 68.5%.

Table 2(d) presents the results for the WPMS crafted instances. Again, the *ilp* approach is the best one, solving 332 of 372 instances, 14 more than the second one, *wpm1*. Our *wpm2<sub>shua</sub>* is the third in solved instances with 271 and the fourth in mean ratio with 62.0%.

As a summary of crafted instances, we can conclude that *ilp* is the best performing approach, and our *wpm2<sub>shua</sub>* is the second in total solved instances.

In Table 3 we can see a summary of the solved instances and mean ratio per category for best solvers. We recall that all solvers accept weights except *qms* that is only for PMS. Our *wpm2<sub>shua</sub>* is the first in solved instances for both PMS industrial and WPMS industrial. In crafted categories it is the second in total solved instances. However, for both PMS crafted and WPMS crafted categories *ilp* is the first in solved instances. We can conclude that our *wpm2<sub>shua</sub>* is the most robust solver across all four PMS and WPMS industrial and crafted categories, followed by *wpm1* and *bincd2*.

## 6 Conclusions and Future Work

From the experimental evaluation, we conclude that the new WPM2 solver is the best performing solver for PMS and WPMS industrial instances and the best on the union of PMS and WPMS industrial and crafted instances. In particular, we have shown that solving to optimality the subformulas defined by covers really works in practice. As future work, we will study how to improve the interaction with the optimization of the subformulas. A portfolio that selects the most suitable optimization approach depending on the structure of the subformula seems another way of achieving additional speed-ups. Finally, we have also shown that SMT technology is an underlying efficient technology for solving the MaxSAT problem.

## References

1. Ansótegui, C., Bofill, M., Palahí, M., Suy, J., Villaret, M.: A Proposal for Solving Weighted CSPs with SMT. In: Proceedings of the 10th International Workshop on Constraint Modelling and Reformulation (ModRef 2011), pp. 5–19 (2011)
2. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving sat-based weighted maxsat solvers. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 86–101. Springer, Heidelberg (2012)
3. Ansótegui, C., Bonet, M.L., Levy, J.: On solving MaxSAT through SAT. In: Proc. of the 12th Int. Conf. of the Catalan Association for Artificial Intelligence (CCIA 2009), pp. 284–292 (2009)
4. Ansótegui, C., Bonet, M.L., Levy, J.: Solving (weighted) partial MaxSAT through satisfiability testing. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 427–440. Springer, Heidelberg (2009)
5. Ansótegui, C., Bonet, M.L., Levy, J.: A new algorithm for weighted partial maxsat. In: Proc. the 24th National Conference on Artificial Intelligence (AAAI 2010) (2010)
6. Ansótegui, C., Bonet, M.L., Levy, J.: Sat-based maxsat algorithms. Artif. Intell. 196, 77–105 (2013)
7. Ansótegui, C., Gabas, J.: Solving maxsat with mip. In: CPAIOR (2013)
8. Bailleux, O., Bouffkhad, 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)
9. Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2010), <http://www.SMT-LIB.org>
10. Berre, D.L.: Sat4j, a satisfiability library for java (2006), <http://www.sat4j.org>
11. Borchers, B., Furman, J.: A two-phase exact algorithm for max-sat and weighted max-sat problems. J. Comb. Optim. 2(4), 299–306 (1998)
12. Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R., Stenico, C.: Satisfiability modulo the theory of costs: Foundations and applications. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 99–113. Springer, Heidelberg (2010)
13. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press (2009)
14. 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)
15. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. JSAT 2(1-4), 1–26 (2006)
16. Fu, Z., Malik, S.: On solving the partial MAX-SAT problem. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 252–265. Springer, Heidelberg (2006)
17. Heras, F., Larrosa, J., Oliveras, A.: MiniMaxSat: A new weighted Max-SAT solver. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 41–55. Springer, Heidelberg (2007)
18. Heras, F., Larrosa, J., Oliveras, A.: Minimaxsat: An efficient weighted max-sat solver. J. Artif. Intell. Res (JAIR) 31, 1–32 (2008)
19. Heras, F., Morgado, A., Marques-Silva, J.: Core-guided binary search algorithms for maximum satisfiability. In: Proc. the 25th National Conference on Artificial Intelligence (AAAI 2011) (2011)

20. Honjyo, K., Tanjo, T.: Shinmaxsat, a Weighted Partial Max-SAT solver inspired by MiniSat+, Information Science and Technology Center, Kobe University
21. Koshimura, M., Zhang, T., Fujita, H., Hasegawa, R.: Qmaxsat: A partial max-sat solver. *JSAT* 8(1/2), 95–100 (2012)
22. Kügel, A.: Improved exact solver for the weighted max-sat problem (to appear)
23. Larrosa, J., Heras, F., de Givry, S.: A logical approach to efficient max-sat solving. *Artif. Intell.* 172(2-3), 204–233 (2008)
24. Li, C.M., Manyà, F., Mohamedou, N., Planes, J.: Exploiting cycle structures in Max-SAT. In: Kullmann, O. (ed.) *SAT 2009*. LNCS, vol. 5584, pp. 467–480. Springer, Heidelberg (2009)
25. Li, C.M., Manyà, F., Planes, J.: New inference rules for Max-SAT. *J. Artif. Intell. Res (JAIR)* 30, 321–359 (2007)
26. Lin, H., Su, K.: Exploiting inference rules to compute lower bounds for Max-SAT solving. In: *IJCAI 2007*, pp. 2334–2339 (2007)
27. Lin, H., Su, K., Li, C.M.: Within-problem learning for efficient lower bound computation in Max-SAT solving. In: *Proc. the 23rd National Conference on Artificial Intelligence (AAAI 2008)*, pp. 351–356 (2008)
28. Manquinho, V., Marques-Silva, J., Planes, J.: Algorithms for weighted boolean optimization. In: Kullmann, O. (ed.) *SAT 2009*. LNCS, vol. 5584, pp. 495–508. Springer, Heidelberg (2009)
29. Manquinho, V.M., Martins, R., Lynce, I.: Improving unsatisfiability-based algorithms for boolean optimization. In: Strichman, O., Szeider, S. (eds.) *SAT 2010*. LNCS, vol. 6175, pp. 181–193. Springer, Heidelberg (2010)
30. Marques-Silva, J., Argelich, J., Graça, A., Lynce, I.: Boolean lexicographic optimization: algorithms & applications. *Ann. Math. Artif. Intell.* 62(3-4), 317–343 (2011)
31. Martins, R., Manquinho, V.M., Lynce, I.: Exploiting cardinality encodings in parallel maximum satisfiability. In: *ICTAI*, pp. 313–320 (2011)
32. Martins, R., Manquinho, V., Lynce, I.: Clause sharing in parallel MaxSAT. In: Hamadi, Y., Schoenauer, M. (eds.) *LION 2012*. LNCS, vol. 7219, pp. 455–460. Springer, Heidelberg (2012)
33. Morgado, A., Heras, F., Marques-Silva, J.: Improvements to core-guided binary search for MaxSAT. In: Cimatti, A., Sebastiani, R. (eds.) *SAT 2012*. LNCS, vol. 7317, pp. 284–297. Springer, Heidelberg (2012)
34. Nieuwenhuis, R., Oliveras, A.: On SAT modulo theories and optimization problems. In: Biere, A., Gomes, C.P. (eds.) *SAT 2006*. LNCS, vol. 4121, pp. 156–169. Springer, Heidelberg (2006)
35. Sebastiani, R.: Lazy Satisfiability Modulo Theories. *Journal on Satisfiability, Boolean Modeling and Computation* 3(3-4), 141–224 (2007)
36. Silva, J.P.M., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Computers* 48(5), 506–521 (1999)

# MinSAT versus MaxSAT for Optimization Problems

Josep Argelich<sup>1</sup>, Chu-Min Li<sup>2</sup>, Felip Manyà<sup>3</sup>, and Zhu Zhu<sup>2</sup>

<sup>1</sup> Dept. of Computer Science, Universitat de Lleida, Lleida, Spain

<sup>2</sup> MIS, Université de Picardie Jules Verne, Amiens, France

<sup>3</sup> Artificial Intelligence Research Institute (IIIA, CSIC), Bellaterra, Spain

**Abstract.** Despite their similarities, MaxSAT and MinSAT use different encodings and solving techniques to cope with optimization problems. In this paper we describe a new weighted partial MinSAT solver, define original MinSAT encodings for relevant combinatorial problems, propose a new testbed for evaluating MinSAT, report on an empirical investigation comparing MinSAT with MaxSAT, and provide new insights into the duality between MinSAT and MaxSAT.

## 1 Introduction

MinSAT is the problem of finding a truth assignment that minimizes the number of satisfied clauses in a CNF formula, and MaxSAT is the problem of finding a truth assignment that maximizes the number of satisfied clauses. When hard and soft clauses, and weights are considered, the problem is known as weighted partial MinSAT/MaxSAT.

The promising results on MaxSAT as a generic approach to solving combinatorial optimization problems [6,16] led us to investigate the opportunities that MinSAT offers in optimization [19,20]. At first sight, it may seem that MaxSAT and MinSAT are so close that it does not pay off to devote efforts to MinSAT, but this is not completely certain. In [19,20], branch-and-bound MinSAT solvers apply upper bounding techniques not applicable in MaxSAT solvers, and this provides a competitive advantage to MinSAT, which solves MaxClique and combinatorial auction instances (that are beyond the reach of current MaxSAT solvers) even faster than using dedicated algorithms. Interestingly, the MinSAT and MaxSAT encodings of these problems are almost identical (the only difference is that the literals in the soft clauses, which are unitary, have opposite polarity). The superiority of MinSAT in [19,20] is due to the MinSAT solving techniques, not to the encoding. The compared MinSAT and MaxSAT solvers use the same data structures but implement different techniques. Since performance depends on both solvers and encodings, our first goal is to define suitable MinSAT encodings of NP-hard problems, and improve the performance by using more efficient encodings.

All the interesting genuine MinSAT encodings defined so far, except for random Min-kSAT instances, only contain unit clauses in the soft part. In the literature we can find experimental investigations solving instances with non-unit soft clauses, but the optimization problem that is actually being solved is a genuine or transformed MaxSAT instance. The first contribution of the paper is the definition of genuine MinSAT encodings of practical optimization problems, containing non-unit soft clauses, that are completely different from their MaxSAT counterparts, and a comparison that provides empirical evidence of the gains that could be achieved. More specifically, we define

novel encodings from weighted MaxCSP to MinSAT that fulfill our requirements. We use weighted MaxCSP as a good source of benchmarks for MinSAT solvers, but not with the aim of competing with weighted MaxCSP solvers in their own territory.

The second contribution is the incorporation of the inference rules defined in [17] into MinSatz [19,20]. It turns out that MaxSAT inference rules can be applied to MinSAT too, because they preserve the cost distribution over all the models. As a result, we have implemented a new version of MinSatz that incorporates inference rules, and provided empirical evidence that inference rules may produce significant speedups.

Last but not least, we would like to highlight that MinSAT solving is a novel and emerging technology with a remarkable potential in optimization. It allows to look at problems from a different perspective because, in contrast to existing Boolean optimization approaches, it works by maximizing the number of violated constraints instead of working by minimizing that number. This fact leads to the definition of different encodings and solving techniques that exploit the duality between MinSAT and MaxSAT.

*Related Work:* The work on MinSAT for solving optimization problems may be divided into three categories: (I) Transformation between MinSAT and MaxSAT: Reductions from MinSAT to PMaxSAT were defined in [18], but these reductions do not generalize to WPMInSAT. This drawback was overcome with the definition of the natural encoding [13], which was improved in [23]. Reductions of WPMInSAT to Group MaxSAT were evaluated in [11]. (II) Branch-and-bound solvers: The only existing WPMInSAT solver, MinSatz [19,20], is based on MaxSatz [17], and implements upper bounds that exploit clique partition algorithms and MaxSAT technology. (III) SAT-based solvers: There exist two WPMInSAT solvers of this class [2,11]. The main difference with SAT-based MaxSAT solvers lies in the way of relaxing soft clauses.

## 2 Preliminaries

A weighted clause is a pair  $(c, w)$ , where  $c$  is a disjunction of literals and  $w$ , its weight, is a natural number or infinity. A clause is hard if its weight is infinity (for simplicity, we omit infinity weights); otherwise it is soft. A Weighted Partial MinSAT (MaxSAT) instance is a multiset of hard clauses and weighted soft clauses. A truth assignment assigns to each propositional variable either 0 or 1. The Weighted Partial MinSAT (MaxSAT) problem, or WPMInSAT (WPMMaxSAT), for an instance  $\phi$  consists in finding an assignment in which the sum of weights of the satisfied (falsified) soft clauses is minimal, and all the hard clauses are satisfied. The Weighted MinSAT (MaxSAT) problem, or WMinSAT (WMaxSAT), is the WPMInSAT (WPMMaxSAT) problem when there are no hard clauses. The Partial MinSAT (MaxSAT) problem, or PMinSAT (PMaxSAT), is the WPMInSAT (WPMMaxSAT) problem when all the soft clauses have the same weight. The MinSAT (MaxSAT) problem is the Partial MinSAT (MaxSAT) problem when there are no hard clauses.

A CSP instance is a triple  $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ , where  $\mathcal{X} = \{X_1, \dots, X_n\}$  is a set of variables,  $\mathcal{D} = \{d(X_1), \dots, d(X_n)\}$  is a set of finite domains, and  $\mathcal{C} = \{C_1, \dots, C_m\}$  is a set of constraints. Each  $C_i = \langle S_i, R_i \rangle$  in  $\mathcal{C}$  is a relation  $R_i$  over a subset of  $S_i = \{X_{i_1}, \dots, X_{i_k}\} \subseteq \mathcal{X}$ , called the scope.  $R_i$  may be represented extensionally as a subset of the Cartesian product  $d(X_{i_1}) \times \dots \times d(X_{i_k})$ . The tuples belonging to  $R_i$  represent the



allowed values and are called goods, and the rest of tuples represent the forbidden values and are called nogoods. An assignment  $v$  for a CSP instance  $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$  is a mapping that assigns to every variable  $X_i \in \mathcal{X}$  an element  $v(X_i) \in d(X_i)$ . It satisfies a constraint  $\langle \{X_{i_1}, \dots, X_{i_k}\}, R_i \rangle \in \mathcal{C}$  iff  $\langle v(X_{i_1}), \dots, v(X_{i_k}) \rangle \in R_i$ . The Constraint Satisfaction Problem (CSP) for an instance  $P$  consists in finding a satisfying assignment for  $P$ .

A Weighted MaxCSP (WMaxCSP) instance is defined as a triple  $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ , where  $\mathcal{X}$  and  $\mathcal{D}$  are variables and domains as in CSP, and  $\mathcal{C}$  is a set of weighted constraints. A weighted constraint  $\langle c, w \rangle$  is just a classical constraint  $c$  plus a weight  $w$  over natural numbers. The cost of an assignment  $v$  is the sum of the weights of all constraints violated by  $v$ . An optimal solution is an assignment with minimal cost. In the particular case where all the constraints have the same weight, it is called the MaxCSP problem.

### 3 Encodings from WMaxCSP to WPMInSAT

We define the MinSAT counterparts of the direct, minimal support and interval-based minimal support encodings from WMaxCSP to WPMInSAT defined in [5]. All these encodings are correct: solving a WMaxCSP instance is equivalent to solving the WPMInSAT instance derived by any of our encodings. Other variants and encodings could be tried. We selected them because they were superior in most of our tests. We assume binary constraints, but the direct encoding is valid for non-binary constraints too.

**Direct Encoding.** A Boolean variable  $x_i$  is associated with each value  $i$  that the CSP variable  $X$  can take. If the domain  $d(X)$  has size  $m$ , the ALO clause of  $X$  is  $x_1 \vee \dots \vee x_m$ , and ensures that  $X$  is given a value. The AMO clauses of  $X$  are the set of clauses  $\{\neg x_i \vee \neg x_j \mid i, j \in d(X), i < j\}$ , and ensure that  $X$  takes no more than one value.

**Definition 1.** *The direct encoding of a WMaxCSP instance  $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$  is the WPMInSAT instance that contains as hard clauses the ALO and AMO clauses for every CSP variable in  $\mathcal{X}$ , and a soft clause  $(\neg x_i \vee \neg y_j, w)$  for every good  $(X = i, Y = j)$  of every weighted constraint  $\langle C, w \rangle$  of  $\mathcal{C}$  with scope  $\{X, Y\}$ .*

Since PMinSAT maximizes the number of falsified soft clauses, the idea behind the PMinSAT encoding of a MaxCSP instance is that we want to force the violation of one soft clause for every satisfied constraint, because of that we negate the goods instead of the nogoods as is done for PMaxSAT encodings of MaxCSP instances in [5]. If an interpretation is compatible with a good of a constraint, then one soft clause of the clauses encoding the constraint is falsified, and if it is not compatible with any good, then all the clauses are satisfied. The idea is the same when weights are added.

*Example 1.* Let  $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$  be the WMaxCSP instance where  $\mathcal{X} = \{X, Y\}$ ,  $d(X) = d(Y) = \{1, 2, 3\}$ ,  $\mathcal{C} = \{X = Y\}$ , and the weight of  $X = Y$  is 3. The direct encoding from WMaxCSP to WPMInSAT is formed by the hard clauses encoding the ALO and AMO conditions of  $X$  and  $Y$ , and the soft clauses  $(\neg x_1 \vee \neg y_1, 3)$ ,  $(\neg x_2 \vee \neg y_2, 3)$ , and  $(\neg x_3 \vee \neg y_3, 3)$ . In contrast to the WPMInSAT direct encoding, which needs a quadratic number of binary soft clauses in the domain size for encoding the equality constraint, the WPMInSAT direct encoding just needs a linear number of binary clauses. So, this example shows that certain constraints can be more compactly defined with MinSAT.

**Minimal Support Encoding.** In the support encoding from CSP to SAT [12,9,22], besides the ALO and AMO clauses, there are clauses that encode the support for a value instead of encoding conflicts. The support for a value  $i$  of a variable  $X$  across a binary constraint with scope  $\{X, Y\}$  is the set of values of  $Y$  which allow  $X = i$ . If  $v_1, \dots, v_k$  are the supporting values of variable  $Y$  for  $X = i$ , the clause  $\neg x_i \vee y_{v_1} \vee \dots \vee y_{v_k}$  (called support clause) is added. There is one support clause for each value in the domain and for each pair of variables  $X, Y$  involved in a constraint. In the support encoding, a clause in each direction is used: one for the pair  $X, Y$  and one for  $Y, X$ , while in the minimal support encoding [5], for every constraint with scope  $\{X, Y\}$ , the added clauses are the support clauses either for all the domain values of  $X$  or for all the domain values of  $Y$ . The minimal support encoding does not maintain arc consistency through unit propagation but usually is better for SAT solvers with learning and for MaxSAT solvers [5]. In our MinSAT setting, we focus on the minimal support encoding because it greatly outperformed the support encoding in our tests. Before defining the encoding, we need to define the *negative support* for a value  $j$  of a CSP variable  $X$  across a binary constraint with scope  $\{X, Y\}$  as the set of values of  $Y$  which forbid  $X = j$ .

**Definition 2.** *The minimal support encoding of a WMaxCSP instance  $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$  is the WPMInSAT instance that contains as hard clauses the ALO and AMO clauses for every CSP variable in  $\mathcal{X}$  and, for every constraint with scope  $\{X, Y\}$  and weight  $w$ , either the soft clause  $(\neg x_i \vee y_{v_1} \vee \dots \vee y_{v_n}, w)$  for every value  $i \in d(X)$ , where  $v_1, \dots, v_n$  is the negative support for value  $i$ , or the soft clause  $(\neg y_j \vee x_{u_1} \vee \dots \vee x_{u_m}, w)$  for every value  $j \in d(Y)$ , where  $u_1, \dots, u_m$  is the negative support for value  $j$ .*

The main difference with the minimal support encoding from WMaxCSP to WPMInSAT [5] is that the support clauses now include the negative support instead of the positive support. This allows the violation of a soft clause for every satisfied constraint. In the experiments, we select the variable that produces support clauses of smaller size. To this end, we give a score of 16 to unit clauses, a score of 4 to binary clauses and a score of 1 to ternary clauses, and select the variable with higher sum of scores.

*Example 2.* Let  $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$  be the MaxCSP instance where  $\mathcal{X} = \{X, Y\}$ ,  $d(X) = d(Y) = \{1, 2, 3\}$ ,  $\mathcal{C} = \{X \neq Y\}$ , and the constraint weight is 2. The minimal support encoding from WMaxCSP to WPMInSAT is formed by the ALO and AMO hard clauses of  $X$  and  $Y$ , and the soft clauses  $(\neg x_1 \vee y_1, 2)$ ,  $(\neg x_2 \vee y_2, 2)$ , and  $(\neg x_3 \vee y_3, 2)$ . Notice that we could also define the minimal support encoding by replacing the previous soft clauses with  $(x_1 \vee \neg y_1, 2)$ ,  $(x_2 \vee \neg y_2, 2)$ , and  $(x_3 \vee \neg y_3, 2)$  if we add the support clauses for the domain values of  $Y$  instead of the domain values of  $X$ . The length of the soft clauses of the minimal support WPMInSAT encoding is linear in the domain size for the inequality constraint whereas in MinSAT all the soft clauses are binary.

If we replace  $\mathcal{C} = \{X \neq Y\}$  with  $\mathcal{C} = \{X \leq Y\}$ , we can derive the minimal support encoding containing the soft clauses  $(\neg x_1, 2)$ ,  $(\neg x_2 \vee y_1, 2)$ , and  $(\neg x_3 \vee y_1 \vee y_2, 2)$ . Interestingly, we derive a unit clause that can be very useful for applying inference rules. In the minimal support encoding from WMaxCSP to WPMInSAT no unit clause is derived. It contains the soft clauses  $(\neg x_2 \vee y_2 \vee y_3, 2)$ , and  $(\neg x_3 \vee y_3, 2)$ .

**Interval-Based Minimal Support Encoding.** An alternative for modeling a CSP variable  $X$  is the use of Boolean variables of the form  $x_i^{\geq}$  (instead of  $x = i$ ), assuming a total ordering on the domain. Variables of the form  $x_i^{\geq}$  are called regular variables, and their intended meaning is that  $x_i^{\geq}$  is true iff  $X \geq i$  [7,3,10]. For simplicity, we assume here that domains are subsets of natural numbers.

One advantage of introducing regular literals (i.e.; regular variables and their negations) is that the number of clauses needed for each CSP variable is linear rather than quadratic in the domain size, they may produce more compact encodings, and are particularly useful for dealing with large domains. We focus on the interval-based minimal support encoding [5] because it outperformed other encodings with regular literals.

Given a MinSAT support clause  $\neg x_j \vee y_{v_1} \vee y_{v_2} \vee \dots \vee y_{v_k}$ , the negative support of variable  $Y$  for  $X = j$  is now encoded in intervals using regular literals. Let us see an example: If the domain of  $Y$  is  $\{1, 2, \dots, 10\}$  and we are given the support clause  $\neg x_2 \vee y_2 \vee y_3 \vee y_6 \vee y_8 \vee y_9$ , then the negative support is represented by the following intervals:  $[2, 3]$ ,  $[6, 6]$ , and  $[8, 9]$ . The interval-based encoding for this clause is as follows:  $x_2^{\geq} \wedge \neg x_3^{\geq} \rightarrow y_2^{\geq}$ ,  $x_2^{\geq} \wedge \neg x_3^{\geq} \wedge y_4^{\geq} \rightarrow y_6^{\geq}$ ,  $x_2^{\geq} \wedge \neg x_3^{\geq} \wedge y_7^{\geq} \rightarrow y_8^{\geq}$ ,  $x_2^{\geq} \wedge \neg x_3^{\geq} \rightarrow \neg y_{10}^{\geq}$ . Notice that the number of clauses needed for each support clause is linear in the largest domain size, and each clause has at most four regular literals.

**Definition 3.** *The interval-based minimal support encoding of a WMaxCSP instance  $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$  is the WPMInSAT instance obtained from the minimal support encoding by (i) replacing the ALO and AMO clauses for every variable  $X \in \mathcal{X}$  with domain size  $m$  with the clauses  $x_m^{\geq} \rightarrow x_{m-1}^{\geq}$ ,  $x_{m-1}^{\geq} \rightarrow x_{m-2}^{\geq}, \dots, x_3^{\geq} \rightarrow x_2^{\geq}$ ; and (ii) replacing every support clause with the corresponding interval-based regular clauses using the negative support; i.e, if the negative support of variable  $Y$  for  $X = i$  can be represented by the intervals  $[l_1, u_1], [l_2, u_2], \dots [l_k, u_k]$ , we add the clauses  $x_i^{\geq} \wedge \neg x_{i+1}^{\geq} \rightarrow y_{l_1}^{\geq}$ ,  $x_i^{\geq} \wedge \neg x_{i+1}^{\geq} \wedge y_{u_1+1}^{\geq} \rightarrow y_{l_2}^{\geq}, \dots, x_i^{\geq} \wedge \neg x_{i+1}^{\geq} \wedge y_{u_k+1}^{\geq} \rightarrow y_{l_k}^{\geq}$ ,  $x_i^{\geq} \wedge \neg x_{i+1}^{\geq} \rightarrow \neg y_{u_k+1}^{\geq}$ .*

*Example 3.* The WPMInSAT interval-based minimal support encoding for the WMaxCSP instance of Example 2 with the constraint  $X \neq Y$  is formed by the hard clauses  $\neg x_3^{\geq} \vee x_2^{\geq}$  and  $\neg y_3^{\geq} \vee y_2^{\geq}$ , and the soft clauses  $(x_2^{\geq} \vee \neg y_2^{\geq}, 2)$ ,  $(\neg x_2^{\geq} \vee x_3^{\geq} \vee y_2^{\geq}, 2)$ ,  $(\neg x_2^{\geq} \vee x_3^{\geq} \vee \neg y_3^{\geq}, 2)$ , and  $(\neg x_3^{\geq} \vee y_3^{\geq}, 2)$ .

## 4 A New WPMInSAT Solver

We implemented a new version of MinSatz [20], which is the only existing branch-and-bound WPMInSAT solver, by incorporating all the inference rules of MaxSatz [17]. Such rules preserve the number of falsified clauses over all the models and can be applied in MinSatz, to help detect contradictions earlier, because it works by maximizing the number of falsified clauses (instead of minimizing that number as in MaxSAT).

For lack of space we just define one rule of [17]: The clauses  $x, y, \neg x \vee \neg y$  are replaced with  $\square, x \vee y$ . The clause  $x \vee y$  ensures that the number of falsified clauses is preserved also when  $x = y = 0$ . In this case, two clauses are falsified.

**Table 1.** Experimental results for random binary MaxCSP, graph coloring, and kbtrees instances. The first column displays the parameters provided to the generators, the second column displays the number of tested instances, and the rest of columns display, for the encoding and solver indicated in the first row, the mean time (seconds) needed to solve an instance among the instances that were solved within a cutoff time of 1800 seconds, followed by the total number of solved instances in parentheses. The best results are in bold.

### Random binary CSP

| $\langle n, d, p_1, p_2 \rangle$ | #    | Toulbar2         | MinSatz (supc) | MaxSatz (isupc)  | Abscon      | MinSatz (isupc) | MaxSatz (supc)   | MinSatz (dir)    | MaxSatz (dir)    | WPM1 (supc)      |
|----------------------------------|------|------------------|----------------|------------------|-------------|-----------------|------------------|------------------|------------------|------------------|
| $\langle 22, 5, 180, 2 \rangle$  | 100  | <b>0.01(100)</b> | 0.37(100)      | <b>0.01(100)</b> | 438.21(98)  | 11.66(100)      | <b>0.01(100)</b> | 937.91(27)       | <b>0.01(100)</b> | <b>0.01(100)</b> |
| $\langle 22, 5, 180, 4 \rangle$  | 100  | <b>0.01(100)</b> | 7.29(100)      | 0.93(100)        | 187.77(100) | 11.59(100)      | 1.65(100)        | 1084.66(27)      | 0.02(100)        | 0.02(100)        |
| $\langle 22, 5, 180, 6 \rangle$  | 100  | <b>0.58(100)</b> | 44.29(100)     | 62.21(100)       | 377.95(100) | 49.73(100)      | 23.20(100)       | 1179.06(2)       | 7.40(100)        | 730.03(32)       |
| $\langle 22, 5, 180, 8 \rangle$  | 100  | <b>2.20(100)</b> | 99.62(100)     | 217.00(100)      | 522.45(100) | 68.60(100)      | 59.37(100)       | 1127.71(2)       | 111.02(100)      | 0.00(0)          |
| $\langle 22, 5, 180, 10 \rangle$ | 100  | <b>4.96(100)</b> | 173.71(100)    | 362.44(100)      | 586.06(100) | 114.64(100)     | 129.01(100)      | 1715.14(3)       | 854.61(93)       | 0.00(0)          |
| $\langle 22, 5, 180, 12 \rangle$ | 100  | <b>7.48(100)</b> | 247.97(100)    | 441.82(100)      | 634.37(100) | 202.55(100)     | 237.51(100)      | 1314.50(5)       | 1487.47(1)       | 0.00(0)          |
| $\langle 22, 5, 180, 14 \rangle$ | 100  | <b>9.52(100)</b> | 285.39(100)    | 359.80(100)      | 513.48(100) | 350.16(100)     | 422.87(100)      | 1390.58(23)      | 0.00(0)          | 0.00(0)          |
| $\langle 22, 5, 180, 16 \rangle$ | 100  | <b>9.74(100)</b> | 276.80(100)    | 224.28(100)      | 366.35(100) | 516.14(100)     | 696.88(99)       | 1033.17(92)      | 0.00(0)          | 0.00(0)          |
| $\langle 22, 5, 180, 18 \rangle$ | 100  | <b>8.49(100)</b> | 225.65(100)    | 124.22(100)      | 173.55(100) | 639.40(100)     | 1021.36(89)      | 334.16(100)      | 0.00(0)          | 0.00(0)          |
| $\langle 22, 5, 180, 20 \rangle$ | 100  | <b>6.47(100)</b> | 176.76(100)    | 54.13(100)       | 72.30(100)  | 764.34(96)      | 1122.14(62)      | 73.99(100)       | 0.00(0)          | 0.00(0)          |
| $\langle 22, 5, 180, 22 \rangle$ | 100  | <b>2.69(100)</b> | 61.97(100)     | 14.75(100)       | 9.34(100)   | 669.01(99)      | 989.45(92)       | 5.41(100)        | 0.00(0)          | 0.00(0)          |
| $\langle 22, 5, 180, 24 \rangle$ | 100  | 0.08(100)        | 0.25(100)      | 0.93(100)        | 0.04(100)   | 195.24(100)     | 44.76(100)       | <b>0.01(100)</b> | 0.00(0)          | 319.82(1)        |
| Total                            | 1200 | <b>1200</b>      | 1200           | 1200             | 1198        | 1195            | 1142             | 581              | 494              | 233              |

### Graph Coloring

| $\langle n, k, c \rangle$   | #   | Toulbar2           | Abscon      | MinSatz (supc) | MaxSatz (isupc) | MinSatz (dir)     | MaxSatz (isupc) | MinSatz (supc) | MaxSatz (dir) | WPM1 (supc) |
|-----------------------------|-----|--------------------|-------------|----------------|-----------------|-------------------|-----------------|----------------|---------------|-------------|
| $\langle 16, 16, 6 \rangle$ | 100 | <b>10.48(100)</b>  | 85.65(100)  | 106.75(100)    | 84.99(100)      | 167.05(100)       | 615.50(75)      | 688.32(63)     | 557.89(30)    | 86.48(91)   |
| $\langle 16, 16, 4 \rangle$ | 100 | 1.98(100)          | 49.32(100)  | 7.22(100)      | 4.92(100)       | <b>1.60(100)</b>  | 26.07(100)      | 22.66(100)     | 88.79(100)    | 668.69(24)  |
| $\langle 18, 18, 6 \rangle$ | 100 | <b>132.01(100)</b> | 502.96(81)  | 649.46(74)     | 576.50(67)      | 1036.78(37)       | 1275.15(52)     | 0.00(0)        | 0.00(0)       | 247.14(40)  |
| $\langle 18, 18, 4 \rangle$ | 100 | 16.35(100)         | 130.90(100) | 65.48(100)     | 50.95(100)      | <b>12.44(100)</b> | 290.03(100)     | 262.91(100)    | 692.70(84)    | 0.00(0)     |
| $\langle 20, 20, 7 \rangle$ | 100 | <b>514.20(63)</b>  | 588.24(40)  | 886.25(5)      | 757.93(11)      | 0.00(0)           | 0.00(0)         | 0.00(0)        | 0.00(0)       | 234.16(34)  |
| $\langle 20, 20, 5 \rangle$ | 100 | <b>478.25(95)</b>  | 1024.63(30) | 872.03(46)     | 899.19(22)      | 1169.70(12)       | 0.00(0)         | 0.00(0)        | 0.00(0)       | 437.06(1)   |
| Total                       | 600 | <b>558</b>         | 451         | 425            | 400             | 349               | 277             | 263            | 214           | 190         |

### Kbtrees

| $t$   | #   | Toulbar2        | MinSatz (supc) | MinSatz (isupc) | Abscon     | MinSatz (dir) | MaxSatz (isupc) | MaxSatz (supc)  | MaxSatz (dir)   | WPM1 (dir) |
|-------|-----|-----------------|----------------|-----------------|------------|---------------|-----------------|-----------------|-----------------|------------|
| 10    | 50  | <b>0.01(50)</b> | 1.46(50)       | 39.81(50)       | 18.72(50)  | 0.00(0)       | <b>0.01(50)</b> | <b>0.01(50)</b> | <b>0.01(50)</b> |            |
| 20    | 50  | <b>0.01(50)</b> | 2.93(50)       | 27.02(50)       | 25.42(50)  | 489.89(50)    | 65.93(50)       | 34.26(50)       | 1.42(50)        | 0.30(50)   |
| 30    | 50  | <b>0.19(50)</b> | 4.58(50)       | 34.21(50)       | 50.73(50)  | 485.40(48)    | 600.77(35)      | 260.17(50)      | 209.61(50)      | 541.71(11) |
| 40    | 50  | <b>1.18(50)</b> | 11.21(50)      | 47.04(50)       | 77.81(50)  | 444.69(49)    | 1509.73(2)      | 925.71(41)      | 0.00(0)         | 0.00(0)    |
| 50    | 50  | <b>2.05(50)</b> | 11.33(50)      | 45.08(50)       | 104.27(50) | 213.32(50)    | 907.35(3)       | 1040.50(18)     | 0.00(0)         | 0.00(0)    |
| 60    | 50  | <b>2.77(50)</b> | 15.04(50)      | 31.25(50)       | 177.85(50) | 90.48(50)     | 930.23(21)      | 1200.68(7)      | 0.00(0)         | 0.00(0)    |
| 70    | 50  | <b>3.49(50)</b> | 19.10(50)      | 28.29(50)       | 306.46(50) | 65.45(50)     | 998.99(37)      | 0.00(0)         | 0.00(0)         | 0.00(0)    |
| 80    | 50  | <b>1.60(50)</b> | 18.88(50)      | 5.79(50)        | 321.73(50) | 8.39(50)      | 244.92(50)      | 0.00(0)         | 0.00(0)         | 0.00(0)    |
| 90    | 50  | <b>0.82(50)</b> | 33.60(50)      | 1.46(50)        | 370.69(50) | 1.83(50)      | 47.07(50)       | 0.00(0)         | 0.00(0)         | 0.00(0)    |
| Total | 450 | <b>450</b>      | 450            | 450             | 450        | 397           | 298             | 216             | 150             | 111        |

MinSatz is also characterized by incorporating the upper bounds, described in [20], that exploit clique partition algorithms and MaxSAT technology. In a sense, the new MinSatz combines efficient MaxSAT techniques with genuine MinSAT techniques.

To show the impact of inference rules in performance, Figure 1 compares the old and new MinSatz solvers on weighted Min2SAT instances, generated uniformly at random, having 2000 variables, and with a number of clauses ranging from 1000 to 3000, and the weights ranging from 1 to 10. We used the computer described below, and solved 100 instances per point. MinSatz-ir refers to MinSatz without inference rules.

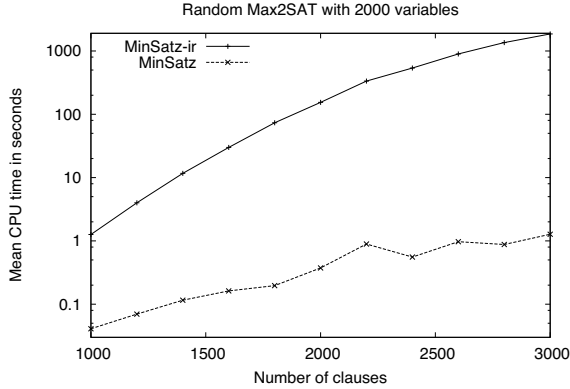


**Fig. 1.** Comparison of Minsatz with and without inference rules on weighted Min2SAT instances

## 5 Experimental Results

We compared the performance of the direct encoding (*dir*), the minimal support encoding (*supc*), and the interval-based support encoding (*isupc*) of different optimization problems with WCSP, WPMInSAT and WPMaXSAT solvers. Experiments were performed on a cluster with Intel Xeon 2.67GHz processors with 4GB of memory.

The solved problems (that were first derived as MaxCSP instances and then translated to the mentioned WPMInSAT and WPMaXSAT encodings) are: (I) *Random binary MaxCSP (model B [21])*: In the class  $\langle n, d, p_1, p_2 \rangle$  with  $n$  variables of domain size  $d$ , we choose a random subset of exactly  $p_1 n(n-1)/2$  constraints (rounded to the nearest integer), each with exactly  $p_2 d^2$  conflicts;  $p_1$  may be thought of as the density and  $p_2$  as the tightness. (II) *Graph coloring*: unsatisfiable graph coloring instances of Culberson [8] with option IID (independent random edge assignment). The parameters of the generator are: number of vertices ( $n$ ), optimum number of colors to get a valid coloring ( $k$ ), and number of colors we use to color the graph ( $c$ ). We solved the problem of finding a coloring that minimizes the number of adjacent vertices with the same color. (III) *Kbtrees*: Clique tree instances with different constraint tightness ( $t$ ), tested in [4].

We used the MaxSAT solvers MaxSatz and WPM1 [1] from the last MaxSAT Evaluation, the new version of MinSatz, and the WCSP solvers Abscon [15] and Toulbar2 [14]. For WPM1 we just show results with the best performing encoding. We also tested WPMIn1 [2] and the WPMInSAT instances produced by transforming the tested

WPMaXSAT instances using the natural flow network encoding [23], but their performance was not competitive.

Table 1 shows the results obtained. As a general comment for the solved testbed, we should say that MinSAT outperforms MaxSAT, the minimal support and interval-based minimal support encodings are better than the direct encoding in MinSAT, and encodings containing fewer and smaller clauses are usually more efficient. Even when our aim is to study the duality and complementarity between MaxSAT and MinSAT, and not to compete with WCSP solvers in their own territory, it is worth mentioning the good behavior of MinSAT wrt Abscon, as well as the still existing gap between MinSAT and Toulbar2 that should stimulate further research on MinSAT. The efforts devoted to WCSP solvers cannot be compared with the recent efforts on WPMinSAT solvers. Recall that WCSP is used here as a source for getting benchmarks for WPMinSAT solvers, and for advancing in the construction of a challenging testbed of MinSAT instances.

In random MaxCSP, the performance of MinSAT versus MaxSAT depends on the number of conflicts. The number of conflicts is related to the clause size in the support encodings, and is related to the number of clauses in the direct encoding. Also observe that *isupc* is superior to *supc* on some subsets of instances on both MaxSAT and MinSAT solvers, showing the relevance of using regular literals in some cases.

In graph coloring, MinSAT solves more instances than MaxSAT but MaxSAT is faster in some subsets, providing again evidence of the complementarity between MinSAT and MaxSAT. We also observe that regular literals are relevant in some cases. There are a substantial number of instances beyond the reach of the tested solvers, and are a challenge for future experimental investigations. Observe that MinSAT outperforms Abscon on some sets.

In the clique tree results, all the MinSAT options are superior to the best MaxSAT option, solving 152 additional instances. Our encodings allowed to solve for the first time a number of instances not solved in [4].

## 6 Conclusions

We have defined original encodings from WMaxCSP to WPMinSAT, created a new testbed for evaluating MinSAT solvers, developed a new WPMinSAT solver, and provided new insights into MinSAT such as the duality between MinSAT and MaxSAT encodings. Our experiments indicate that MaxSAT solving and MinSAT solving are complementary. Depending on the structure of the instances, more compact and efficient encodings are produced either with MinSAT or with MaxSAT. In a similar way, the solving techniques of MinSAT outperform the techniques of MaxSAT for some instances, and vice versa.

MaxSAT and MinSAT are important optimization problems that deserve to be compared, and their duality and complementarity deserve to be analyzed. Moreover, as MinSAT-based problem solving is a new research topic, there are not as many available benchmarks as in MaxSAT for evaluating solvers and stimulating the development of new ones. We hope the results of the paper could contribute to gain new insights on these problems, advance the state of the art of MinSAT-based problem solving, and provide tools for developing and evaluating MinSAT solvers.

## References

1. Ansótegui, C., Bonet, M.L., Levy, J.: A new algorithm for weighted partial MaxSAT. In: Proceedings of the 24th AAAI Conference on Artificial Intelligence, AAAI 2008, Atlanta, USA, pp. 3–8 (2010)
2. Ansótegui, C., Li, C.M., Manyà, F., Zhu, Z.: A SAT-based approach to MinSAT. In: Proceedings of the 15th International Conference of the Catalan Association for Artificial Intelligence, CCIA 2012, Alacant, Spain, pp. 185–189. IOS Press (2012)
3. Ansótegui, C., Manyà, F.: Mapping Problems with Finite-Domain Variables to Problems with Boolean Variables. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 1–15. Springer, Heidelberg (2005)
4. Argelich, J., Cabiscol, A., Lynce, I., Manyà, F.: Sequential Encodings from Max-CSP into Partial Max-SAT. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 161–166. Springer, Heidelberg (2009)
5. Argelich, J., Cabiscol, A., Lynce, I., Manyà, F.: Efficient encodings from CSP into SAT, and from MaxCSP into MaxSAT. *Multiple-Valued Logic and Soft Computing* 19(1-3), 3–23 (2012)
6. Argelich, J., Li, C.M., Manyà, F., Planes, J.: The first and second Max-SAT evaluations. *Journal on Satisfiability, Boolean Modeling and Computation* 4, 251–278 (2008)
7. Béjar, R., Hähnle, R., Manyà, F.: A modular reduction of regular logic to classical logic. In: Proceedings, 31st International Symposium on Multiple-Valued Logics (ISMVL), Warsaw, Poland, pp. 221–226. IEEE CS Press, Los Alamitos (2001)
8. Culberson, J.: Graph coloring page: The flat graph generator (1995), <http://web.cs.ualberta.ca/~joe/Coloring/Generators/flat.html>
9. Gent, I.P.: Arc consistency in SAT. In: Proceedings of the 15th European Conference on Artificial Intelligence, ECAI, Lyon, France, pp. 121–125 (2002)
10. Gent, I.P., Nightingale, P.: A new encoding of AllDifferent into SAT. In: Proceedings of the 3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems, CP 2004 Workshop, Toronto, Canada, pp. 95–110 (2004)
11. Heras, F., Morgado, A., Planes, J., Marques-Silva, J.: Iterative SAT solving for minimum satisfiability. In: Proceedings of the IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, pp. 922–927 (2012)
12. Kasif, S.: On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence* 45, 275–286 (1990)
13. Kügel, A.: Natural Max-SAT encoding of Min-SAT. In: Hamadi, Y., Schoenauer, M. (eds.) LION 6. LNCS, vol. 7219, pp. 431–436. Springer, Heidelberg (2012)
14. Larrosa, J., Schiex, T.: In the quest of the best form of local consistency for weighted CSP. In: Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI 2003, Acapulco, México, pp. 239–244 (2003)
15. Lecoutre, C., Tabary, S.: Abscon 112: Towards more robustness. In: Proceedings of the 3rd International Constraint Solver Competition in CP 2008 (2008)
16. Li, C.M., Manyà, F.: MaxSAT, hard and soft constraints. In: Biere, A., van Maaren, H., Walsh, T. (eds.) *Handbook of Satisfiability*, pp. 613–631. IOS Press (2009)
17. Li, C.M., Manyà, F., Planes, J.: New inference rules for Max-SAT. *Journal of Artificial Intelligence Research* 30, 321–359 (2007)
18. Li, C.M., Manyà, F., Quan, Z., Zhu, Z.: Exact MinSAT solving. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 363–368. Springer, Heidelberg (2010)
19. Li, C.M., Zhu, Z., Manyà, F., Simon, L.: Minimum satisfiability and its applications. In: Proceedings of the 22nd International Joint Conference on Artificial Intelligence, IJCAI 2011, Barcelona, Spain, pp. 605–610 (2011)

20. Li, C.M., Zhu, Z., Manyà, F., Simon, L.: Optimizing with minimum satisfiability. *Artificial Intelligence* 190, 32–44 (2012)
21. Smith, B.M., Dyer, M.E.: Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence* 81, 155–181 (1996)
22. Walsh, T.: SAT v CSP. In: Dechter, R. (ed.) *CP 2000*. LNCS, vol. 1894, pp. 441–456. Springer, Heidelberg (2000)
23. Zhu, Z., Li, C.-M., Manyà, F., Argelich, J.: A New Encoding from MinSAT into MaxSAT. In: Milano, M. (ed.) *CP 2012*. LNCS, vol. 7514, pp. 455–463. Springer, Heidelberg (2012)



# Adaptive Parameterized Consistency<sup>\*</sup>

Amine Balafrej<sup>1,2</sup>, Christian Bessiere<sup>1</sup>,  
Remi Coletta<sup>1</sup>, and El Houssine Bouyahf<sup>2</sup>

<sup>1</sup> CNRS, University of Montpellier, France

<sup>2</sup> LIMARF/FSR, University Mohammed V Agdal, Rabat, Morocco  
{balafrej,bessiere,coletta}@lirmm.fr, bouyahf@fsr.ac.ma

**Abstract.** State-of-the-art constraint solvers uniformly maintain the same level of local consistency (usually arc consistency) on all the instances. We propose *parameterized local consistency*, an original approach to adjust the level of consistency depending on the instance and on which part of the instance we propagate. We do not use as parameter one of the features of the instance, as done for instance in portfolios of solvers. We use as parameter the *stability* of values, which is a feature based on the state of the arc consistency algorithm during its execution. Parameterized local consistencies choose to enforce arc consistency or a higher level of local consistency on a value depending on whether the stability of the value is above or below a given threshold. We also propose a way to dynamically adapt the parameter, and thus the level of local consistency, during search. This approach allows us to get a good trade-off between the number of values pruned and the computational cost. We validate our approach on various problems from the CSP competition.

## 1 Introduction

Enforcing constraint propagation by applying local consistency during search is one of the strengths of constraint programming (CP). It allows the constraint solver to remove locally inconsistent values. This leads to a reduction of the search space. Arc consistency is the oldest and most well-known way of propagating constraints [2]. It has the nice feature that it does not modify the structure of the constraint network. It just prunes infeasible values. Arc consistency is the standard level of consistency maintained in constraint solvers. Several other local consistencies pruning only values and stronger than arc consistency have been proposed, such as max restricted path consistency or singleton arc consistency [7]. These local consistencies are seldom used in practice because of the high computational cost of maintaining them during search. However, on some problems, maintaining arc consistency is not a good choice because of the high number of ineffective revisions of constraints that penalize the CPU time. For instance, Stergiou observed that when solving the scen11 radio link frequency assignment problem (RLFAP) with an algorithm maintaining arc consistency, only 27 out of the 4103 constraints of the problem were identified as causing a domain wipe out and 1921 constraints did not prune any value [10].

---

<sup>\*</sup> This work has been funded by the EU project ICON (FP7-284715).

Choosing the right level of local consistency for solving a problem requires finding the good trade-off between the ability of this local consistency to remove inconsistent values, and the cost of the algorithm that enforces it. Stergiou suggests to take advantage of the power of strong consistencies to reduce the search space while avoiding the high cost of maintaining them in the whole network. His method results in a heuristic approach based on the monitoring of propagation events to dynamically adapt the level of local consistency (arc consistency or max restricted path consistency) to individual constraints. This prunes more values than arc consistency and less than max restricted path consistency. The level of propagation obtained is not characterized by a local consistency property. Depending on the order of propagation we can converge on different closures. When dealing with global constraints, some work propose to weaken arc consistency instead of strengthening it. In [8], Katriel et al. proposed a randomized filtering scheme for AllDifferent and Global Cardinality Constraint. In [9], Sellmann introduced the concept of approximated consistency for optimization constraints and provided filtering algorithms for Knapsack Constraints based on bounds with guaranteed accuracy.

In this paper we define the notion of stability of values. This is an original notion not based on characteristics of the instance to solve but based on the state of the arc consistency algorithm during its propagation. Based on this notion, we propose *parameterized consistencies*, an original approach to adjust the level of consistency inside a given instance. The intuition is that if a value is hard to prove arc consistent (i.e., the value is not stable for arc consistency), this value will perhaps be pruned by a stronger local consistency. The parameter  $p$  specifies the threshold of stability of a value  $v$  below which we will enforce a higher consistency to  $v$ . A parameterized consistency  $p$ -LC is thus an intermediate level of consistency between arc consistency and another consistency LC, stronger than arc consistency. The strength of  $p$ -LC depends on the parameter  $p$ . This approach allows us to find a trade-off between the pruning power of the local consistency and the computational cost of the algorithm that achieves it. We apply  $p$ -LC to the case where LC is max restricted path consistency. We describe the algorithm  $p$ -maxRPC3 (based on maxRPC3 [1]) that achieves  $p$ -max restricted path consistency. Then, we propose  $ap$ -LC, an adaptive variant of  $p$ -LC which adapts dynamically and locally the level of local consistency during search. Finally, we experimentally assess the practical relevance of parameterized local consistency. We show that by making good choices for the parameter  $p$  we take advantage of both arc consistency light computational cost and LC effectiveness of pruning. In the best cases, a solver using  $p$ -LC explores the same number of nodes as LC with a number of constraint checks lower than LC, resulting in a CPU-time lower than both arc consistency-based or LC-based solvers.

## 2 Background

A *constraint network* is defined as a set of  $n$  variables  $X = \{x_1, \dots, x_n\}$ , a set of ordered domains  $D = \{D(x_1), \dots, D(x_n)\}$ , and a set of  $e$  constraints

$C = \{c_1, \dots, c_e\}$ . Each constraint  $c_k$  is defined by a pair  $(\text{var}(c_k), \text{sol}(c_k))$ , where  $\text{var}(c_k)$  is an ordered subset of  $X$ , and  $\text{sol}(c_k)$  is a set of combinations of values (tuples) satisfying  $c_k$ . In the following, we restrict ourselves to binary constraints because the local consistency (maxRPC) we use here to instantiate our approach is defined on the binary case only. However, the notions we introduce can be extended to non-binary constraints, by using maxRPWC for instance [4]. A binary constraint  $c$  between  $x_i$  and  $x_j$  will be denoted by  $c_{ij}$ , and  $\Gamma(x_i)$  will denote the set of variables  $x_j$  involved in a constraint with  $x_i$ .

A value  $v_j \in D(x_j)$  is called an *arc consistent support* (AC support) for  $v_i \in D(x_i)$  on  $c_{ij}$  if  $(v_i, v_j) \in \text{sol}(c_{ij})$ . A value  $v_i \in D(x_i)$  is *arc consistent* (AC) if and only if for all  $x_j \in \Gamma(x_i)$   $v_i$  has an AC support  $v_j \in D(x_j)$  on  $c_{ij}$ . A domain  $D(x_i)$  is arc consistent if it is non empty and all values in  $D(x_i)$  are arc consistent. A network is arc consistent if all domains in  $D$  are arc consistent. If enforcing arc consistency on a network  $N$  leads to a domain wipe out, we say that  $N$  is arc inconsistent.

A tuple  $(v_i, v_j) \in D(x_i) \times D(x_j)$  is *path consistent* (PC) if and only if for any third variable  $x_k$  there exists a value  $v_k \in D(x_k)$  such that  $v_k$  is an AC support for both  $v_i$  and  $v_j$ . In such a case,  $v_k$  is called *witness* for the path consistency of  $(v_i, v_j)$ .

A value  $v_j \in D(x_j)$  is a *max restricted path consistent* (maxRPC) support for  $v_i \in D(x_i)$  on  $c_{ij}$  if and only if it is an AC support and the tuple  $(v_i, v_j)$  is path consistent. A value  $v_i \in D(x_i)$  is max restricted path consistent on a constraint  $c_{ij}$  if and only if  $\exists v_j \in D(x_j)$  maxRPC support for  $v_i$  on  $c_{ij}$ . A value  $v_i \in D(x_i)$  is max restricted path consistent iff for all  $x_j \in \Gamma(x_i)$   $v_i$  has a maxRPC support  $v_j \in D(x_j)$  on  $c_{ij}$ . A domain  $D(x_i)$  is maxRPC if it is non empty and all values in  $D(x_i)$  are maxRPC. A network is maxRPC if all domains in  $D$  are maxRPC.

We say that a local consistency  $LC_1$  is stronger than a local consistency  $LC_2$  ( $LC_2 \preceq LC_1$ ) if  $LC_2$  holds on any constraint network on which  $LC_1$  holds.

The problem of deciding whether a constraint network has solutions is called the *constraint satisfaction problem* (CSP), and it is NP-complete. Solving a CSP is done by backtrack search that maintains some level of consistency between each branching step.

### 3 Parameterized Consistency

In this section we present an original approach to parameterize a level of consistency LC stronger than arc consistency so that it degenerates to arc consistency when the parameter equals 0, to LC when the parameters equals 1, and to levels in between when the parameter is between 0 and 1. The idea behind this is to be able to adjust the level of consistency to the instance to be solved, hoping that such an adapted level of consistency will prune significantly more values than arc consistency while being less time consuming than LC.

Parameterized consistency is based on the concept of stability of values. We first need to define the 'distance to end' of a value in a domain. This captures how far a value is from the last in its domain. In the following,  $\text{rank}(v, S)$  is the position of value  $v$  in the ordered set of values  $S$ .

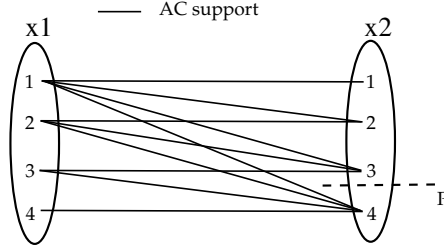


**Fig. 1.** Stability of supports on the example of the constraint  $x_1 \leq x_2$  with the domains  $D(x_1) = D(x_2) = \{1, 2, 3, 4\}$ .  $(x_1, 4)$  is not  $p$ -stable for AC.

**Definition 1 (Distance to end of a value).** *The distance to end of a value  $v_i \in D(x_i)$  is the ratio*

$$\Delta(x_i, v_i) = (|D_o(x_i)| - \text{rank}(v_i, D_o(x_i))) / |D_o(x_i)|,$$

where  $D_o(x_i)$  is the initial domain of  $x_i$ .

We see that the first value in  $D_o(x_i)$  has distance  $(|D_o(x_i)| - 1) / |D_o(x_i)|$  and the last one has distance 0. Thus,  $\forall v_i \in D(x_i), 0 \leq \Delta(x_i, v_i) < 1$ .

We can now give the definition of what we call the parameterized stability of a value for arc consistency. The idea is to define stability for values based on the distance to the end of their AC supports. For instance, consider the constraint  $x_1 \leq x_2$  with the domains  $D(x_1) = D(x_2) = \{1, 2, 3, 4\}$  (see Figure 1).  $\Delta(x_2, 1) = (4 - 1) / 4 = 0.75$ ,  $\Delta(x_2, 2) = 0.5$ ,  $\Delta(x_2, 3) = 0.25$  and  $\Delta(x_2, 4) = 0$ . If  $p = 0.2$ , the value  $(x_1, 4)$  is not  $p$ -stable for AC, because the first and only AC support of  $(x_1, 4)$  in the ordering used to look for supports, that is  $(x_2, 4)$ , has a distance to end smaller than the threshold  $p$ . Proving that the pair  $(4, 4)$  is inconsistent (by a stronger consistency) could lead to the pruning of  $(x_1, 4)$ . In other words, applying a stronger consistency on  $(x_1, 4)$  has more chances to lead to its removal than applying it on for instance  $(x_1, 1)$ , which had no difficulty to find its first AC support (distance to en of  $(x_2, 1)$  is 0.75).

**Definition 2 ( $p$ -stability for AC).** *A value  $v_i \in D(x_i)$  is  $p$ -stable for AC on  $c_{ij}$  iff  $v_i$  has an AC support  $v_j \in D(x_j)$  on  $c_{ij}$  such that  $\Delta(x_j, v_j) \geq p$ . A value  $v_i \in D(x_i)$  is  $p$ -stable for AC iff  $\forall x_j \in \Gamma(x_i)$ ,  $v_i$  is  $p$ -stable for AC on  $c_{ij}$ .*

We are now ready to give the first definition of parameterized local consistency. This first definition can be applied to any local consistency LC for which the consistency of a value on a constraint is well defined. This is the case for instance for all triangle-based consistencies [6,2].

**Definition 3 (Constraint-based  $p$ -LC).** *Let LC be a local consistency stronger than AC for which the LC consistency of a value on a constraint is defined. A value  $v_i \in D(x_i)$  is constraint-based  $p$ -LC on  $c_{ij}$  iff it is  $p$ -stable for*

*AC on  $c_{ij}$ , or it is LC on  $c_{ij}$ . A value  $v_i \in D(x_i)$  is constraint-based  $p$ -LC iff  $\forall c_{ij}, v_i$  is constraint-based  $p$ -LC on  $c_{ij}$ . A constraint network is constraint-based  $p$ -LC iff all values in all domains in  $D$  are constraint-based  $p$ -LC.*

**Theorem 1.** *Let LC be a local consistency stronger than AC for which the LC consistency of a value on a constraint is defined. Let  $p_1$  and  $p_2$  be two parameters in  $[0..1]$ . If  $p_1 < p_2$  then  $AC \preceq$  constraint-based  $p_1$ -LC  $\preceq$  constraint-based  $p_2$ -LC  $\preceq$  LC.*

**Proof.** Suppose that there exist two parameters  $p_1, p_2$  such that  $0 \leq p_1 < p_2 \leq 1$ , and suppose that there exists a  $p_2$ -LC constraint network  $N$  that contains a  $p_2$ -LC value  $(x_i, v_i)$  that is  $p_1$ -LC inconsistent. Let  $c_{ij}$  be the constraint on which  $(x_i, v_i)$  is  $p_1$ -LC inconsistent. Then,  $\nexists v_j \in D(x_j)$  that is an AC support for  $(x_i, v_i)$  on  $c_{ij}$  such that  $\Delta(x_j, v_j) \geq p_1$ . Thus,  $v_i$  is not  $p_2$ -stable for AC on  $c_{ij}$ . In addition,  $v_i$  is not LC on  $c_{ij}$ . Therefore,  $v_i$  is not  $p_2$ -LC, and  $N$  is not  $p_2$ -LC. ■

Definition 3 can be modified to a more coarse-grained version that is not dependent on the consistency of values on a constraint. It will have the advantage to apply to any type of strong local consistency, even those, like singleton arc consistency, for which the consistency of a value on a constraint is not defined.

**Definition 4 (Value-based  $p$ -LC).** *Let LC be a local consistency stronger than AC. A value  $v_i \in D(x_i)$  is value-based  $p$ -LC if and only if it is  $p$ -stable for AC or it is LC. A constraint network is value-based  $p$ -LC if and only if all values in all domains in  $D$  are value-based  $p$ -LC.*

**Theorem 2.** *Let LC be a local consistency stronger than AC. Let  $p_1$  and  $p_2$  be two parameters in  $[0..1]$ . If  $p_1 < p_2$  then  $AC \preceq$  value-based  $p_1$ -LC  $\preceq$  value-based  $p_2$ -LC  $\preceq$  LC.*

**Proof.** Suppose that there exist two parameters  $p_1, p_2$  such that  $0 \leq p_1 < p_2 \leq 1$ , and suppose that there exists a  $p_2$ -LC constraint network  $N$  that contains a  $p_2$ -LC value  $(x_i, v_i)$  that is  $p_1$ -LC-inconsistent.  $v_i$  is  $p_1$ -LC-inconsistent means that:

1.  $v_i$  is not  $p_1$ -stable for AC:  $\exists c_{ij}$  on which  $v_i$  is not  $p_1$ -stable for AC. Then  $\nexists v_j \in D(x_j)$  that is an AC support for  $(x_i, v_i)$  on  $c_{ij}$  such that  $\Delta(x_j, v_j) \geq p_1$ . Therefore,  $v_i$  is not  $p_2$ -stable for AC on  $c_{ij}$ , then  $v_i$  is not  $p_2$ -stable for AC.
2.  $v_i$  is LC inconsistent

(1) and (2) imply that  $v_i$  is not  $p_2$ -LC, and  $N$  is not  $p_2$ -LC. ■

For both types of definitions of  $p$ -LC, we have the following property on the extreme cases ( $p = 0, p = 1$ ).

**Corollary 1.** *Let  $LC_1$  and  $LC_2$  be two local consistencies stronger than AC. We have: value-based  $0$ -LC<sub>2</sub> = AC and value-based  $1$ -LC<sub>2</sub> = LC. If the  $LC_1$  consistency of a value on a constraint is defined, we also have: constraint-based  $0$ -LC<sub>1</sub> = AC and constraint-based  $1$ -LC<sub>1</sub> = LC.*

## 4 Parameterized maxRPC: $p$ -maxRPC

To illustrate the benefit of our approach, we apply *parameterized consistency* to maxRPC to obtain the  $p$ -maxRPC level of consistency that achieves a consistency level between AC and maxRPC.

**Definition 5 ( $p$ -maxRPC).** *A value, a network, are  $p$ -maxRPC if and only if they are constraint-based  $p$ -maxRPC.*

From Theorem 1 and Corollary 1 we derive the following corollary.

**Corollary 2.** *For any two parameters  $p_1, p_2, 0 \leq p_1 < p_2 \leq 1$ ,  $AC \preceq p_1$ -maxRPC  $\preceq p_2$ -maxRPC  $\preceq$  maxRPC.  $0$ -maxRPC = AC and  $1$ -maxRPC = maxRPC.*

---

### Algorithm 1. Initialization( $X, D, C, Q$ )

---

```

1 begin
2   foreach  $x_i \in X$  do
3     foreach  $v_i \in D(x_i)$  do
4       foreach  $x_j \in \Gamma(x_i)$  do
5          $p$ -support  $\leftarrow$  false
6         foreach  $v_j \in D(x_j)$  do
7           if  $(v_i, v_j) \in c_{ij}$  then
8              $LastAC_{x_i, v_i, x_j} \leftarrow v_j$ 
9             if  $\Delta(x_j, v_j) \geq p$  then
10                $p$ -support  $\leftarrow$  true
11                $LastPC_{x_i, v_i, x_j} \leftarrow v_j$ 
12               break;
13             if searchPCwit( $v_i, v_j$ ) then
14                $p$ -support  $\leftarrow$  true
15                $LastPC_{x_i, v_i, x_j} \leftarrow v_j$ 
16               break;
17           if  $\neg p$ -support then
18             remove  $v_i$  from  $D(x_i)$ 
19              $Q \leftarrow Q \cup \{x_i\}$ 
20             break;
21   if  $D(x_i) = \emptyset$  then return false
22 return true

```

---

We propose an algorithm for  $p$ -maxRPC, based on maxRPC3, the best existing maxRPC algorithm. We do not describe maxRPC3 in full detail as it can be found in [1]. We only describe procedures where changes to maxRPC3 are necessary to design  $p$ -maxRPC3, a coarse grained algorithm that performs  $p$ -maxRPC. We use light grey to emphasize the modified parts of the original maxRPC3 algorithm.

**Algorithm 2.** checkPCsupLoss( $v_j, x_i$ )

---

```

1 begin
2   if  $LastAC_{x_j, v_j, x_i} \in D(x_i)$  then  $b_i \leftarrow \max>LastPC_{x_j, v_j, x_i} + 1, LastAC_{x_j, v_j, x_i}$ 
3   else  $b_i \leftarrow \max>LastPC_{x_j, v_j, x_i} + 1, LastAC_{x_j, v_j, x_i} + 1$ 
4   foreach  $v_i \in D(x_i), v_i \geq b_i$  do
5     if  $(v_j, v_i) \in c_{ji}$  then
6       if  $LastAC_{x_j, v_j, x_i} \notin D(x_i) \ \& \ LastAC_{x_j, v_j, x_i} > LastPC_{x_j, v_j, x_i}$  then
7          $LastAC_{x_j, v_j, x_i} \leftarrow v_i$ 
8         if  $\Delta(x_i, v_i) \geq p$  then  $LastPC_{x_j, v_j, x_i} \leftarrow v_i$  return true
9         if searchPCwit( $v_j, v_i$ ) then  $LastPC_{x_j, v_j, x_i} \leftarrow v_i$  return true
10  return false

```

---

**Algorithm 3.** checkPCwitLoss( $x_j, v_j, x_i$ )

---

```

1 begin
2   foreach  $x_k \in \Gamma(x_j) \cap \Gamma(x_i)$  do
3     witness  $\leftarrow false$ 
4     if  $v_k \leftarrow LastPC_{x_j, v_j, x_k} \in D(x_k)$  then
5       if  $\Delta(x_k, v_k) \geq p$  then witness  $\leftarrow true$ 
6     else
7       if  $LastAC_{x_j, v_j, x_i} \in D(x_i) \ \& \ LastAC_{x_j, v_j, x_i} = LastAC_{x_k, v_k, x_i}$ 
8       OR  $LastAC_{x_j, v_j, x_i} \in D(x_i) \ \& \ (LastAC_{x_j, v_j, x_i}, v_k) \in c_{ik}$ 
9       OR  $LastAC_{x_k, v_k, x_i} \in D(x_i) \ \& \ (LastAC_{x_k, v_k, x_i}, v_j) \in c_{ij}$ 
10      then witness  $\leftarrow true$ 
11     else
12       if searchACsup( $x_j, v_j, x_i$ ) & searchACsup( $x_k, v_k, x_i$ ) then
13         foreach  $v_i \in D(x_i), v_i \geq \max>LastAC_{x_j, v_j, x_i}, LastAC_{x_k, v_k, x_i}$ 
14         do
15           if  $(v_j, v_i) \in c_{ji} \ \& \ (v_k, v_i) \in c_{ki}$  then
16             witness  $\leftarrow true$ 
17             break;
18   if  $\neg witness \ \& \ \neg checkPCsupLoss(v_j, x_k)$  then return false
19 return true

```

---

maxRPC3 uses a propagation list  $Q$  where it inserts the variables whose domains have changed. It also uses two other data structures: LastAC and LastPC. For each value  $(x_i, v_i)$  LastAC $_{x_i, v_i, x_j}$  stores the smallest AC support for  $(x_i, v_i)$  on  $c_{ij}$  and LastPC $_{x_i, v_i, x_j}$  stores the smallest PC support for  $(x_i, v_i)$  on  $c_{ij}$  (i.e., the smallest AC support  $(x_j, v_j)$  for  $(x_i, v_i)$  on  $c_{ij}$  such that  $(v_i, v_j)$  is PC). This algorithm consists in two phases: initialization and propagation.

In the initialization phase (algorithm 1) maxRPC3 checks if each value  $(x_i, v_i)$  has a maxRPC-support  $(x_j, v_j)$  on each constraint  $c_{ij}$ . If not, it removes  $v_i$  from  $D(x_i)$  and inserts  $x_i$  in  $Q$ . To check if a value  $(x_i, v_i)$  has a maxRPC-support on a constraint  $c_{ij}$ , maxRPC3 looks first for an AC-support  $(x_j, v_j)$  for  $(x_i, v_i)$  on  $c_{ij}$ , then it checks if  $(v_i, v_j)$  is PC. In this last step, changes were necessary to obtain  $p$ -maxRPC3 (lines 9-12). We check if  $(v_i, v_j)$  is PC (line 13) only if  $\Delta(x_j, v_j)$  is smaller than the parameter  $p$  (line 9).

The propagation phase of maxRPC3 consists in propagating the effect of deletions. While  $Q$  is non empty, maxRPC3 extracts a variable  $x_i$  from  $Q$  and checks for each value  $(x_j, v_j)$  of each neighboring variable  $x_j \in \Gamma(x_i)$  if it is not maxRPC because of deletions of values in  $D(x_i)$ . A value  $(x_j, v_j)$  becomes maxRPC inconsistent in two cases: if its unique PC-support  $(x_i, v_i)$  on  $c_{ij}$  has been deleted, or if we deleted the unique witness  $(x_i, v_i)$  for a pair  $(v_j, v_k)$  such that  $(x_k, v_k)$  is the unique PC-support for  $(x_j, v_j)$  on  $c_{jk}$ . So, to propagate deletions, maxRPC3 checks if the last maxRPC support (last known support) of  $(x_j, v_j)$  on  $c_{ij}$  still belongs to the domain of  $x_i$ , otherwise it looks for the next support (algorithm 2). If such a support does not exist, it removes the value  $v_j$  and adds the variable  $x_j$  to  $Q$ . Then if  $(x_j, v_j)$  has not been removed in the previous step, maxRPC3 checks (algorithm 3) whether there is still a witness for each pair  $(v_j, v_k)$  such that  $(x_k, v_k)$  is the PC support for  $(x_j, v_j)$  on  $c_{jk}$ . If not, it looks for the next maxRPC support for  $(x_j, v_j)$  on  $c_{jk}$ . If such a support does not exist, it removes  $v_j$  from  $D(x_j)$  and adds the variable  $x_j$  to  $Q$ .

In the propagation phase also, we modified maxRPC3 to check if the values are still  $p$ -maxRPC instead of checking if they are maxRPC. In  $p$ -maxRPC3, the last  $p$ -maxRPC support for  $(x_j, v_j)$  on  $c_{ij}$  is the last AC support if  $(x_j, v_j)$  is  $p$ -stable for AC on  $c_{ij}$ . If not, it is the last PC support. Thus,  $p$ -maxRPC3 checks if the last  $p$ -maxRPC support (last known support) of  $(x_j, v_j)$  on  $c_{ij}$  still belongs to the domain of  $x_i$ . If not, it looks (algorithm 2) for the next AC support  $(x_i, v_i)$  on  $c_{ij}$ , and checks if  $(v_i, v_j)$  is PC (line 9) only when  $\Delta(x_i, v_i) < p$  (line 8). If no  $p$ -maxRPC support exists,  $p$ -maxRPC3 removes the value and adds the variable  $x_j$  to  $Q$ . If the value  $(x_j, v_j)$  has not been removed in the previous phase,  $p$ -maxRPC3 checks (algorithm 3) whether there is still a witness for each pair  $(v_j, v_k)$  such that  $(x_k, v_k)$  is the  $p$ -maxRPC support for  $v_j$  on  $c_{jk}$  and  $\Delta(x_k, v_k) < p$ . If not, it looks for the next  $p$ -maxRPC support for  $v_j$  on  $c_{jk}$ . If such a support does not exist, it removes  $v_j$  from  $D(x_j)$  and adds the variable  $x_j$  to  $Q$ .

## 5 Experimental Validation of $p$ -maxRPC

To validate the approach of parameterized local consistency, we made a first basic experiment. The purpose of this experiment is to see if there exist problems on which a given level of  $p$ -maxRPC, with a  $p$  uniform on all the constraint network and static during the whole search is more efficient than AC or maxRPC, or both.



We have implemented the algorithms that achieve  $p$ -maxRPC as described in the previous section in our own binary constraint solver, in addition to maxRPC (maxRPC3 version [1]) and AC (AC2001 version [3]). All these algorithms are maintained during search. We tested these algorithms on several classes of problems of the International Constraint Solver Competition 09<sup>1</sup>. We have only selected problems involving binary constraints. To isolate the effect of propagation, we used the lexicographic ordering for variables and values. We set the CPU time-out to one hour. Our experiments were conducted on a 12-core Genuine Intel machine with 16Gb of RAM running at 2.92GHz.

On each instance of our experiment, we ran AC, max-RPC, and  $p$ -maxRPC for all values of  $p$  in  $\{0.1, 0.2, \dots, 0.9\}$ . Performance has been measured in terms of CPU time in seconds, the number of visited nodes (NODE) and the number of constraint checks (CCK). Results are presented as "CPU time ( $p$ )", where  $p$  is the parameter for which  $p$ -maxRPC gives the best result.

Table 1 reports the performance of AC, maxRPC, and  $p$ -maxRPC for the value of  $p$  producing the best CPU time, on Radio Link Frequency Assignment Problems (RLFAPs), Geom problems, and queens knights problems. The CPU time of the best algorithm is highlighted with bold. On RLFAP and Geom, we observe the existence of a parameter  $p$  where  $p$ -maxRPC is faster than *both* AC and maxRPC for most instances of these two classes of problems. On the queens-knight problem, however, AC is always the best algorithm. In Figures 2 and 3, we try to understand more closely what makes  $p$ -maxRPC better or worse than AC and maxRPC. Figures 2 and 3 plot the performance (CPU, NODE and CCK) of  $p$ -maxRPC for all values of  $p$  from 0 to 1 by steps of 0.1 against performance of AC and maxRPC. Figure 2 shows an instance where  $p$ -maxRPC solves the problem faster than AC and maxRPC for values of  $p$  in the range  $[0.3..0.8]$ . We observe that  $p$ -maxRPC is faster than AC and maxRPC when it reduces the size of the search space as much as maxRPC (same number of nodes visited) with a number of CCK closer to the number of CCK produced by AC. Figure 3 shows an instance where the CPU time for  $p$ -maxRPC is never better than *both* AC and maxRPC. We see that if the CPU time for  $p$ -maxRPC is two to three times better than maxRPC, it fails to improve AC because the number of constraint checks performed by  $p$ -maxRPC is much higher than the number of constraint checks performed by AC, whereas the number of nodes visited by  $p$ -maxRPC is not significantly reduced compared to the number of nodes visited by AC. From these observations, it thus seems that  $p$ -maxRPC outperforms AC and maxRPC when it finds a compromise between the number of nodes visited (the power of maxRPC) and the number of CCK needed to maintain (the light cost of AC).

In Figures 2 and 3 we can see that the CPU time for 1-maxRPC (respectively 0-maxRPC) is greater than the CPU time for maxRPC (respectively AC) although the two consistencies are equivalent. The reason is that  $p$ -maxRPC performs tests on the distances. For  $p = 0$ , we also explain this difference by the fact that  $p$ -maxRPC maintains data structures that AC does not use.

---

<sup>1</sup> <http://cpai.ucc.ie/09/>

**Table 1.** Performance (CPU time, nodes and constraint checks) of AC,  $p$ -maxRPC, and maxRPC on various instances

|                  |        | AC            | $p$ -maxRPC           | maxRPC        |
|------------------|--------|---------------|-----------------------|---------------|
| scen1-f8         | CPU(s) | Time-out      | <b>1.39</b> (0.2)     | 6.10          |
|                  | #nodes | –             | 927                   | 917           |
|                  | #ccks  | –             | 1,397,440             | 26,932,990    |
| scen2-f24        | CPU(s) | Time-out      | <b>0.13</b> (0.3)     | 0.65          |
|                  | #nodes | –             | 201                   | 201           |
|                  | #ccks  | –             | 296,974               | 3,462,070     |
| scen3-f10        | CPU(s) | Time-out      | <b>0.89</b> (0.5)     | 2.80          |
|                  | #nodes | –             | 469                   | 408           |
|                  | #ccks  | –             | 874,930               | 13,311,797    |
| geo50-20d4-75-26 | CPU(s) | 111.48        | 17.80 (1.0)           | <b>15.07</b>  |
|                  | #nodes | 477,696       | 3,768                 | 3,768         |
|                  | #ccks  | 96,192,822    | 40,784,017            | 40,784,017    |
| geo50-20d4-75-43 | CPU(s) | 1,671.35      | <b>1,264.36</b> (0.5) | 1,530.02      |
|                  | #nodes | 4,118,134     | 555,259               | 279,130       |
|                  | #ccks  | 1,160,664,461 | 1,801,402,535         | 3,898,964,831 |
| geo50-20d4-75-46 | CPU(s) | 1,732.22      | <b>371.30</b> (0.6)   | 517.35        |
|                  | #nodes | 3,682,394     | 125,151               | 64,138        |
|                  | #ccks  | 1,516,856,615 | 584,743,023           | 1,287,674,430 |
| geo50-20d4-75-84 | CPU(s) | 404.63        | <b>0.44</b> (0.6)     | 0.56          |
|                  | #nodes | 2,581,794     | 513                   | 333           |
|                  | #ccks  | 293,092,144   | 800,657               | 1,606,047     |
| queensK10-5-add  | CPU(s) | <b>27.14</b>  | 30.79 (0.2)           | 98.44         |
|                  | #nodes | 82,208        | 81,033                | 78,498        |
|                  | #ccks  | 131,098,933   | 148,919,686           | 954,982,880   |
| queensK10-5-mul  | CPU(s) | <b>43.89</b>  | 83.27 (0.1)           | 300.74        |
|                  | #nodes | 74,968        | 74,414                | 70,474        |
|                  | #ccks  | 104,376,698   | 140,309,576           | 1,128,564,278 |

## 6 Adaptative Parameterized Consistency: $ap$ -maxRPC

In the previous section, we have defined  $p$ -maxRPC, a version of parameterized consistency where the strong local consistency is maxRPC. We have performed some initial experiments where  $p$  has the same value during the whole search and everywhere in the constraint network. However, the algorithm we proposed to enforce  $p$ -maxRPC does not specify how  $p$  is chosen. In this section, we propose two possible ways to dynamically and locally adapt the parameter  $p$  in order to solve the problem faster than both AC and maxRPC. Instead of using a single parameter  $p$  during the whole search and for the whole constraint network, we propose to use several local parameters and to adapt the level of local consistency by dynamically adjusting the value of the different local parameters during search. The idea is to concentrate the effort of propagation by increasing the level of consistency in the most difficult parts of the instance. We can determine these

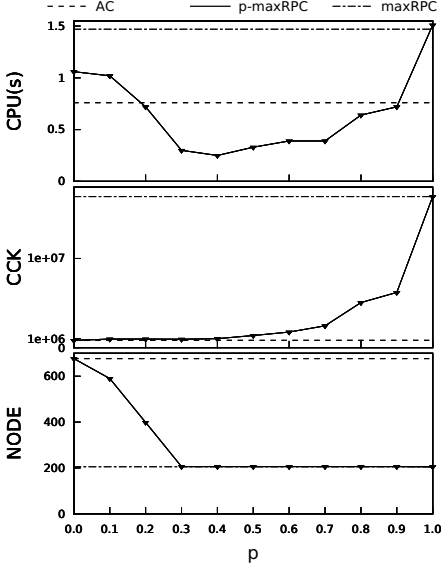


**Fig. 2.** Instance where  $p$ -maxRPC outperforms both AC and maxRPC

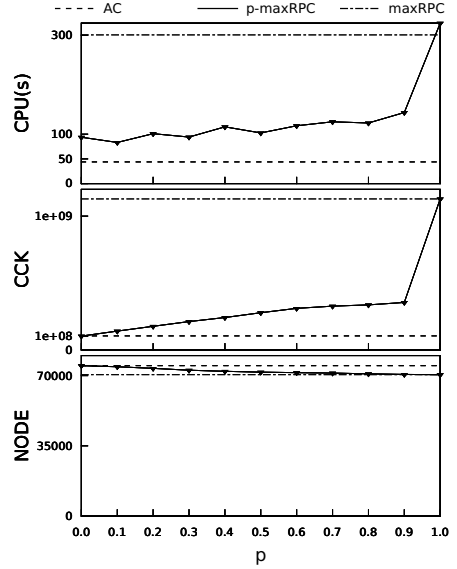


**Fig. 3.** Instance where AC outperforms  $p$ -maxRPC

difficult parts using heuristics based on conflicts in the same vein as the weight of a constraint or the weighted degree of a variable in [5].

### 6.1 Constraint-Based $ap$ -maxRPC : $apc$ -maxRPC

The first technique we propose, called constraint-based  $ap$ -maxRPC, assigns a parameter  $p(c_k)$  to each constraint  $c_k$  in  $C$ . We define this parameter to be correlated to the *weight* of the constraint. The idea is to apply a higher level of consistency in parts of the problem where the constraints are the most active.

**Definition 6 (The weight of a constraint [5]).** *The weight  $w(c_k)$  of a constraint  $c_k \in C$  is an integer that is incremented every time a domain wipe out occurs while performing propagation on this constraint.*

We define the adaptive parameter  $p(c_k)$  local to constraint  $c_k$  in such a way that it is greater when the weight  $w(c_k)$  is higher wrt to other constraints.

$$\forall c_k \in C, p(c_k) = \frac{w(c_k) - \min_{c \in C}(w(c))}{\max_{c \in C}(w(c)) - \min_{c \in C}(w(c))} \quad (1)$$

Equation 1 is normalized so that we are guaranteed that  $0 \leq p(c_k) \leq 1$  for all  $c_k \in C$  and that there exists  $c_{k_1}$  with  $p(c_{k_1}) = 0$  (the constraint with lowest weight) and  $c_{k_2}$  with  $p(c_{k_2}) = 1$  (the constraint with highest weight).

We are ready to define adaptive parameterized consistency based on constraints.

**Definition 7 (constraint-based *ap*-maxRPC).** A value  $v_i \in D(x_i)$  is constraint-based *ap*-maxRPC (or *apc*-maxRPC) on a constraint  $c_{ij}$  if and only if it is constraint-based  $p(c_{ij})$ -maxRPC. A value  $v_i \in D(x_i)$  is *apc*-maxRPC iff  $\forall c_{ij}, v_i$  is *apc*-maxRPC on  $c_{ij}$ . A constraint network is *apc*-maxRPC iff all values in all domains in  $D$  are *apc*-maxRPC.

## 6.2 Variable-Based *ap*-maxRPC: *apx*-maxRPC

The technique proposed in Section 6.1 can only be used on consistencies where the consistency of a value on a constraint is defined. We give a second technique which can be used on constraint-based or variable-based local consistencies differently. We instantiate our definitions to maxRPC but the extension to other consistencies is direct. We call this new technique variable-based *ap*-maxRPC. We need to define the weighted degree of a variable as the aggregation of the weights of all constraints involving it.

**Definition 8 (The weighted degree of a variable [5]).** The weighted degree  $wdeg(x_i)$  of a variable  $x_i$  is the sum of the weights of the constraints involving  $x_i$  and one other uninstantiated variable.

We associate each variable with an adaptive local parameter based on its weighted degree.

$$\forall x_i \in X, p(x_i) = \frac{wdeg(x_i) - \min_{x \in X}(wdeg(x))}{\max_{x \in X}(wdeg(x)) - \min_{x \in X}(wdeg(x))} \quad (2)$$

As in Equation 1, we see that the local parameter is normalized so that we are guaranteed that  $0 \leq p(x_i) \leq 1$  for all  $x_i \in X$  and that there exists  $x_{k_1}$  with  $p(x_{k_1}) = 0$  (the variable with lowest weighted degree) and  $x_{k_2}$  with  $p(x_{k_2}) = 1$  (the variable with highest weighted degree).

**Definition 9 (variable-based *ap*-maxRPC).** A value  $v_i \in D(x_i)$  is variable-based *ap*-maxRPC (or *apx*-maxRPC) if and only if it is value-based  $p(x_i)$ -maxRPC. A constraint network is *apx*-maxRPC iff all values in all domains in  $D$  are *apx*-maxRPC.

## 7 Experimental Evaluation of *ap*-maxRPC

In Section 5 we have shown that maintaining a static form of *p*-maxRPC during the whole search can lead to a promising trade-off between computational effort and pruning when all algorithms follow the same static variable ordering. In this section, we want to put our contributions in the real context of a solver using the best known variable ordering heuristic, *dom/wdeg*, though it is known that this heuristic is so good that it reduces a lot the differences in performance that other features of the solver could provide. We have compared the two variants of adaptive parameterized consistency introduced in Section 6 to AC and maxRPC.

We ran the four algorithms on radio link frequency assignment problems, geom problems, and queens knights problems.

Table 2 reports some representative results. A first observation is that, thanks to the *dom/wdeg* heuristic, we were able to solve more instances before the cutoff of one hour, especially the scen11 variants of RLFAP. A second observation is that *apc*-maxRPC and *apx*-maxRPC are both faster than at least one of the two extreme consistencies (AC and maxRPC) on all instances except scen7-w1-f4 and geo50-20-d4-75-30. Third, when *apx*-maxRPC and/or *apc*-maxRPC are faster than both AC and maxRPC (scen1-f9, scen2-f25, scen11-f9, scen11-f10 and scen11-f11), we observe that the gap in performance in terms of nodes and CCKs between AC and maxRPC is significant. Except for scen7-w1-f4, the number of nodes visited by AC is three to five times greater than the number of nodes visited by maxRPC and the number of constraint checks performed by maxRPC is twelve to sixteen times greater than the number of constraint checks performed by AC. For the Geom instances the CPU time of the *ap*-maxRPC algorithms is between AC and maxRPC, and it is never lower than the CPU time of AC. This probably means that when solving these instances with the *dom/wdeg* heuristic, there is no need for sophisticated local consistencies. In general we see that the *ap*-maxRPC algorithms fail to improve both the two extreme consistencies simultaneously for the instances where the performance gap between AC and maxRPC is low.

If we compare *apx*-maxRPC to *apc*-maxRPC, we observe that although *apx*-maxRPC is coarser in its design than *apc*-maxRPC, *apx*-maxRPC is often faster than *apc*-maxRPC. We can explain this by the fact that the constraints initially all have the same weight equal to 1. Hence, all local parameters  $ap(c_k)$  initially have the same value 0, so that *apc*-maxRPC starts resolution by applying AC everywhere. It will start enforcing some amount of maxRPC only after the first wipe-out occurred. On the contrary, in *apx*-maxRPC, when constraints all have the same weight, the local parameter  $p(x_i)$  is correlated to the degree of the variable  $x_i$ . As a result, *apx*-maxRPC benefits from the filtering power of maxRPC even before the first wipe-out.

In Table 2, we reported only the results on a few representative instances. Table 3 summarizes the whole set of experiments. It shows the average CPU time for each algorithm on all instances of the different classes of problems tested. We considered only the instances solved before the cutoff of one hour by at least one of the four algorithms. To compute the average CPU time of an algorithm on a class of problems, we add the CPU time needed to solve each instance solved before the cutoff of one hour, and for the instances not solved before the cutoff, we add one hour. We observe that the adaptive approach is, on average, faster than the two extreme consistencies AC and maxRPC, except on the Geom class.

In *apx*-maxRPC and *apc*-maxRPC, we update the local parameters  $p(x_i)$  or  $p(c_k)$  at each node in the search tree. We could wonder if such a frequent update does not produce too much overhead. To answer this question we performed a simple experiment in which we update the local parameters every 10 nodes only.

**Table 2.** Performance (CPU time, nodes and constraint checks) of AC, variable-based *ap*-maxRPC (*apx*-maxRPC), constraint-based *ap*-maxRPC (*apc*-maxRPC), and maxRPC on various instances

|                  |        | AC            | <i>apx</i> -maxRPC | <i>apc</i> -maxRPC | maxRPC        |
|------------------|--------|---------------|--------------------|--------------------|---------------|
| scen1-f9         | CPU(s) | 90.34         | <b>31.17</b>       | 33.40              | 41.56         |
|                  | #nodes | 2,291         | 1,080              | 1,241              | 726           |
|                  | #ccks  | 3,740,502     | 3,567,369          | 2,340,417          | 50,045,838    |
| scen2-f25        | CPU(s) | 70.57         | 46.40              | <b>27.22</b>       | 81.40         |
|                  | #nodes | 12,591        | 4,688              | 3,928              | 3,002         |
|                  | #ccks  | 15,116,992    | 38,239,829         | 8,796,638          | 194,909,585   |
| scen6-w2         | CPU(s) | 7.30          | 1.25               | 2.63               | <b>0.01</b>   |
|                  | #nodes | 2,045         | 249                | 610                | 0             |
|                  | #ccks  | 2,401,057     | 1,708,812          | 1,914,113          | 85,769        |
| scen7-w1-f4      | CPU(s) | 0.28          | <b>0.17</b>        | 0.54               | 0.30          |
|                  | #nodes | 567           | 430                | 523                | 424           |
|                  | #ccks  | 608,040       | 623,258            | 584,308            | 1,345,473     |
| scen11-f9        | CPU(s) | 2,718.65      | <b>1,110.80</b>    | 1,552.20           | 2,005.61      |
|                  | #ccks  | 103,506       | 40,413             | 61,292             | 32,882        |
|                  | #nodes | 227,751,301   | 399,396,873        | 123,984,968        | 3,637,652,122 |
| scen11-f10       | CPU(s) | 225.29        | <b>83.89</b>       | 134.46             | 112.18        |
|                  | #ccks  | 9,511         | 3,510              | 4,642              | 2,298         |
|                  | #nodes | 12,972,427    | 17,778,458         | 6,717,485          | 156,005,235   |
| scen11-f11       | CPU(s) | 156.76        | <b>39.39</b>       | 93.69              | 76.95         |
|                  | #ccks  | 7,050         | 2,154              | 3,431              | 1,337         |
|                  | #nodes | 7,840,552     | 10,006,821         | 5,143,592          | 91,518,348    |
| scen11-f12       | CPU(s) | 139.91        | 69.50              | 88.76              | <b>61.92</b>  |
|                  | #ccks  | 7,050         | 2,597              | 3,424              | 1,337         |
|                  | #nodes | 7,827,974     | 11,327,536         | 5,144,835          | 91,288,023    |
| geo50-20d4-75-19 | CPU(s) | <b>242.13</b> | 553.53             | 657.72             | 982.34        |
|                  | #nodes | 195,058       | 114,065            | 160,826            | 71,896        |
|                  | #ccks  | 224,671,319   | 594,514,132        | 507,131,322        | 2,669,750,690 |
| geo50-20d4-75-30 | CPU(s) | <b>0.84</b>   | 1.01               | 1.07               | 1.02          |
|                  | #nodes | 359           | 115                | 278                | 98            |
|                  | #ccks  | 261,029       | 432,705            | 313,168            | 1,880,927     |
| geo50-20d4-75-84 | CPU(s) | <b>0.02</b>   | 0.09               | 0.05               | 0.29          |
|                  | #nodes | 59            | 54                 | 59                 | 52            |
|                  | #ccks  | 33,876        | 80,626             | 32,878             | 697,706       |
| queensK20-5-mul  | CPU(s) | 787.35        | 2,345.43           | <b>709.45</b>      | Time-out      |
|                  | #codes | 55,596        | 40,606             | 41,743             | –             |
|                  | #ccks  | 347,596,389   | 6,875,941,876      | 379,826,516        | –             |
| queensK15-5-add  | CPU(s) | 24.69         | 17.01              | <b>14.98</b>       | 35.05         |
|                  | #codes | 24,639        | 12,905             | 12,677             | 11,595        |
|                  | #ccks  | 90,439,795    | 91,562,150         | 58,225,434         | 394,073,525   |

**Table 3.** Average CPU time of AC, variable-based *ap*-maxRPC (*apx*-maxRPC), constraint-based *ap*-maxRPC (*apc*-maxRPC), and maxRPC on all instances of each class of problems tested, when the local parameters are updated at each node

|              |         | AC           | <i>apx</i> -maxRPC | <i>apc</i> -maxRPC | maxRPC |
|--------------|---------|--------------|--------------------|--------------------|--------|
| Average(CPU) | geom    | <b>69.28</b> | 180.57             | 191.03             | 279.30 |
|              | scen    | 18.95        | 9.63               | <b>8.30</b>        | 13.94  |
|              | scen11  | 810.15       | <b>325.90</b>      | 467.28             | 564.17 |
|              | queensK | 135.95       | 395.41             | <b>121.75</b>      | 610.51 |

**Table 4.** Average CPU time of AC, variable-based *ap*-maxRPC (*apx*-maxRPC), constraint-based *ap*-maxRPC (*apc*-maxRPC), and maxRPC on all instances of each class of problems tested, when the local parameters are updated every 10 nodes

|              |         | AC           | <i>apx</i> -maxRPC | <i>apc</i> -maxRPC | maxRPC |
|--------------|---------|--------------|--------------------|--------------------|--------|
| Average(CPU) | geom    | <b>69.28</b> | 147.20             | 189.42             | 279.30 |
|              | scen    | 18.95        | <b>7.40</b>        | 8.86               | 13.94  |
|              | scen11  | 810.15       | <b>311.74</b>      | 417.97             | 564.17 |
|              | queensK | 135.95       | 269.51             | <b>117.18</b>      | 610.52 |

We re-ran the whole set of experiments with this new setting. Table 4 reports the CPU time average results. We observe that when the local parameters are updated every 10 nodes, the gain for the adaptive approach is, on average, greater than when the local parameters are updated at each node. This gives room for improvement, by trying to adapt the frequency of update of these parameters.

## 8 Conclusion

We have introduced the notion of stability of values for arc consistency, a notion based on the depth of their supports in their domain. We have used this notion to propose parameterized consistency, a technique that allows to define levels of local consistency of increasing strength between arc consistency and a given strong local consistency. We have instantiated the generic parameterized consistency approach to max restricted path consistency. We have experimentally shown that the concept of parameterized consistency is viable. Then we have introduced two techniques which allow us to make the parameter adaptable dynamically and locally during search. We have evaluated these two techniques experimentally and we have observed that adapting the level of local consistency during search using the parameterized consistency concept is a promising approach that can outperform both MAC and a strong local consistency on many problems.

## References

1. Balafoutis, T., Paparrizou, A., Stergiou, K., Walsh, T.: New algorithms for max restricted path consistency. *Constraints* 16(4), 372–406 (2011)
2. Bessiere, C.: Constraint propagation. In: Rossi, F., van Beek, P., Walsh, T. (eds.) *Handbook of Constraint Programming*, ch. 3, Elsevier (2006)
3. Bessiere, C., Régin, J.C., Yap, R.H.C., Zhang, Y.: An optimal coarse-grained arc consistency algorithm. *Artif. Intell.* 165(2), 165–185 (2005)
4. Bessiere, C., Stergiou, K., Walsh, T.: Domain filtering consistencies for non-binary constraints. *Artif. Intell.* 172(6-7), 800–822 (2008)
5. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: *ECAI*, pp. 146–150 (2004)
6. Debruyne, R., Bessiere, C.: Domain filtering consistencies. *Journal of Artificial Intelligence Research* 14, 205–230 (2001)
7. Debruyne, R., Bessière, C.: Some practicable filtering techniques for the constraint satisfaction problem. In: *IJCAI* (1), pp. 412–417 (1997)
8. Katriel, I., Van Hentenryck, P.: Randomized filtering algorithms. Technical Report CS-06-09, Brown University (June 2006)
9. Sellmann, M.: Approximated consistency for knapsack constraints. In: Rossi, F. (ed.) *CP 2003. LNCS*, vol. 2833, pp. 679–693. Springer, Heidelberg (2003)
10. Stergiou, K.: Heuristics for dynamically adapting propagation in constraint satisfaction problems. *AI Commun.* 22, 125–141 (2009)



# Global Inverse Consistency for Interactive Constraint Satisfaction\*

Christian Bessiere<sup>1</sup>, Hélène Fargier<sup>2</sup>, and Christophe Lecoutre<sup>3</sup>

<sup>1</sup> LIRMM-CNRS, University of Montpellier, France

<sup>2</sup> IRIT-CNRS, University of Toulouse, France

<sup>3</sup> CRIL-CNRS, University of Artois, Lens, France

bessiere@lirmm.fr, fargier@irit.fr, lecoutre@cril.fr

**Abstract.** Some applications require the interactive resolution of a constraint problem by a human user. In such cases, it is highly desirable that the person who interactively solves the problem is not given the choice to select values that do not lead to solutions. We call this property *global inverse consistency*. Existing systems simulate this either by maintaining arc consistency after each assignment performed by the user or by compiling offline the problem as a multi-valued decision diagram. In this paper, we define several questions related to global inverse consistency and analyse their complexity. Despite their theoretical intractability, we propose several algorithms for enforcing global inverse consistency and we show that the best version is efficient enough to be used in an interactive setting on several configuration and design problems. We finally extend our contribution to the inverse consistency of tuples.

## 1 Introduction

Constraint Programming (CP) is widely used to express and solve combinatorial problems. Once the problem is modelled as a constraint network, efficient solving techniques generate a solution satisfying the constraints, if such a solution exists. However, there are situations where the user has strong opinions about the way to build good solutions to the problem but some of the desirable/undesirable combinations will become clear only once some of the variables are assigned. In this case, the constraint solver should be there to assist the user in the solution design and to ensure her choices remain in the feasible space, removing the combinatorial complexity from her shoulders. See the Synthia system for protein design as an early example of using CP to interactively solve a problem [12]. Another well known example of such an interactive solving of constraint-based models is product configuration [7,1]. The person modelling the product as a constraint network for the company knows its technical and marketing requirements. She models the feasibility, availability and/or marketing constraints about the product. This constraint network captures the catalog of possible products, which may contain billions of solutions, but in an intentional and compact way. Nevertheless, the modeller does not know the constraints or preferences of the customer(s). Now, this

---

\* This work has been funded by the ANR (“Agence Nationale de la Recherche”) project BR4CP (ANR-11-BS02-008).

is the customer who will look for solutions, with her own constraints and preferences on the price, the colour, or any other configurable feature.

These applications refer to an interactive solving process where the user selects values for variables according to her own preferences and the system checks the constraints of the network, until all variables are assigned and satisfy all constraints of the network. This solving policy raises an important issue: the person who interactively solves the problem should not be led to a dead-end where satisfying all constraints of the network is impossible. Existing interactive solving systems address this issue either by compiling the constraint network into a multivalued decision diagram (MDD) at the modelling phase [1,9,10] or by enforcing arc consistency on the network after each assignment performed by the user [12]. Compiling the constraint network as a MDD can require a significant amount of time and space. That is why compilation is performed offline (before the solving session). As a consequence, configurators based on a MDD compilation are restricted to static constraint networks: non-unary constraints can neither be added nor removed once the network compiled. It is thus not possible for the user to perform complex requirements, e.g., she is interested in travelling to Venezia only during the carnival period. Arc and dynamic arc consistencies require a lighter computational effort but the user can be trapped in dead-ends, which is very risky from a commercial point of view. It has been shown in [5] that arc consistency (and even higher levels of local consistency) can be very bad approximations of the ideal state where all values remaining in the network can be extended to solutions.

The message of our paper is that for many of the problems that require interactive solving of the problem, and especially for real problems, it is computationally feasible to maintain the domains of the variables in a state where they only contain those values which belong to a complete solution extending the current choices of the user. Inspired by the nomenclature used in [6] and [15], we call this level of consistency *global inverse consistency* (GIC).

Our contribution addresses several aspects. First, we formally characterise the questions that underlie the interactive constraint solving loop and we show that they are all NP-hard. Second, we provide several algorithms with increasing sophistication to address those tasks and we experimentally show that the most efficient one is efficient enough to be used in an interactive constraint solving loop of several non trivial configuration and design problems. Third, we finally extend all these contributions to the *positive consistency* of constraints, which is a problem closely related to GIC that appears in configuration.

## 2 Background

A (discrete) constraint network (CN)  $N$  is composed of a finite set of  $n$  variables, denoted by  $vars(N)$ , and a finite set of  $e$  constraints, denoted by  $cons(N)$ . Each variable  $x$  has a domain which is the finite set of values that can be assigned to  $x$ . The initial domain of a variable  $x$  is denoted by  $dom^{init}(x)$  whereas the current domain of  $x$  is denoted by  $dom(x)$ ; we always have  $dom(x) \subseteq dom^{init}(x)$ . The maximum domain size for a given CN will be denoted by  $d$ . To simplify, a variable-value pair  $(x, a)$  such that  $x \in vars(N)$  and  $a \in dom(x)$  is called a value of  $N$ . Each constraint  $c$  involves

an ordered set of variables, called the *scope* of  $c$  and denoted by  $scp(c)$ , and is semantically defined by a relation, denoted by  $rel(c)$ , which contains the set of tuples allowed for the variables involved in  $c$ . The *arity* of a constraint  $c$  is the size of  $scp(c)$ , and will usually be denoted by  $r$ .

An *instantiation*  $I$  of a set  $X = \{x_1, \dots, x_k\}$  of variables is a set  $\{(x_1, a_1), \dots, (x_k, a_k)\}$  such that  $\forall i \in 1..k, a_i \in dom^{init}(x_i)$ ;  $X$  is denoted by  $vars(I)$  and each  $a_i$  is denoted by  $I[x_i]$ . An instantiation  $I$  on a CN  $N$  is an instantiation of a set  $X \subseteq vars(N)$ ; it is *complete* if  $vars(I) = vars(N)$ .  $I$  is *valid* on  $N$  iff  $\forall (x, a) \in I, a \in dom(x)$ .  $I$  *covers* a constraint  $c$  iff  $scp(c) \subseteq vars(I)$ , and  $I$  *satisfies* a constraint  $c$  with  $scp(c) = \{x_1, \dots, x_r\}$  iff (i)  $I$  covers  $c$  and (ii) the tuple  $(I[x_1], \dots, I[x_r]) \in rel(c)$ . An instantiation  $I$  on a CN  $N$  is *locally consistent* iff (i)  $I$  is valid on  $N$  and (ii) every constraint of  $N$  covered by  $I$  is satisfied by  $I$ . A *solution* of  $N$  is a complete locally consistent instantiation on  $N$ ;  $sols(N)$  denotes the set of solutions of  $N$ . A CN  $N$  is *satisfiable* iff  $sols(N) \neq \emptyset$ .

The ubiquitous example of constraint propagation is enforcement of *generalised arc consistency* (GAC) which removes values from domains without reducing the set of solutions of the constraint network. A value  $(x, a)$  of a CN  $N$  is GAC on  $N$  iff for every constraint  $c$  of  $N$  involving  $x$ , there exists a valid instantiation  $I$  of  $scp(c)$  such that  $I$  satisfies  $c$  and  $I[x] = a$ .  $N$  is GAC iff every value of  $N$  is GAC. Enforcing GAC means removing GAC-inconsistent values from domains until the constraint network is GAC. In this paper, we shall refer to MAC which is an algorithm considered to be among the most efficient generic approaches for the solution of CNs. MAC [17] explores the search space depth-first and enforces (generalised) arc consistency after each decision taken (variable assignment or value refutation) during search. A *past* variable is a variable explicitly assigned by the search algorithm whereas a *future* variable is a variable not (explicitly) assigned. The set of future variables of a CN  $N$  is denoted by  $vars^{fut}(N)$ .

### 3 Problems Raised by Interactive Constraint Solving

In this section we formally characterise the questions that underlie the interactive constraint solving loop and we study their theoretical complexity.

#### 3.1 Formalization

We first need to define global inverse consistency.

**Definition 1 (Global Inverse Consistency).** A value  $(x, a)$  of a CN  $N$  is globally inverse consistent (GIC) iff  $\exists I \in sols(N) \mid I[x] = a$ . A CN  $N$  is GIC iff every value of  $N$  is GIC.

The *GIC closure* of  $N$  is the CN obtained from  $N$  by removing all the values that do not belong to a solution of  $N$ . The obvious problems that follow are to check whether a constraint network is GIC or not, and to enforce GIC.

**Problem 1 (Deciding GIC).** Given a CN  $N$ , is  $N$  GIC?

**Problem 2 (Computing GIC).** *Given a CN  $N$ , compute the GIC closure of  $N$ .*

As we are interested in interactive solving, we define the problem of restoring (maintaining) GIC after the user has performed a variable assignment.

**Problem 3 (Restoring GIC).** *Given a CN  $N$  that is GIC, and a value  $(x, a)$  of  $N$ , restore GIC after the assignment  $x = a$  has been performed.*

In a configuration setting, as soon as some mandatory variables have been set, the user can ask for an automatic completion of the remaining variables. Hence the definition of following problem:

**Problem 4 (Solving a GIC network).** *Given a CN  $N$  that is GIC, find a solution to  $N$ .*

### 3.2 Complexity Results

Not surprisingly, the basic questions related to GIC (Problems 1 and 2) are intractable.

**Theorem 1 (Problem 1).** *Deciding whether a constraint network  $N$  is GIC is NP-complete, even if  $N$  is satisfiable.*

*Proof.* We first prove membership to NP. For each value  $(x, a)$  of  $N$ , it is sufficient to provide a solution  $I$  of  $N$  such that the projection  $I[x]$  of  $I$  on variable  $x$  is equal to  $a$ . This certificate has size  $n \cdot n \cdot d$  and can be checked in polynomial time.

Completeness for NP is proved by reducing 3COL to the problem of deciding whether a satisfiable CN is GIC. Take any instance of the 3COL problem, that is, a graph  $G = (V, E)$ . Consider the CN  $N$  where  $\text{vars}(N) = \{x_i \mid i \in V\}$ ,  $\text{dom}(x_i) = \{0, 1, 2, 3\}$ ,  $\forall i \in V$ , and  $\text{cons}(N) = \{(x_i \neq x_j) \vee (x_i = 0 \wedge x_j = 0) \mid (i, j) \in E\}$ . Clearly  $[0, \dots, 0]$  is a solution of  $N$ , and by construction,  $N$  has other solutions iff  $G$  is 3-colourable. Now, if  $G$  is 3-colourable,  $N$  is GIC because colours are completely interchangeable. Therefore,  $N$  is GIC iff  $G$  is 3-colourable.  $\square$

Our proof shows that hardness for deciding GIC holds for binary CNs (i.e., CNs only involving binary constraints). We have another proof, inspired from that used in Theorem 3 in [2], that shows that deciding GIC is still hard for Boolean domains and quaternary constraints.

**Theorem 2 (Problem 2).** *Computing the GIC closure of a constraint network  $N$  is NP-hard and NP-easy, even if  $N$  is satisfiable.*

*Proof.* We prove NP-easiness by showing that a polynomial number of calls to a NP oracle are sufficient to build the GIC closure of  $N$ . For each value  $(x, a)$  of  $N$ , we ask the NP oracle whether  $N$  with the extra constraint  $x = a$  is satisfiable (we call this an *inverse check*). Once all values have been tested, we build the GIC closure of  $N$  by removing from each  $\text{dom}(x)$  all values  $a$  for which the oracle test returned 'no'. Hardness is a direct corollary of Theorem 1.  $\square$

Notice that the two previous intractability results are still valid when the CN is satisfiable, as is the case at the beginning of an interactive resolution session.

We finally show that Problems 3 and 4 are unfortunately not easier than checking GIC or enforcing GIC from scratch. But they are not harder.

**Theorem 3 (Problem 3).** *Given a CN  $N$  that is GIC, and a value  $(y, b)$  of  $N$ , computing the GIC closure of the CN  $N'$ , where  $\text{vars}(N') = \text{vars}(N)$  and  $\text{cons}(N') = \text{cons}(N) \cup \{y = b\}$  is NP-hard and NP-easy.*

*Proof.* NP-easiness is proved as in the proof of Theorem 2 by showing that a polynomial number of calls to a NP oracle are sufficient to build the GIC closure of  $N'$ . For each value  $(x, a)$  of  $N$  (except values  $(y, a)$  with  $a \neq b$ ), we ask the NP oracle whether  $N'$  with the extra constraint  $x = a$  is satisfiable. Once all values have been tested, we build the closure of  $N'$  by removing from  $\text{dom}(y)$  all values  $a \neq b$  and removing from each  $\text{dom}(x)$  all values  $a$  for which the oracle test returned 'no'. Hardness is a direct corollary of Theorem 7 in [2].  $\square$

**Theorem 4 (Problem 4).** *Generating a solution to a GIC constraint network cannot be done in polynomial time, unless  $P = NP$ .*

*Proof.* The following proof is derived from [16]. But it is also a corollary of the recent and more complex Theorem 3.1 in [8].

Suppose we have an algorithm  $A$  that generates a solution to a GIC constraint network  $N$  in time bounded by a polynomial  $p(|N|)$ . Take any instance of the 3COL problem, that is, a graph  $G = (V, E)$ . Consider the CN  $N$  where  $\text{vars}(N) = \{x_i \mid i \in V\}$ ,  $\text{dom}(x_i) = \{0, 1, 2\}$ ,  $\forall i \in V$ , and  $\text{cons}(N) = \{x_i \neq x_j \mid (i, j) \in E\}$ .  $N$  has a solution iff  $G$  is 3-colourable. Now, if  $G$  is 3-colourable,  $N$  is GIC because colours are completely interchangeable. Thus, it is sufficient to run  $A$  during  $p(|N|)$  steps. If it returns a solution to  $N$ , then the 3COL instance is satisfiable. Otherwise, the 3COL instance is unsatisfiable. Therefore, as 3COL is NP-complete, there cannot exist a polynomial algorithm for generating a solution to a GIC constraint network, unless  $P = NP$ .  $\square$

## 4 GIC Algorithms

In this section, we introduce four algorithms to enforce global inverse consistency. These GIC algorithms use increasingly sophisticated data structures and techniques that have recently proved their worth in filtering algorithms proposed in the literature; e.g., see [14, 18]. To simplify our presentation, we assume that the CNs are satisfiable, which is the case in interactive resolution, allowing us to avoid handling domain wipe-outs in the GIC procedures. Note that these algorithms can be used to enforce GIC, but also to maintain it during a user-driven search. This is why we refer to the set  $\text{vars}^{fut}(N)$  of future variables in some instructions.

The first algorithm, GIC1, described in Algorithm 1, is really basic: it will be used as our baseline during our experiments. For each value  $a$  in the domain of a future variable  $x$ , a solution for the CN  $N$  where  $x$  is assigned the value  $a$ , denoted by  $N|_{x=a}$ , is sought using a complete search algorithm. This search algorithm, called here searchSolutionFor, either returns the first solution that can be found, or the special value *nil*. Our implementation choice will be the algorithm MAC that maintains (G)AC during a backtrack search [17]. Hence, in Algorithm 1, when it is proved with searchSolutionFor that no solution exists, i.e.,  $I = \text{nil}$ , the value  $a$  can be deleted. Note that, in contrary to weaker forms of consistency, when a value is pruned there is no need for GIC to repeat the process of iterating over the values remaining in the CN.

**Algorithm 1.** GIC1( $N$ : CN)

---

```

1 foreach variable  $x \in vars^{fut}(N)$  do
2   foreach value  $a \in dom(x)$  do
3      $I \leftarrow \text{searchSolutionFor}(N|_{x=a})$ 
4     if  $I = nil$  then
5        $\_$  remove  $a$  from  $dom(x)$ 

```

---

**Algorithm 2.** handleSolution2/3( $x$ : variable,  $I$ : instantiation)

---

```

1 foreach variable  $y \in vars^{fut}(N) \mid y \text{ is revised after } x$  do
2   if  $\text{stamp}[y][I[y]] \neq \text{time}$  then
3      $\_$   $\text{stamp}[y][I[y]] \leftarrow \text{time}$ 
4      $\_$   $\text{nbGic}[y] ++$ 

```

---

**Algorithm 3.** isValid( $X$ : set of variables,  $I$ : instantiation): Boolean

---

```

1 foreach variable  $x \in X$  do
2   if  $I[x] \notin dom(x)$  then
3      $\_$  return false
4 return true

```

---

**Algorithm 4.** GIC2/3( $N$ : CN)

---

**Data:** GIC3 is obtained by considering light grey coloured instructions between lines 5 and 6, and after line 10

---

```

1  $\text{time} ++$ 
2 foreach variable  $x \in vars^{fut}(N)$  do
3    $\_$   $\text{nbGic}[x] \leftarrow 0$ 
4 foreach variable  $x \in vars^{fut}(N) \mid \text{nbGic}[x] < |dom(x)|$  do
5   foreach value  $a \in dom(x) \mid \text{stamp}[x][a] < \text{time}$  do
6     if  $\text{isValid}(vars(N), \text{residue}[x][a])$  then
7        $\_$  handleSolution2/3( $x, \text{residue}[x][a]$ )
8        $\_$  continue
9      $I \leftarrow \text{searchSolutionFor}(N|_{x=a})$ 
10    if  $I = nil$  then
11       $\_$  remove  $a$  from  $dom(x)$ 
12    else
13       $\_$  handleSolution2/3( $x, I$ )
14       $\_$   $\text{residue}[x][a] \leftarrow I$ 

```

---

The second algorithm, GIC2 described in Algorithm 4 (ignoring light grey lines), uses timestamping. This is useful when GIC is maintained during a user-driven search. We use an integer variable `time` for counting time, and we introduce a two-dimensional array `stamp` that associates with each value  $(x, a)$  of the CN the last time (value of `stamp[x][a]`) a solution was found for that value. We also assume that variables are implicitly totally ordered (for example, in lexicographic order). Then, the idea is to increment the value of the variable `time` whenever a new call to GIC2 is performed (see line 1) and to test `time` against each value  $(x, a)$  of the CN (see line 5) to determine whether it is necessary or not to search for a solution for  $(x, a)$ . When a solution  $I$  is found, function `handleSolution2/3` is called at line 10 in order to update stamps. Actually, we only update the stamps of values in  $I$  corresponding to variables that are processed after  $x$  in the loop of revisions (line 4) in Algorithm 4. These are the variables that have not been processed yet by the loop at line 4 of Algorithm 4. Finally, by further introducing a one-dimensional array `nbGic` that associates with each variable  $x$  of the CN the number of values in  $dom(x)$  that have been proved to be GIC, it is possible to avoid some iterations of loop 5; see initialization at lines 2-3, testing at line 4 and update at line 4 of Algorithm 2.

The third algorithm, GIC3, described in Algorithm 4 when considering light grey lines, can be seen as a refinement of GIC2 obtained by exploiting residues, which correspond to solutions that have been previously found. Here, we introduce a two-dimensional array `residue` that associates with each value  $(x, a)$  of the CN the last solution found for this value (potentially, during another call to GIC3). Because residual solutions may not be valid anymore, for each value  $(x, a)$  we need to test the validity of `residue[x][a]` by calling the function `isValid`; see instructions between lines 5 and 6. If the residue is valid, we call `handleSolution2/3` to update the other data structures, and we continue with the next value in the domain of  $x$ . A validity test, Algorithm 3, only checks that all values in a given complete instantiation are still present in the current domains. Of course, when a new solution is found, we record it as a residue; see instruction after line 10.

Our last algorithm, GIC4 described in Algorithm 6, is based on an original use of simple tabular reduction [18]. The principle is to record all solutions found during the enforcement of GIC in a table, so that an (adaptation of an) algorithm such as STR2 [13] can be applied. The current table is given by all elements of an array `solutions` at indices ranging from 1 to `nbSolutions`. As for STR2, we introduce two sets of variables called  $S^{val}$  and  $S^{sup}$ . The former allows us to limit validity control of solutions to the variables whose domains have changed recently (i.e., since the last execution of GIC4). This is made possible by reasoning from domain cardinalities, as performed at lines 3 and 26–27 with the array `lastSize`. The latter ( $S^{sup}$ ) contains any future variable  $x$  for which at least one value is not in the array `gicValues[x]`, meaning that it has still to be proved GIC. Related details can be found in [13]. After the initialization of  $S^{val}$  and  $S^{sup}$  (lines 1–8), each instantiation `solutions[i]` of the current table is processed (lines 11–16). If it remains valid (hence, a solution), we update structures `gicValues` and  $S^{sup}$  by calling the function `handleSolution4`. Otherwise, this instantiation is deleted by swapping it with the last one. The rest of the algorithm (lines 17–25) just tries to find a solution support for each value not present in `gicValues`. When a

**Algorithm 5.** handleSolution4( $I$  : instantiation)

---

```

1 foreach variable  $x \in S^{sup}$  do
2   if  $I[x] \notin \text{gicValues}[x]$  then
3      $\text{gicValues}[x] \leftarrow \text{gicValues}[x] \cup \{I[x]\}$ 
4     if  $|\text{gicValues}[x]| = |\text{dom}(x)|$  then
5        $S^{sup} \leftarrow S^{sup} \setminus \{x\}$ 

```

---

**Algorithm 6.** GIC4( $N$ : CN)

---

```

// Initialization of structures
1  $S^{val} \leftarrow \emptyset$ 
2 foreach variable  $x \in \text{vars}(N)$  do
3   if  $|\text{dom}(x)| \neq \text{lastSize}[x]$  then
4      $S^{val} \leftarrow S^{val} \cup \{x\}$ 
5  $S^{sup} \leftarrow \emptyset$ 
6 foreach variable  $x \in \text{vars}^{fut}(N)$  do
7    $\text{gicValues}[x] \leftarrow \emptyset$ 
8    $S^{sup} \leftarrow S^{sup} \cup \{x\}$ 

// The table of current solutions is traversed
9  $i \leftarrow 1$ 
10 while  $i \leq \text{nbSolutions}$  do
11   if  $\text{isValid}(S^{val}, \text{solutions}[i])$  then
12     handleSolution4( $\text{solutions}[i]$ )
13      $i++$ 
14   else
15      $\text{solutions}[i] \leftarrow \text{solutions}[\text{nbSolutions}]$ 
16      $\text{nbSolutions}--$ 

// Search for values not currently supported is performed
17 foreach variable  $x \in S^{sup}$  do
18   foreach value  $a \in \text{dom}(x) \setminus \text{gicValues}[x]$  do
19      $I \leftarrow \text{searchSolutionFor}(N|_{x=a})$ 
20     if  $I = \text{nil}$  then
21       remove  $a$  from  $\text{dom}(x)$ 
22     else
23        $\text{nbSolutions}++$ 
24        $\text{solutions}[\text{nbSolutions}] \leftarrow I$ 
25       handleSolution4( $I$ )

26 foreach variable  $x \in \text{vars}^{fut}(N)$  do
27    $\text{lastSize}[x] \leftarrow |\text{dom}(x)|$ 

```

---



new solution is found, it is recorded in the current table (lines 23–24) and `handleSolution4` is called (line 25).

**Theorem 5.** *Algorithms GIC1, GIC2, GIC3 and GIC4 enforce GIC.*

*Proof.* (sketch) This is immediate for GIC1. For GIC2 and GIC3, the use of timestamps and residues permits us to avoid useless inverse checks. For GIC4, the same arguments as those used for proving that STR2 enforces GAC hold. Simply, additional inverse checks are performed for values not collected (in `gicValues`) during the traversal of the current table.  $\square$

The worst-case space complexity (for the specific data structures) of GIC1 is  $O(1)$ . For GIC2 and GIC3, this is  $O(nd)$  because `nbGic` is  $O(n)$ , `stamp` and `residue` are  $O(nd)$ . For GIC4,  $S^{val}$ ,  $S^{sup}$  and `lastSize` are  $O(n)$ , `gicValues` is  $O(nd)$ , and the structure `solutions` is  $O(n^2d)$ . The time complexity of the GIC algorithms can be expressed in term of the number of calls to the (oracle) `searchSolutionFor`. For GIC1, this is  $O(nd)$ . For GIC2, in the *best-case*, only  $d$  calls are necessary, one call permitting to prove (through timestamping) that  $n$  values are GIC. For GIC3 and GIC4, still in the *best-case* and assuming the case of maintaining GIC (i.e., after the assignment of a variable by the user), no call to the oracle is necessary (residues and the current table permit alone to prove that all values are GIC). This rough analysis of time complexity suggests that GIC3 and GIC4 might be the best options.

## 5 Tuple Inverse Consistency

Up to this point, we have based our analysis on the last part of the interactive resolution process, i.e., the specification of a solution of the constraint network by the user. This allowed us to make the simplifying assumptions that the user is only looking at the domains of the variables. After each variable assignment, she just wants to know which values remain feasible for non assigned variables.

The situation is different at the modelling phase, e.g., the engineers of the company dynamically build the set of constraints that define the configurable product. At this point, GIC is also a crucial functionality, not for deriving a solution (a end product), but to ensure that each of the options proposed in the catalog (each of the values in the domains of the constraint network) is present in at least one end product. It is meaningless to propose (and advertise on) a sophisticated air bag system when it cannot equip any car in practice.

In that modelling phase, the need for information on the extensibility to solutions is not restricted to domains, but extends to (some of) the constraints of the model. Many constraints have actually a double meaning. Following the standard semantics of constraints, the first one is negative: technical constraints forbid combinations of variables. The second one is positive: the possibilities that are left by some (generally, table) constraints *have to* be effective. Let us assume, for instance, that a constraint means 'The level of equipment of vehicles with type M3 engine can be middle level or luxurious'. If some other constraint excludes the vehicles M3 engine for luxurious level of equipment, the specification of the product is considered as inconsistent. This property has been called *positive consistency* in [2] and actually refers to the extensibility to a solution of each of the tuples allowed by the constraint of interest:

**Definition 2 (Tuple Inverse Consistency).** *Given a CN  $N$ , a tuple  $\tau$  on a set of variable  $X$  is said to be inverse consistent (TIC) in  $N$  iff there exists a solution  $I$  of  $N$  such that  $\forall x \in X, I[x] = \tau[x]$ .*

**Definition 3 (Positive Consistency).** *A constraint  $c$  is positively consistent in  $N$  iff for any valid tuple  $\tau \in \text{rel}(c)$ ,  $\tau$  is TIC.*

The positive closure of a constraint  $c$  is the constraint obtained from  $c$  by removing from  $\text{rel}(c)$  all the valid tuples that are not TIC in  $N$ . The obvious problem that follows is to check whether a constraint is positively consistent or not.

**Problem 5 (Deciding Positive Consistency).** *Given a CN  $N$  and a constraint  $c$  of  $N$ , is  $c$  positively consistent in  $N$ ?*

Deciding positive consistency has been shown to be NP-hard, even when the constraint network is known to be satisfiable ([2]). The other problem of interest is to restore positive consistency on a constraint after the user has refined her model by adding a constraint to the network.

**Problem 6 (Restoring Positive Consistency).** *Given a CN  $N$ , given a positive consistent constraint  $c$  in  $N$ , given any extra constraint  $c'$  not in  $\text{cons}(N)$ , compute the new positive closure of  $c$  in the network obtained from  $N$  by adding  $c'$  to  $\text{cons}(N)$ .*

## 6 Experiments

In order to show the practical interest of our approach, we have performed several experiments mainly using a computer with processors Intel(R) Core(TM) i7-2820QM CPU 2.30GHz; for random instances, we used a cluster of Xeon 3.0GHz with 13GB of RAM. Our main purpose was to determine whether maintaining GIC is a viable option for configuration-like problem instances and for interactive puzzle creation, as well as to compare the relative efficiency of the four GIC algorithms described in Section 4.

**Table 1.** Features of six Renault configuration instances

|            | $n$ | $d$ | $e$ | $r$ | $t$    | $D$   | $T$     |
|------------|-----|-----|-----|-----|--------|-------|---------|
| souffleuse | 32  | 12  | 35  | 3   | 55     | 145   | 350     |
| megane     | 99  | 42  | 113 | 10  | 48,721 | 396   | 194,838 |
| master     | 158 | 324 | 195 | 12  | 26,911 | 732   | 183,701 |
| small      | 139 | 16  | 147 | 8   | 222    | 340   | 3,044   |
| medium     | 148 | 20  | 174 | 10  | 2,718  | 424   | 9,532   |
| big        | 268 | 324 | 332 | 12  | 26,881 | 1,273 | 225,989 |

In Table 1, we show relevant features of car configuration instances, generated with the help of our industrial partner Renault. For each of the six instances currently available,<sup>1</sup> we indicate

<sup>1</sup> See <http://www.irit.fr/~Helene.Fargier/BR4CP/benches.html>

- the number of variables ( $n$ ),
- the size of the greatest domain ( $d$ ),
- the number of constraints ( $e$ ),
- the greatest constraint arity ( $r$ ),
- the size of the greatest table ( $t$ ),
- the total number of values ( $D = \sum_{x \in vars(N)} |dom(x)|$ ),
- and the total number of tuples ( $T = \sum_{c \in cons(N)} |rel(c)|$ ).

The left part of Table 2 presents the CPU time required to establish GIC on the six Renault configuration instances. Clearly GIC1 is outperformed by the three other algorithms, which have here rather similar efficiency. The right part of Table 2 aims at simulating the behaviour of a configuration software user who makes the variable choices and value selections. It presents the CPU time required to maintain GIC along a single branch built by performing random variable assignments (random variable assignment simulates the user, who chooses the variables and the values according to her preference). Specifically, variables and values are randomly selected in turn, and after each assignment, GIC is systematically enforced to maintain this property. Of course, no conflict (dead-end) can occur along the branch due to the strength of GIC, which is why we use the term of greedy executions. CPU times are given on average for 100 executions (different random orderings). For all instances, GIC3 and GIC4 are maintained very fast, whereas on the biggest instances, GIC2 requires a few seconds and GIC1 around ten seconds.

**Table 2.** CPU time (in seconds) to establish GIC on Renault configuration instances, and to maintain it (average over 100 random greedy executions)

|            | Establishing GIC with |      |      |      | Maintaining GIC with |      |             |             |
|------------|-----------------------|------|------|------|----------------------|------|-------------|-------------|
|            | GIC1                  | GIC2 | GIC3 | GIC4 | GIC1                 | GIC2 | GIC3        | GIC4        |
| souffleuse | 0.02                  | 0.01 | 0.01 | 0.01 | 0.13                 | 0.07 | <b>0.02</b> | <b>0.02</b> |
| megane     | 2.94                  | 0.71 | 0.72 | 0.71 | 4.26                 | 1.18 | 0.05        | <b>0.04</b> |
| master     | 2.45                  | 1.35 | 1.33 | 1.33 | 9.81                 | 3.57 | 0.07        | <b>0.06</b> |
| small      | 0.14                  | 0.02 | 0.03 | 0.03 | 0.32                 | 0.05 | <b>0.01</b> | <b>0.01</b> |
| medium     | 0.26                  | 0.04 | 0.05 | 0.04 | 0.35                 | 0.04 | <b>0.01</b> | <b>0.01</b> |
| big        | 4.19                  | 1.16 | 1.10 | 1.10 | 12.6                 | 2.60 | <b>0.05</b> | <b>0.05</b> |

One great advantage of GIC is that it guarantees that a conflict can never occur during a configuration session. However, one may wonder whether the risk of failure(s) is really important in user-driven searches that use a weaker consistency such as GAC or a partial form of it (Forward Checking). Table 3 shows the number of conflicts (sum over 100 executions using random orderings) encountered when following a MAC or a nFC2 [3] strategy. The number of conflict situations can be very large with nFC2 (for two instances, we even report the impossibility of finding a solution within 10 minutes with some random orderings). For MAC, the number of failures is rather small but the risk is not null (for example, the risk is equal to 5% for megane).

**Table 3.** Number of conflicts encountered when running nFC2 and MAC (sum over 100 random executions)

|      | souffleuse | megane  | master   | small | medium | big      |
|------|------------|---------|----------|-------|--------|----------|
| nFC2 | 252,605    | 313,910 | time-out | 3,728 | 7,824  | time-out |
| MAC  | 0          | 7       | 5        | 0     | 3      | 3        |

The encouraging results obtained on Renault configuration instances led us to test other problems, in particular to get a better picture of the relative efficiency of the various GIC algorithms. For example, on classical Crossword instances (see Table 4), GIC1 is once again clearly outperformed while the three other algorithms are quite close, where there is still a small benefit of using GIC4.

**Table 4.** CPU time (in seconds) to establish GIC on some Crosswords instances, and to maintain it on average over 100 random greedy executions

|           | Establishing GIC with |      |      |      | Maintaining GIC with |      |      |             |
|-----------|-----------------------|------|------|------|----------------------|------|------|-------------|
|           | GIC1                  | GIC2 | GIC3 | GIC4 | GIC1                 | GIC2 | GIC3 | GIC4        |
| ogd-vg5-5 | 2.25                  | 0.67 | 0.67 | 0.67 | 2.34                 | 0.79 | 0.73 | <b>0.70</b> |
| ogd-vg5-6 | 6.40                  | 2.18 | 2.19 | 2.19 | 7.42                 | 2.82 | 2.58 | <b>2.48</b> |
| ogd-vg5-7 | 25.8                  | 9.91 | 9.87 | 9.84 | 33.4                 | 15.2 | 14.3 | <b>13.8</b> |

**Table 5.** CPU time (in seconds) to establish GIC on Puzzle instances, and to maintain it on average over 100 random greedy executions until a unique solution is found

|                 | Establishing GIC with |      |      |      | Maintaining GIC with |             |      |             |
|-----------------|-----------------------|------|------|------|----------------------|-------------|------|-------------|
|                 | GIC1                  | GIC2 | GIC3 | GIC4 | GIC1                 | GIC2        | GIC3 | GIC4        |
| sudoku-9x9      | 1.58                  | 0.32 | 0.32 | 0.31 | 15.3                 | 2.71        | 2.10 | <b>1.74</b> |
| sudoku-16x16    | 6.04                  | 0.51 | 0.50 | 0.50 | 246                  | 25.5        | 26.5 | <b>18.9</b> |
| magicSquare-4x4 | 0.96                  | 0.26 | 0.28 | 0.28 | 1.63                 | <b>0.69</b> | 0.71 | 0.71        |
| magicSquare-5x5 | 14.7                  | 3.01 | 3.10 | 2.99 | 55.1                 | 15.9        | 15.6 | <b>13.7</b> |

It is worthwhile to note that GIC is a nice property that can be useful when puzzles, where hints are specified, have to be conceived. Typically, one looks for puzzles where only one solution exists. One way of building such puzzles is to add hints in sequence, while maintaining GIC, until all domains become singleton. For example, this is a possible approach for constructing Sudoku and Magic Square grids, with the advantage that the user can choose freely the position of the hints.<sup>2</sup> On the left part of Table 5, we report the time to enforce GIC on empty Sudoku grids of size 9x9 and 16x16, and on empty Magic square of size 4x4 and 5x5, and on the right part, the average time required to maintain GIC until a fixed point is reached, meaning that after several hints have been randomly selected and propagated, we have the guarantee of having a one-solution puzzle. GIC4 is a clear winner, with for example, a 30% speedup over GIC2

<sup>2</sup> However, we are not claiming that maintaining GIC is the unique answer to this problem.

and GIC3 on sudoku-16x16, and more than one order of magnitude over GIC1. Overall, the results we obtain show that MIC, i.e., maintaining GIC, is a practicable solution (at least for some problems) as the average time between each decision of the user is small with GIC4.

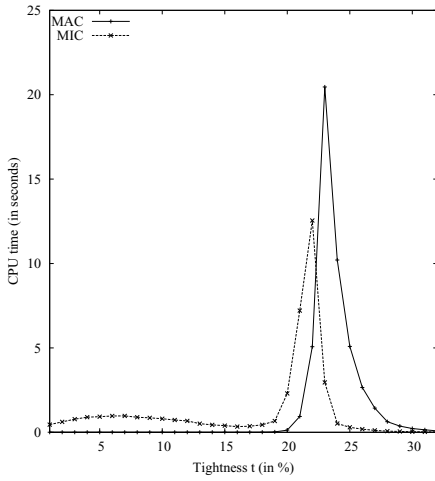
The efficiency of MIC on structured under-constrained instances piqued our curiosity. So we decided to compare MIC (embedding GIC4) and MAC on series of binary random instances generated from Model RB [20]. For the class  $RB(2, 30, 0.8, 3, t)$ , see [19], we obtain instances with 30 variables, 15 values per domain and 306 binary constraints of tightness  $t$ , and for the class  $RB(2, 40, 0.8, 3, t)$ , instances with 40 variables, 19 values per domain and 443 binary constraints of tightness  $t$ . For each value of  $t$  ranging from 0.01 to 0.50 (step of 0.01), a series of 100 instances was generated so as to observe the behaviour of MIC on both under-constrained instances and over-constrained instances; the theoretical threshold is around 0.23. Figure 1 shows the average CPU time of MIC and MAC on series of class  $RB(2, 30, 0.8, 3, t)$ . On the left, Figure 1(a), the ordering of variables and values is random (simulating a free user-driven search). MIC outperforms MAC when the ordering is random and the tightness is greater than or equal to 0.23. That means that the strong inference capability of MIC do pay off for the unsatisfiable instances. On the right, Figure 1(b), the variable ordering heuristic is *dom/wdeg* [4] and the value ordering heuristic is lexico. Obviously, MAC with *dom/wdeg* is clearly faster than MIC. However, if used in a context of interactive resolution, the *dom/wdeg* ranking of the variables drives the user, who is not free anymore in the choices of its variables. It may ask her to assign first variables that are meaningless to her, restricting her future choices on important variables. The outcome will be a solution which is very bad with respect to the preferences of the user. All of this suggests that MIC can be efficient enough to be used in practice, except for a (small) region of satisfiable instances lying at the left of the threshold point. Figure 2 shows similar results with respect to series of class  $RB(2, 40, 0.8, 3, t)$ .

One other practical issue we are interested in is the effectiveness of positive consistency. Hence, we tested to establish positive consistency on existing constraints of the Renault configuration instances, see Table 6. The algorithm we used here is a simple adaptation of GIC1 to tuples (so, certainly, several optimizations are possible). A few hundreds of seconds are necessary to ensure the positive consistency of all existing constraints of the biggest instances.

**Table 6.** CPU time (in seconds) and filtering in term of the number of tuples deleted when establishing positive consistency on Renault configuration instances

|                  | souffleuse | megane  | master | small | medium | big     |
|------------------|------------|---------|--------|-------|--------|---------|
| CPU              | 0.68       | 352     | 368    | 2.6   | 4.2    | 613     |
| # tuples removed | 0          | 138,493 | 90,874 | 240   | 5,425  | 105,020 |

Finally, in our last experiment, for each constraint network, we randomly select a constraint of interest  $c_i$  for which positive consistency must be ensured (as if the modeller were asking for the positive consistency of this constraint), and we randomly select a set  $C$  containing 10% of the set of constraints. We initially consider the CN without



(a) With random ordering

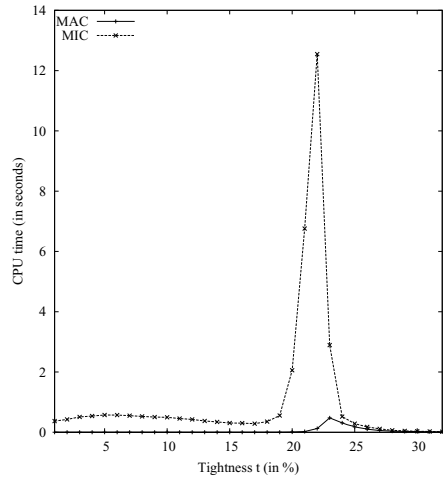
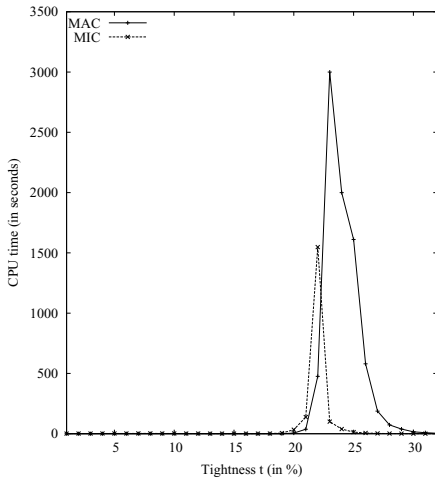
(b) With heuristic *dom/wdeg*

**Fig. 1.** Mean search cost (100 instances) of solving instances in class  $RB(2, 30, 0.8, 3, t)$  with MAC and MIC



(a) With random ordering

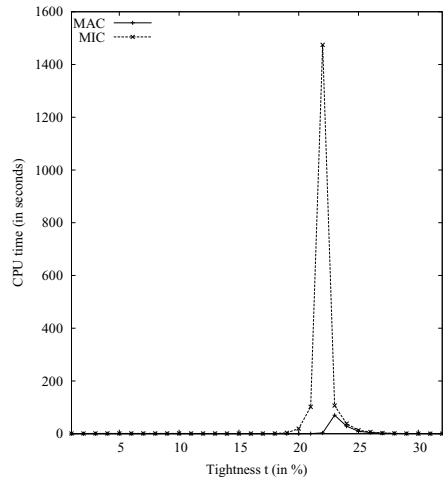
(b) With heuristic *dom/wdeg*

**Fig. 2.** Mean search cost (100 instances) of solving instances in class  $RB(2, 40, 0.8, 3, t)$  with MAC and MIC

the constraints in  $C$ , and we enforce positive consistency on  $c_i$ . Then we simulate a session of product modelling: we post each constraint in  $C$  in turn and maintain positive consistency on  $c_i$ . In our implementation (not detailed here due to lack of space), we use residues, i.e., a solution stored for each tuple of  $c_i$ . The first line of Table 7 shows the average CPU time to maintain positive consistency on the constraint of interest. For the second line, the constraint of interest is not randomly chosen but set to the constraint with the largest table. The obtained results are rather promising (except for the instance megane).

**Table 7.** Dynamic positive consistency filtering on Renault configuration instances (average CPU time over 100 executions)

|         | megane | master | big  |
|---------|--------|--------|------|
| random  | 9.97   | 10.1   | 36.4 |
| largest | 106.6  | 11.4   | 20.6 |

## 7 Conclusion

We have analysed the problems that arise in applications that require the interactive resolution of a constraint problem by a human user. The central notion is global inverse consistency of the network because it ensures that the person who interactively solves the problem is not given the choice to select values that do not lead to solutions. We have shown that deciding, computing, or restoring global inverse consistency, and other related problems are all NP-hard. We have proposed several algorithms for enforcing global inverse consistency and we have shown that the best version is efficient enough to be used in an interactive setting on several configuration and design problems. This is a great advantage compared to existing techniques usually used in configurators. As opposed to techniques maintaining arc consistency, our algorithms give an exact picture of the values remaining feasible. As opposed to compiling offline the problem as a multi-valued decision diagram, our algorithms can deal with constraint networks that change over time (e.g., an extra non-unary constraint posted by a customer who does not want to buy a car with more than 100,000 miles except if it is a Volvo). We have finally extended our contribution to the inverse consistency of tuples, which is useful at the modelling phase of configuration problems.

One direct perspective of this work is to try computing diverse solutions when enforcing GIC. This should permit, on average, to reduce the number of search runs. Some techniques developed in [11] might be useful.

## References

1. Amilhastre, J., Fargier, H., Marquis, P.: Consistency restoration and explanations in dynamic CSPs - application to configuration. *Artificial Intelligence* 135(1-2), 199–234 (2002)
2. Astesana, J.M., Cosserrat, L., Fargier, H.: Constraint-based vehicle configuration: A case study. In: *Proceedings of ICTAI 2010*, pp. 68–75 (2010)

3. Bessiere, C., Meseguer, P., Freuder, E.C., Larrosa, J.: On Forward Checking for non-binary constraint satisfaction. *Artificial Intelligence* 141, 205–224 (2002)
4. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: *Proceedings of ECAI 2004*, pp. 146–150 (2004)
5. Debruyne, R., Bessiere, C.: Domain filtering consistencies. *Journal of Artificial Intelligence Research* 14, 205–230 (2001)
6. Freuder, E.C., Elfe, C.D.: Neighborhood inverse consistency preprocessing. In: *Proceedings of AAAI 1996*, Portland, Oregon, pp. 202–208 (1996)
7. Gelle, E., Weigel, R.: Interactive configuration using constraint satisfaction techniques. In: *Proceedings of PACT 1996*, pp. 37–44 (1996)
8. Gottlob, G.: On minimal constraint networks. *Artificial Intelligence* 191–192, 42–60 (2012)
9. Hadzic, T., Andersen, H.R.: Interactive reconfiguration in power supply restoration. In: van Beek, P. (ed.) *CP 2005*. LNCS, vol. 3709, pp. 767–771. Springer, Heidelberg (2005)
10. Hadzic, T., Hansen, E.R., O’Sullivan, B.: Layer compression in decision diagrams. In: *Proceedings of ICTAI 2008*, pp. 19–26 (2008)
11. Hebrard, E., Hnich, B., O’Sullivan, B., Walsh, T.: Finding diverse and similar solutions in constraint programming. In: *Proceedings of AAAI 2005*, pp. 372–377 (2005)
12. Janssen, P., Jégou, P., Nougier, B., Vilarem, M.C., Castro, B.: SYNTHIA: Assisted design of peptide synthesis plans. *New Journal of Chemistry* 14(12), 969–976 (1990)
13. Lecoutre, C.: STR2: Optimized simple tabular reduction for table constraints. *Constraints* 16(4), 341–371 (2011)
14. Lecoutre, C., Hemery, F.: A study of residual supports in arc consistency. In: *Proceedings of IJCAI 2007*, pp. 125–130 (2007)
15. Martinez, D.: Résolution interactive de problèmes de satisfaction de contraintes. PhD thesis, Supaero, Toulouse, France (1998)
16. Papadimitriou, C.: Private communication (1999)
17. Sabin, D., Freuder, E.C.: Contradicting conventional wisdom in constraint satisfaction. In: Borning, A. (ed.) *PPCP 1994*. LNCS, vol. 874, pp. 10–20. Springer, Heidelberg (1994)
18. Ullmann, J.R.: Partition search for non-binary constraint satisfaction. *Information Science* 177, 3639–3678 (2007)
19. Xu, K., Boussemart, F., Hemery, F., Lecoutre, C.: Random constraint satisfaction: easy generation of hard (satisfiable) instances. *Artificial Intelligence* 171(8–9), 514–534 (2007)
20. Xu, K., Li, W.: Exact phase transitions in random constraint satisfaction problems. *Journal of Artificial Intelligence Research* 12, 93–103 (2000)



# Counting Spanning Trees to Guide Search in Constrained Spanning Tree Problems

Simon Brockbank, Gilles Pesant, and Louis-Martin Rousseau

<sup>1</sup> École polytechnique de Montréal, Montreal, Canada

<sup>2</sup> CIRRELT, Université de Montréal, Montreal, Canada

{simon.brockbank,gilles.pesant,louis-martin.rousseau}@polymtl.ca

**Abstract.** Counting-based branching heuristics such as maxSD were shown to be effective on a variety of constraint satisfaction problems. These heuristics require that we equip each family of constraints with a dedicated algorithm to compute the local solution density of variable assignments, much as what has been done with filtering algorithms to apply local inference. This paper derives an exact polytime algorithm to compute solution densities for a spanning tree constraint, starting from a known result about the number of spanning trees in a graph. We then empirically compare branching heuristics based on that result with other generic heuristics.

## 1 Introduction

Constraint programming is a powerful approach that can be used to solve combinatorial problems. However its success depends heavily on heuristics that can guide the search toward promising areas of the search tree. One can design a heuristic dedicated to the particular problem at hand or rely on out-of-the-box generic heuristics that have shown good performance on a variety of problems. The last decade has witnessed renewed interest in the design of robust generic branching heuristics (e.g. [9,6]). In particular Zanarini and Pesant[15] introduced branching heuristics based on the concept of *solution density*, i.e. the proportion of solutions local to a constraint featuring a given variable-value assignment.

**Definition 1 (solution density).** *Given a constraint  $c(x_1, \dots, x_n)$ , its number of solutions  $\#c(x_1, \dots, x_n)$ , respective finite domains  $D_i$   $1 \leq i \leq n$ , a variable  $x_i$  in the scope of  $c$ , and a value  $d \in D_i$ , we will call*

$$\sigma(x_i, d, c) = \frac{\#c(x_1, \dots, x_{i-1}, d, x_{i+1}, \dots, x_n)}{\#c(x_1, \dots, x_n)}$$

*the solution density of pair  $(x_i, d)$  in  $c$ . It measures how often a certain assignment is part of a solution to  $c$ .*

Specialized algorithms have been designed to compute solution densities for several families of constraints[8]. In this paper we propose an exact polytime algorithm that computes solution densities for a spanning tree constraint.

**Definition 2 (Spanning Tree Constraint (adapted from [4])).** *Given an undirected graph  $G(V, E)$  and set variable  $T \subseteq E$ , constraint  $\text{spanningTree}(G, T)$  restricts  $T$  to be a spanning tree of  $G$ .*

For the sake of conforming to the previous definition of solution density, especially important if we are to allow the combination of solution density information from different constraints, we instead represent  $T$  as an array of boolean variables.

The rest of the paper is organized as follows: Section 2 exposes the related work, Section 3 describes our algorithm to compute solution densities for the spanning tree constraint, Section 4 discusses how our data structures are updated in the course of backtrack search, and Section 5 provides supporting empirical evidence of branching based on solution density.

## 2 Related Work

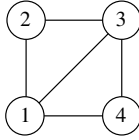
Research in the CP community about imposed tree structures has focused so far on filtering algorithms and not on branching heuristics. Beldiceanu et al.[2] introduced the tree constraint, which addresses the digraph partitioning problem from a constraint programming perspective. In their work a constraint that enforces a set of vertex-disjoint anti-arborescences is proposed. They achieve domain consistency in  $\mathcal{O}(nm)$  time, where  $n$  is the number of vertices and  $m$  is the number of edges in the graph. Their pruning relies on the identification of strong articulation points in the graph and of roots and sinks (to evaluate the minimum and maximum number of trees required to partition the graph).

Dooms and Katriel[3] introduced the MST constraint, requiring the tree variable to represent a minimum spanning tree of the graph on which the constraint is defined. Many variants of the minimum spanning tree problem, such as minimum  $k$ -spanning tree and Steiner tree are known to be NP-hard, even though its basic version can be solved in polynomial time. Those problems can be modeled by combining the MST constraint and other constraints. The authors proposed polytime bound consistent filtering algorithms for several restrictions of this constraint. They proceed by classifying edges in three sets: mandatory, possible, and forbidden. Afterwards Dooms and Katriel[4] proposed a weighted spanning tree constraint, in which both the tree and the weight of the edges are variables, and considered several filtering algorithms. In their work a set variable is used, indicating which edges are tree edges.

The filtering proposed by Dooms and Katriel[4] was then simplified and improved by Régim[10], who proposed an incremental filtering algorithm by maintaining a connected component tree which represents disjoint trees merging operations in Kruskal's algorithm, and by computing lowest common ancestors on that tree. Domain consistency was thus achieved in  $\mathcal{O}(m + n \log n)$  time. Subsequently, Régim et al.[11] improved the time complexity of that filtering and also considered mandatory edges.

### 3 Computing Solution Densities

The *Laplacian matrix*  $L(G)$  of a graph  $G$  is formed by subtracting the adjacency matrix of  $G$  from the diagonal matrix whose  $i^{th}$  entry is equal to the degree of vertex  $i$  in  $G$ . Henceforth for notational convenience we will refer to it simply as  $L$ . For example Figure 1 shows a graph and its Laplacian matrix.



$$L = \begin{pmatrix} 3 & -1 & -1 & -1 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix}$$

**Fig. 1.** The kite graph and its Laplacian matrix

The  $(i, j)$ -*minor* of a square matrix  $M$ , denoted  $M_{ij}$ , is the determinant of the sub-matrix obtained by removing from  $M$  its  $i^{th}$  row and  $j^{th}$  column. The Laplacian matrix has the interesting property that its  $(i, j)$ -minor, for any row  $i$  and column  $j$ , is equal to the number of spanning trees of the corresponding graph.

**Theorem 1 (Kirchhoff's Matrix-Tree Theorem [13]).** *Denote by  $\tau(G)$  the number of spanning trees of graph  $G$  on  $n$  vertices. For any  $1 \leq i, j \leq n$ ,*

$$\tau(G) = L_{ij}.$$

So the number of solutions to a **spanningTree** constraint can be computed as the determinant of a  $(n - 1) \times (n - 1)$  matrix, in  $\mathcal{O}(n^3)$  time.

If we remove the first row and column of the Laplacian matrix at Figure 1, the resulting minor is  $2 \times (3 \times 2 - (-1) \times (-1)) - (-1) \times (-1 \times 2 - (-1) \times 0) = 8$  and one can easily verify that there are eight possible spanning trees for that graph.

But we are interested in computing the solution density of an edge  $(i, j) \in E$ . One way to approach this is by counting the number of spanning trees not using that edge,  $\tau(G \setminus \{(i, j)\})$ , and then dividing that by the total number of spanning trees, yielding the solution density of the corresponding variable being assigned value 0 (i.e.  $(i, j) \notin T$ ):

$$\sigma((i, j), 0, \text{spanningTree}(G, T)) = \frac{\tau(G \setminus \{(i, j)\})}{\tau(G)}.$$

Let  $L' = L(G \setminus \{(i, j)\})$ . How different is  $L'$  from  $L$ ? It will be identical except for entries  $\ell_{ii}$ ,  $\ell_{jj}$ ,  $\ell_{ij}$ , and  $\ell_{ji}$ . Since we can choose any row and column of  $L$  to compute our minor, consider removing row and column  $i$ . Then the only difference is  $\ell'_{jj} = \ell_{jj} - 1$ . The *Sherman-Morrison formula* [12] tells us that if

$M'$  is obtained from matrix  $M$  by replacing its  $j^{th}$  column,  $(M)_j$ , by column vector  $u$  then

$$\det(M') = (1 + e_j^\top M^{-1}(u - (M)_j))\det(M).$$

In our case  $(u - (M)_j) = -e_j$  so the right-hand side of the previous equation simplifies to  $(1 - e_j^\top M^{-1}e_j)\det(M) = (1 - m_{jj}^{-1})\det(M)$ . So finally we have

$$\sigma((i, j), 0, \text{spanningTree}(G, T)) = \frac{L'_{ii}}{L_{ii}} = \frac{(1 - m_{jj}^{-1})L_{ii}}{L_{ii}} = 1 - m_{jj}^{-1},$$

and of course

$$\sigma((i, j), 1, \text{spanningTree}(G, T)) = m_{jj}^{-1}.$$

Computing solution densities turns out to be quite simple: for each edge  $(i, j)$  incident with vertex  $i$  such that  $j < i$  (respectively  $j > i$ ), the corresponding value is the  $j^{th}$  (respectively  $(j - 1)^{th}$ ) entry on the diagonal of the inverse of  $M$ , the sub-matrix of Laplacian matrix  $L$  obtained by removing its  $i^{th}$  row and column. Repeating this from every vertex of a vertex cover of  $G$  provides solution densities for every edge. If  $\gamma$  is the size of the vertex cover used then the whole procedure takes  $\mathcal{O}(\gamma n^3)$  time.

*Example 1.* Let  $M$  be the sub-matrix of  $L$  obtained by removing its first row and column as before. Then

$$M^{-1} = \begin{pmatrix} 5/8 & 2/8 & 1/8 \\ 2/8 & 4/8 & 2/8 \\ 1/8 & 2/8 & 5/8 \end{pmatrix}$$

and the solution density of edges  $(1, 2)$ ,  $(1, 3)$ , and  $(1, 4)$  being used in  $T$  is respectively  $\frac{5}{8}$ ,  $\frac{4}{8}$ , and  $\frac{5}{8}$ .

## 4 Integration into Backtrack Search

In this section we describe some of the implementation details and issues. As branching decisions are made and domain filtering is applied, some edges of  $G$  will be required in  $T$  and others, forbidden. These changes must be reflected in our data structures. Our data structures are reversible so that they are restored upon backtracking.

We use a heuristic greedy algorithm to compute our initial vertex cover — it may be worthwhile spending the time to compute a minimum vertex cover but that cover will need to be revised as vertices are merged following edge contractions. A simple way to update a vertex cover containing vertex  $j$  when edge  $(i, j)$  is contracted is to replace it with vertex  $i$ .

#### 4.1 Updating the Laplacian Matrix

If edge  $(i, j)$  is forbidden it is simply removed from the graph. To reflect that in the Laplacian matrix we simply add one to entries  $\ell_{ij}$  and  $\ell_{ji}$ . The degree of each endpoint is also updated by subtracting one to  $\ell_{ii}$  and  $\ell_{jj}$ .

If edge  $(i, j)$  is required we contract it in the graph, so that  $(i, j)$  is implicitly part of the spanning tree. To update the Laplacian matrix we start by adding to vertex  $i$  all the edges  $(j, k)$ :  $\ell_{ik} \leftarrow \ell_{ik} + \ell_{jk}$ . This may create multiple edges. The degree of vertex  $i$ ,  $\ell_{ii}$ , is also updated accordingly. Then, since vertex  $j$  is now merged with  $i$ , we remove all the edges connected to it, by setting to zero row and column  $j$  of the Laplacian matrix. Finally we set  $\ell_{jj}$  to 1 so that minors will be computed correctly when they include row and column  $j$ .

*Example 2.* Recall Figure 1 and suppose edge  $(1, 2)$  is now required for the spanning tree: we contract it and merge vertex 2 with 1. The new Laplacian matrix will be (note the double edge  $(1, 3)$ ):

$$L = \begin{pmatrix} 3 & 0 & -2 & -1 \\ 0 & 1 & 0 & 0 \\ -2 & 0 & 3 & -1 \\ -1 & 0 & -1 & 2 \end{pmatrix}$$

#### 4.2 Updating Solution Densities

The solution densities will change and we would like to avoid recomputing them from scratch. Given the inverse of matrix  $M$  can we incrementally compute the inverse of a slightly different matrix  $M'$ ? The *Sherman-Morrison formula* further reveals that if  $M'$  is obtained from  $M$  by replacing its  $i^{th}$  column,  $(M)_i$ , by column vector  $u$  as before then

$$M'^{-1} = M^{-1} - \frac{(M^{-1}(u - (M)_i))(e_i^\top M^{-1})}{1 + e_i^\top M^{-1}(u - (M)_i)}.$$

This can be computed in  $\mathcal{O}(n^2)$  time.

In some cases we can lower that time complexity considerably. Consider forbidden edge  $(i, j)$ . For any edge  $(i, k)$  whose solution density was obtained through the inverse of a sub-matrix removing row and column  $i$  from  $L$ , removing edge  $(i, j)$  only changes one entry in that sub-matrix, as we saw before, and the previous formula simplifies to

$$M'^{-1} = M^{-1} - \frac{(M^{-1} \cdot (-e_j)) \cdot (e_j^\top \cdot M^{-1})}{1 - m_{jj}^{-1}} = M^{-1} + \frac{1}{1 - m_{jj}^{-1}} \cdot Q$$

where  $Q = (q_{hk})$  is an  $(n-1) \times (n-1)$  matrix with  $q_{hk} = m_{hj}^{-1} \cdot m_{jk}^{-1}$ . Because we only need the  $k^{th}$  entry on the diagonal,  $m_{kk}^{-1} + (m_{kj}^{-1})^2 / (1 - m_{jj}^{-1})$ , the update for that edge takes constant time. What preceded equally applies for any edge  $(j, k)$  with a sub-matrix removing row and column  $j$  from  $L$ .

*Example 3.* Recall that for the graph at Figure 1 the solution density of edge  $(1, 4)$  is  $\frac{5}{8}$ . Suppose edge  $(1, 2)$  is now forbidden in the spanning tree. The updated solution density will be  $\frac{5}{8} + (m_{42}^{-1})^2 / (1 - m_{22}^{-1}) = \frac{5}{8} + (\frac{1}{8})^2 / (1 - \frac{5}{8}) = \frac{2}{3}$ .

## 5 Experiments

To demonstrate the effectiveness of using solution density information from a **spanningTree** constraint to guide a branching heuristic on some constrained spanning tree problems, we consider finding degree-constrained spanning trees of a graph. Note that the special case of a maximum degree of 2 corresponds to the Hamiltonian path problem. We created some graphs using a generator designed to produce hard Hamiltonian path instances for backtracking algorithms [14]. We used the IBM ILOG CP v1.6 solver for our implementation and performed our experiments on a AMD Opteron 2.2GHz with 1GB of memory. Our current implementation does not include the incremental algorithm described in Section 4.2 so the times reported are with matrix inversions computed from scratch at every search tree node. We report comparative results between **maxSD**, impact-based search (**IBS**), and random variable and value selection (**random**). Heuristic **maxSD** considers solution density information from each constraint and branches on the variable-value pair corresponding to the highest solution density observed. For **IBS** impacts are initialized by probing at the root node. At a search tree node the five best variables according to the approximated impact are identified. For that subset, we compute node impacts and branch on the best variable (highest impact) and value (lowest impact). This is consistent with what is suggested in the IBM ILOG solver documentation. For **random** we report the average of ten runs.

We used simple filtering rules for our constraint — our objective is not to solve that problem in the best way possible but rather to evaluate a counting-based branching heuristic. The first one forces each vertex to have degree at least one in the tree by lower bounding the sum of the variables corresponding to the edges incident to it. The second one fixes the number of edges that can be part of the spanning tree: as a spanning tree is formed by  $n - 1$  edges, the sum of all variables must equal that value. Finally, since a tree is acyclic, we maintain the connected component in which each vertex lies, removing any extraneous intra connected component edges. In addition to the **spanningTree** constraint, we add to our model for each vertex  $i$  an upper bound on the sum of the variables corresponding to the edges incident to  $i$ .

We first generated random graphs of 15, 20, 25, 30, and 35 vertices (10 instances each). The generator ensures the existence of a Hamiltonian path. Turning first to a degree-2 bound, Table 1 left indicates that using **maxSD** effectively guides the search to a solution in several orders of magnitude fewer backtracks than the other two branching heuristics. Even though **maxSD** appears slower on small graphs, as displayed in Table 1 right, as the graphs become larger, this approach becomes faster than **IBS** and **random**.

**Table 1.** Number of backtracks (left) and time in seconds (right) before finding a spanning tree of maximum degree 2. Each line represents an average over 10 instances.

| n  | maxSD | IBS     | random   | n  | maxSD | IBS   | random |
|----|-------|---------|----------|----|-------|-------|--------|
| 15 | 0.2   | 229.8   | 49.0     | 15 | 0.029 | 0.001 | 0.001  |
| 20 | 1.5   | 533.0   | 976.6    | 20 | 0.080 | 0.012 | 0.020  |
| 25 | 2.1   | 1772.3  | 5919.6   | 25 | 0.187 | 0.085 | 0.173  |
| 30 | 71.7  | 12517.1 | 91454.4  | 30 | 0.815 | 0.897 | 1.873  |
| 35 | 112.2 | 18405.4 | 139861.3 | 35 | 1.769 | 4.742 | 14.646 |

**Table 2.** Number of backtracks (left) and time in seconds (right) before finding a spanning tree of maximum degree 3. Each line represents an average over 10 instances.

| n  | maxSD | IBS   | random  | n  | maxSD | IBS   | random |
|----|-------|-------|---------|----|-------|-------|--------|
| 15 | 0.0   | 225.6 | 1.3     | 15 | 0.039 | 0.002 | 0.001  |
| 20 | 0.0   | 315.2 | 53.2    | 20 | 0.100 | 0.013 | 0.001  |
| 25 | 0.0   | 446.7 | 882.0   | 25 | 0.222 | 0.021 | 0.311  |
| 30 | 0.0   | 495.1 | 18589.8 | 30 | 0.441 | 0.039 | 0.093  |
| 35 | 0.0   | 566.8 | 20001.4 | 35 | 0.852 | 0.063 | 2.333  |

We then turn to a degree-3 bound (see Table 2). It clearly demonstrates that using solution densities to find spanning trees in random graphs is a very effective approach. A maximum degree of 3 is much less restrictive than a maximum degree of 2 and more spanning trees in that graph will have that property. Therefore the first few spanning trees found satisfy all constraints. For all graphs, the solution density branching heuristic finds a spanning tree without any backtrack, unlike the other approaches. Despite not having to backtrack, maxSD remains slower than IBS on these instances since the latter only requires a few hundred backtracks.

**Table 3.** Number of backtracks (left) and time in seconds (right) before finding a Hamiltonian path in crossroad graphs. Each line represents an average over 10 instances.

| n | maxSD | IBS      | random   | n | maxSD | IBS     | random |
|---|-------|----------|----------|---|-------|---------|--------|
| 3 | 0.2   | 7721.9   | 8530.5   | 3 | 0.085 | 0.255   | 0.062  |
| 4 | 0.1   | 262011.7 | 191195.8 | 4 | 0.280 | 26.379  | 3.674  |
| 5 | 0.4   | 162353.0 | -        | 5 | 0.676 | 586.679 | -      |

We also generated *crossroad graphs* using the same graph generator. These graphs are made up of small subgraphs only connected to each other via "bridge" edges. We generated 10 instances each of crossroad graphs containing 3, 4, and 5 subgraphs (with up to 35 vertices in total) and then tried to find a Hamiltonian path (spanning tree of degree 2). Results are shown in Table 3.

Using `maxSD` on these hard graphs is very effective, always finding a solution in much fewer backtracks than the other approaches. For the instances made up of 5 subgraphs, `random` could not solve a single instance within 2 hours of computing time. Here `maxSD` is also orders of magnitude faster than the other two branching heuristics.

## 6 Conclusion

We presented a new algorithm that computes exact solution densities for the spanning tree constraint in  $\mathcal{O}(\gamma n^3)$  time, where  $\gamma$  is the size of a vertex cover for the graph, and updates solution densities in  $\mathcal{O}(\gamma n^2)$  time, even in some cases achieving constant time updates per edge. Building the Laplacian matrix of a graph and inverting selected sub-matrices, the proportion of spanning trees including a certain edge of the graph can be calculated. By relying on that information, search can be oriented towards areas of the search space with high solution density with respect to the spanning tree structure and we gave some empirical evidence that this helps solve constrained spanning tree problems.

As future work we would like to try other types of constrained spanning tree problems. There are several application areas that involve finding spanning trees, such as network design, telecommunication, or transportation. Examples of these problems are the degree-constrained problem [7], the hop-constrained problem [5] or the diameter-constrained minimum spanning tree [1]. We also plan to investigate the compatibility of our solution density algorithm with more powerful filtering algorithms and variants of the constraint as proposed in the literature. For example the Matrix-Tree Theorem to count the number of spanning trees has already been generalized to directed graphs.

## References

1. Abdalla, A., Deo, N.: Random-tree diameter and the diameter-constrained mst. *Int. J. Comput. Math.* 79(6), 651–663 (2002)
2. Beldiceanu, N., Flener, P., Lorca, X.: The tree constraint. In: Barták, R., Milano, M. (eds.) *CPAIOR 2005*. LNCS, vol. 3524, pp. 64–78. Springer, Heidelberg (2005)
3. Dooms, G., Katriel, I.: The minimum spanning tree constraint. In: Benhamou, F. (ed.) *CP 2006*. LNCS, vol. 4204, pp. 152–166. Springer, Heidelberg (2006)
4. Dooms, G., Katriel, I.: The “not-too-heavy spanning tree” constraint. In: Van Hentenryck, P., Wolsey, L.A. (eds.) *CPAIOR 2007*. LNCS, vol. 4510, pp. 59–70. Springer, Heidelberg (2007)
5. Gouveia, L., Simonetti, L., Uchoa, E.: Modeling hop-constrained and diameter-constrained minimum spanning tree problems as steiner tree problems over layered graphs. *Math. Program.* 128(1-2), 123–148 (2011)
6. 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)
7. Narula, S.C., Ho, C.A.: Degree-constrained minimum spanning tree. *Computers & OR* 7(4), 239–249 (1980)



8. 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)
9. Refalo, P.: Impact-Based Search Strategies for Constraint Programming. In: Wallace, M. (ed.) *CP 2004. LNCS*, vol. 3258, pp. 557–571. Springer, Heidelberg (2004)
10. Régim, J.-C.: Simpler and incremental consistency checking and arc consistency filtering algorithms for the weighted spanning tree constraint. In: Trick, M.A. (ed.) *CPAIOR 2008. LNCS*, vol. 5015, pp. 233–247. Springer, Heidelberg (2008)
11. Régim, J.-C., Rousseau, L.-M., Rueher, M., van Hoeve, W.-J.: The weighted spanning tree constraint revisited. In: Lodi, A., Milano, M., Toth, P. (eds.) *CPAIOR 2010. LNCS*, vol. 6140, pp. 287–291. Springer, Heidelberg (2010)
12. Sherman, J., Morrison, W.J.: Adjustment of an inverse matrix corresponding to a change in one element of a given matrix. *Annals of Mathematical Statistics* 21, 124–127 (1950)
13. Tutte, W.T.: *Graph Theory. Encyclopedia of Mathematics and Its Applications*, vol. 21, 333 pages. Cambridge University Press (2001)
14. Vandegriend, B.: Finding hamiltonian cycles: Algorithms, graphs and performance. Master's thesis, University of Alberta (1998), <http://webdocs.cs.ualberta.ca/~joe/Theses/HCArchive/main.html>
15. Zanarini, A., Pesant, G.: Solution counting algorithms for constraint-centered search heuristics. In: Bessière, C. (ed.) *CP 2007. LNCS*, vol. 4741, pp. 743–757. Springer, Heidelberg (2007)

# On the Reduction of the CSP Dichotomy Conjecture to Digraphs<sup>\*</sup>

Jakub Bulín<sup>1</sup>, Dejan Delić<sup>2</sup>, Marcel Jackson<sup>3</sup>, and Todd Niven<sup>3</sup>

<sup>1</sup> Faculty of Mathematics and Physics,  
Charles University in Prague, Czech Republic

<sup>2</sup> Department of Mathematics, Ryerson University, Canada

<sup>3</sup> Department of Mathematics, La Trobe University, Australia  
m.g.jackson@latrobe.edu.au, ddelic@ryerson.ca,  
{jakub.bulin,toddniven}@gmail.com

**Abstract.** It is well known that the constraint satisfaction problem over general relational structures can be reduced in polynomial time to digraphs. We present a simple variant of such a reduction and use it to show that the algebraic dichotomy conjecture is equivalent to its restriction to digraphs and that the polynomial reduction can be made in logspace. We also show that our reduction preserves the bounded width property, i.e., solvability by local consistency methods. We discuss further algorithmic properties that are preserved and related open problems.

## 1 Introduction

A fundamental problem in constraint programming is to understand the computational complexity of constraint satisfaction problems (CSPs). While it is well known that the class of all constraint problems is NP-complete, there are many subclasses of problems for which there are efficient solving methods. One way to restrict the instances is to only allow a fixed set of constraint relations, often referred to as a *constraint language* [5] or *fixed template*. Classifying the computational complexity of fixed template CSPs has been a major focus in the theoretical study of constraint satisfaction. In particular it is of interest to know which templates produce polynomial time solvable problems to help provide more efficient solution techniques.

The study of fixed template CSPs dates back to the 1970's with the work of Montanari [18] and Schaefer [19]. A standout result from this era is that of Schaefer who showed that the CSPs arising from constraint languages over 2-element domains satisfy a *dichotomy*. The decision problem for fixed template

---

<sup>\*</sup> The first author was supported by the grant projects GAČR 201/09/H012, GA UK 67410, SVV-2013-267317; the second author gratefully acknowledges support by the Natural Sciences and Engineering Research Council of Canada in the form of a Discovery Grant; the third and fourth were supported by ARC Discovery Project DP1094578; the first and fourth authors were also supported by the Fields Institute.

CSPs over finite domains belong to the class NP, and Schaefer showed that in the 2-element domain case, a constraint language is either solvable in polynomial time or NP-complete. Dichotomies cannot be expected for decision problems in general, since (under the assumption that  $P \neq NP$ ) there are many problems in NP that are neither solvable in polynomial time, nor NP-complete [15]. Another important dichotomy was proved by Hell and Nešetřil [9]. They showed that if a fixed template is a finite simple graph (the vertices make up the domain and the edges make up the only allowable constraints), then the corresponding CSP is either polynomial time solvable or NP-complete. The decision problem for a graph constraint language can be rephrased as graph homomorphism problem (a graph homomorphism is a function from the vertices of one graph to another such that the edges are preserved). Specifically, given a fixed graph  $\mathcal{H}$  (the constraint language), an instance is a graph  $\mathcal{G}$  together with the question “Is there a graph homomorphism from  $\mathcal{G}$  to  $\mathcal{H}$ ?”. In this sense, 3-colorability corresponds to  $\mathcal{H}$  being the complete graph on 3 vertices. The notion of graph homomorphism problems naturally extends to directed graph (digraph) homomorphism problems and to relational structure homomorphism problems.

These early examples of dichotomies, by Schaefer, Hell and Nešetřil, form the basis of a larger project of classifying the complexity of fixed template CSPs. Of particular importance in this project is to prove the so-called *CSP Dichotomy Conjecture* of Feder and Vardi [8] dating back to 1993. It states that the CSPs relating to a fixed constraint language over a finite domain are either polynomial time solvable or NP-complete. To date this conjecture remains unanswered, but it has driven major advances in the study of CSPs.

One such advance is the algebraic connection revealed by Jeavons, Cohen and Gyssens [13] and later refined by Bulatov, Jeavons and Krokhin [5]. This connection associates with each finite domain constraint language  $\mathbb{A}$  a finite algebraic structure  $\mathbf{A}$ . The properties of this algebraic structure are deeply linked with the computational complexity of the constraint language. In particular, for a fixed constraint language  $\mathbb{A}$ , if there does not exist a particular kind of operation, known as a Taylor polymorphism, then the class of problems determined by  $\mathbb{A}$  is NP-complete. Bulatov, Jeavons and Krokhin [5] go on to conjecture that all other constraint languages over finite domains determine polynomial time CSPs (a stronger form of the CSP Dichotomy Conjecture, since it describes where the split between polynomial time and NP-completeness lies). This conjecture is often referred to as the *Algebraic CSP Dichotomy Conjecture*. Many important results have been built upon this algebraic connection. Bulatov [6] extended Schaefer’s [19] result on 2-element domains to prove the CSP Dichotomy Conjecture for 3-element domains. Barto, Kozik and Niven [3] extended Hell and Nešetřil’s result [9] on simple graphs to constraint languages consisting of a finite digraph with no sources and no sinks. Barto and Kozik [2] gave a complete algebraic description of the constraint languages over finite domains that are solvable by local consistency methods (these problems are said to be of *bounded width*) and as a consequence it is decidable to determine whether a constraint language can be solved by such methods.

In their seminal paper, Feder and Vardi [8] not only conjectured a dichotomy, they also reduced the problem of proving the dichotomy conjecture to the particular case of digraph homomorphism problems, and even to digraph homomorphism problems where the digraph is balanced (here balanced means that its vertices can be partitioned into levels). Specifically, for every template  $\mathbb{A}$  (a finite relational structure of finite type) there is a balanced digraph (digraphs are particular kinds of relational structures)  $\mathcal{D}(\mathbb{A})$  such that the CSP over  $\mathbb{A}$  is polynomial time equivalent to that over  $\mathcal{D}(\mathbb{A})$ .

## 2 The Main Results

In general, fixed template CSPs can be modelled as relational structure homomorphism problems [8]. For detailed formal definitions of relational structures, homomorphisms and polymorphisms, see Section 3.

Let  $\mathbb{A}$  be a finite structure with signature  $\mathcal{R}$  (the fixed template), then the *constraint satisfaction problem for  $\mathbb{A}$*  is the following decision problem.

### Constraint Satisfaction Problem for $\mathbb{A}$ .

|                                                                       |
|-----------------------------------------------------------------------|
| <b>CSP(<math>\mathbb{A}</math>)</b>                                   |
| INSTANCE: A finite $\mathcal{R}$ -structure $\mathbb{X}$ .            |
| QUESTION: Is there a homomorphism from $\mathbb{X}$ to $\mathbb{A}$ ? |

The dichotomy conjecture [8] can be stated as follows:

**CSP Dichotomy Conjecture.** *Let  $\mathbb{A}$  be a finite relational structure. Then CSP( $\mathbb{A}$ ) is solvable in polynomial time or NP-complete.*

The dichotomy conjecture is equivalent to its restriction to digraphs [8], and thus can be restated as follows:

**CSP Dichotomy Conjecture.** *Let  $\mathbb{H}$  be a finite digraph. Then CSP( $\mathbb{H}$ ) is solvable in polynomial time or NP-complete.*

Every finite relational structure  $\mathbb{A}$  has a unique *core* substructure  $\mathbb{A}'$  (see Section 3.3 for the precise definition) such that CSP( $\mathbb{A}$ ) and CSP( $\mathbb{A}'$ ) are identical problems, i.e., the “yes” and “no” instances are precisely the same. The algebraic dichotomy conjecture [5] is the following:

**Algebraic CSP Dichotomy Conjecture.** *Let  $\mathbb{A}$  be a finite relational structure that is a core. If  $\mathbb{A}$  has a Taylor polymorphism then CSP( $\mathbb{A}$ ) is solvable in polynomial time, otherwise CSP( $\mathbb{A}$ ) is NP-complete.*

Indeed, perhaps the above conjecture should be called the *algebraic tractability conjecture* since it is known that if a core  $\mathbb{A}$  does not possess a Taylor polymorphism, then CSP( $\mathbb{A}$ ) is NP-complete [5].

Feder and Vardi [8] proved that every fixed template CSP is polynomial time equivalent to a digraph CSP. This article will provide the following theorem, which replaces “polynomial time” with “logspace” and reduces the algebraic dichotomy conjecture to digraphs.

**Theorem 1.** *Let  $\mathbb{A}$  be a finite relational structure. There is a finite digraph  $\mathcal{D}(\mathbb{A})$  such that*

- (i)  *$\text{CSP}(\mathbb{A})$  and  $\text{CSP}(\mathcal{D}(\mathbb{A}))$  are logspace equivalent,*
- (ii)  *$\mathbb{A}$  has a Taylor polymorphism if and only if  $\mathcal{D}(\mathbb{A})$  has a Taylor polymorphism, and*
- (iii)  *$\mathbb{A}$  is a core if and only if  $\mathcal{D}(\mathbb{A})$  is a core.*

*Furthermore, if  $\mathbb{A}$  is a core, then  $\text{CSP}(\mathbb{A})$  has bounded width if and only if  $\text{CSP}(\mathcal{D}(\mathbb{A}))$  has bounded width.*

*Proof.* To prove (i), one reduction follows from Lemma 3 and Lemma 1. The other reduction is Lemma 4.

To prove (ii) we employ Theorem 2; it suffices to show that  $\mathbb{A}$  has a WNU polymorphism if and only if  $\mathcal{D}(\mathbb{A})$  has a WNU polymorphism. The forward implication (which is the crucial part of our proof) is proved in Lemma 8 and the converse follows from Lemma 3 and Lemma 2. Item (iii) is Corollary 1.

The preservation of bounded width follows from Corollary 1, Lemma 8 and Theorem 3.  $\square$

See Remark 1 in Section 4 for the size of  $\mathcal{D}(\mathbb{A})$ . The “Taylor polymorphism” in Theorem 1 (ii) can be replaced by many other polymorphism properties, but space constraints do not allow us to elaborate here.

As a direct consequence of Theorem 1 (ii) and (iii) above, we can restate the algebraic dichotomy conjecture:

**Algebraic CSP Dichotomy Conjecture.** *Let  $\mathbb{H}$  be a finite digraph that is a core. If  $\mathbb{H}$  has a Taylor polymorphism then  $\text{CSP}(\mathbb{H})$  is solvable in polynomial time, otherwise  $\text{CSP}(\mathbb{H})$  is NP-complete.*

### 3 Background and Definitions

We approach fixed template constraint satisfaction problems from the “homomorphism problem” point of view. For background on the homomorphism approach to CSPs, see [8], and for background on the algebraic approach to CSP, see [5].

A *relational signature*  $\mathcal{R}$  is a (in our case finite) set of *relation symbols*  $R_i$ , each with an associated arity  $k_i$ . A (finite) *relational structure*  $\mathbb{A}$  *over relational signature*  $\mathcal{R}$  (called an  $\mathcal{R}$ -*structure*) is a finite set  $A$  (the *domain*) together with a relation  $R_i \subseteq A^{k_i}$ , for each relation symbol  $R_i$  of arity  $k_i$  in  $\mathcal{R}$ . A *CSP template* is a fixed finite  $\mathcal{R}$ -structure, for some signature  $\mathcal{R}$ .

For simplicity we do not distinguish the relation with its associated relation symbol, however to avoid ambiguity, sometimes we write  $R^{\mathbb{A}}$  to indicate that  $R$  belongs to  $\mathbb{A}$ . We will often refer to the domain of relational structure  $\mathbb{A}$  simply by  $A$ . When referring to a fixed relational structure, we may simply specify it as  $\mathbb{A} = (A; R_1, R_2, \dots, R_k)$ . For technical reasons we require that all the relations of a relational structure are nonempty.

### 3.1 Notation

For a positive integer  $n$  we denote the set  $\{1, 2, \dots, n\}$  by  $[n]$ . We write tuples using boldface notation, e.g.  $\mathbf{a} = (a_1, a_2, \dots, a_k) \in A^k$  and when ranging over tuples we use superscript notation, e.g.  $(\mathbf{r}^1, \mathbf{r}^2, \dots, \mathbf{r}^l) \in R^l \subseteq (A^k)^l$ , where  $\mathbf{r}^i = (r_1^i, r_2^i, \dots, r_k^i)$ , for  $i = 1, \dots, l$ .

Let  $R_i \subseteq A^{k_i}$  be relations of arity  $k_i$ , for  $i = 1, \dots, n$ . Let  $k = \sum_{i=1}^n k_i$  and  $l_i = \sum_{j < i} k_j$ . We write  $R_1 \times \dots \times R_n$  to mean the  $k$ -ary relation

$$\{(a_1, \dots, a_k) \in A^k \mid (a_{l_i+1}, \dots, a_{l_i+k_i}) \in R_i \text{ for } i = 1, \dots, n\}.$$

An  $n$ -ary operation on a set  $A$  is simply a mapping  $f : A^n \rightarrow A$ ; the number  $n$  is the *arity* of  $f$ . Let  $f$  be an  $n$ -ary operation on  $A$  and let  $k > 0$ . We define  $f^{(k)}$  to be the  $n$ -ary operation obtained by applying  $f$  coordinatewise on  $A^k$ . That is, we define the  $n$ -ary operation  $f^{(k)}$  on  $A^k$  by

$$f^{(k)}(\mathbf{a}^1, \dots, \mathbf{a}^n) = (f(a_1^1, \dots, a_1^n), \dots, f(a_k^1, \dots, a_k^n)),$$

for  $\mathbf{a}^1, \dots, \mathbf{a}^n \in A^k$ .

We will be particularly interested in so-called idempotent operations. An  $n$ -ary operation  $f$  is said to be *idempotent* if it satisfies the equation

$$f(x, x, \dots, x) = x.$$

### 3.2 Homomorphisms, Cores and Polymorphisms

We begin with the notion of a relational structure homomorphism.

**Definition 1.** Let  $\mathbb{A}$  and  $\mathbb{B}$  be relational structures in the same signature  $\mathcal{R}$ . A homomorphism from  $\mathbb{A}$  to  $\mathbb{B}$  is a mapping  $\varphi$  from  $A$  to  $B$  such that for each  $n$ -ary relation symbol  $R$  in  $\mathcal{R}$  and each  $n$ -tuple  $\mathbf{a} \in A^n$ , if  $\mathbf{a} \in R^{\mathbb{A}}$ , then  $\varphi(\mathbf{a}) \in R^{\mathbb{B}}$ , where  $\varphi$  is applied to  $\mathbf{a}$  coordinatewise.

We write  $\varphi : \mathbb{A} \rightarrow \mathbb{B}$  to mean that  $\varphi$  is a homomorphism from  $\mathbb{A}$  to  $\mathbb{B}$ , and  $\mathbb{A} \rightarrow \mathbb{B}$  to mean that there exists a homomorphism from  $\mathbb{A}$  to  $\mathbb{B}$ .

An *isomorphism* is a bijective homomorphism  $\varphi$  such that  $\varphi^{-1}$  is a homomorphism. A homomorphism  $\mathbb{A} \rightarrow \mathbb{A}$  is called an *endomorphism*. An isomorphism from  $\mathbb{A}$  to  $\mathbb{A}$  is an *automorphism*. It is an easy fact that if  $\mathbb{A}$  is finite, then every surjective endomorphism is an automorphism.

A finite relational structure  $\mathbb{A}'$  is a *core* if every endomorphism  $\mathbb{A}' \rightarrow \mathbb{A}'$  is surjective (and therefore an automorphism). For every  $\mathbb{A}$  there exists a relational structure  $\mathbb{A}'$  such that  $\mathbb{A} \rightarrow \mathbb{A}'$  and  $\mathbb{A}' \rightarrow \mathbb{A}$  and  $\mathbb{A}'$  is minimal with respect to these properties. The structure  $\mathbb{A}'$  is called the *core of  $\mathbb{A}$* . The core of  $\mathbb{A}$  is unique (up to isomorphism) and  $\text{CSP}(\mathbb{A})$  and  $\text{CSP}(\mathbb{A}')$  are the same decision problems. Equivalently, the core of  $\mathbb{A}$  can be defined as a minimal induced substructure that  $\mathbb{A}$  retracts onto. (See [10] for details on cores for graphs, cores for relational structures are a natural generalisation.)

The notion of *polymorphism* is central in the so called algebraic approach to CSP. Polymorphisms are a natural generalization of endomorphisms to higher arity operations.

**Definition 2.** Given an  $\mathcal{R}$ -structure  $\mathbb{A}$ , an  $n$ -ary polymorphism of  $\mathbb{A}$  is an  $n$ -ary operation  $f$  on  $A$  such that  $f$  preserves the relations of  $\mathbb{A}$ . That is, if  $\mathbf{a}^1, \dots, \mathbf{a}^n \in R$ , for some  $k$ -ary relation  $R$  in  $\mathcal{R}$ , then  $f^{(k)}(\mathbf{a}^1, \dots, \mathbf{a}^n) \in R$ .

Thus, an endomorphism is a 1-ary polymorphism.

In this paper we will be interested in the following kind of polymorphisms.

**Definition 3.** A weak near-unanimity (WNU) polymorphism is an  $n$ -ary idempotent polymorphism  $\omega$ , for some  $n \geq 3$ , that satisfies the following equations (for all  $x, y$ ):

$$\omega(x, \dots, x, y) = \omega(x, \dots, x, y, x) = \dots = \omega(y, x, \dots, x).$$

We call the above WNU equations.

Note that since we assume that a WNU polymorphism  $\omega$  is idempotent it also satisfies the equation

$$\omega(x, x, \dots, x) = x.$$

Of particular interest, with respect to the algebraic dichotomy conjecture, are Taylor polymorphisms. We will not need to explicitly define Taylor polymorphisms (and only need consider WNU polymorphisms) by the following theorem.

**Theorem 2.** [17] *A finite relational structure  $\mathbb{A}$  has a Taylor polymorphism if and only if  $\mathbb{A}$  has a WNU polymorphism.*

Weak near-unanimity polymorphisms can be also used to characterise CSPs of bounded width (see [2] for a detailed explanation of the bounded width algorithm).

**Theorem 3.** [2,17] *Let  $\mathbb{A}$  be a finite relational structure that is a core. Then  $\text{CSP}(\mathbb{A})$  is of bounded width if and only if  $\mathbb{A}$  has WNU polymorphisms of all but finitely many arities.*

### 3.3 Primitive Positive Definability

A first order formula is called *primitive positive* if it is an existential conjunction of atomic formulæ. Since we only refer to relational signatures, a primitive positive formula is simply an existential conjunct of formulæ of the form  $x = y$  or  $(x_1, x_2, \dots, x_n) \in R$ , where  $R$  is a relation symbol of arity  $n$ .

For example, if we have a binary relation symbol  $E$  in our signature, then the formula

$$\psi(x, y) = (\exists z)((x, z) \in E \wedge (z, y) \in E),$$

pp-defines a binary relation in which elements  $a, b$  are related if there is a directed path of length 2 from  $a$  to  $b$  in  $E$ .

**Definition 4.** A relational structure  $\mathbb{B}$  is primitive positive definable in  $\mathbb{A}$  (or  $\mathbb{A}$  pp-defines  $\mathbb{B}$ ) if

- (i) the set  $B$  is a subset of  $A$  and is definable by a primitive positive formula interpreted in  $\mathbb{A}$ , and
- (ii) each relation  $R$  in the signature of  $\mathbb{B}$  is definable on the set  $B$  by a primitive positive formula interpreted in  $\mathbb{A}$ .

The following result relates the above definition to the complexity of CSPs.

**Lemma 1.** [13] *Let  $\mathbb{A}$  be a finite relational structure that pp-defines  $\mathbb{B}$ . Then,  $\text{CSP}(\mathbb{B})$  is polynomial time (indeed, logspace) reducible to  $\text{CSP}(\mathbb{A})$ .*

It so happens that, if  $\mathbb{A}$  pp-defines  $\mathbb{B}$ , then  $\mathbb{B}$  inherits the polymorphisms of  $\mathbb{A}$ . See [5] for a detailed explanation.

**Lemma 2.** [5] *Let  $\mathbb{A}$  be a finite relational structure that pp-defines  $\mathbb{B}$ . If  $\varphi$  is a polymorphism of  $\mathbb{A}$ , then its restriction to  $B$  is a polymorphism of  $\mathbb{B}$ .*

In particular, if  $\mathbb{A}$  pp-defines  $\mathbb{B}$  and  $\mathbb{A}$  has a WNU polymorphism  $\omega$ , then  $\omega$  restricted to  $B$  is a WNU polymorphism of  $\mathbb{B}$ .

In the case that  $\mathbb{A}$  pp-defines  $\mathbb{B}$  and  $\mathbb{B}$  pp-defines  $\mathbb{A}$ , we say that  $\mathbb{A}$  and  $\mathbb{B}$  are *pp-equivalent*. In this case,  $\text{CSP}(\mathbb{A})$  and  $\text{CSP}(\mathbb{B})$  are essentially the same problems (they are logspace equivalent) and  $\mathbb{A}$  and  $\mathbb{B}$  have the same polymorphisms.

*Example 1.* Let  $\mathbb{A} = (A; R_1, \dots, R_n)$ , where each  $R_i$  is  $k_i$ -ary, and define  $R = R_1 \times \dots \times R_n$ . Then the structure  $\mathbb{A}' = (A; R)$  is pp-equivalent to  $\mathbb{A}$ .

Indeed, let  $k = \sum_{i=1}^n k_i$  be the arity of  $R$  and  $l_i = \sum_{j < i} k_j$  for  $i = 1, \dots, n$ . The relation  $R$  is pp-definable from  $R_1, \dots, R_n$  using the formula

$$\Psi(x_1, \dots, x_k) = \bigwedge_{i=1}^n (x_{l_i+1}, \dots, x_{l_i+k_i}) \in R_i.$$

The relation  $R_1$  can be defined from  $R$  by the primitive positive formula

$$\Psi(x_1, \dots, x_{k_1}) = (\exists y_{k_1+1}, \dots, \exists y_k)((x_1, \dots, x_{k_1}, y_{k_1+1}, \dots, y_k) \in R)$$

and the remaining  $R_i$ 's can be defined similarly.

Example 1 shows that when proving Theorem 1 we can restrict ourselves to relational structures with a single relation.

### 3.4 Digraphs

A *directed graph*, or *digraph*, is a relational structure  $\mathbb{G}$  with a single binary relation symbol  $E$  as its signature. We typically call the members of  $G$  and  $E^{\mathbb{G}}$  *vertices* and *edges*, respectively. We usually write  $a \rightarrow b$  to mean  $(a, b) \in E^{\mathbb{G}}$ , if there is no ambiguity.

A special case of relational structure homomorphism (see Definition 1), is that of digraph homomorphism. That is, given digraphs  $\mathbb{G}$  and  $\mathbb{H}$ , a function  $\varphi : G \rightarrow H$  is a homomorphism if  $(\varphi(a), \varphi(b)) \in E^{\mathbb{H}}$  whenever  $(a, b) \in E^{\mathbb{G}}$ .



**Definition 5.** For  $i = 1, \dots, n$ , let  $\mathbb{G}_i = (G_i, E_i)$  be digraphs. The direct product of  $\mathbb{G}_1, \dots, \mathbb{G}_n$ , denoted by  $\prod_{i=1}^n \mathbb{G}_i$ , is the digraph with vertices  $\prod_{i=1}^n G_i$  (the cartesian product of the sets  $G_i$ ) and edge relation

$$\{(\mathbf{a}, \mathbf{b}) \in (\prod_{i=1}^n G_i)^2 \mid (a_i, b_i) \in E_i \text{ for } i = 1 \dots, n\}.$$

If  $\mathbb{G}_1 = \dots = \mathbb{G}_n = \mathbb{G}$  then we write  $\mathbb{G}^n$  to mean  $\prod_{i=1}^n \mathbb{G}_i$ .

With the above definition in mind, an  $n$ -ary polymorphism on a digraph  $\mathbb{G}$  is simply a digraph homomorphism from  $\mathbb{G}^n$  to  $\mathbb{G}$ .

**Definition 6.** A digraph  $\mathbb{P}$  is an oriented path if it consists of a sequence of vertices  $v_0, v_1, \dots, v_k$  such that precisely one of  $(v_{i-1}, v_i), (v_i, v_{i-1})$  is an edge, for each  $i = 1, \dots, k$ . We require oriented paths to have a direction; we denote the initial vertex  $v_0$  and the terminal vertex  $v_k$  by  $\iota\mathbb{P}$  and  $\tau\mathbb{P}$ , respectively.

Given a digraph  $\mathbb{G}$  and an oriented path  $\mathbb{P}$ , we write  $a \xrightarrow{\mathbb{P}} b$  to mean that we can walk in  $\mathbb{G}$  from  $a$  following  $\mathbb{P}$  to  $b$ , i.e., there exists a homomorphism  $\varphi : \mathbb{P} \rightarrow \mathbb{G}$  such that  $\varphi(\iota\mathbb{P}) = a$  and  $\varphi(\tau\mathbb{P}) = b$ . Note that for every  $\mathbb{P}$  there exists a primitive positive formula  $\psi(x, y)$  such that  $a \xrightarrow{\mathbb{P}} b$  if and only if  $\psi(a, b)$  is true in  $\mathbb{G}$ . If there exists an oriented path  $\mathbb{P}$  such that  $a \xrightarrow{\mathbb{P}} b$ , we say that  $a$  and  $b$  are *connected*. If vertices  $a$  and  $b$  are connected, then the *distance* from  $a$  to  $b$  is the number of edges in the shortest oriented path connecting them. Connectedness forms an equivalence relation on  $G$ ; its classes are called the *connected components* of  $\mathbb{G}$ . We say that a digraph is connected if it consists of a single connected component.

A connected digraph is *balanced* if it admits a *level function*  $\text{lvl} : G \rightarrow \mathbb{N}$ , where  $\text{lvl}(b) = \text{lvl}(a) + 1$  whenever  $(a, b)$  is an edge, and the minimum level is 0. The maximum level is called the *height* of the digraph. Oriented paths are natural examples of balanced digraphs.

By a *zigzag* we mean the oriented path  $\bullet \rightarrow \bullet \leftarrow \bullet \rightarrow \bullet$  and a *single edge* is the path  $\bullet \rightarrow \bullet$ . For oriented paths  $\mathbb{P}$  and  $\mathbb{P}'$ , the *concatenation* of  $\mathbb{P}$  and  $\mathbb{P}'$ , denoted by  $\mathbb{P} \dot{+} \mathbb{P}'$ , is the oriented path obtained by identifying  $\tau\mathbb{P}$  with  $\iota\mathbb{P}'$ .

Our digraph reduction as described in Section 4 relies on oriented paths obtained by concatenation of zigzags and single edges. For example, the path in Figure 1 is a concatenation of a single edge followed by two zigzags and two more single edges (for clarity, we organise its vertices into levels).

## 4 The Reduction to Digraphs

In this section we take an arbitrary finite relational structure  $\mathbb{A}$  and construct a balanced digraph  $\mathcal{D}(\mathbb{A})$  such that  $\text{CSP}(\mathbb{A})$  and  $\text{CSP}(\mathcal{D}(\mathbb{A}))$  are logspace equivalent.

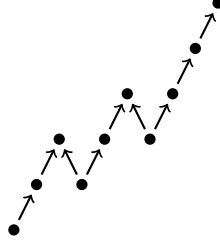


**Fig. 1.** A minimal oriented path

Let  $\mathbb{A} = (A; R_1, \dots, R_n)$  be a finite relational structure, where  $R_i$  is of arity  $k_i$ , for  $i = 1, \dots, n$ . Let  $k = \sum_{i=1}^n k_i$  and let  $R$  be the  $k$ -ary relation  $R_1 \times \dots \times R_n$ . For  $\mathcal{I} \subseteq [k]$  define  $\mathbb{Q}_{\mathcal{I},l}$  to be a single edge if  $l \in \mathcal{I}$ , and a zigzag if  $l \in [k] \setminus \mathcal{I}$ .

We define the oriented path  $\mathbb{Q}_{\mathcal{I}}$  (of height  $k+2$ ) by

$$\mathbb{Q}_{\mathcal{I}} = \bullet \rightarrow \bullet \dot{+} \mathbb{Q}_{\mathcal{I},1} \dot{+} \mathbb{Q}_{\mathcal{I},2} \dot{+} \dots \dot{+} \mathbb{Q}_{\mathcal{I},k} \dot{+} \bullet \rightarrow \bullet$$

Instead of  $\mathbb{Q}_{\emptyset}, \mathbb{Q}_{\emptyset,l}$  we write just  $\mathbb{Q}, \mathbb{Q}_l$ , respectively. For example, the oriented path in Figure 1 is  $\mathbb{Q}_{\mathcal{I}}$  where  $k = 3$  and  $\mathcal{I} = \{3\}$ . We will need the following observation.

**Observation.** *Let  $\mathcal{I}, \mathcal{J} \subseteq [k]$ . A homomorphism  $\varphi : \mathbb{Q}_{\mathcal{I}} \rightarrow \mathbb{Q}_{\mathcal{J}}$  exists, if and only if  $\mathcal{I} \subseteq \mathcal{J}$ . In particular  $\mathbb{Q} \rightarrow \mathbb{Q}_{\mathcal{I}}$  for all  $\mathcal{I} \subseteq [k]$ . Moreover, if  $\varphi$  exists, it is unique and surjective.*

We are now ready to define the digraph  $\mathcal{D}(\mathbb{A})$ .

**Definition 7.** *For every  $e = (a, \mathbf{r}) \in A \times R$  we define  $\mathbb{P}_e$  to be the path  $\mathbb{Q}_{\{i:a=r_i\}}$ . The digraph  $\mathcal{D}(\mathbb{A})$  is obtained from the digraph  $(A \cup R; A \times R)$  by replacing every  $e = (a, \mathbf{r}) \in A \times R$  by the oriented path  $\mathbb{P}_e$  (identifying  $\iota \mathbb{P}_e$  with  $a$  and  $\tau \mathbb{P}_e$  with  $\mathbf{r}$ ).*

(We often write  $\mathbb{P}_{e,l}$  to mean  $\mathbb{Q}_{\mathcal{I},l}$  where  $\mathbb{P}_e = \mathbb{Q}_{\mathcal{I}}$ .)

*Example 2.* Consider the relational structure  $\mathbb{A} = (\{0, 1\}; R)$  where  $R = \{(0, 1), (1, 0)\}$ , i.e.,  $\mathbb{A}$  is the 2-cycle. Figure 2 is a visual representation of  $\mathcal{D}(\mathbb{A})$ .

*Remark 1.* The number of vertices in  $\mathcal{D}(\mathbb{A})$  is  $(3k+1)|R||A| + (1-2k)|R| + |A|$  and the number of edges is  $(3k+2)|R||A| - 2k|R|$ . The construction of  $\mathcal{D}(\mathbb{A})$  can be performed in logspace (under any reasonable encoding).

*Proof.* The vertices of  $\mathcal{D}(\mathbb{A})$  consist of the elements of  $A \cup R$ , along with vertices from the connecting paths. The number of vertices lying strictly within the connecting paths would be  $(3k+1)|R||A|$  if every  $\mathbb{P}_e$  was  $\mathbb{Q}$ . We need to deduct 2 vertices whenever there is a single edge instead of a zigzag and there are  $\sum_{(a,\mathbf{r}) \in A \times R} |\{i : a = r_i\}| = k|R|$  such instances. The number of edges is counted very similarly.  $\square$

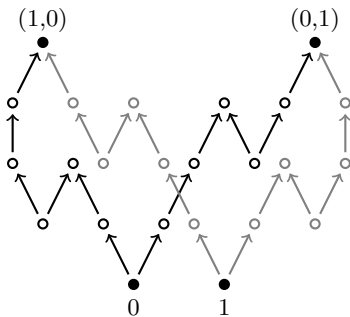


**Fig. 2.**  $\mathcal{D}(\mathbb{A})$  where  $\mathbb{A}$  is the 2-cycle

*Remark 2.* Note that if we apply this construction to itself (that is,  $\mathcal{D}(\mathcal{D}(\mathbb{A}))$ ) then we obtain balanced digraphs of height 4. When applied to digraphs, the  $\mathcal{D}$  construction is identical to that given by Feder and Vardi [8, Theorem 13].

The following lemma, together with Lemma 1, shows that  $\text{CSP}(\mathbb{A})$  reduces to  $\text{CSP}(\mathcal{D}(\mathbb{A}))$  in logspace.

**Lemma 3.**  $\mathbb{A}$  is pp-definable from  $\mathcal{D}(\mathbb{A})$ .

*Proof.* Example 1 demonstrates that  $\mathbb{A}$  is pp-equivalent to  $(A; R)$ . We now show that  $\mathcal{D}(\mathbb{A})$  pp-defines  $(A; R)$ , from which it follows that  $\mathcal{D}(\mathbb{A})$  pp-defines  $\mathbb{A}$ .

Note that  $\mathbb{Q} \rightarrow \mathbb{P}_e$  for all  $e \in A \times R$ , and  $\mathbb{Q}_{\{i\}} \rightarrow \mathbb{P}_{(a,r)}$  if and only if  $a = r_i$ . The set  $A$  is pp-definable in  $\mathcal{D}(\mathbb{A})$  by  $A = \{x \mid (\exists y)(x \xrightarrow{\mathbb{Q}} y)\}$  and the relation  $R$  can be defined as the set  $\{(x_1, \dots, x_k) \mid (\exists y)(x_i \xrightarrow{\mathbb{Q}_{\{i\}}} y \text{ for all } i \in [k])\}$ , which is also a primitive positive definition.  $\square$

It is not, in general, possible to pp-define  $\mathcal{D}(\mathbb{A})$  from  $\mathbb{A}$ .<sup>1</sup> Nonetheless the following lemma is true.

**Lemma 4.**  $\text{CSP}(\mathcal{D}(\mathbb{A}))$  reduces in logspace to  $\text{CSP}(\mathbb{A})$ .

The proof of Lemma 4 is rather technical, though broadly follows the polynomial process described in the proof of [8, Theorem 13] (as mentioned, our construction coincides with theirs in the case of digraphs). Details of the argument will be presented in a subsequent expanded version of this article.

Lemma 3 and Lemma 4 complete the proof of part (i) of Theorem 1. As this improves the oft-mentioned polynomial time equivalence of general CSPs with digraph CSPs, we now present it as stand-alone statement.

**Theorem 4.** Every fixed finite template CSP is logspace equivalent to the CSP over some finite digraph.

<sup>1</sup> Using the definition of pp-definability as described in this paper, this is true for cardinality reasons. However, a result of Kazda [14] can be used to show that the statement remains true even for more general definitions of pp-definability.

## 5 Preserving Cores

In what follows, let  $\mathbb{A}$  be a fixed finite relational structure. Without loss of generality we may assume that  $\mathbb{A} = (A; R)$ , where  $R$  is a  $k$ -ary relation (see Example 1).

**Lemma 5.** *The endomorphisms of  $\mathbb{A}$  and  $\mathcal{D}(\mathbb{A})$  are in one-to-one correspondence.*

*Proof.* We first show that every endomorphism  $\varphi$  of  $\mathbb{A}$  can be extended to an endomorphism  $\overline{\varphi}$  of  $\mathcal{D}(\mathbb{A})$ . Let  $\overline{\varphi}(a) = \varphi(a)$  for  $a \in A$ , and let  $\overline{\varphi}(\mathbf{r}) = \varphi^{(k)}(\mathbf{r})$  for  $\mathbf{r} \in R$ . Note that  $\varphi^{(k)}(\mathbf{r}) \in R$  since  $\varphi$  is an endomorphism of  $\mathbb{A}$ .

Let  $c \in \mathcal{D}(\mathbb{A}) \setminus (A \cup R)$  and let  $e = (a, \mathbf{r})$  be such that  $c \in \mathbb{P}_e$ . Define  $e' = (\varphi(a), \varphi^{(k)}(\mathbf{r}))$ . If  $\mathbb{P}_{e,l}$  is a single edge for some  $l \in [k]$ , then  $r_l = a$  and  $\varphi(r_l) = \varphi(a)$ , and therefore  $\mathbb{P}_{e',l}$  is a single edge. Thus there exists a (unique) homomorphism  $\mathbb{P}_e \rightarrow \mathbb{P}_{e'}$ . Define  $\overline{\varphi}(c)$  to be the image of  $c$  under this homomorphism, completing the definition of  $\overline{\varphi}$ .

We now show that every endomorphism  $\Phi$  of  $\mathcal{D}(\mathbb{A})$  is of the form  $\overline{\varphi}$ , for some endomorphism  $\varphi$  of  $\mathbb{A}$ . Let  $\Phi$  be an endomorphism of  $\mathcal{D}(\mathbb{A})$ . Let  $\varphi$  be the restriction of  $\Phi$  to  $A$ . By Lemma 2 and Lemma 3,  $\varphi$  is an endomorphism of  $\mathbb{A}$ . For every  $e = (a, \mathbf{r})$ , the endomorphism  $\Phi$  maps  $\mathbb{P}_e$  onto  $\mathbb{P}_{(\varphi(a), \Phi(\mathbf{r}))}$ . If we set  $a = r_l$ , then  $\mathbb{P}_{e,l}$  is a single edge. In this case it follows that  $\mathbb{P}_{(\varphi(a), \Phi(\mathbf{r})),l}$  is also a single edge. Thus, by the construction of  $\mathcal{D}(\mathbb{A})$  the  $l^{\text{th}}$  coordinate of  $\Phi(\mathbf{r})$  is  $\varphi(a) = \varphi(r_l)$ . This proves that the restriction of  $\Phi$  to  $R$  is  $\varphi^{(k)}$  and therefore  $\Phi = \overline{\varphi}$ .  $\square$

The following corollary is Theorem 1 (iii).

**Corollary 1.**  *$\mathbb{A}$  is a core if and only if  $\mathcal{D}(\mathbb{A})$  is a core.*

*Proof.* To prove the corollary we need to show that an endomorphism  $\varphi$  of  $\mathbb{A}$  is surjective if and only if  $\overline{\varphi}$  (from Lemma 5) is surjective. Clearly, if  $\overline{\varphi}$  is surjective then so is  $\varphi$ .

Assume  $\varphi$  is surjective (and therefore an automorphism of  $\mathbb{A}$ ). It follows that  $\varphi^{(k)}$  is surjective on  $R$  and therefore  $\overline{\varphi}$  is a bijection when restricted to the set  $A \cup R$ . Let  $a \in A$  and  $\mathbf{r} \in R$ . By definition we know that  $\overline{\varphi}$  maps  $\mathbb{P}_{(a,\mathbf{r})}$  homomorphically onto  $\mathbb{P}_{(\varphi(a), \varphi^{(k)}(\mathbf{r}))}$ . Since  $\varphi$  has an inverse  $\varphi^{-1}$ , it follows that  $\overline{\varphi}^{-1}$  maps  $\mathbb{P}_{(\varphi(a), \varphi^{(k)}(\mathbf{r}))}$  homomorphically onto  $\mathbb{P}_{(a,\mathbf{r})}$ . Thus  $\mathbb{P}_{(a,\mathbf{r})}$  and  $\mathbb{P}_{(\varphi(a), \varphi^{(k)}(\mathbf{r}))}$  are isomorphic, completing the proof.  $\square$

To complete the proof of Theorem 1, it remains to show that our reduction preserves WNUs.

## 6 The Reduction Preserves WNUs

For  $m > 0$  let  $\Delta_m$  denote the connected component of the digraph  $\mathcal{D}(\mathbb{A})^m$  containing the *diagonal* (i.e., the set  $\{(c, c, \dots, c) \mid c \in \mathcal{D}(\mathbb{A})\}$ ).

**Lemma 6.** *The elements of the diagonal are all connected in  $\mathcal{D}(\mathbb{A})^m$ . Furthermore,  $A^m \subseteq \Delta_m$  and  $R^m \subseteq \Delta_m$ .*

*Proof.* The first statement follows from the fact that  $\mathcal{D}(\mathbb{A})$  is connected. To see that  $A^m \subseteq \Delta_m$  and  $R^m \subseteq \Delta_m$ , fix  $a \in A$ , and so by definition,  $(a, a, \dots, a) \in \Delta_m$ . Let  $(\mathbf{r}^1, \dots, \mathbf{r}^m) \in R^m$  and for every  $i \in [m]$  let  $\varphi_i : \mathbb{Q} \rightarrow \mathbb{P}_{(a, \mathbf{r}^i)}$ . The homomorphism defined by  $x \mapsto (\varphi_1(x), \dots, \varphi_m(x))$  witnesses  $(a, \dots, a) \xrightarrow{\mathbb{Q}} (\mathbf{r}^1, \dots, \mathbf{r}^m)$  in  $\mathcal{D}(\mathbb{A})^m$ . Hence  $R^m \subseteq \Delta_m$ . A similar argument gives  $A^m \subseteq \Delta_m$ .  $\square$

The following lemma shows that there is only one non-trivial connected component of  $\mathcal{D}(\mathbb{A})^m$  that contains tuples (whose entries are) on the same level in  $\mathcal{D}(\mathbb{A})$ ; namely  $\Delta_m$ .

**Lemma 7.** *Let  $m > 0$  and let  $\Gamma$  be a connected component of  $\mathcal{D}(\mathbb{A})^m$  containing an element  $\mathbf{c}$  such that  $\text{lvl}(c_1) = \dots = \text{lvl}(c_m)$ . Then every element  $\mathbf{d} \in \Gamma$  is of the form  $\text{lvl}(d_1) = \dots = \text{lvl}(d_m)$  and the following hold.*

- (i) *If  $\mathbf{c} \rightarrow \mathbf{d}$  is an edge in  $\Gamma$  such that  $\mathbf{c} \notin A^m$  and  $\mathbf{d} \notin R^m$ , then there exist  $e_1, \dots, e_m \in A \times R$  and  $l \in [k]$  such that  $\mathbf{c}, \mathbf{d} \in \prod_{i=1}^m \mathbb{P}_{e_i, l}$ .*
- (ii) *Either  $\Gamma = \Delta_m$  or  $\Gamma$  is one-element.*

*Proof.* First observe that if an element  $\mathbf{d}$  is connected in  $\mathcal{D}(\mathbb{A})^m$  to an element  $\mathbf{c}$  with  $\text{lvl}(c_1) = \dots = \text{lvl}(c_m)$ , then there is an oriented path  $\mathbb{Q}'$  such that  $\mathbf{c} \xrightarrow{\mathbb{Q}'} \mathbf{d}$  from which it follows that  $\text{lvl}(d_1) = \dots = \text{lvl}(d_m)$ .

To prove (i), let  $\mathbf{c} \rightarrow \mathbf{d}$  be an edge in  $\Gamma$  such that  $\mathbf{c} \notin A^m$  and  $\mathbf{d} \notin R^m$ . For  $i = 1, \dots, m$  let  $e_i$  be such that  $c_i \in \mathbb{P}_{e_i}$  and let  $l = \text{lvl}(c_1)$ . The claim now follows immediately from the construction of  $\mathcal{D}(\mathbb{A})$ .

It remains to prove (ii). If  $|\Gamma| > 1$ , then there is an edge  $\mathbf{c} \rightarrow \mathbf{d}$  in  $\Gamma$ . If  $\mathbf{c} \in A^m$  or  $\mathbf{d} \in R^m$ , then the result follows from Lemma 6. Otherwise, from (i), there exists  $l \in [k]$  and  $e_i = (a_i, \mathbf{r}^i)$  such that  $\mathbf{c}, \mathbf{d} \in \prod_{i=1}^m \mathbb{P}_{e_i, l}$ .

For every  $i \in [m]$  we can walk from  $c_i$  to  $\iota \mathbb{P}_{e_i, l}$  following the path  $\bullet \rightarrow \bullet \leftarrow \bullet$ ; and so  $\mathbf{c}$  and  $(\iota \mathbb{P}_{e_1, l}, \dots, \iota \mathbb{P}_{e_m, l})$  are connected. For every  $i \in [m]$  there exists a homomorphism  $\varphi_i : \mathbb{Q} \rightarrow \mathbb{P}_{e_i}$  such that  $\varphi_i(\iota \mathbb{Q}) = a_i$  and  $\varphi_i(\iota \mathbb{Q}_l) = \iota \mathbb{P}_{e_i, l}$ . The homomorphism  $\mathbb{Q} \rightarrow \mathcal{D}(\mathbb{A})^m$  defined by  $x \mapsto (\varphi_1(x), \dots, \varphi_m(x))$  shows that  $(a_1, \dots, a_m)$  and  $(\iota \mathbb{P}_{e_1, l}, \dots, \iota \mathbb{P}_{e_m, l})$  are connected. By transitivity,  $(a_1, \dots, a_m)$  is connected to  $\mathbf{c}$  and therefore  $(a_1, \dots, a_m) \in \Gamma$ . Using (i) we obtain  $\Gamma = \Delta_m$ .  $\square$

We are now ready to prove the main ingredient of Theorem 1 (ii). The proof of Lemma 8 is similar in essence to the proof of Lemma 5, although more complicated.

**Lemma 8.** *If  $\mathbb{A}$  has an  $m$ -ary WNU polymorphism, then  $\mathcal{D}(\mathbb{A})$  has an  $m$ -ary WNU polymorphism.*

*Proof.* Let  $\omega$  be an  $m$ -ary WNU polymorphism of  $\mathbb{A}$ . We construct an  $m$ -ary operation  $\bar{\omega}$  on  $\mathcal{D}(\mathbb{A})$ . We split the definition into several cases and subcases. Let  $\mathbf{c} \in \mathcal{D}(\mathbb{A})^m$ .

**Case 1.**  $|\{\text{lvl}(c_1), \dots, \text{lvl}(c_m)\}| > 1$ .

1a If there exists  $i \in [m]$  such that  $|\{\text{lvl}(c_j) \mid j \neq i\}| = 1$ , we define  $\overline{\omega}(\mathbf{c}) = c_i$ .

1b Else let  $\overline{\omega}(\mathbf{c}) = c_1$ .

**Case 2.**  $\text{lvl}(c_1) = \dots = \text{lvl}(c_m)$ , but  $\mathbf{c} \notin \Delta_m$ .

2a If there exists  $i \in [m]$  such that  $|\{c_j \mid j \neq i\}| = 1$ , we define  $\overline{\omega}(\mathbf{c}) = c_i$ .

2b Else let  $\overline{\omega}(\mathbf{c}) = c_1$ .

**Case 3.**  $\mathbf{c} \in \Delta_m$ .

3a If  $\{c_1, \dots, c_m\} \subseteq A$ , we define  $\overline{\omega}(\mathbf{c}) = \omega(\mathbf{c})$ .

3b If  $\{c_1, \dots, c_m\} \subseteq R$ , we define  $\overline{\omega}(\mathbf{c}) = \omega^{(k)}(\mathbf{c})$ .

3c Else, there exists  $\mathbf{d} \in \Delta_m \setminus (A^m \cup R^m)$  such that  $\mathbf{c} \rightarrow \mathbf{d}$  or  $\mathbf{d} \rightarrow \mathbf{c}$  in  $\mathcal{D}(\mathbb{A})^m$ . By Lemma 7 (ii), there exist  $l \in [k]$  and  $e_i = (a_i, \mathbf{r}^i)$  such that  $\mathbf{c} \in \prod_{i=1}^m \mathbb{P}_{e_i, l}$ . Let  $e = (a, \mathbf{r})$ , where  $a = \omega(a_1, \dots, a_m)$  and  $\mathbf{r} = \omega^{(k)}(\mathbf{r}^1, \dots, \mathbf{r}^m)$ . We set  $\overline{\omega}(\mathbf{c}) = \Phi(\mathbf{c})$ , where  $\Phi : \prod_{i=1}^m \mathbb{P}_{e_i, l} \rightarrow \mathbb{P}_{e, l}$  is defined as follows.

1. If  $\mathbb{P}_{e, l}$  is a single edge, then we set

$$\Phi(\mathbf{u}) = \begin{cases} \iota \mathbb{P}_{e, l} & \text{if } \text{lvl}(u_1) = \dots = \text{lvl}(u_m) = \text{lvl}(\iota \mathbb{P}_{e, l}) \\ \tau \mathbb{P}_{e, l} & \text{otherwise.} \end{cases}$$

2. If  $\mathbb{P}_{e, l}$  is a zigzag, then let  $I = \{i \in [m] \mid \mathbb{P}_{e_i, l} \text{ is a zigzag}\}$ . For every  $i \in I$  let  $\phi_i : \mathbb{P}_{e_i, l} \rightarrow \mathbb{P}_{e, l}$  be the unique isomorphism. We define  $\Phi(\mathbf{u})$  to be the vertex from  $\{\phi_i(u_i) : i \in I\}$  with minimal distance from  $\iota \mathbb{P}_{e, l}$ .

Let us first comment on correctness of the definition. In subcase 3b,  $\omega^{(k)}(\mathbf{c}) \in R$  follows from the fact that  $\omega$  preserves  $R$ . In subcase 3c, if  $\mathbb{P}_{e, l}$  is a zigzag, then  $I \neq \emptyset$ . Indeed, if all the  $\mathbb{P}_{e_i, l}$ 's were single edges, then  $r_l = \omega(r_l^1, \dots, r_l^m) = \omega(a_1, \dots, a_m) = a$  and so  $\mathbb{P}_{e, l}$  would also be a single edge. The  $e_i$ 's are uniquely determined by  $\mathbf{c}$ , and the choice of  $l$  is unique as well, with one exception: if  $l < k$  and  $c_i = \tau \mathbb{P}_{e_i, l} = \iota \mathbb{P}_{e_i, l+1}$  for every  $i \in [m]$ , then we have  $\mathbf{d} \rightarrow \mathbf{c} \rightarrow \mathbf{d}'$  for some  $\mathbf{d}, \mathbf{d}' \in \mathcal{D}(\mathbb{A})^m$  and we can choose  $l+1$  instead of  $l$ . However, it is not hard to see that the value assigned to  $\overline{\omega}(\mathbf{c})$  is the same in both cases, namely it is the vertex  $\tau \mathbb{P}_{e, l} = \iota \mathbb{P}_{e, l+1}$  (see property (b) below).

We need the following properties of the mapping  $\Phi$  defined in 3c.

- (a)  $\Phi$  is a homomorphism.
- (b)  $\Phi(\iota \mathbb{P}_{e_1, l}, \dots, \iota \mathbb{P}_{e_m, l}) = \iota \mathbb{P}_{e, l}$  and  $\Phi(\tau \mathbb{P}_{e_1, l}, \dots, \tau \mathbb{P}_{e_m, l}) = \tau \mathbb{P}_{e, l}$ .
- (c)  $\Phi$  does not depend on the ordering of the tuple  $(e_1, \dots, e_m)$ .
- (d) If  $e_1 = \dots = e_m = e$ , then  $\Phi : \mathbb{P}_{e, l}^m \rightarrow \mathbb{P}_{e, l}$  is idempotent, i.e.,  $\Phi(u, \dots, u) = u$  for all  $u \in \mathbb{P}_{e, l}$ .

All of the above properties follow easily from the definition of  $\Phi$ . We leave the verification to the reader. It remains to prove that  $\overline{\omega}$  is a WNU polymorphism of  $\mathcal{D}(\mathbb{A})$ .

*Claim.*  $\overline{\omega}$  is idempotent and satisfies the WNU equations.

Let  $c, d \in \mathcal{D}(\mathbb{A})$ . Note that all of the tuples  $(c, \dots, c, d), (c, \dots, d, c), \dots, (d, c, \dots, c)$  fall into the same subcase; the possibilities are 3a, 3b, 3c and if  $c \neq d$ , then also 1a or 2a. In subcases 1a and 2a the definition does not depend on the ordering of the input tuple at all; therefore the WNU equations hold (and since  $c \neq d$ , idempotency does not apply).

In case 3 we use the fact that  $\omega$  and  $\omega^{(k)}$  are idempotent and satisfy the WNU equations. In 3a and 3b the claim follows immediately. In 3c note that  $e$  is the same for all of the tuples  $(c, \dots, c, d), \dots, (d, c, \dots, c)$ , and if  $c = d$ , then  $e_1 = \dots = e_m = e$ . The WNU equations follow from property (c) of the mapping  $\Phi$ , and idempotency follows from (d).

*Claim.*  $\bar{\omega}$  is a polymorphism of  $\mathcal{D}(\mathbb{A})$ .

Let  $\mathbf{c}, \mathbf{d} \in \mathcal{D}(\mathbb{A})^m$  be such that  $\mathbf{c} \rightarrow \mathbf{d}$  is an edge in  $\mathcal{D}(\mathbb{A})^m$ , that is,  $c_i \rightarrow d_i$  for all  $i \in [m]$ . Both the tuples  $\mathbf{c}$  and  $\mathbf{d}$  fall into the same case and, by Lemma 7 (iii), it cannot be case 2. If it is case 1, then they also fall into the same subcase and it is easily seen that  $\bar{\omega}(\mathbf{c}) \rightarrow \bar{\omega}(\mathbf{d})$ .

If  $\mathbf{c}$  falls into subcase 3a, then  $\mathbf{d}$  falls into 3c. Let  $e_i$  be such that  $d_i \in \mathbb{P}_{e_i}$ . As  $c_i \rightarrow d_i$ , it follows that  $e_i = (c_i, \mathbf{r}^i)$  for some  $\mathbf{r}^i \in R$  and  $d_i = \iota \mathbb{P}_{e_i, 1}$ . Let us define  $c = \omega(\mathbf{c})$ ,  $\mathbf{r} = \omega^{(k)}(\mathbf{r}^1, \dots, \mathbf{r}^m)$  and  $e = (c, \mathbf{r})$ . Now  $\bar{\omega}(\mathbf{d})$  is the result of the mapping  $\Phi$  applied to the tuple of initial vertices of the  $\mathbb{P}_{e_i, 1}$ 's, which (by property (b)) is the initial vertex of  $\mathbb{P}_{e, 1}$ . So  $\bar{\omega}(\mathbf{c}) = c \rightarrow \iota \mathbb{P}_{e, 1} = \bar{\omega}(\mathbf{d})$ . The argument is similar if  $\mathbf{d}$  falls into 3b.

It remains to verify that  $\bar{\omega}(\mathbf{c}) \rightarrow \bar{\omega}(\mathbf{d})$  if both  $\mathbf{c}$  and  $\mathbf{d}$  fall into subcase 3c. Both  $\bar{\omega}(\mathbf{c})$  and  $\bar{\omega}(\mathbf{d})$  are defined using the mapping  $\Phi : \prod_{i=1}^m \mathbb{P}_{e_i, l} \rightarrow \mathbb{P}_{e, l}$ . Since  $\Phi$  is a homomorphism, we have  $\bar{\omega}(\mathbf{c}) = \Phi(\mathbf{c}) \rightarrow \Phi(\mathbf{d}) = \bar{\omega}(\mathbf{d})$ , concluding the proof.  $\square$

## 7 Discussion

The algebraic dichotomy conjecture proposes a polymorphism characterisation of tractability for core CSPs. A number of other algorithmic properties are also either proved or conjectured to correspond to the existence of polymorphisms with special equational properties. For instance, solvability by the few subpowers algorithm (a generalization of Gaussian elimination) as described in Idziak et al. [11] has a polymorphism characterisation [4], as well as problems of bounded width (see Theorem 3). The final statement in Theorem 1 already shows that  $\mathbb{A}$  has bounded width if and only if  $\mathcal{D}(\mathbb{A})$  has bounded width. Kazda [14] showed that every digraph with a Maltsev polymorphism must have a majority polymorphism, which is not the case for finite relational structures in general. In a later version of the present article we will show that Theorem 1(ii) extends to include almost all commonly encountered polymorphism properties aside from Maltsev. For instance the CSP over  $\mathbb{A}$  is solvable by the few subpowers algorithm if and only if the same is true for  $\mathcal{D}(\mathbb{A})$ . Among other conditions preserved under our reduction to digraphs is that of arc consistency (or width 1 problems [7]) and problems of bounded strict width [8]. The following example is a powerful consequence of the result.

*Example 3.* Let  $\mathbb{A}$  be the structure on  $\{0, 1\}$  with a single 4-ary relation

$$\{(0, 0, 0, 1), (0, 1, 1, 1), (1, 0, 1, 1), (1, 1, 0, 1)\}.$$

Clearly  $\mathbb{A}$  is a core. The polymorphisms of  $\mathbb{A}$  can be shown to be the idempotent term functions of the two element group, and from this it follows that  $\text{CSP}(\mathbb{A})$  is solvable by the few subpowers algorithm of [11], but is not bounded width. Then the CSP over the digraph  $\mathcal{D}(\mathbb{A})$  is also solvable by few subpowers but is not bounded width (that is, is not solvable by local consistency check).

Prior to the announcement of this example it had been temporarily conjectured by some researchers that solvability by the few subpowers algorithm implied solvability by local consistency check in the case of digraphs (this was the opening conjecture in Maróti's keynote presentation at the Second International Conference on Order, Algebra and Logics in Krakow 2011 for example). With 78 vertices and 80 edges, Example 3 also serves as a simpler alternative to the 368-vertex, 432-edge digraph whose CSP was shown by Atserias in [1, §4.2] to be tractable but not solvable by local consistency check.

There are also conjectured polymorphism classifications of the property of solvability within nondeterministic logspace and within logspace; see Larose and Tesson [16]. The required polymorphism conditions are among those we can show are preserved under the transition from  $\mathbb{A}$  to  $\mathcal{D}(\mathbb{A})$ . It then follows that these conjectures are true provided they can be established in the restricted case of CSPs over digraphs.

**Open Problems.** We conclude our paper with some further research directions. It is possible to show that the logspace reduction in Lemma 4 *cannot* be replaced by first order reductions. Is there a different construction that translates general CSPs to digraph CSPs with first order reductions in both directions?

Feder and Vardi [8] and Atserias [1] provide polynomial time reductions of CSPs to digraph CSPs. We vigorously conjecture that their reductions preserve the properties of possessing a WNU polymorphism (and of being cores; but this is routinely verified). Do these or other constructions preserve the precise arity of WNU polymorphisms? What other polymorphism properties are preserved? Do they preserve the bounded width property? Can they preserve *conservative* polymorphisms (the polymorphisms related to list homomorphism problems)? The third and fourth authors with Kowalski [12] have recently shown that a minor variation of the  $\mathcal{D}$  construction in the present article preserves  $k$ -ary WNU polymorphisms (and can serve as an alternative to  $\mathcal{D}$  in Theorem 1), but always fails to preserve many other polymorphism properties (such as those witnessing strict width).

**Acknowledgements.** The authors would like to thank Libor Barto, Marcin Kozik, Miklós Maróti and Barnaby Martin for their thoughtful comments and discussions.

## References

1. Atserias, A.: On digraph coloring problems and treewidth duality. European J. Combin. 29(4), 796–820 (2008), <http://dx.doi.org/10.1016/j.ejc.2007.11.004>



2. Barto, L., Kozik, M.: Constraint satisfaction problems of bounded width. In: 2009 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2009), pp. 595–603. IEEE Computer Soc., Los Alamitos (2009), <http://dx.doi.org/10.1109/FOCS.2009.32>
3. Barto, L., Kozik, M., Niven, T.: The CSP dichotomy holds for digraphs with no sources and no sinks (a positive answer to a conjecture of Bang-Jensen and Hell). *SIAM J. Comput.* 38(5), 1782–1802 (2008/2009), <http://dx.doi.org/10.1137/070708093>
4. Berman, J., Idziak, P., Marković, P., McKenzie, R., Valeriote, M., Willard, R.: Varieties with few subalgebras of powers. *Trans. Amer. Math. Soc.* 362(3), 1445–1473 (2010), <http://dx.doi.org/10.1090/S0002-9947-09-04874-0>
5. Bulatov, A., Jeavons, P., Krokhin, A.: Classifying the complexity of constraints using finite algebras. *SIAM J. Comput.* 34(3), 720–742 (2005), <http://dx.doi.org/10.1137/S0097539700376676>
6. Bulatov, A.A.: A dichotomy theorem for constraint satisfaction problems on a 3-element set. *J. ACM* 53(1), 66–120 (2006), <http://dx.doi.org/10.1145/1120582.1120584>
7. Dalmau, V., Pearson, J.: Closure functions and width 1 problems. In: Jaffar, J. (ed.) *CP 1999*. LNCS, vol. 1713, pp. 159–173. Springer, Heidelberg (1999)
8. 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 (electronic) (1999), <http://dx.doi.org/10.1137/S0097539794266766>
9. Hell, P., Nešetřil, J.: On the complexity of  $H$ -coloring. *J. Combin. Theory Ser. B* 48(1), 92–110 (1990), [http://dx.doi.org/10.1016/0095-8956\(90\)90132-J](http://dx.doi.org/10.1016/0095-8956(90)90132-J)
10. Hell, P., Nešetřil, J.: Graphs and homomorphisms. *Oxford Lecture Series in Mathematics and its Applications*, vol. 28. Oxford University Press, Oxford (2004), <http://dx.doi.org/10.1093/acprof:oso/9780198528173.001.0001>
11. Idziak, P., Marković, P., McKenzie, R., Valeriote, M., Willard, R.: Tractability and learnability arising from algebras with few subpowers. *SIAM J. Comput.* 39(7), 3023–3037 (2010), <http://dx.doi.org/10.1137/090775646>
12. Jackson, M., Kowalski, T., Niven, T.: Digraph related constructions and the complexity of digraph homomorphism problems. *arXiv:1304.4986 [math.CO]* (2013)
13. Jeavons, P., Cohen, D., Gyssens, M.: Closure properties of constraints. *J. ACM* 44(4), 527–548 (1997), <http://dx.doi.org/10.1145/263867.263489>
14. Kazda, A.: Maltsev digraphs have a majority polymorphism. *European J. Combin.* 32(3), 390–397 (2011), <http://dx.doi.org/10.1016/j.ejc.2010.11.002>
15. Ladner, R.E.: On the structure of polynomial time reducibility. *J. Assoc. Comput. Mach.* 22, 155–171 (1975)
16. Larose, B., Tesson, P.: Universal algebra and hardness results for constraint satisfaction problems. *Theor. Comput. Sci.* 410(18), 1629–1647 (2009)
17. Maróti, M., McKenzie, R.: Existence theorems for weakly symmetric operations. *Algebra Universalis* 59(3–4), 463–489 (2008), <http://dx.doi.org/10.1007/s00012-008-2122-9>
18. Montanari, U.: Networks of constraints: fundamental properties and applications to picture processing. *Information Sci.* 7, 95–132 (1974)
19. Schaefer, T.J.: The complexity of satisfiability problems. In: *Conference Record of the Tenth Annual ACM Symposium on Theory of Computing* (San Diego, Calif., 1978), pp. 216–226. ACM, New York (1978)

# A Scalable Approximate Model Counter<sup>\*,\*\*</sup>

Supratik Chakraborty<sup>1</sup>, Kuldeep S. Meel<sup>2</sup>, and Moshe Y. Vardi<sup>2</sup>

<sup>1</sup> Indian Institute of Technology Bombay, India

<sup>2</sup> Department of Computer Science, Rice University

**Abstract.** *Propositional model counting* ( $\#SAT$ ), i.e., counting the number of satisfying assignments of a propositional formula, is a problem of significant theoretical and practical interest. Due to the inherent complexity of the problem, *approximate model counting*, which counts the number of satisfying assignments to within given tolerance and confidence level, was proposed as a practical alternative to exact model counting. Yet, approximate model counting has been studied essentially only theoretically. The only reported implementation of approximate model counting, due to Karp and Luby, worked only for DNF formulas. A few existing tools for CNF formulas are *bounding model counters*; they can handle realistic problem sizes, but fall short of providing counts within given tolerance and confidence, and, thus, are not approximate model counters.

We present here a novel algorithm, as well as a reference implementation, that is the first scalable approximate model counter for CNF formulas. The algorithm works by issuing a polynomial number of calls to a SAT solver. Our tool, **ApproxMC**, scales to formulas with tens of thousands of variables. Careful experimental comparisons show that **ApproxMC** reports, with high confidence, bounds that are close to the exact count, and also succeeds in reporting bounds with small tolerance and high confidence in cases that are too large for computing exact model counts.

## 1 Introduction

Propositional model counting, also known as  $\#SAT$ , concerns counting the number of models (satisfying truth assignments) of a given propositional formula. This problem has been the subject of extensive theoretical investigation since its

---

\* Authors would like to thank Henry Kautz and Ashish Sabhrwal for their valuable help in experiments, and Tracy Volz for valuable comments on the earlier drafts. Work supported in part by NSF grants CNS 1049862 and CCF-1139011, by NSF Expeditions in Computing project “ExCAPE: Expeditions in Computer Augmented Program Engineering,” by BSF grant 9800096, by a gift from Intel, by a grant from Board of Research in Nuclear Sciences, India, and by the Shared University Grid at Rice funded by NSF under Grant EIA-0216467, and a partnership between Rice University, Sun Microsystems, and Sigma Solutions, Inc.

\*\* A longer version of this paper is available at <http://www.cs.rice.edu/CS/Verification/Projects/ApproxMC/>

introduction by Valiant [35] in 1979. Several interesting applications of  $\#SAT$  have been studied in the context of probabilistic reasoning, planning, combinatorial design and other related fields [24,4,9]. In particular, probabilistic reasoning and inferencing have attracted considerable interest in recent years [13], and stand to benefit significantly from efficient propositional model counters.

Theoretical investigations of  $\#SAT$  have led to the discovery of deep connections in complexity theory [3,29,33]:  $\#SAT$  is  $\#P$ -complete, where  $\#P$  is the set of counting problems associated with decision problems in the complexity class  $NP$ . Furthermore,  $P^{\#SAT}$ , that is, a polynomial-time machine with a  $\#SAT$  oracle, can solve all problems in the entire polynomial hierarchy. In fact, the polynomial-time machine only needs to make one  $\#SAT$  query to solve any problem in the polynomial hierarchy. This is strong evidence for the hardness of  $\#SAT$ .

In many applications of model counting, such as in probabilistic reasoning, the exact model count may not be critically important, and approximate counts are sufficient. Even when exact model counts are important, the inherent complexity of the problem may force one to work with approximate counters in practice. In [31], Stockmeyer showed that counting models within a specified tolerance factor can be achieved in deterministic polynomial time using a  $\Sigma_2^P$ -oracle. Karp and Luby presented a fully polynomial randomized approximation scheme for counting models of a DNF formula [18]. Building on Stockmeyer's result, Jerrum, Valiant and Vazirani [16] showed that counting models of CNF formulas within a specified tolerance factor can be solved in random polynomial time using an oracle for  $SAT$ .

On the implementation front, the earliest approaches to  $\#SAT$  were based on DPLL-style  $SAT$  solvers and computed exact counts. These approaches consisted of incrementally counting the number of solutions by adding appropriate multiplication factors after a partial solution was found. This idea was formalized by Birnbaum and Lozinkii [6] in their model counter CDP. Subsequent model counters such as *Relsat* [17], *Cachet* [26] and *sharpSAT* [32] improved upon this idea by using several optimizations such as component caching, clause learning, look-ahead and the like. Techniques based on Boolean Decision Diagrams and their variants [23,21], or d-DNNF formulae [8], have also been used to compute exact model counts. Although exact model counters have been successfully used in small- to medium-sized problems, scaling to larger problem instances has posed significant challenges in practice. Consequently, a large class of practical applications has remained beyond the reach of exact model counters.

To counter the scalability challenge, more efficient techniques for counting models approximately have been proposed. These counters can be broadly divided into three categories. Counters in the first category are called  $(\varepsilon, \delta)$  counters, following Karp and Luby's terminology [18]. Let  $\varepsilon$  and  $\delta$  be real numbers such that  $0 < \varepsilon \leq 1$  and  $0 < \delta \leq 1$ . For every propositional formula  $F$  with  $\#F$  models, an  $(\varepsilon, \delta)$  counter computes a number that lies in the interval  $[(1 + \varepsilon)^{-1} \#F, (1 + \varepsilon) \#F]$  with probability at least  $1 - \delta$ . We say that  $\varepsilon$  is the

*tolerance* of the count, and  $1 - \delta$  is its *confidence*. The counter described in this paper and also that due to Karp and Luby [18] belong to this category. The approximate-counting algorithm of Jerrum et al. [16] also belongs to this category; however, their algorithm does not lend itself to an implementation that scales in practice. Counters in the second category are called *lower (or upper) bounding counters*, and are parameterized by a confidence probability  $1 - \delta$ . For every propositional formula  $F$  with  $\#F$  models, an upper (resp., lower) bounding counter computes a number that is at least as large (resp., as small) as  $\#F$  with probability at least  $1 - \delta$ . Note that bounding counters *do not* provide any tolerance guarantees. The large majority of approximate counters used in practice are bounding counters. Notable examples include **SampleCount** [14], **BPCount** [20], **MBound** (and **Hybrid-MBound**) [12], and **MiniCount** [20]. The final category of counters is called *guarantee-less counters*. These counters provide no guarantees at all but they can be very efficient and provide good approximations in practice. Examples of guarantee-less counters include **ApproxCount** [36], **SearchTreeSampler** [10], **SE** [25] and **SampleSearch** [11].

Bounding both the tolerance and confidence of approximate model counts is extremely valuable in applications like probabilistic inference. Thus, designing  $(\varepsilon, \delta)$  counters that scale to practical problem sizes is an important problem. Earlier work on  $(\varepsilon, \delta)$  counters has been restricted largely to theoretical treatments of the problem. The only counter in this category that we are aware of as having been implemented is due to Karp and Luby [22]. Karp and Luby’s original implementation was designed to estimate reliabilities of networks with failure-prone links. However, the underlying Monte Carlo engine can be used to approximately count models of DNF, *but not* CNF, formulas.

The counting problems for both CNF and DNF formulae are  $\#P$ -complete. While the DNF representation suits some applications, most modern applications of model counting (e.g. probabilistic inference) use the CNF representation. Although *exact* counting for DNF and CNF formulae are polynomially inter-reducible, there is no known polynomial reduction for the corresponding *approximate* counting problems. In fact, Karp and Luby remark in [18] that it is highly unlikely that their randomized approximate algorithm for DNF formulae can be adapted to work for CNF formulae. Thus, there has been no prior implementation of  $(\varepsilon, \delta)$  counters for CNF formulae *that scales in practice*. In this paper, we present the first such counter. As in [16], our algorithm runs in random polynomial time using an oracle for SAT. Our extensive experiments show that our algorithm scales, with low error, to formulae arising from several application domains involving tens of thousands of variables.

The organization of the paper is as follows. We present preliminary material in Section 2, and related work in Section 3. In Section 4, we present our algorithm, followed by its analysis in Section 5. Section 6 discusses our experimental methodology, followed by experimental results in Section 7. Finally, we conclude in Section 8.

## 2 Notation and Preliminaries

Let  $\Sigma$  be an alphabet and  $R \subseteq \Sigma^* \times \Sigma^*$  be a binary relation. We say that  $R$  is an  $\mathcal{NP}$ -relation if  $R$  is polynomial-time decidable, and if there exists a polynomial  $p(\cdot)$  such that for every  $(x, y) \in R$ , we have  $|y| \leq p(|x|)$ . Let  $L_R$  be the language  $\{x \in \Sigma^* \mid \exists y \in \Sigma^*, (x, y) \in R\}$ . The language  $L_R$  is said to be in  $\mathcal{NP}$  if  $R$  is an  $\mathcal{NP}$ -relation. The set of all satisfiable propositional logic formulae in CNF is a language in  $\mathcal{NP}$ . Given  $x \in L_R$ , a *witness* or *model* of  $x$  is a string  $y \in \Sigma^*$  such that  $(x, y) \in R$ . The set of all models of  $x$  is denoted  $R_x$ . For notational convenience, fix  $\Sigma$  to be  $\{0, 1\}$  without loss of generality. If  $R$  is an  $\mathcal{NP}$ -relation, we may further assume that for every  $x \in L_R$ , every witness  $y \in R_x$  is in  $\{0, 1\}^n$ , where  $n = p(|x|)$  for some polynomial  $p(\cdot)$ .

Let  $R \subseteq \{0, 1\}^* \times \{0, 1\}^*$  be an  $\mathcal{NP}$  relation. The *counting problem* corresponding to  $R$  asks “Given  $x \in \{0, 1\}^*$ , what is  $|R_x|$ ?” If  $R$  relates CNF propositional formulae to their satisfying assignments, the corresponding counting problem is called  $\#SAT$ . The primary focus of this paper is on  $(\varepsilon, \delta)$  counters for  $\#SAT$ . The randomized  $(\varepsilon, \delta)$  counters of Karp and Luby [18] for DNF formulas are *fully polynomial*, which means that they run in time polynomial in the size of the input formula  $F$ ,  $1/\varepsilon$  and  $\log(1/\delta)$ . The randomized  $(\varepsilon, \delta)$  counters for CNF formulas in [16] and in this paper are however fully polynomial *with respect to a SAT oracle*.

A special class of hash functions, called *r-wise independent* hash functions, play a crucial role in our work. Let  $n, m$  and  $r$  be positive integers, and let  $H(n, m, r)$  denote a family of  $r$ -wise independent hash functions mapping  $\{0, 1\}^n$  to  $\{0, 1\}^m$ . We use  $\Pr[X : \mathcal{P}]$  to denote the probability of outcome  $X$  when sampling from a probability space  $\mathcal{P}$ , and  $h \xleftarrow{R} H(n, m, r)$  to denote the probability space obtained by choosing a hash function  $h$  uniformly at random from  $H(n, m, r)$ . The property of  $r$ -wise independence guarantees that for all  $\alpha_1, \dots, \alpha_r \in \{0, 1\}^m$  and for all distinct  $y_1, \dots, y_r \in \{0, 1\}^n$ ,  $\Pr[\bigwedge_{i=1}^r h(y_i) = \alpha_i : h \xleftarrow{R} H(n, m, r)] = 2^{-mr}$ . For every  $\alpha \in \{0, 1\}^m$  and  $h \in H(n, m, r)$ , let  $h^{-1}(\alpha)$  denote the set  $\{y \in \{0, 1\}^n \mid h(y) = \alpha\}$ . Given  $R_x \subseteq \{0, 1\}^n$  and  $h \in H(n, m, r)$ , we use  $R_{x,h,\alpha}$  to denote the set  $R_x \cap h^{-1}(\alpha)$ . If we keep  $h$  fixed and let  $\alpha$  range over  $\{0, 1\}^m$ , the sets  $R_{x,h,\alpha}$  form a partition of  $R_x$ . Following the notation in [5], we call each element of such a partition a *cell* of  $R_x$  *induced* by  $h$ . It was shown in [5] that if  $h$  is chosen uniformly at random from  $H(n, m, r)$  for  $r \geq 1$ , then the expected size of  $R_{x,h,\alpha}$ , denoted  $\mathbb{E}[|R_{x,h,\alpha}|]$ , is  $|R_x|/2^m$ , for each  $\alpha \in \{0, 1\}^m$ .

The specific family of hash functions used in our work, denoted  $H_{xor}(n, m, 3)$ , is based on randomly choosing bits from  $y \in \{0, 1\}^n$  and xor-ing them. This family of hash functions has been used in earlier work [12], and has been shown to be 3-independent in [15]. Let  $h(y)[i]$  denote the  $i^{th}$  component of the bit-vector obtained by applying hash function  $h$  to  $y$ . The family  $H_{xor}(n, m, 3)$  is defined as  $\{h(y) \mid (h(y))[i] = a_{i,0} \oplus (\bigoplus_{k=1}^n a_{i,k} \cdot y[k]), a_{i,j} \in \{0, 1\}, 1 \leq i \leq m, 0 \leq j \leq n\}$ , where  $\oplus$  denotes the xor operation. By randomly choosing the  $a_{i,j}$ ’s, we can randomly choose a hash function from this family.

### 3 Related Work

Sipser pioneered a hashing based approach in [30], which has subsequently been used in theoretical [34,5] and practical [15,12,7] treatments of approximate counting and (near-)uniform sampling. Earlier implementations of counters that use the hashing-based approach are MBound and Hybrid-MBound [12]. Both these counters use the same family of hashing functions, i.e.,  $H_{xor}(n, m, 3)$ , that we use. Nevertheless, there are significant differences between our algorithm and those of MBound and Hybrid-MBound. Specifically, we are able to exploit properties of the  $H_{xor}(n, m, 3)$  family of hash functions to obtain a fully polynomial  $(\varepsilon, \delta)$  counter with respect to a SAT oracle. In contrast, both MBound and Hybrid-MBound are bounding counters, and cannot provide bounds on tolerance. In addition, our algorithm requires no additional parameters beyond the tolerance  $\varepsilon$  and confidence  $1 - \delta$ . In contrast, the performance and quality of results of both MBound and Hybrid-MBound, depend crucially on some hard-to-estimate parameters. It has been our experience that the right choice of these parameters is often domain dependent and difficult.

Jerrum, Valiant and Vazirani [16] showed that if  $R$  is a self-reducible  $\mathcal{NP}$  relation (such as SAT), the problem of generating models *almost uniformly* is polynomially inter-reducible with approximately counting models. The notion of almost uniform generation requires that if  $x$  is a problem instance, then for every  $y \in R_x$ , we have  $(1 + \varepsilon)^{-1}\varphi(x) \leq \Pr[y \text{ is generated}] \leq (1 + \varepsilon)\varphi(x)$ , where  $\varepsilon > 0$  is the specified tolerance and  $\varphi(x)$  is an appropriate function. Given an almost uniform generator  $\mathcal{G}$  for  $R$ , an input  $x$ , a tolerance bound  $\varepsilon$  and an error probability bound  $\delta$ , it is shown in [16] that one can obtain an  $(\varepsilon, \delta)$  counter for  $R$  by invoking  $\mathcal{G}$  polynomially (in  $|x|$ ,  $1/\varepsilon$  and  $\log_2(1/\delta)$ ) many times, and by using the generated samples to estimate  $|R_x|$ . For convenience of exposition, we refer to this approximate-counting algorithm as the JVV algorithm (after the last names of the authors).

An important feature of the JVV algorithm is that it uses the almost uniform generator  $\mathcal{G}$  as a black box. Specifically, the details of how  $\mathcal{G}$  works is of no consequence. Prima facie, this gives us freedom in the choice of  $\mathcal{G}$  when implementing the JVV algorithm. Unfortunately, while there are theoretical constructions of uniform generators in [5], we are not aware of any implementation of an almost uniform generator that scales to CNF formulas involving thousands of variables. The lack of a scalable and almost uniform generator presents a significant hurdle in implementing the JVV algorithm for practical applications. It is worth asking if we can make the JVV algorithm work without requiring  $\mathcal{G}$  to be an almost uniform generator. A closer look at the proof of correctness of the JVV algorithm [16] shows it relies crucially on the ability of  $\mathcal{G}$  to ensure that the probabilities of generation of any two distinct models of  $x$  differ by a factor in  $O(\varepsilon^2)$ . As discussed in [7], existing algorithms for randomly generating models either provide this guarantee but scale very poorly in practice (e.g., the algorithms in [5,37]), or scale well in practice without providing the above guarantee (e.g., the algorithms in [7,15,19]). Therefore, using an existing generator as a black box in the JVV algorithm would not give us an  $(\varepsilon, \delta)$  model counter that

scales in practice. The primary contribution of this paper is to show that a scalable  $(\varepsilon, \delta)$  counter can indeed be designed by using the same insights that went into the design of a *near uniform* generator, UniWit [7], but without using the generator as a black box in the approximate counting algorithm. Note that near uniformity, as defined in [7], is an even more relaxed notion of uniformity than almost uniformity. We leave the question of whether a near uniform generator can be used as a black box to design an  $(\varepsilon, \delta)$  counter as part of future work.

The central idea of UniWit, which is also shared by our approximate model counter, is the use of  $r$ -wise independent hashing functions to randomly partition the space of all models of a given problem instance into “small” cells. This idea was first proposed in [5], but there are two novel insights that allow UniWit [7] to scale better than other hashing-based sampling algorithms [5,15], while still providing guarantess on the quality of sampling. These insights are: (i) the use of computationally efficient linear hashing functions with low degrees of independence, and (ii) a drastic reduction in the size of “small” cells, from  $n^2$  in [5] to  $n^{1/k}$  (for  $2 \leq k \leq 3$ ) in [7], and even further to a constant in the current paper. We continue to use these key insights in the design of our approximate model counter, although UniWit is not used explicitly in the model counter.

## 4 Algorithm

We now describe our approximate model counting algorithm, called **ApproxMC**. As mentioned above, we use 3-wise independent linear hashing functions from the  $H_{xor}(n, m, 3)$  family, for an appropriate  $m$ , to randomly partition the set of models of an input formula into “small” cells. In order to test whether the generated cells are indeed small, we choose a random cell and check if it is non-empty and has no more than *pivot* elements, where *pivot* is a threshold that depends only on the tolerance bound  $\varepsilon$ . If the chosen cell is not small, we randomly partition the set of models into twice as many cells as before by choosing a random hashing function from the family  $H_{xor}(n, m + 1, 3)$ . The above procedure is repeated until either a randomly chosen cell is found to be non-empty and small, or the number of cells exceeds  $\frac{2^{n+1}}{pivot}$ . If all cells that were randomly chosen during the above process were either empty or not small, we report a counting failure and return  $\perp$ . Otherwise, the size of the cell last chosen is scaled by the number of cells to obtain an  $\varepsilon$ -approximate estimate of the model count.

The procedure outlined above forms the core engine of **ApproxMC**. For convenience of exposition, we implement this core engine as a function **ApproxMC****Core**. The overall **ApproxMC** algorithm simply invokes **ApproxMC****Core** sufficiently many times, and returns the median of the non- $\perp$  values returned by **ApproxMC****Core**. The pseudocode for algorithm **ApproxMC** is shown below.

**Algorithm** ApproxMC( $F, \varepsilon, \delta$ )

```

1: counter  $\leftarrow 0$ ;  $C \leftarrow \text{emptyList}$ ;
2: pivot  $\leftarrow 2 \times \text{ComputeThreshold}(\varepsilon)$ ;
3:  $t \leftarrow \text{ComputeIterCount}(\delta)$ ;
4: repeat:
5:    $c \leftarrow \text{ApproxMCCore}(F, \textit{pivot})$ ;
6:   counter  $\leftarrow \textit{counter} + 1$ ;
7:   if ( $c \neq \perp$ )
8:     AddToList( $C, c$ );
9: until (counter  $< t$ );
10: finalCount  $\leftarrow \text{FindMedian}(C)$ ;
11: return finalCount;

```

**Algorithm** ComputeThreshold( $\varepsilon$ )

```

1: return  $\lceil 3e^{1/2} (1 + \frac{1}{\varepsilon})^2 \rceil$ ;

```

**Algorithm** ComputeIterCount( $\delta$ )

```

1: return  $\lceil 35 \log_2(3/\delta) \rceil$ ;

```

Algorithm ApproxMC takes as inputs a CNF formula  $F$ , a tolerance  $\varepsilon$  ( $0 < \varepsilon \leq 1$ ) and  $\delta$  ( $0 < \delta \leq 1$ ) such that the desired confidence is  $1 - \delta$ . It computes two key parameters: (i) a threshold *pivot* that depends only on  $\varepsilon$  and is used in ApproxMCCore to determine the size of a “small” cell, and (ii) a parameter  $t$  ( $\geq 1$ ) that depends only on  $\delta$  and is used to determine the number of times ApproxMCCore is invoked. The particular choice of functions to compute the parameters *pivot* and  $t$  aids us in proving theoretical guarantees for ApproxMC in Section 5. Note that *pivot* is in  $\mathcal{O}(1/\varepsilon^2)$  and  $t$  is in  $\mathcal{O}(\log_2(1/\delta))$ . All non- $\perp$  estimates of the model count returned by ApproxMCCore are stored in the list  $C$ . The function AddToList( $C, c$ ) updates the list  $C$  by adding the element  $c$ . The final estimate of the model count returned by ApproxMC is the median of the estimates stored in  $C$ , computed using FindMedian( $C$ ). We assume that if the list  $C$  is empty, FindMedian( $C$ ) returns  $\perp$ .

The pseudocode for algorithm ApproxMCCore is shown below.

**Algorithm** ApproxMCCore( $F, \textit{pivot}$ )

```

/* Assume  $z_1, \dots, z_n$  are the variables of  $F$  */
1:  $S \leftarrow \text{BoundedSAT}(F, \textit{pivot} + 1)$ ;
2: if ( $|S| \leq \textit{pivot}$ )
3:   return  $|S|$ ;
4: else
5:    $l \leftarrow \lfloor \log_2(\textit{pivot}) \rfloor - 1$ ;  $i \leftarrow l - 1$ ;
6:   repeat
7:      $i \leftarrow i + 1$ ;
8:     Choose  $h$  at random from  $H_{xor}(n, i - l, 3)$ ;
9:     Choose  $\alpha$  at random from  $\{0, 1\}^{i-l}$ ;
10:     $S \leftarrow \text{BoundedSAT}(F \wedge (h(z_1, \dots, z_n) = \alpha), \textit{pivot} + 1)$ ;
11:   until ( $1 \leq |S| \leq \textit{pivot}$ ) or ( $i = n$ );
12:   if ( $|S| > \textit{pivot}$  or  $|S| = 0$ ) return  $\perp$ ;
13:   else return  $|S| \cdot 2^{i-l}$ ;

```



Algorithm **ApproxMCCore** takes as inputs a CNF formula  $F$  and a threshold  $pivot$ , and returns an  $\varepsilon$ -approximate estimate of the model count of  $F$ . We assume that **ApproxMCCore** has access to a function **BoundedSAT** that takes as inputs a proposition formula  $F'$  that is the conjunction of a CNF formula and xor constraints, as well as a threshold  $v \geq 0$ . **BoundedSAT**( $F', v$ ) returns a set  $S$  of models of  $F'$  such that  $|S| = \min(v, \#F')$ . If the model count of  $F$  is no larger than  $pivot$ , then **ApproxMCCore** returns the exact model count of  $F$  in line 3 of the pseudocode. Otherwise, it partitions the space of all models of  $F$  using random hashing functions from  $H_{xor}(n, i-l, 3)$  and checks if a randomly chosen cell is non-empty and has at most  $pivot$  elements. Lines 8–10 of the repeat-until loop in the pseudocode implement this functionality. The loop terminates if either a randomly chosen cell is found to be small and non-empty, or if the number of cells generated exceeds  $\frac{2^{n+1}}{pivot}$  (if  $i = n$  in line 11, the number of cells generated is  $2^{n-l} \geq \frac{2^{n+1}}{pivot}$ ). In all cases, unless the cell that was chosen last is empty or not small, we scale its size by the number of cells generated by the corresponding hashing function to compute an estimate of the model count. If, however, all randomly chosen cells turn out to be empty or not small, we report a counting error by returning  $\perp$ .

**Implementation Issues:** There are two steps in algorithm **ApproxMCCore** (lines 8 and 9 of the pseudocode) where random choices are made. Recall from Section 2 that choosing a random hash function from  $H_{xor}(n, m, 3)$  requires choosing random bit-vectors. It is straightforward to implement these choices and also the choice of a random  $\alpha \in \{0, 1\}^{i-l}$  in line 9 of the pseudocode, if we have access to a source of independent and uniformly distributed random bits. Our implementation uses pseudo-random sequences of bits generated from nuclear decay processes and made available at HotBits [2]. We download and store a sufficiently long sequence of random bits in a file, and access an appropriate number of bits sequentially whenever needed. We defer experimenting with sequences of bits obtained from other pseudo-random generators to a future study.

In lines 1 and 10 of the pseudocode for algorithm **ApproxMCCore**, we invoke the function **BoundedSAT**. Note that if  $h$  is chosen randomly from  $H_{xor}(n, m, 3)$ , the formula for which we seek models is the conjunction of the original (CNF) formula and xor constraints encoding the inclusion of each witness in  $h^{-1}(\alpha)$ . We therefore use a SAT solver optimized for conjunctions of xor constraints and CNF clauses as the back-end engine. Specifically, we use CryptoMiniSAT (version 2.9.2) [1], which also allows passing a parameter indicating the maximum number of witnesses to be generated.

Recall that **ApproxMCCore** is invoked  $t$  times with the same arguments in algorithm **ApproxMC**. Repeating the loop of lines 6–11 in the pseudocode of **ApproxMCCore** in each invocation can be time consuming if the values of  $i-l$  for which the loop terminates are large. In [7], a heuristic called *leap-frogging* was proposed to overcome this bottleneck in practice. With leap-frogging, we register the smallest value of  $i-l$  for which the loop terminates during the first few invocations of **ApproxMCCore**. In all subsequent invocations of **ApproxMCCore** with the same arguments, we start iterating the loop of lines 6–11 by initializing  $i-l$

to the smallest value registered from earlier invocations. Our experiments indicate that leap-frogging is extremely efficient in practice and leads to significant savings in time after the first few invocations of **ApproxMCCore**. A theoretical analysis of leapfrogging is deferred to future work.

## 5 Analysis of **ApproxMC**

The following result, a minor variation of Theorem 5 in [28], about Chernoff-Hoeffding bounds plays an important role in our analysis.

**Theorem 1.** *Let  $\Gamma$  be the sum of  $r$ -wise independent random variables, each of which is confined to the interval  $[0, 1]$ , and suppose  $\mathbb{E}[\Gamma] = \mu$ . For  $0 < \beta \leq 1$ , if  $r \leq \lfloor \beta^2 \mu e^{-1/2} \rfloor \leq 4$  then  $\Pr[|\Gamma - \mu| \geq \beta\mu] \leq e^{-r/2}$ .*

Let  $F$  be a CNF propositional formula with  $n$  variables. The next two lemmas show that algorithm **ApproxMCCore**, when invoked from **ApproxMC** with arguments  $F$ ,  $\varepsilon$  and  $\delta$ , behaves like an  $(\varepsilon, d)$  model counter for  $F$ , for a fixed confidence  $1 - d$  (possibly different from  $1 - \delta$ ). Throughout this section, we use the notations  $R_F$  and  $R_{F,h,\alpha}$  introduced in Section 2.

**Lemma 1.** *Let algorithm **ApproxMCCore**, when invoked from **ApproxMC**, return  $c$  with  $i$  being the final value of the loop counter in **ApproxMCCore**. Then,  $\Pr\left[(1 + \varepsilon)^{-1} \cdot |R_F| \leq c \leq (1 + \varepsilon) \cdot |R_F| \mid c \neq \perp \text{ and } i \leq \log_2 |R_F|\right] \geq 1 - e^{-3/2}$ .*

*Proof.* Referring to the pseudocode of **ApproxMCCore**, the lemma is trivially satisfied if  $|R_F| \leq \text{pivot}$ . Therefore, the only non-trivial case to consider is when  $|R_F| > \text{pivot}$  and **ApproxMCCore** returns from line 13 of the pseudocode. In this case, the count returned is  $2^{i-l} \cdot |R_{F,h,\alpha}|$ , where  $l = \lfloor \log_2(\text{pivot}) \rfloor - 1$  and  $\alpha, i$  and  $h$  denote (with abuse of notation) the values of the corresponding variables and hash functions in the final iteration of the repeat-until loop in lines 6–11 of the pseudocode.

For simplicity of exposition, we assume henceforth that  $\log_2(\text{pivot})$  is an integer. A more careful analysis removes this restriction with only a constant factor scaling of the probabilities. From the pseudocode of **ApproxMCCore**, we know that  $\text{pivot} = 2 \left\lceil 3e^{1/2} \left(1 + \frac{1}{\varepsilon}\right)^2 \right\rceil$ . Furthermore, the value of  $i$  is always in  $\{l, \dots, n\}$ . Since  $\text{pivot} < |R_F| \leq 2^n$  and  $l = \lfloor \log_2 \text{pivot} \rfloor - 1$ , we have  $l < \log_2 |R_F| \leq n$ . The lemma is now proved by showing that for every  $i$  in  $\{l, \dots, \lfloor \log_2 |R_F| \rfloor\}$ ,  $h \in H(n, i - l, 3)$  and  $\alpha \in \{0, 1\}^{i-l}$ , we have  $\Pr\left[(1 + \varepsilon)^{-1} \cdot |R_F| \leq 2^{i-l} |R_{F,h,\alpha}| \leq (1 + \varepsilon) \cdot |R_F|\right] \geq (1 - e^{-3/2})$ .

For every  $y \in \{0, 1\}^n$  and for every  $\alpha \in \{0, 1\}^{i-l}$ , define an indicator variable  $\gamma_{y,\alpha}$  as follows:  $\gamma_{y,\alpha} = 1$  if  $h(y) = \alpha$ , and  $\gamma_{y,\alpha} = 0$  otherwise. Let us fix  $\alpha$  and  $y$  and choose  $h$  uniformly at random from  $H(n, i - l, 3)$ . The random choice of  $h$  induces a probability distribution on  $\gamma_{y,\alpha}$ , such that  $\Pr[\gamma_{y,\alpha} = 1] = \Pr[h(y) = \alpha] = 2^{-(i-l)}$ , and  $\mathbb{E}[\gamma_{y,\alpha}] = \Pr[\gamma_{y,\alpha} = 1] = 2^{-(i-l)}$ . In addition, the 3-wise independence of hash functions chosen from  $H(n, i - l, 3)$  implies that for

every distinct  $y_a, y_b, y_c \in R_F$ , the random variables  $\gamma_{y_a, \alpha}$ ,  $\gamma_{y_b, \alpha}$  and  $\gamma_{y_c, \alpha}$  are 3-wise independent.

Let  $\Gamma_\alpha = \sum_{y \in R_F} \gamma_{y, \alpha}$  and  $\mu_\alpha = \mathbb{E}[\Gamma_\alpha]$ . Clearly,  $\Gamma_\alpha = |R_{F, h, \alpha}|$  and  $\mu_\alpha = \sum_{y \in R_F} \mathbb{E}[\gamma_{y, \alpha}] = 2^{-(i-l)} |R_F|$ . Since  $|R_F| > \text{pivot}$  and  $i \leq \log_2 |R_F|$ , using the expression for  $\text{pivot}$ , we get  $3 \leq \left[ e^{-1/2} (1 + \frac{1}{\varepsilon})^{-2} \cdot \frac{|R_F|}{2^{i-l}} \right]$ . Therefore, using Theorem 1,  $\Pr \left[ |R_F| \cdot \left( 1 - \frac{\varepsilon}{1+\varepsilon} \right) \leq 2^{i-l} |R_{F, h, \alpha}| \leq (1 + \frac{\varepsilon}{1+\varepsilon}) |R_F| \right] \geq 1 - e^{-3/2}$ . Simplifying and noting that  $\frac{\varepsilon}{1+\varepsilon} < \varepsilon$  for all  $\varepsilon > 0$ , we obtain  $\Pr \left[ (1 + \varepsilon)^{-1} \cdot |R_F| \leq 2^{i-l} |R_{F, h, \alpha}| \leq (1 + \varepsilon) \cdot |R_F| \right] \geq 1 - e^{-3/2}$ .

**Lemma 2.** *Given  $|R_F| > \text{pivot}$ , the probability that an invocation of ApproxMCCore from ApproxMC returns non- $\perp$  with  $i \leq \log_2 |R_F|$ , is at least  $1 - e^{-3/2}$ .*

*Proof.* Let us denote  $\log_2 |R_F| - l = \log_2 |R_F| - (\lfloor \log_2(\text{pivot}) \rfloor - 1)$  by  $m$ . Since  $|R_F| > \text{pivot}$  and  $|R_F| \leq 2^n$ , we have  $l < m + l \leq n$ . Let  $p_i$  ( $l \leq i \leq n$ ) denote the conditional probability that ApproxMCCore( $F, \text{pivot}$ ) terminates in iteration  $i$  of the repeat-until loop (lines 6–11 of the pseudocode) with  $1 \leq |R_{F, h, \alpha}| \leq \text{pivot}$ , given  $|R_F| > \text{pivot}$ . Since the choice of  $h$  and  $\alpha$  in each iteration of the loop are independent of those in previous iterations, the conditional probability that ApproxMCCore( $F, \text{pivot}$ ) returns non- $\perp$  with  $i \leq \log_2 |R_F| = m + l$ , given  $|R_F| > \text{pivot}$ , is  $p_l + (1 - p_l)p_{l+1} + \dots + (1 - p_l)(1 - p_{l+1}) \dots (1 - p_{m+l-1})p_{m+l}$ . Let us denote this sum by  $P$ . Thus,  $P = p_l + \sum_{i=l+1}^{m+l} \prod_{k=l}^{i-1} (1 - p_k) p_i \geq \left( p_l + \sum_{i=l+1}^{m+l-1} \prod_{k=l}^{i-1} (1 - p_k) p_i \right) p_{m+l} + \prod_{s=l}^{m+l-1} (1 - p_s) p_{m+l} = p_{m+l}$ . The lemma is now proved by using Theorem 1 to show that  $p_{m+l} \geq 1 - e^{-3/2}$ .

It was shown in Lemma 1 that  $\Pr \left[ (1 + \varepsilon)^{-1} \cdot |R_F| \leq 2^{i-l} |R_{F, h, \alpha}| \leq (1 + \varepsilon) \cdot |R_F| \right] \geq 1 - e^{-3/2}$  for every  $i \in \{l, \dots, \lfloor \log_2 |R_F| \rfloor\}$ ,  $h \in H(n, i - l, 3)$  and  $\alpha \in \{0, 1\}^{i-l}$ . Substituting  $\log_2 |R_F| = m + l$  for  $i$ , re-arranging terms and noting that the definition of  $m$  implies  $2^{-m} |R_F| = \text{pivot}/2$ , we get  $\Pr \left[ (1 + \varepsilon)^{-1} (\text{pivot}/2) \leq |R_{F, h, \alpha}| \leq (1 + \varepsilon) (\text{pivot}/2) \right] \geq 1 - e^{-3/2}$ . Since  $0 < \varepsilon \leq 1$  and  $\text{pivot} > 4$ , it follows that  $\Pr \left[ 1 \leq |R_{F, h, \alpha}| \leq \text{pivot} \right] \geq 1 - e^{-3/2}$ . Hence,  $p_{m+l} \geq 1 - e^{-3/2}$ .

**Theorem 2.** *Let an invocation of ApproxMCCore from ApproxMC return  $c$ . Then  $\Pr [c \neq \perp \text{ and } (1 + \varepsilon)^{-1} \cdot |R_F| \leq c \leq (1 + \varepsilon) \cdot |R_F|] \geq (1 - e^{-3/2})^2 > 0.6$ .*

*Proof sketch:* It is easy to see that the required probability is at least as large as  $\Pr [c \neq \perp \text{ and } i \leq \log_2 |R_F| \text{ and } (1 + \varepsilon)^{-1} \cdot |R_F| \leq c \leq (1 + \varepsilon) \cdot |R_F|]$ . From Lemmas 1 and 2, the latter probability is  $\geq (1 - e^{-3/2})^2$ .

We now turn to proving that the confidence can be raised to at least  $1 - \delta$  for  $\delta \in (0, 1]$  by invoking ApproxMCCore  $\mathcal{O}(\log_2(1/\delta))$  times, and by using the median of the non- $\perp$  counts thus returned. For convenience of exposition, we use  $\eta(t, m, p)$  in the following discussion to denote the probability of at least  $m$  heads in  $t$  independent tosses of a biased coin with  $\Pr[\text{heads}] = p$ . Clearly,  $\eta(t, m, p) = \sum_{k=m}^t \binom{t}{k} p^k (1 - p)^{t-k}$ .

**Theorem 3.** *Given a propositional formula  $F$  and parameters  $\varepsilon$  ( $0 < \varepsilon \leq 1$ ) and  $\delta$  ( $0 < \delta \leq 1$ ), suppose  $\text{ApproxMC}(F, \varepsilon, \delta)$  returns  $c$ . Then  $\Pr \left[ (1 + \varepsilon)^{-1} \cdot |R_F| \leq c \leq (1 + \varepsilon) \cdot |R_F| \right] \geq 1 - \delta$ .*

*Proof.* Throughout this proof, we assume that  $\text{ApproxMCCore}$  is invoked  $t$  times from  $\text{ApproxMC}$ , where  $t = \lceil 35 \log_2(3/\delta) \rceil$  (see pseudocode for  $\text{ComputeIterCount}$  in Section 4). Referring to the pseudocode of  $\text{ApproxMC}$ , the final count returned by  $\text{ApproxMC}$  is the median of non- $\perp$  counts obtained from the  $t$  invocations of  $\text{ApproxMCCore}$ . Let  $\text{Err}$  denote the event that the median is not in  $[(1 + \varepsilon)^{-1} \cdot |R_F|, (1 + \varepsilon) \cdot |R_F|]$ . Let “ $\#non\perp = q$ ” denote the event that  $q$  (out of  $t$ ) values returned by  $\text{ApproxMCCore}$  are non- $\perp$ . Then,  $\Pr[\text{Err}] = \sum_{q=0}^t \Pr[\text{Err} \mid \#non\perp = q] \cdot \Pr[\#non\perp = q]$ .

In order to obtain  $\Pr[\text{Err} \mid \#non\perp = q]$ , we define a 0-1 random variable  $Z_i$ , for  $1 \leq i \leq t$ , as follows. If the  $i^{\text{th}}$  invocation of  $\text{ApproxMCCore}$  returns  $c$ , and if  $c$  is either  $\perp$  or a non- $\perp$  value that does not lie in the interval  $[(1 + \varepsilon)^{-1} \cdot |R_F|, (1 + \varepsilon) \cdot |R_F|]$ , we set  $Z_i$  to 1; otherwise, we set it to 0. From Theorem 2,  $\Pr[Z_i = 1] = p < 0.4$ . If  $Z$  denotes  $\sum_{i=1}^t Z_i$ , a necessary (but not sufficient) condition for event  $\text{Err}$  to occur, given that  $q$  non- $\perp$ s were returned by  $\text{ApproxMCCore}$ , is  $Z \geq (t - q + \lceil q/2 \rceil)$ . To see why this is so, note that  $t - q$  invocations of  $\text{ApproxMCCore}$  must return  $\perp$ . In addition, at least  $\lceil q/2 \rceil$  of the remaining  $q$  invocations must return values outside the desired interval. To simplify the exposition, let  $q$  be an even integer. A more careful analysis removes this restriction and results in an additional constant scaling factor for  $\Pr[\text{Err}]$ . With our simplifying assumption,  $\Pr[\text{Err} \mid \#non\perp = q] \leq \Pr[Z \geq (t - q + q/2)] = \eta(t, t - q/2, p)$ . Since  $\eta(t, m, p)$  is a decreasing function of  $m$  and since  $q/2 \leq t - q/2 \leq t$ , we have  $\Pr[\text{Err} \mid \#non\perp = q] \leq \eta(t, t/2, p)$ . If  $p < 1/2$ , it is easy to verify that  $\eta(t, t/2, p)$  is an increasing function of  $p$ . In our case,  $p < 0.4$ ; hence,  $\Pr[\text{Err} \mid \#non\perp = q] \leq \eta(t, t/2, 0.4)$ .

It follows from above that  $\Pr[\text{Err}] = \sum_{q=0}^t \Pr[\text{Err} \mid \#non\perp = q] \cdot \Pr[\#non\perp = q] \leq \eta(t, t/2, 0.4) \cdot \sum_{q=0}^t \Pr[\#non\perp = q] = \eta(t, t/2, 0.4)$ . Since  $\binom{t}{t/2} \geq \binom{t}{k}$  for all  $t/2 \leq k \leq t$ , and since  $\binom{t}{t/2} \leq 2^t$ , we have  $\eta(t, t/2, 0.4) = \sum_{k=t/2}^t \binom{t}{k} (0.4)^k (0.6)^{t-k} \leq \binom{t}{t/2} \sum_{k=t/2}^t (0.4)^k (0.6)^{t-k} \leq 2^t \sum_{k=t/2}^t (0.6)^t (0.4/0.6)^k \leq 2^t \cdot 3 \cdot (0.6 \times 0.4)^{t/2} \leq 3 \cdot (0.98)^t$ . Since  $t = \lceil 35 \log_2(3/\delta) \rceil$ , it follows that  $\Pr[\text{Err}] \leq \delta$ .

**Theorem 4.** *Given an oracle for SAT,  $\text{ApproxMC}(F, \varepsilon, \delta)$  runs in time polynomial in  $\log_2(1/\delta)$ ,  $|F|$  and  $1/\varepsilon$  relative to the oracle.*

*Proof.* Referring to the pseudocode for  $\text{ApproxMC}$ , lines 1–3 take time no more than a polynomial in  $\log_2(1/\delta)$  and  $1/\varepsilon$ . The repeat-until loop in lines 4–9 is repeated  $t = \lceil 35 \log_2(3/\delta) \rceil$  times. The time taken for each iteration is dominated by the time taken by  $\text{ApproxMCCore}$ . Finally, computing the median in line 10 takes time linear in  $t$ . The proof is therefore completed by showing that  $\text{ApproxMCCore}$  takes time polynomial in  $|F|$  and  $1/\varepsilon$  relative to the SAT oracle.

Referring to the pseudocode for  $\text{ApproxMCCore}$ , we find that  $\text{BoundedSAT}$  is called  $\mathcal{O}(|F|)$  times. Each such call can be implemented by at most  $\text{pivot} + 1$

calls to a SAT oracle, and takes time polynomial in  $|F|$  and  $pivot + 1$  relative to the oracle. Since  $pivot + 1$  is in  $\mathcal{O}(1/\varepsilon^2)$ , the number of calls to the SAT oracle, and the total time taken by all calls to **BoundedSAT** in each invocation of **ApproxMCCore** is a polynomial in  $|F|$  and  $1/\varepsilon$  relative to the oracle. The random choices in lines 8 and 9 of **ApproxMCCore** can be implemented in time polynomial in  $n$  (hence, in  $|F|$ ) if we have access to a source of random bits. Constructing  $F \wedge h(z_1, \dots, z_n) = \alpha$  in line 10 can also be done in time polynomial in  $|F|$ .

## 6 Experimental Methodology

To evaluate the performance and quality of results of **ApproxMC**, we built a prototype implementation and conducted an extensive set of experiments. The suite of benchmarks represents problems from practical domains as well as problems of theoretical interest. In particular, we considered a wide range of model counting benchmarks from different domains including grid networks, plan recognition, DQMR networks, Langford sequences, circuit synthesis, random  $k$ -CNF and logistics problems [27,20]. The suite consisted of benchmarks ranging from 32 variables to 229100 variables in CNF representation. The complete set of benchmarks (numbering above 200) is available at <http://www.cs.rice.edu/CS/Verification/Projects/ApproxMC/>.

All our experiments were conducted on a high-performance computing cluster. Each individual experiment was run on a single node of the cluster; the cluster allowed multiple experiments to run in parallel. Every node in the cluster had two quad-core Intel Xeon processors with 4GB of main memory. We used 2500 seconds as the timeout for each invocation of **BoundedSAT** in **ApproxMCCore**, and 20 hours as the timeout for **ApproxMC**. If an invocation of **BoundedSAT** in line 10 of the pseudo-code of **ApproxMCCore** timed out, we repeated the iteration (lines 6-11 of the pseudocode of **ApproxMCCore**) without incrementing  $i$ . The parameters  $\varepsilon$  (tolerance) and  $\delta$  (confidence being  $1 - \delta$ ) were set to 0.75 and 0.1 respectively. With these parameters, **ApproxMC** successfully computed counts for benchmarks with upto 33,000 variables.

We implemented leap-frogging, as described in [7], to estimate initial values of  $i$  from which to start iterating the repeat-until loop of lines 6–11 of the pseudocode of **ApproxMCCore**. To further optimize the running time, we obtained tighter estimates of the iteration count  $t$  used in algorithm **ApproxMC**, compared to those given by algorithm **ComputelaterCount**. A closer examination of the proof of Theorem 3 shows that it suffices to have  $\eta(t, t/2, 0.4) \leq \delta$ . We therefore pre-computed a table that gave the smallest  $t$  as a function of  $\delta$  such that  $\eta(t, t/2, 0.4) \leq \delta$ . This sufficed for all our experiments and gave smaller values of  $t$  (we used  $t=41$  for  $\delta=0.1$ ) compared to those given by **ComputelaterCount**.

For purposes of comparison, we also implemented and conducted experiments with the exact counter **Cachet** [26] by setting a timeout of 20 hours on the same computing platform. We compared the running time of **ApproxMC** with that of **Cachet** for several benchmarks, ranging from benchmarks on which **Cachet** ran very efficiently to those on which **Cachet** timed out. We also measured the

quality of approximation produced by **ApproxMC** as follows. For each benchmark on which **Cachet** did not time out, we obtained the approximate count from **ApproxMC** with parameters  $\varepsilon = 0.75$  and  $\delta = 0.1$ , and checked if the approximate count was indeed within a factor of 1.75 from the exact count. Since the theoretical guarantees provided by our analysis are conservative, we also measured the relative error of the counts reported by **ApproxCount** using the  $L_1$  norm, for all benchmarks on which **Cachet** did not time out. For an input formula  $F_i$ , let  $A_{F_i}$  (resp.,  $C_{F_i}$ ) be the count returned by **ApproxCount** (resp., **Cachet**). We computed the  $L_1$  norm of the relative error as  $\frac{\sum_i |A_{F_i} - C_{F_i}|}{\sum_i C_{F_i}}$ .

Since **Cachet** timed out on most large benchmarks, we compared **ApproxMC** with state-of-the-art bounding counters as well. As discussed in Section 1, bounding counters do not provide any tolerance guarantees. Hence their guarantees are significantly weaker than those provided by **ApproxMC**, and a direct comparison of performance is not meaningful. Therefore, we compared the sizes of the intervals (i.e., difference between upper and lower bounds) obtained from existing state-of-the-art bounding counters with those obtained from **ApproxMC**. To obtain intervals from **ApproxMC**, note that Theorem 3 guarantees that if  $\text{ApproxMC}(F, \varepsilon, \delta)$  returns  $c$ , then  $\Pr[\frac{c}{1+\varepsilon} \leq |R_F| \leq (1+\varepsilon) \cdot c] \geq 1 - \delta$ . Therefore, **ApproxMC** can be viewed as computing the interval  $[\frac{c}{1+\varepsilon}, (1+\varepsilon) \cdot c]$  for the model count, with confidence  $\delta$ . We considered state-of-the-art lower bounding counters, viz. **MBound** [12], **Hybrid-MBound** [12], **SampleCount** [14] and **BPCount** [20], to compute a lower bound of the model count, and used **MiniCount** [20] to obtain an upper bound. We observed that **SampleCount** consistently produced better (i.e. larger) lower bounds than **BPCount** for our benchmarks. Furthermore, the authors of [12] advocate using **Hybrid-MBound** instead of **MBound**. Therefore, the lower bound for each benchmark was obtained by taking the maximum of the bounds reported by **Hybrid-MBound** and **SampleCount**.

We set the confidence value for **MiniCount** to 0.99 and **SampleCount** and **Hybrid-MBound** to 0.91. For a detailed justification of these choices, we refer the reader to the full version of our paper. Our implementation of **Hybrid-MBound** used the “conservative” approach described in [12], since this provides the best lower bounds with the required confidence among all the approaches discussed in [12]. Finally, to ensure fair comparison, we allowed all bounding counters to run for 20 hours on the same computing platform on which **ApproxMC** was run.

## 7 Results

The results on only a subset of our benchmarks are presented here for lack of space. Figure 1 shows how the running times of **ApproxMC** and **Cachet** compared on this subset of our benchmarks. The y-axis in the figure represents time in seconds, while the x-axis represents benchmarks arranged in ascending order of running time of **ApproxMC**. The comparison shows that although **Cachet** performed better than **ApproxMC** initially, it timed out as the “difficulty” of problems increased. **ApproxMC**, however, continued to return bounds with the specified tolerance and confidence, for many more difficult and larger problems. Eventually, however, even **ApproxMC** timed out for very large problem instances.

Our experiments clearly demonstrate that there is a large class of practical problems that lie beyond the reach of exact counters, but for which we can still obtain counts with  $(\varepsilon, \delta)$ -style guarantees in reasonable time. This suggests that given a model counting problem, it is advisable to run **Cachet** initially with a small timeout. If **Cachet** times out, **ApproxMC** should be run with a larger timeout. Finally, if **ApproxMC** also times out, counters with much weaker guarantees but shorter running times, such as bounding counters, should be used.

Figure 2 compares the model count computed by **ApproxMC** with the bounds obtained by scaling the exact count obtained from **Cachet** by the tolerance factor (1.75) on a subset of our benchmarks. The y-axis in this figure represents the model count on a log-scale, while the x-axis represents the benchmarks arranged in ascending order of the model count. The figure shows that in all cases, the count reported by **ApproxMC** lies within the specified tolerance of the exact count. Although we have presented results for only a subset of our benchmarks

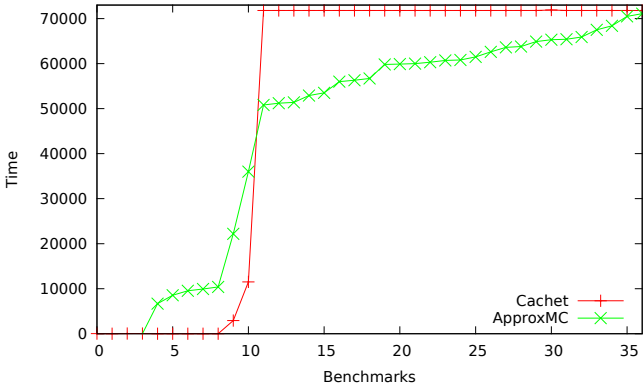


**Fig. 1.** Performance comparison between **ApproxMC** and **Cachet**. The benchmarks are arranged in increasing order of running time of **ApproxMC**.

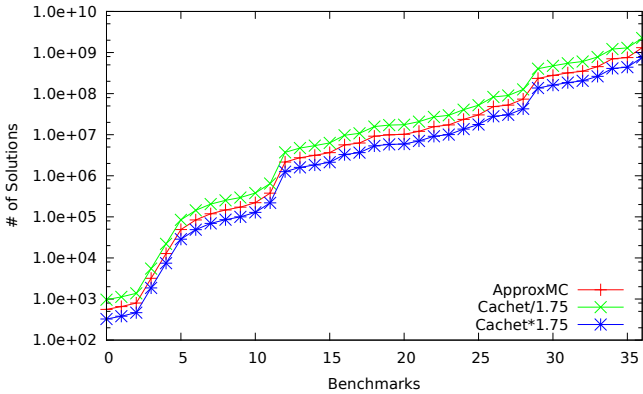


**Fig. 2.** Quality of counts computed by **ApproxMC**. The benchmarks are arranged in increasing order of model counts.

(37 in total) in Figure 2 for reasons of clarity, the counts reported by **ApproxMC** were found to be within the specified tolerance of the exact counts for *all* 95 benchmarks for which **Cachet** reported exact counts. We also found that the  $L_1$  norm of the relative error, considering all 95 benchmarks for which **Cachet** returned exact counts, was 0.033. Thus, **ApproxMC** has approximately 4% error in practice – much smaller than the theoretical guarantee of 75% with  $\varepsilon = 0.75$ .

Figure 3 compares the sizes of intervals computed using **ApproxMC** and using state-of-the-art bounding counters (as described in Section 6) on a subset of our benchmarks. The comparison clearly shows that the sizes of intervals computed using **ApproxMC** are consistently smaller than the sizes of the corresponding intervals obtained from existing bounding counters. Since smaller intervals with comparable confidence represent better approximations, we conclude that **ApproxMC** computes better approximations than a combination of existing bounding counters. In all cases, **ApproxMC** improved the upper bounds from **MiniCount** significantly; it also improved lower bounds from **SampleCount** and **MBound** to a lesser extent. For details, please refer to the full version.

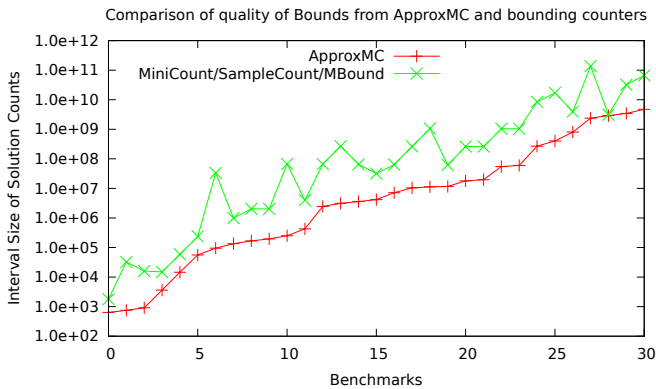


**Fig. 3.** Comparison of interval sizes from **ApproxMC** and those from bounding counters. The benchmarks are arranged in increasing order of model counts.

## 8 Conclusion and Future Work

We presented **ApproxMC**, the first  $(\varepsilon, \delta)$  approximate counter for CNF formulae that scales in practice to tens of thousands of variables. We showed that **ApproxMC** reports bounds with small tolerance in theory, and with much smaller error in practice, with high confidence. Extending the ideas in this paper to probabilistic inference and to count models of SMT constraints is an interesting direction of future research.



## References

1. CryptoMiniSAT, <http://www.msoos.org/cryptominisat2/>
2. HotBits, <http://www.fourmilab.ch/hotbits>
3. Angluin, D.: On counting problems and the polynomial-time hierarchy. *Theoretical Computer Science* 12(2), 161–173 (1980)
4. Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and complexity results for #SAT and bayesian inference. In: *Proc. of FOCS*, pp. 340–351 (2004)
5. Bellare, M., Goldreich, O., Petrank, E.: Uniform generation of NP-witnesses using an NP-oracle. *Information and Computation* 163(2), 510–526 (1998)
6. Birnbaum, E., Lozinskii, E.L.: The good old Davis-Putnam procedure helps counting models. *Journal of Artificial Intelligence Research* 10(1), 457–477 (1999)
7. Chakraborty, S., Meel, K.S., Vardi, M.Y.: A scalable and nearly uniform generator of SAT witnesses. In: Sharygina, N., Veith, H. (eds.) *CAV 2013. LNCS*, vol. 8044, pp. 608–623. Springer, Heidelberg (2013)
8. Darwiche, A.: New advances in compiling CNF to decomposable negation normal form. In: *Proc. of ECAI*, pp. 328–332. Citeseer (2004)
9. Domshlak, C., Hoffmann, J.: Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research* 30(1), 565–620 (2007)
10. Ermon, S., Gomes, C.P., Selman, B.: Uniform solution sampling using a constraint solver as an oracle. In: *Proc. of UAI* (2012)
11. Gogate, V., Dechter, R.: Samplesearch: Importance sampling in presence of determinism. *Artificial Intelligence* 175(2), 694–729 (2011)
12. Gomes, C.P., Sabharwal, A., Selman, B.: Model counting: A new strategy for obtaining good bounds. In: *Proc. of AAAI*, pp. 54–61 (2006)
13. Gomes, C.P., Sabharwal, A., Selman, B.: Model counting. In: Biere, A., Heule, M., Maaren, H.V., Walsh, T. (eds.) *Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications*, vol. 185, pp. 633–654. IOS Press (2009)
14. Gomes, C.P., Hoffmann, J., Sabharwal, A., Selman, B.: From sampling to model counting. In: *Proc. of IJCAI*, pp. 2293–2299 (2007)
15. Gomes, C.P., Sabharwal, A., Selman, B.: Near-uniform sampling of combinatorial spaces using XOR constraints. In: *Proc. of NIPS*, pp. 670–676 (2007)
16. Jerrum, M.R., Valiant, L.G., Vazirani, V.V.: Random generation of combinatorial structures from a uniform distribution. *Theoretical Computer Science* 43(2-3), 169–188 (1986)
17. Bayardo Jr., R.J., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: *Proc. of AAAI*, pp. 203–208 (1997)
18. Karp, R.M., Luby, M., Madras, N.: Monte-Carlo approximation algorithms for enumeration problems. *Journal of Algorithms* 10(3), 429–448 (1989)
19. Kitchen, N., Kuehlmann, A.: Stimulus generation for constrained random simulation. In: *Proc. of ICCAD*, pp. 258–265 (2007)
20. Kroc, L., Sabharwal, A., Selman, B.: Leveraging belief propagation, backtrack search, and statistics for model counting. In: Trick, M.A. (ed.) *CPAIOR 2008. LNCS*, vol. 5015, pp. 127–141. Springer, Heidelberg (2008)
21. Löbbing, M., Wegener, I.: The number of knight’s tours equals 33,439,123,484,294 – counting with binary decision diagrams. *The Electronic Journal of Combinatorics* 3(1), R5 (1996)
22. Luby, M.G.: Monte-Carlo Methods for Estimating System Reliability. PhD thesis, EECS Department, University of California, Berkeley (June 1983)

23. Minato, S.: Zero-suppressed bdds for set manipulation in combinatorial problems. In: Proc. of Design Automation Conference, pp. 272–277 (1993)
24. Roth, D.: On the hardness of approximate reasoning. *Artificial Intelligence* 82(1), 273–302 (1996)
25. Rubinfeld, R.: Stochastic enumeration method for counting np-hard problems. In: *Methodology and Computing in Applied Probability*, pp. 1–43 (2012)
26. Sang, T., Bacchus, F., Beame, P., Kautz, H., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: Proc. of SAT (2004)
27. Sang, T., Bearne, P., Kautz, H.: Performing bayesian inference by weighted model counting. In: Proc. of AAAI, pp. 475–481 (2005)
28. Schmidt, J.P., Siegel, A., Srinivasan, A.: Chernoff-Hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics* 8, 223–250 (1995)
29. Simon, J.: On the difference between one and many. In: Salomaa, A., Steinby, M. (eds.) *ICALP 1977. LNCS*, vol. 52, pp. 480–491. Springer, Heidelberg (1977)
30. Sipser, M.: A complexity theoretic approach to randomness. In: Proc. of STOC, pp. 330–335 (1983)
31. Stockmeyer, L.: The complexity of approximate counting. In: Proc. of STOC, pp. 118–126 (1983)
32. Thurley, M.: sharpSAT – counting models with advanced component caching and implicit BCP. In: Biere, A., Gomes, C.P. (eds.) *SAT 2006. LNCS*, vol. 4121, pp. 424–429. Springer, Heidelberg (2006)
33. Toda, S.: On the computational power of PP and (+)P. In: Proc. of FOCS, pp. 514–519. IEEE (1989)
34. Trevisan, L.: Lecture notes on computational complexity. Notes written in Fall (2002), <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.71.9877&rep=rep1&type=pdf>
35. Valiant, L.G.: The complexity of enumeration and reliability problems. *SIAM Journal on Computing* 8(3), 410–421 (1979)
36. Wei, W., Selman, B.: A new approach to model counting. In: Bacchus, F., Walsh, T. (eds.) *SAT 2005. LNCS*, vol. 3569, pp. 324–339. Springer, Heidelberg (2005)
37. Yuan, J., Aziz, A., Pixley, C., Albin, K.: Simplifying boolean constraint solving for random simulation-vector generation. *IEEE Trans. on CAD of Integrated Circuits and Systems* 23(3), 412–420 (2004)

# Dominance Driven Search

Geoffrey Chu<sup>2</sup> and Peter J. Stuckey<sup>1,2</sup>

<sup>1</sup> National ICT Australia, Victoria Laboratory

<sup>2</sup> Department of Computing and Information Systems,  
University of Melbourne, Australia  
{gchu,pjs}@csse.unimelb.edu.au

**Abstract.** Recently, a generic method for identifying and exploiting dominance relations using *dominance breaking constraints* was proposed. In this method, sufficient conditions for a solution to be dominated are identified and these conditions are used to generate dominance breaking constraints which prune off the dominated solutions. We propose to use these dominance relations in a different way in order to boost the search for good/optimal solutions. In the new method, which we call *dominance jumping*, when search reaches a point where all solutions in the current domain are dominated, rather than simply backtrack as in the original dominance breaking method, we jump to the subtree which dominates the current subtree. This new strategy allows the solver to move from a bad subtree to a good one, significantly increasing the speed with which good solutions can be found. Experiments across a range of problems show that the method can be very effective when the original search strategy was not very good at finding good solutions.

## 1 Introduction

Recently, a generic method for identifying and exploiting dominance relations using *dominance breaking constraints* was proposed in [3]. This method analyzes the effects of different assignment mappings on the satisfiability and objective value of solutions, and finds sufficient conditions under which a solution is dominated by (i.e., no better than) another one. Dominance breaking constraints are then generated which prune off these dominated solutions.

While symmetry and dominance breaking constraints are very powerful and can produce orders of magnitude speedup on a wide range of problems, it is well known that static symmetry breaking constraints (e.g., [20,4,14,21,8]) can conflict with the search strategy, leading to less speedup or even a slowdown [10]. Static dominance breaking constraints, which are a generalization of static symmetry breaking constraints, suffer from a similar problem. In optimization problems, dominance breaking constraints prevent the solver from finding any solution which is dominated. While such dominated solutions may not be optimal, they may nevertheless improve the current best solution and allow additional pruning through branch and bound. If the search strategy quickly guides the search to good/optimal non-dominated solutions, then there is no conflict between the search and the dominance breaking constraints. However, if we have a search strategy which keeps pushing the search into subtrees with only dominated solutions, then the search will potentially conflict with the dominance breaking constraints.

Consider a situation where the next 100 solutions in the search tree are all dominated, but one or more of them does improve the current best solution. A search without dominance breaking constraints will go through that portion of the search space without the pruning provided by the dominance breaking constraints, but once one of these better solutions are found, it will gain some additional pruning from branch and bound. A search with dominance breaking constraints will have the pruning provided by the dominance breaking constraints, but none of the pruning from the better solution that it could have found among these 100 solutions. Thus it is possible that the solver is actually slower at finding the optimal solution with dominance breaking constraints than without.

In symmetry breaking, the potential conflict between search strategy and static symmetry breaking constraints can be overcome by using a dynamic symmetry breaking method such as symmetry breaking during search (SBDS) [1,11] or symmetry breaking by dominance detection (SBDD) [5]. In this paper, we propose a different way to overcome this problem in the dominance case. We propose an altered method called *dominance jumping*, which exploits the information contained in the dominance relations in a different way. When a dominance breaking constraint prunes a partial assignment (because it is dominated by another), instead of simply backtracking as a normal CP solver would do, we jump to the subtree represented by the partial assignment which dominates the current one. Then, rather than getting stuck in a subtree where all the solutions are dominated and therefore not discoverable by a search with dominance breaking constraints, the dominance jumping can take us to a subtree with non-dominated solutions, allowing the search to find good non-dominated solutions and benefit from the additional pruning provided by branch and bound.

The rest of the paper is organized as follows: in the next section we introduce notation, and recall the approach to creating dominance breaking constraint described in [3]. In Section 3 we describe dominance jumping. In Section 4 we give experimental results comparing dominance breaking and dominance jumping. In Section 5 we examine related work, and finally in Section 6 we conclude.

## 2 Definitions

### 2.1 Constraints, Literals, and COPs

Let  $\equiv$  denote syntactical identity,  $\Rightarrow$  denote logical implication and  $\Leftrightarrow$  denote logical equivalence. We define variables and constraints in a problem independent way. A variable  $v$  is a mathematical quantity capable of assuming any value from a set of values called the *default domain* of  $v$ . Each variable is typed, e.g., Boolean or Integer, and its type determines its default domain, e.g.,  $\{0, 1\}$  for Boolean variables and  $\mathbb{Z}$  for Integer variables. Given a set of variables  $V$ , let  $\Theta_V$  denote the set of valuations over  $V$  where each variable in  $V$  is assigned to a value in its default domain. A constraint  $c$  over a set of variables  $V$  is defined by a set of valuations  $\text{solns}(c) \subseteq \Theta_V$ . Given a valuation  $\theta$  over  $V' \supset V$ , we say  $\theta$  satisfies  $c$  if the restriction of  $\theta$  onto  $V$  is in  $\text{solns}(c)$ . Otherwise, we say that  $\theta$  violates  $c$ . A domain  $D$  over variables  $V$  is a set of *unary constraints*, one for each variable in  $V$ . In an abuse of notation, if a symbol  $A$  refers to a set of constraints  $\{c_1, \dots, c_n\}$ , we will often also use the symbol  $A$  to refer to the constraint  $c_1 \wedge \dots \wedge c_n$ . This allows us to avoid repetitive use of conjunction symbols.

A *Constraint Satisfaction Problem* (CSP) is a tuple  $P \equiv (V, D, C)$ , where  $V$  is a set of variables,  $D$  is a domain over  $V$ , and  $C$  is a set of  $n$ -ary constraints. A valuation  $\theta$  over  $V$  is a *solution* of  $P$  if it satisfies every constraint in  $D$  and  $C$ . The aim of a CSP is to find a solution or to prove that none exist. In a *Constraint Optimization Problem* (COP)  $P \equiv (V, D, C, f)$ , we also have an objective function  $f$  mapping  $\Theta_V$  to an ordered set, e.g.,  $\mathbb{Z}$  or  $\mathbb{R}$ , and we wish to minimize or maximize  $f$  over the solutions of  $P$ . In this paper, we deal with finite domain problems only, i.e., where the initial domain  $D$  constrains each variable to take values from a finite set of values.

CP solvers solve CSP's by interleaving search with inference. We begin with the original problem at the root of the search tree. At each node in the search tree, we propagate the constraints to try to infer variable/value pairs which can no longer be taken in any solution in this subtree. Such pairs are removed from the current domain. If some variable's domain becomes empty, then the subtree has no solution and the solver backtracks. If all the variables are assigned and no constraint is violated, then a solution has been found and the solver can terminate. If inference is unable to detect either of the above two cases, the solver further divides the problem into a number of more constrained subproblems and searches each of those in turn. COP's are typically solved via branch and bound where we solve a series of CSP's with increasingly tight bounds on the objective value.

## 2.2 Dominance Breaking

Without loss of generality, assume that we are dealing with a minimization problem. We say that assignment  $\theta_1$  dominates  $\theta_2$  if either: 1)  $\theta_1$  is a solution and  $\theta_2$  is a non-solution, or 2) they are both solutions or both non-solutions and  $f(\theta_1) \leq f(\theta_2)$ . In [3], a generic method for identifying and exploiting dominance relations via dominance breaking constraints was proposed. The method can be briefly outlined as follows:

- Step 1 Choose a refinement of the objective function  $f'$  with the property that  $\forall \theta_1, \theta_2, f(\theta_1) < f(\theta_2)$  implies  $f'(\theta_1) < f'(\theta_2)$ .
- Step 2 Find mappings  $\sigma : \Theta_V \rightarrow \Theta_V$  which are likely to map solutions to better solutions.
- Step 3 For each  $\sigma$ , find a constraint  $scond(\sigma)$  s.t. if  $\theta \in solns(C \wedge D \wedge scond(\sigma))$ , then  $\sigma(\theta) \in solns(C \wedge D)$ .
- Step 4 For each  $\sigma$ , find a constraint  $ocond(\sigma)$  s.t. if  $\theta \in solns(C \wedge D \wedge ocond(\sigma))$ , then  $f'(\sigma(\theta)) < f'(\theta)$ .
- Step 5 For each  $\sigma$ , post the dominance breaking constraint  $db(\sigma) \equiv \neg(scond(\sigma) \wedge ocond(\sigma))$ .

The method analyzes the effects of different assignment mappings  $\sigma$  on the satisfiability and objective value of solutions, and finds sufficient conditions  $scond(\sigma) \wedge ocond(\sigma)$  under which a solution is dominated by another one. Dominance breaking constraints  $db(\sigma) \equiv \neg(scond(\sigma) \wedge ocond(\sigma))$  are then generated which prune off these dominated solutions. We now restate the main theorem from [3] showing the correctness of the dominance breaking constraints generated by this method.

**Theorem 1.** *Given a finite domain COP  $P \equiv (V, D, C, f)$ , a refinement of the objective function  $f'$  satisfying  $\forall \theta_1, \theta_2, f(\theta_1) < f(\theta_2)$  implies  $f'(\theta_1) < f'(\theta_2)$ , a set of mappings  $S$ , and for each mapping  $\sigma \in S$  constraints  $scond(\sigma)$  and  $ocond(\sigma)$  satisfying:  $\forall \sigma \in S$ ,*

if  $\theta \in \text{solns}(C \wedge D \wedge \text{scond}(\sigma))$ , then  $\sigma(\theta) \in \text{solns}(C \wedge D)$ , and:  $\forall \sigma \in S$ , if  $\theta \in \text{solns}(C \wedge D \wedge \text{ocond}(\sigma))$ , then  $f'(\sigma(\theta)) < f'(\theta)$ , we can add all of the dominance breaking constraints  $db(\sigma) \equiv \neg(\text{scond}(\sigma) \wedge \text{ocond}(\sigma))$  to  $P$  without changing its satisfiability or optimal value.

A proof of this theorem and more details on the method can be found in [3].

*Example 1.* Consider the 0-1 knapsack problem where  $x_i$  are 0-1 variables, we have constraint  $\sum w_i x_i \leq W$  and we have objective  $f = -\sum v_i x_i$ , where  $w_i$  and  $v_i$  are constants.

*Step 1* Initially, let us not refine the objective function leaving  $f' = f$ .

*Step 2* Consider mappings which swap the values of two variables, i.e.,  $\forall i < j, \sigma_{i,j}$  swaps  $x_i$  and  $x_j$ .

*Step 3* A sufficient condition for  $\sigma_{i,j}$  to map the current solution to another solution is:  $\text{scond}(\sigma_{i,j}) \equiv w_i x_j + w_j x_i \leq w_i x_i + w_j x_j$ . Rearranging, we get:  $(w_i - w_j)(x_i - x_j) \geq 0$ .

*Step 4* A sufficient condition for  $\sigma_{i,j}$  to map the current solution to an assignment with a better objective value is:  $\text{ocond}(\sigma_{i,j}) \equiv v_i x_j + v_j x_i > v_i x_i + v_j x_j$ . Rearranging, we get:  $(v_i - v_j)(x_i - x_j) < 0$ .

*Step 5* For each  $\sigma_{i,j}$ , we can post the dominance breaking constraint:  $db(\sigma_{i,j}) \equiv \neg(\text{scond}(\sigma_{i,j}) \wedge \text{ocond}(\sigma_{i,j}))$ . After simplifying, we have  $db(\sigma_{i,j}) \equiv x_i \leq x_j$  if  $w_i \geq w_j$  and  $v_i < v_j$ ,  $db(\sigma_{i,j}) \equiv x_i \geq x_j$  if  $w_i \leq w_j$  and  $v_i > v_j$ , and  $db(\sigma_{i,j}) \equiv \text{true}$  for all other cases.

These dominance breaking constraints ensure that if one item has worse value and greater or equal weight to another, then it cannot be chosen without choosing the other also.

We can refine the objective to get stronger dominance breaking constraints. In Step 1, we can tie break solutions with equal objective value by the weight used, and then lexicographically, i.e.,  $f' = \text{lex}(f, \sum w_i x_i, x_1, \dots, x_n)$ . In Step 4, we have:  $\forall i < j, \text{ocond}(\sigma_{i,j}) \equiv \sigma(f') < f' \equiv ((v_i - v_j)(x_i - x_j) < 0) \vee ((v_i - v_j)(x_i - x_j) = 0 \wedge (w_i - w_j)(x_i - x_j) > 0) \vee ((v_i - v_j)(x_i - x_j) = 0 \wedge (w_i - w_j)(x_i - x_j) = 0 \wedge x_j < x_i)$ . In Step 5, after simplifying, in addition to the dominance breaking constraints we had before, we would also have:  $db(\sigma_{i,j}) \equiv x_i \leq x_j$  if  $w_i > w_j$  and  $v_i = v_j$ ,  $db(\sigma_{i,j}) \equiv x_i \geq x_j$  if  $w_i < w_j$  and  $v_i = v_j$ , and  $db(\sigma_{i,j}) \equiv x_i \leq x_j$  if  $w_i = w_j$  and  $v_i = v_j$  which is a symmetry breaking constraint.  $\square$

### 3 Dominance Jumping

A propagator for a dominance breaking constraint can do two things: 1) it can check the consistency of the current domain w.r.t. the dominance breaking constraint, producing failure if it is inconsistent, and 2) it can prune off variable/value pairs which, if taken, will cause inconsistency. In the original method, the failure and propagation produced by these propagators are treated the same as any other propagator in the system. In our altered method, whenever dominance jumping is active, we modify this behavior as follows: 1) we check consistency only and never prune any values using the dominance breaking constraints, 2) when a failure is detected, we perform a *dominance jump* rather than a normal backtrack.

As can be seen from the definitions in Section 2, each dominance breaking constraint  $db(\sigma)$  is generated from an assignment mapping  $\sigma$ . When a domain  $D$  is failed by  $db(\sigma)$ , it means that every solution  $\theta$  in  $D$  is dominated by a corresponding solution  $\sigma(\theta)$ . Rather than simply failing and backtracking, we can instead perform a *dominance jump* to get to the part of the search tree that contains these better solutions. Let us extend  $\sigma$  to also map domains to domains via  $solns(\sigma(D)) \equiv \{\sigma(\theta) \mid \theta \in solns(D)\}$ . Then, if  $D$  is failed by  $db(\sigma)$ , the domain  $\sigma(D)$  must contain solutions which dominate those in the current domain  $D$ . We want to calculate this new domain  $\sigma(D)$  and jump to there. In this paper, we consider only  $\sigma$  which are literal mappings, i.e., assignment mappings which map each equality literal  $x = v$  to the same or another equality literal  $x' = v'$  in all assignments. All of the  $\sigma$  used in [3] are literal mappings, and indeed we expect that most practically useful mappings for the method proposed in [3] will be literal mappings. Let us extend  $\sigma$  to map equality literals to equality literals and disequality literals to disequality literals such that if  $\sigma(x = v) = (x' = v')$ , then  $\sigma(x \neq v) = (x' \neq v')$ .

Any domain  $D$  can be expressed as a set of disequality literals  $lits_D$  representing which variable/value pairs from the original domain has been pruned. For example, suppose the initial domain  $D_{init}$  of  $x_1, x_2$  were  $\{1, 2, 3, 4, 5\}$  and the current domain  $D$  is  $x_1 \in \{1, 3, 5\}, x_2 \in \{2, 3, 4\}$ . Then  $lits_D \equiv \{x_1 \neq 2, x_1 \neq 4, x_2 \neq 1, x_2 \neq 5\}$ . Using the set of literals  $\sigma(lits_D) \equiv \{\sigma(l) \mid l \in lits_D\}$  as decisions from the root node will take us to the search space  $\sigma(D)$ . We do this by backtracking to the deepest level such that all previous decisions in the current search path are in  $\sigma(lits_D)$ . We then suspend the normal search strategy and draw decisions from  $\sigma(lits_D)$  until it is either exhausted, or some conflict occurs. After that, we resume the normal search strategy. If  $D$  had variables which were fixed, we can use those equality literals in  $lits_D$  instead so we need to make fewer decisions to get to  $\sigma(D)$ . Similarly, if  $\sigma$  happens to also map inequality literals to inequality literals (e.g., in a mapping which swaps variables), we can use those to reduce the number of decisions.

*Example 2.* Consider the Photo problem. A group of people wants to take a group photo where they stand in one line. Each person has preferences regarding who they want to stand next to. We want to find the arrangement which satisfies the most preferences.

We can model this as follows. Let  $x_i \in \{1, \dots, n\}$  for  $i = 1, \dots, n$  be variables where  $x_i$  represent the person in the  $i$ th place. Let  $p$  be a 2d integer array where  $p[i][j] = p[j][i] = 2$  if person  $i$  and  $j$  both want to stand next to each other,  $p[i][j] = p[j][i] = 1$  if only one of them wants to stand next to the other, and  $p[i][j] = p[j][i] = 0$  if neither want to stand next to each other. The only constraint is:  $alldiff([x_1, \dots, x_n])$ . The objective function to be maximised is given by:  $f = \sum_{i=1}^{n-1} p[x_i][x_{i+1}]$ . In MiniZinc [18] it would be modelled as:

```
int: n;                                     % number of people
set of int: Person = 1..n;
array[Person, Person] of 0..2: p;          % preferences

array[Person] of Person: x;                % person in position i;

constraint alldifferent(x);

solve maximize sum(i in 1..n-1)(p[x[i], x[i+1]]);
```

As described in [3], if we consider the mappings  $\sigma_{i,j}$  which flip a subsequence of the  $x$ 's from the  $i$ th position to the  $j$ th (i.e., map  $x_i$  to  $x_j$ ,  $x_{i+1}$  to  $x_{j-1}$ , ...,  $x_j$  to  $x_i$ ), we can generate a number of dominance breaking constraints. For  $2 \geq i < j \leq n-1$ ,  $db(\sigma_{i,j}) \equiv p[x_{i-1}][x_i] + p[x_j][x_{j+1}] + (x_i < x_j) > p[x_{i-1}][x_j] + p[x_i][x_{j+1}]$ . For  $i = 1, 2 \leq j$ ,  $db(\sigma_{i,j}) \equiv p[x_j][x_{j+1}] + (x_i < x_j) > p[x_i][x_{j+1}]$ . For  $i \leq n-1, j = n$ ,  $db(\sigma_{i,j}) \equiv p[x_{i-1}][x_i] + (x_i < x_j) > p[x_{i-1}][x_j]$ . For  $i = 1, j = n$ ,  $db(\sigma_{i,j}) \equiv (x_i < x_j) > 0$ .

We now illustrate the difference between dominance breaking and dominance jumping on a simple example. Suppose  $n = 6$  and person 1 wants to stand next to person 6, person 6 wants to stand next to person 2, person 2 wants to stand next to person 5, person 5 wants to stand next to person 3, and person 3 wants to stand next to person 4. This is expressed by the MiniZinc data file:

```
n = 6;
p = [ | 0, 0, 0, 0, 0, 1
      | 0, 0, 0, 0, 1, 1
      | 0, 0, 0, 1, 1, 0
      | 0, 0, 1, 0, 0, 0
      | 0, 1, 1, 0, 0, 0
      | 1, 1, 0, 0, 0, 0 | ];
```

Suppose we use a naive search strategy such as labelling the  $x_i$  in order, trying the lowest value available in the domain first. With neither dominance breaking nor dominance jumping, it takes the search 51 conflicts to reach an optimal solution  $x_1 = 1, x_2 = 6, x_3 = 2, x_4 = 5, x_5 = 3, x_6 = 4$ . With dominance breaking, the search proceeds as follows. At the first decision level, we try  $x_1 = 1$ . At the second decision level, we try  $x_2 = 2$ . At this point, the constraint  $p[x_1][x_2] + (x_2 < x_6) > p[x_1][x_6]$  propagates to force  $p[x_1][x_6] = 0$ , which forces  $x_6 \neq 6$ . At the third decision level, we try  $x_3 = 3$ . At this point, the constraint  $p[x_2][x_3] + (x_3 < x_6) > p[x_2][x_6]$  propagates to force  $p[x_2][x_6] = 0$ , which forces  $x_6 \neq 5$ , which forces  $x_6 = 4$ . The constraint  $p[x_3][x_4] + (x_4 < x_6) > p[x_3][x_6]$  now propagates and forces  $p[x_3][x_4] \geq 2$  because  $p[x_3][x_6] = p[3][4] = 1$ , and  $x_4$  is either 5 or 6 so  $(x_4 < x_6) = 0$ . But then  $x_4 \neq 5$  and  $x_4 \neq 6$  and we have a failure. We then backtrack and continue the search. After another 25 conflicts, we reach the optimal solution. The search tree is shown in Figure 1.

With dominance jumping, the search proceeds as follows. We make the 5 decisions  $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 5$ , which forces  $x_6 = 6$ . At this point, a number of dominance breaking constraints are detected to be violated, telling us that certain mappings can improve the solution. For example,  $db(\sigma_{2,6})$  is violated. Using the mapping  $\sigma_{2,6}$  to perform a jump means that we backtrack to decision level 1, and then try the decisions  $x_2 = 6, x_3 = 5, x_4 = 4, x_5 = 3, x_6 = 2$ . After these, we again detect that a dominance breaking constraint is violated, e.g.,  $db(\sigma_{3,6})$ . Performing this jump means that we backtrack to decision level 2 and try  $x_3 = 2, x_4 = 3, x_5 = 4, x_6 = 5$ . This violates  $db(\sigma_{4,6})$ . Applying this jump causes us to backtrack to decision level 3 and try  $x_4 = 5, x_5 = 4, x_6 = 3$ . This violates  $db(\sigma_{5,6})$ . Applying this jump causes us to backtrack to decision level 4 and try  $x_5 = 3, x_6 = 4$ , finally giving a non-dominated solution of  $x_1 = 1, x_2 = 6, x_3 = 2, x_4 = 5, x_5 = 3, x_6 = 4$ . In this case, it only took 4 conflicts and 4 jumps to bring us to the optimal solution. The search tree is shown in Figure 2.  $\square$



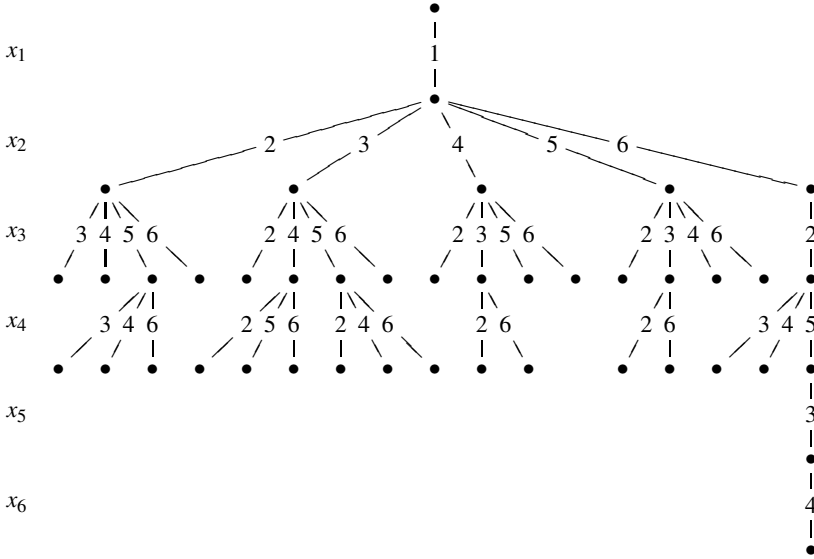


**Fig. 1.** Search tree with dominance breaking

The effect of dominance jumping is significantly different from dominance breaking. When a “bad” decision is made (e.g.,  $x_2 = 2$  after  $x_1 = 1$  in Example 2), dominance breaking is incapable of “fixing” the problem. Instead, it just helps the solver to fail that bad subtree quicker so that it can backtrack out of the bad decision. However, in general, it still takes exponential time to undo the bad decision. Dominance jumping on the other hand, can potentially fix a bad decision and replace it with a good one very quickly. In Example 2, after  $x_1 = 1, x_2 = 2$ , if some  $x_i$  is set to 6, it is likely that flipping the subsequence from 2 to  $i$  improves the objective. Thus dominance jumping will jump to a subtree where  $x_1 = 1, x_2 = 6$ , immediately undoing the bad decision  $x_2 = 2$ .

Note that dominance jumping does not require all variables involved in the dominance breaking constraint to be fixed in order to jump.

*Example 3.* Consider a simple problem:  $x_1 + x_2 + x_3 + x_4 \leq 9 \wedge alldiff(x_1, x_2, x_3, x_4)$  with  $D(x_i) = [1..6]$ . All variables are symmetric. So  $\sigma_{i,j}$  which swaps the values of  $x_i$  and  $x_j$  is a mapping that preserves solutions. Using a lexicographic objective  $f'$  we can compute  $db(\sigma_{i,j}) = x_i \leq x_j$  for  $i < j$ . Imagine we label  $x_2 = 1$ , then propagation causes  $D(x_1) = [2..6]$  and  $db(\sigma_{1,2})$  fails. We compute  $\sigma_{1,2}(D)$  as  $D(x_2) = [2..6]$  and  $D(x_1) = \{1\}$ . The dominance jump goes to the root and then sets  $x_1 = 1$  (which will set  $D(x_2) = [2..6]$ ) and then continues its search. Of course for this trivial example dominance breaking is clearly superior.  $\square$

Both dominance breaking and dominance jumping are optimality preserving. As Theorem 1 states, the addition of dominance breaking constraints to the problem does not change its satisfiability or optimal value. Performing dominance jumping will not

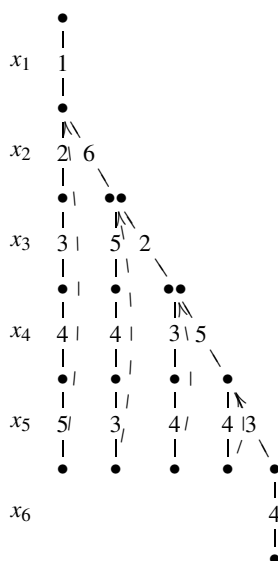


**Fig. 2.** Search tree with dominance jumping

change its satisfiability or optimal value either. When a dominance jump is performed, nothing is actually pruned. We are only doing a restart and temporarily using a different search strategy to guide the solver to another part of the search space. The subtree we jumped away from is not counted as fully searched, so dominance jumping can never prune off optimal solutions of the problem. However, there can be issues of termination when dominance jumping is used, as if we keep jumping, we may never complete a full search of the search tree. Nogood learning techniques such as [13,19,6]) can be used to overcome this problem. Such techniques record nogoods which tell us which parts of the search tree the solver has proved contains no solution better than the current best solution. Such nogoods allow the solver to keep track of which subtrees have been exhaustively searched. If we keep all such nogoods permanently, the search will always terminate and will be complete, guaranteeing that the correct optimal value is found. We implemented dominance jumping in the nogood learning solver *CHUFFED* which already has nogood learning built in, and thus it is capable of performing a complete search with dominance jumping.

The effectiveness of dominance jumping depends significantly on how good the original search strategy was. If the original search strategy was already good enough that it very quickly leads the search to an optimal solution, then dominance jumping is largely pointless. On the other hand, if the original search strategy was quite naive, or is not designed for finding good solutions, then dominance jumping can be very useful. It is quite common that the search strategies used in CP solvers are not good at finding good solutions. This is because many of them are designed to reduce the size of the search tree, rather than to order the subtrees so that good solutions are found first. Many of the common dynamic search strategies such as first fail [12], dom/wdeg and its variants [2], and activity based search [17] are purely designed to make the search

tree smaller and makes no effort whatsoever to try to guide the solver to good solutions quickly. But when dominance jumping is used on top of them, even very naive search strategies which are bad at finding solutions can turn into good ones, as a significant amount of information about the objective and the structure of the problem is contained in the dominance relations. By exploiting this through dominance jumping, we can bring the solver to a good subtree even if the search strategy initially sent it to a bad one.

While dominance jumping can be effective in the solution finding phase of an optimization problem, it is completely useless in the proof of optimality phase. In that phase, we are no longer trying to find any solutions. Jumping around in the search space will simply make it more difficult to complete the proof of optimality. Nogood learning can prevent thrashing behavior caused by the jumping and allow a complete proof of optimality to be derived. However, for large/hard problems, this may require an unreasonably large number of nogoods to be stored, causing the solver to run out of memory. Ideally, if a proof of optimality is desired, we should use some dynamic method to switch off dominance jumping and go back to pure dominance breaking. Such strategies will be explored in future work.

## 4 Experiments

The experiments were performed on Xeon Pro 2.4GHz processors using the state of the art CP solver CHUFFED. We use a time out of 600 seconds. We compare the base solver vs dominance breaking and dominance jumping. We use the 7 optimization problems used in [3]: *Photo- $n$*  the photo problem of Example 2 with  $n$  people, *Knapsack- $n$*  0-1 knapsack problem of Example 1 with  $n$  objects, *Nurse- $n$*  nurse scheduling problem [15] with 15 nurses and  $n$  days, *RCPSP* resource constrained scheduling problem J120 instances, *Talent- $n$*  talent scheduling problem [9] with  $n$  scenes, *SteelMill- $n$*  steel mill scheduling [10] with  $n$  orders, and *PCBoard- $n-m$*  PB board manufacturing problem with  $n$  components and  $m$  devices. We use basic in order fixed search strategies that are not specifically designed to find good solutions. MiniZinc models and data for the problems can be found at: [www.cs.mu.oz.au/~pjs/dom-jump/](http://www.cs.mu.oz.au/~pjs/dom-jump/)

Results are shown in Table 1: *opt* is average of the best solution found; *otime* is the geometric mean of time to find the best solution; *etime* is the geometric mean of time to find a solution at least as good as the worst of the best solution found by dominance breaking and the best solution found by dominance jumping, so we can directly compare *etime* to see how much time each took to get the same quality solution; *stime* is the geometric mean of the time to solve the instance completely (timeouts count as 600); and finally *svd* the number of instances solved to optimality is given in brackets. The best values out of the three methods are given in bold. When there is a tie on the best value, we tie-break on the time required to achieve the value.

The results show that both dominance breaking and dominance jumping substantially improve upon solving without dominance information. Dominance jumping is clearly better at finding good solutions faster. The average best solution found is almost always better. Dominance jumping almost always wins in *etime*, the only exception is in smaller knapsacks and in nurse scheduling where dominance breaking is obviously far superior. Notice how, as the difficulty of the instances grows with size, dominance jumping becomes more advantageous.

**Table 1.** Comparison of the solver with nothing (none), with dominance breaking constraints (dominance breaking), and with dominance jumping.

| Problem      | none    |       |       |     | dominance breaking |       |              |              | dominance jumping |                |       |              |              |           |
|--------------|---------|-------|-------|-----|--------------------|-------|--------------|--------------|-------------------|----------------|-------|--------------|--------------|-----------|
|              | opt     | etime | time  | svd | opt                | etime | time         | svd          | opt               | etime          | time  | svd          |              |           |
| Photo-16     | 19.5    | 16.43 | 50.13 | 18  | 19.7               | 1.03  | 1.03         | <b>11.40</b> | <b>20</b>         | <b>19.7</b>    | 0.10  | <b>0.10</b>  | 16.64        | 20        |
| Photo-18     | 21.4    | 39.45 | 182.7 | 15  | 21.5               | 2.27  | 2.27         | <b>76.26</b> | <b>20</b>         | <b>21.5</b>    | 0.29  | <b>0.29</b>  | 80.62        | 19        |
| Photo-20     | 21.4    | 179.6 | 393.3 | 5   | 23.15              | 31.06 | 31.06        | 262.4        | 7                 | <b>23.15</b>   | 1.16  | <b>1.16</b>  | <b>232.9</b> | <b>9</b>  |
| Photo-22     | 22.8    | 185.1 | 368.7 | 4   | 25.15              | 51.56 | 38.9         | 294.7        | 6                 | <b>25.4</b>    | 2.18  | <b>0.76</b>  | <b>244.4</b> | <b>7</b>  |
| Photo-24     | 21.55   | 213.5 | 596.5 | 1   | 26.75              | 147.9 | 147.9        | 586.7        | 3                 | <b>27.15</b>   | 2.00  | <b>0.52</b>  | <b>495.6</b> | <b>4</b>  |
| Knapsack-100 | 1827.95 | 222.3 | 600   | 0   | <b>2583.2</b>      | 0.30  | <b>0.30</b>  | <b>0.93</b>  | <b>20</b>         | 2583.2         | 0.60  | 0.60         | 1.52         | 20        |
| Knapsack-150 | 3605.75 | 240.6 | 600   | 0   | <b>5810.35</b>     | 13.8  | <b>13.82</b> | <b>47.62</b> | <b>19</b>         | 5810.35        | 27.71 | 27.71        | 82.26        | 19        |
| Knapsack-200 | 5910.55 | 299.2 | 600   | 0   | <b>10422.4</b>     | 180.1 | <b>126.2</b> | <b>261.0</b> | <b>3</b>          | 10415.8        | 220.7 | 206.2        | 491.9        | 2         |
| Knapsack-250 | 8732.25 | 342.4 | 600   | 0   | 16235.25           | 253.9 | 186.2        | 600          | 0                 | <b>16235.5</b> | 327.1 | <b>151.7</b> | 600          | 0         |
| Knapsack-300 | 12000   | 288.9 | 600   | 0   | 23212              | 272.6 | 157.2        | 600          | 0                 | <b>23294.9</b> | 302.2 | <b>123.0</b> | 600          | 0         |
| Nurse-14     | 149.2   | 61.35 | 61.82 | 18  | <b>151.35</b>      | 50.99 | <b>49.14</b> | <b>52.33</b> | <b>19</b>         | 150.2          | 72.45 | 72.45        | 74.06        | 18        |
| Nurse-21     | 137.1   | 416.5 | 600   | 0   | <b>172.7</b>       | 248.0 | <b>217.6</b> | 600          | 0                 | 172.4          | 280.7 | 254.3        | 600          | 0         |
| Nurse-28     | 161.5   | 400.5 | 600   | 0   | 222.4              | 245.1 | <b>213.5</b> | 600          | 0                 | <b>222.45</b>  | 310.2 | 277.6        | 600          | 0         |
| Nurse-35     | 187.5   | 355.0 | 600   | 0   | <b>275.75</b>      | 339.8 | <b>164.7</b> | 600          | 0                 | 275.1          | 223.7 | 196.4        | 600          | 0         |
| Nurse-42     | 213.1   | 382.3 | 600   | 0   | <b>321.9</b>       | 377.4 | <b>193.3</b> | 600          | 0                 | 320.55         | 332.9 | 332.9        | 600          | 0         |
| RCPSP        | 110.9   | 31.37 | 41.62 | 72  | 114.17             | 7.51  | 7.51         | 13.97        | 57                | <b>110.86</b>  | 12.44 | <b>3.92</b>  | <b>15.83</b> | <b>72</b> |
| Talent-16    | 106.1   | 7.58  | 19.03 | 20  | 106.05             | 0.92  | 0.92         | 3.25         | 20                | <b>106.05</b>  | 0.41  | <b>0.41</b>  | <b>2.79</b>  | <b>20</b> |
| Talent-18    | 149.45  | 127.1 | 239.3 | 16  | 147                | 5.22  | 5.22         | 17.26        | 20                | <b>147</b>     | 2.73  | <b>2.73</b>  | <b>15.81</b> | <b>20</b> |
| Talent-20    | 270.2   | 321.4 | 497.3 | 5   | 184.45             | 27.22 | 27.22        | 81.48        | 20                | <b>184.45</b>  | 14.85 | <b>14.85</b> | <b>73.40</b> | <b>20</b> |
| Talent-22    | 387.1   | 369.2 | 600   | 0   | 270.9              | 34.32 | 34.32        | 353.2        | 13                | <b>204.05</b>  | 51.4  | <b>21.10</b> | <b>310.5</b> | <b>14</b> |
| Talent-24    | 566.65  | 323.8 | 600   | 0   | 322.25             | 161.6 | 154.9        | 547.1        | 2                 | <b>260.1</b>   | 123.4 | <b>18.27</b> | <b>510.7</b> | <b>3</b>  |
| SteelMill-40 | 5.45    | 71.02 | 129.3 | 10  | 1.6                | 42.30 | 40.80        | 54.60        | 15                | <b>0.65</b>    | 20.24 | <b>9.33</b>  | <b>21.75</b> | <b>17</b> |
| SteelMill-45 | 8.2     | 105.7 | 269.6 | 6   | 1.55               | 121.7 | 121.7        | 134.1        | 14                | <b>0.35</b>    | 46.44 | <b>31.92</b> | <b>52.83</b> | <b>18</b> |
| SteelMill-50 | 16.35   | 319.6 | 560.1 | 1   | 9.85               | 91.00 | 90.73        | 332.1        | 10                | <b>1.1</b>     | 142.1 | <b>30.75</b> | <b>198.9</b> | <b>16</b> |
| SteelMill-55 | 25.95   | 305.0 | 600   | 0   | 16.05              | 67.26 | 60.19        | 419.3        | 6                 | <b>2.9</b>     | 212.2 | <b>23.59</b> | <b>275.6</b> | <b>13</b> |
| SteelMill-60 | 32.55   | 274.4 | 584.8 | 1   | 19.9               | 59.56 | 54.65        | 497.0        | 2                 | <b>5</b>       | 224.7 | <b>23.82</b> | <b>399.0</b> | <b>8</b>  |
| PCBoard-6-8  | 206.25  | 298.8 | 341.1 | 8   | 217.7              | 17.50 | 17.50        | <b>24.36</b> | <b>20</b>         | <b>217.7</b>   | 1.29  | <b>1.29</b>  | 68.19        | 14        |
| PCBoard-6-9  | 225.94  | 440.2 | 532.8 | 2   | <b>246.7</b>       | 84.81 | 80.04        | <b>142.5</b> | <b>20</b>         | 246.6          | 8.39  | <b>8.39</b>  | 438.5        | 3         |
| PCBoard-7-9  | 242.15  | 418.6 | 600   | 0   | 277.7              | 123.5 | 111.4        | <b>498.5</b> | <b>7</b>          | <b>282.5</b>   | 21.95 | <b>5.56</b>  | 600          | 0         |
| PCBoard-7-10 | 266.9   | 383.7 | 600   | 0   | 282.55             | 6.32  | 6.32         | 600          | 0                 | <b>319.15</b>  | 28.48 | <b>0.86</b>  | 600          | 0         |
| PCBoard-8-10 | 293.9   | 379.2 | 600   | 0   | 308.4              | 4.35  | 4.35         | <b>597</b>   | <b>1</b>          | <b>357.9</b>   | 30.32 | <b>0.38</b>  | 600          | 0         |

Unsurprisingly dominance jumping is usually better at proving optimality having better *stime* in most of the smaller instances. Surprisingly dominance jumping actually turns out to be preferable to dominance breaking even for proving optimality on RCPSP, Talent and SteelMill. For these problems the proof of optimality is not the larger part of the search space, that is once we find the optimal solution for these problems it is often not too difficult to prove it optimal.

Next we compare the three methods using 3 different search heuristics for the Photo problem, to see how the search strategy affects the effectiveness of dominance breaking and dominance jumping. The first is the basic inorder fixed search used above (inorder). The second greedily finds a person who most wants to stand next to an already assigned person and assigns them next to them (greedy). The third uses the first fail heuristic to pick which variable to assign next (first-fail).

The results in Table 2 show that dominance jumping is still much better at reaching a good solution than dominance breaking, regardless of the search strategy. The results clearly illustrate that the biggest advantage of dominance jumping arises in the inorder search, which does not try to look for good solutions. But dominance jumping is still advantageous over dominance breaking using the greedy search, although to a lesser degree. Using first-fail dominance breaking is better at proving optimality, since first-fail search also concentrates on reducing the search space, but as the size of the problem grows, its advantage reduces, until for Photo-24 dominance jumping is superior in proving optimality as well.

**Table 2.** Comparison of dominance breaking and dominance jumping given different search heuristics

| Problem  | Search     | none       |              |              |            | dominance breaking |              |              |              | dominance jumping |            |              |              |              |            |
|----------|------------|------------|--------------|--------------|------------|--------------------|--------------|--------------|--------------|-------------------|------------|--------------|--------------|--------------|------------|
|          |            | <i>opt</i> | <i>otime</i> | <i>stime</i> | <i>svd</i> | <i>opt</i>         | <i>otime</i> | <i>etime</i> | <i>stime</i> | <i>svd</i>        | <i>opt</i> | <i>otime</i> | <i>etime</i> | <i>stime</i> | <i>svd</i> |
| Photo-16 | inorder    | 19.65      | 7.37         | 11.47        | 18         | 19.70              | 0.79         | 0.79         | 2.35         | 20                | 19.70      | 0.08         | 0.08         | 0.98         | 20         |
|          | greedy     | 19.70      | 1.29         | 3.44         | 19         | 19.70              | 0.45         | 0.45         | 1.71         | 20                | 19.70      | 0.04         | 0.04         | 0.56         | 20         |
|          | first-fail | 19.70      | 0.72         | 1.22         | 20         | 19.70              | 0.24         | 0.24         | 0.65         | 20                | 19.70      | 0.13         | 0.13         | 0.76         | 20         |
| Photo-18 | inorder    | 21.45      | 18.47        | 45.25        | 19         | 21.50              | 1.42         | 1.42         | 9.71         | 20                | 21.50      | 0.20         | 0.20         | 5.60         | 20         |
|          | greedy     | 21.50      | 0.69         | 7.80         | 19         | 21.50              | 0.39         | 0.39         | 4.09         | 20                | 21.50      | 0.08         | 0.08         | 2.72         | 20         |
|          | first-fail | 21.50      | 1.05         | 2.79         | 20         | 21.50              | 0.28         | 0.28         | 0.97         | 20                | 21.50      | 0.24         | 0.24         | 1.62         | 20         |
| Photo-20 | inorder    | 22.50      | 161.35       | 228.98       | 10         | 23.20              | 16.74        | 16.74        | 54.97        | 16                | 23.20      | 1.23         | 1.23         | 15.92        | 16         |
|          | greedy     | 23.00      | 5.36         | 38.38        | 12         | 23.20              | 3.26         | 3.26         | 19.01        | 18                | 23.20      | 0.10         | 0.10         | 3.29         | 18         |
|          | first-fail | 23.20      | 4.90         | 12.95        | 17         | 23.20              | 1.17         | 1.17         | 3.45         | 20                | 23.20      | 0.54         | 0.54         | 3.46         | 20         |
| Photo-22 | inorder    | 23.70      | 137.69       | 254.99       | 55         | 25.35              | 44.46        | 44.46        | 102.18       | 13                | 25.40      | 2.09         | 1.57         | 12.27        | 14         |
|          | greedy     | 25.15      | 8.28         | 60.94        | 11         | 25.40              | 9.54         | 9.54         | 36.68        | 15                | 25.45      | 0.60         | 0.60         | 5.38         | 15         |
|          | first-fail | 25.45      | 13.94        | 19.71        | 16         | 25.50              | 3.36         | 3.36         | 6.42         | 20                | 25.50      | 2.33         | 2.33         | 6.55         | 20         |
| Photo-24 | inorder    | 22.45      | 216.79       | 519.93       | 3          | 26.85              | 87.71        | 87.71        | 350.3        | 7                 | 27.10      | 1.36         | 0.71         | 61.88        | 8          |
|          | greedy     | 26.60      | 11.75        | 184.26       | 7          | 26.95              | 16.52        | 16.52        | 171.90       | 8                 | 27.35      | 0.66         | 0.23         | 27.71        | 12         |
|          | first-fail | 26.55      | 43.17        | 88.59        | 12         | 27.45              | 13.10        | 13.10        | 33.29        | 19                | 27.45      | 5.12         | 5.12         | 20.97        | 19         |

## 5 Related Work

Dominance breaking constraints were introduced only recently in [3] and there has been little work analyzing how they may conflict with the search or how that problem can be overcome. The closest related work is that for the special case of symmetry breaking. In this case, potential conflicts between the search and static symmetry breaking constraints can be overcome by using dynamic symmetry breaking techniques such as SBDS [1,11] or SBDD [5]. However, neither of these methods obviously generalize to the dominance case. In the case of symmetry, we have sets of equally good symmetric subtrees. The policy in SBDS/SBDD is to search the first member of each such set encountered during search, and to prune all other members as soon as they are encountered. In dominance breaking however, we have pairs of subtrees where one may be strictly better than the other (i.e., contains a strictly better solution). The ordering between them is not up to us to decide as it is determined by the search strategy, and we cannot simply decide to always search the first of the pair and prune the second. We could try a different policy such as: if we encounter the good one first, we prune the bad one later, and if we encounter the bad one first, we search both. Indeed, such a policy was proposed in [7]. However, checking whether a subtree is dominated by any of the previously searched subtrees is extremely complex in general, and is much harder than simply determining whether it is dominated by some (possibly not yet explored) subtree. In [7], it is shown how an incomplete version of such a dominance check can be performed using greedy local search for the Travelling Salesman Problem. However, it is not clear how complete it is or whether it can generalise to other problems. Also, even if the dominance check can be performed efficiently, such a method will still perform more search in general than the dominance jumping method presented here. In dominance jumping, regardless of the order in which we encounter the pair of subtrees, we will only search the good one and will always skip the bad one, because if we encounter the bad one first, we will simply immediately jump to the good one.

Dominance jumping shares some features with local search/repair methods such as min-conflict search [16]. However, such methods typically travel through the space of infeasible solutions and jump at every node. Dominance jumping on the other hand is

a systematic search which remains within feasible space, and only occasionally jumps when we are guaranteed to get to a better subtree.

Dominance jumping is also related to best first search. When best first search reaches a node which is recognized as possibly dominated (since the lower bound on the objective is substantially worse than another part of the search tree), it jumps to what it thinks is the best node and explores from there. In this case the jump is a heuristic, unlike in dominance jumping where we have a proof that the current node is suboptimal and we jump to a strictly better node.

## 6 Conclusion

We have developed a new method called dominance jumping to exploit the dominance relations identified by the method proposed in [3]. Rather than failing and backtracking as in the original method, we use the dominance relation to jump to a different part of the search tree that dominates the current subtree. Unlike static dominance breaking constraints, the new method will not conflict with the search strategy. Experimental evidence shows that the method allows good/optimal solutions to be found much more quickly on a wide range of problems. Important future work is to examine how to automatically determine when during search to switch from dominance jumping to dominance breaking, so that we can take advantage of the strengths of both approaches.

**Acknowledgments.** NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

## References

1. Backofen, R., Will, S.: Excluding symmetries in constraint-based search. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 73–87. Springer, Heidelberg (1999)
2. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: Proc. of ECAI 2004, pp. 146–150 (2004)
3. Chu, G., Stuckey, P.J.: A generic method for identifying and exploiting dominance relations. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 6–22. Springer, Heidelberg (2012)
4. Crawford, J.M., Ginsberg, M.L., Luks, E.M., Roy, A.: Symmetry-breaking predicates for search problems. In: Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning, pp. 148–159. Morgan Kaufmann (1996)
5. Fahle, T., Schamberger, S., Sellmann, M.: Symmetry breaking. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 93–107. Springer, Heidelberg (2001)
6. 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)
7. Focacci, F., Shaw, P.: Pruning sub-optimal search branches using local search. In: CPAIOR, vol. 2, pp. 181–189 (2002)
8. Frisch, A.M., Jefferson, C., Miguel, I.: Constraints for Breaking More Row and Column Symmetries. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 318–332. Springer, Heidelberg (2003)
9. de la Banda, M.G., Stuckey, P.J., Chu, G.: Solving talent scheduling with dynamic programming. *INFORMS Journal on Computing* 23(1), 120–137 (2011)
10. 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)

11. Gent, I., Smith, B.M.: Symmetry breaking in constraint programming. In: 14th European Conference on Artificial Intelligence, pp. 599–603 (2000)
12. Haralick, R.M., Elliott, G.L.: Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence* (1980)
13. Lecoutre, C., Sais, L., Tabary, S., Vidal, V.: Nogood Recording from Restarts. In: Veloso, M.M. (ed.) *Proc. of IJCAI 2007*, pp. 131–136 (2007)
14. Luks, E.M., Roy, A.: The Complexity of Symmetry-Breaking Formulas. *Ann. Math. Artif. Intell.* 41(1), 19–45 (2004)
15. Miller, H.E., Pierskalla, W.P., Rath, G.J.: Nurse scheduling using mathematical programming. *Operations Research*, 857–870 (1976)
16. Minton, S., Johnston, M.D., Philips, A.B., Laird, P.: Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence* 58(1), 161–205 (1992)
17. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: *Procs. of DAC 2001*, pp. 530–535 (2001)
18. 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)
19. 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)
20. Puget, J.F.: On the Satisfiability of Symmetrical Constrained Satisfaction Problems. In: Komorowski, J., Raś, Z.W. (eds.) *ISMIS 1993. LNCS*, vol. 689, pp. 350–361. Springer, Heidelberg (1993)
21. Puget, J.F.: Breaking symmetries in all different problems. In: Kaelbling, L.P., Saffiotti, A. (eds.) *Proc. of IJCAI 2005*, pp. 272–277. Professional Book Center (2005)

# Tractable Combinations of Global Constraints

David A. Cohen<sup>1</sup>, Peter G. Jeavons<sup>2</sup>,  
Evgenij Thorstensen<sup>2,\*</sup>, and Stanislav Živný<sup>3,\*\*</sup>

<sup>1</sup> Department of Computer Science, Royal Holloway, University of London, UK  
`d.cohen@rhul.ac.uk`

<sup>2</sup> Department of Computer Science, University of Oxford, UK  
`firstname.lastname@cs.ox.ac.uk`

<sup>3</sup> Department of Computer Science, University of Warwick, UK  
`s.zivny@warwick.ac.uk`

**Abstract.** We study the complexity of constraint satisfaction problems involving global constraints, i.e., special-purpose constraints provided by a solver and represented implicitly by a parametrised algorithm. Such constraints are widely used; indeed, they are one of the key reasons for the success of constraint programming in solving real-world problems.

Previous work has focused on the development of efficient propagators for individual constraints. In this paper, we identify a new tractable class of constraint problems involving global constraints of unbounded arity. To do so, we combine structural restrictions with the observation that some important types of global constraint do not distinguish between large classes of equivalent solutions.

## 1 Introduction

Constraint programming (CP) is widely used to solve a variety of practical problems such as planning and scheduling [23,30], and industrial configuration [1,22]. The theoretical properties of constraint problems, in particular the computational complexity of different types of problem, have been extensively studied and quite a lot is known about what restrictions on the general *constraint satisfaction problem* are sufficient to make it tractable [2,7,11,17,20,25].

However, much of this theoretical work has focused on problems where each constraint is represented *explicitly*, by a table of allowed assignments.

In practice, however, a lot of the success of CP is due to the use of special-purpose constraint types for which the software tools provide dedicated algorithms [28,16,31]. Such constraints are known as *global constraints* and are usually represented *implicitly* by an algorithm in the solver. This algorithm may take as a parameter a *description* that specifies exactly which kinds of assignments a particular instance of this constraint should allow.

Theoretical work on global constraints has to a large extent focused on developing efficient algorithms to achieve various kinds of local *consistency* for

---

\* Supported by EPSRC grant EP/G055114/1.

\*\* Supported by a Senior Research Fellowship from Warwick's DIMAP.



individual constraints. This is generally done by pruning from the domains of variables those values that cannot lead to a satisfying assignment [5,29]. Another strand of research has explored when it is possible to replace global constraints by collections of explicitly represented constraints [6]. These techniques allow faster implementations of algorithms for *individual constraints*, but do not shed much light on the complexity of problems with multiple *overlapping* global constraints, which is something that practical problems frequently require.

As an example, consider the following family of constraint problems involving clauses and cardinality constraints of unbounded arity.

*Example 1.* Consider a family of constraint problems on a set of Boolean variables  $\{x_1, x_2, \dots, x_{3n}\}$  (where  $n = 2, 3, 4, \dots$ ), with the following five constraints:

- $C_1$  is the binary clause  $x_1 \vee x_{2n+1}$ ;
- $C_2$  is a cardinality constraint on  $\{x_1, x_2, \dots, x_n\}$  specifying that exactly one of these variables takes the value 1;
- $C_3$  is a cardinality constraint on  $\{x_{2n+1}, x_{2n+2}, \dots, x_{3n}\}$  specifying that exactly one of these variables takes the value 1;
- $C_4$  is a cardinality constraint on  $\{x_2, x_3, \dots, x_{3n}\} - \{x_{2n+1}\}$  specifying that exactly  $n+1$  of these variables takes the value 1;
- $C_5$  is the clause  $\neg x_{n+1} \vee \neg x_{n+2} \vee \dots \vee \neg x_{2n}$ .

This problem is illustrated in Figure 1.

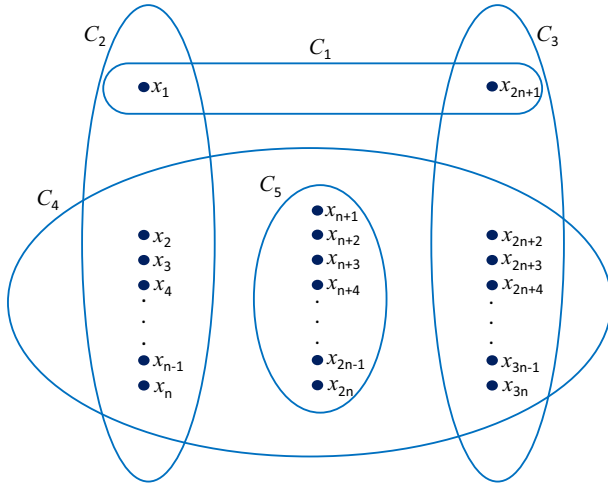


**Fig. 1.** The structure of the constraint problems in Example 1

This family of problems is not included in any previously known tractable class, but will be shown to be tractable using the results of this paper.

As discussed in [9], when the constraints in a family of problems have unbounded arity, the way that the constraints are *represented* can significantly affect the complexity. Previous work in this area has assumed that the global constraints have specific representations, such as propagators [19], negative constraints [10], or GDNF/decision diagrams [9], and exploited properties particular to that representation. In contrast, here we investigate the conditions that yield efficiently solvable classes of constraint problems with global constraints, without requiring any specific representation. Many global constraints have succinct representations, so even problems with very simple structures are known to be hard in some cases [24,29]. We will therefore need to impose some restrictions on the properties of the individual global constraints, as well as on the problem structure.

To obtain our results, we define a notion of equivalence on assignments and a new width measure that identifies variables that are constrained in exactly the same way. We then show that we can replace variables that are equated under our width measure with a single new variable whose domain represents the possible equivalence classes of assignments. Both of these simplification steps, merging variables and equating assignments, can be seen as techniques for eliminating symmetries in the original problem formulation. We describe some sufficient conditions under which these techniques provide a polynomial-time reduction to a known tractable case, and hence identify new tractable classes of constraint problems involving global constraints.

## 2 Global Constraints and Constraint Problems

In order to be more precise about the way in which global constraints are represented, we will extend the standard definition of a constraint problem.

**Definition 1 (Variables and assignments).** *Let  $V$  be a set of variables, each with an associated set of domain elements. We denote the set of domain elements (the domain) of a variable  $v$  by  $D(v)$ . We extend this notation to arbitrary subsets of variables,  $W$ , by setting  $D(W) = \bigcup_{v \in W} D(v)$ .*

*An assignment of a set of variables  $V$  is a function  $\theta : V \rightarrow D(V)$  that maps every  $v \in V$  to an element  $\theta(v) \in D(v)$ . We denote the restriction of  $\theta$  to a set of variables  $W \subseteq V$  by  $\theta|_W$ . We also allow the special assignment  $\perp$  of the empty set of variables. In particular, for every assignment  $\theta$ , we have  $\theta|_\emptyset = \perp$ .*

Global constraints have traditionally been defined, somewhat vaguely, as constraints without a fixed arity, possibly also with a compact representation of the constraint relation. For example, in [23] a global constraint is defined as “a constraint that captures a relation between a non-fixed number of variables”.

Below, we offer a precise definition similar to the one in [5], where the authors define global constraints for a domain  $D$  over a list of variables  $\sigma$  as being given intensionally by a function  $D^{|\sigma|} \rightarrow \{0, 1\}$  computable in polynomial time. Our definition differs from this one in that we separate the general *algorithm* of

a global constraint (which we call its *type*) from the specific description. This separation allows us a better way of measuring the size of a global constraint, which in turn helps us to establish new complexity results.

**Definition 2 (Global constraints).** A global constraint type is a parametrised polynomial-time algorithm that determines the acceptability of an assignment of a given set of variables.

Each global constraint type,  $e$ , has an associated set of descriptions,  $\Delta(e)$ . Each description  $\delta \in \Delta(e)$  specifies appropriate parameter values for the algorithm  $e$ . In particular, each  $\delta \in \Delta(e)$  specifies a set of variables, denoted by  $\text{vars}(\delta)$ .

A global constraint  $e[\delta]$ , where  $\delta \in \Delta(e)$ , is a function that maps assignments of  $\text{vars}(\delta)$  to the set  $\{0, 1\}$ . Each assignment that is allowed by  $e[\delta]$  is mapped to 1, and each disallowed assignment is mapped to 0. The extension or constraint relation of  $e[\delta]$  is the set of assignments,  $\theta$ , of  $\text{vars}(\delta)$  such that  $e[\delta](\theta) = 1$ . We also say that such assignments satisfy the constraint, while all other assignments falsify it.

When we are only interested in describing the set of assignments that satisfy a constraint, and not in the complexity of determining membership in this set, we will sometimes abuse notation by writing  $\theta \in e[\delta]$  to mean  $e[\delta](\theta) = 1$ .

As can be seen from the definition above, a global constraint is not usually explicitly represented by listing all the assignments that satisfy it. Instead, it is represented by some description  $\delta$  and some algorithm  $e$  that allows us to check whether the constraint relation of  $e[\delta]$  includes a given assignment. To stay within the complexity class NP, this algorithm is required to run in polynomial time. As the algorithms for many common global constraints are built into modern constraint solvers, we measure the *size* of a global constraint's representation by the size of its description.

*Example 2 (EGC).* A very general global constraint type is the *extended global cardinality* constraint type [26,29]. This form of global constraint is defined by specifying for every domain element  $a$  a finite set of natural numbers  $K(a)$ , called the cardinality set of  $a$ . The constraint requires that the number of variables which are assigned the value  $a$  is in the set  $K(a)$ , for each possible domain element  $a$ .

Using our notation, the description  $\delta$  of an EGC global constraint specifies a function  $K_\delta : D(\text{vars}(\delta)) \rightarrow \mathcal{P}(\mathbb{N})$  that maps each domain element to a set of natural numbers. The algorithm for the EGC constraint then maps an assignment  $\theta$  to 1 if and only if, for every domain element  $a \in D(\text{vars}(\delta))$ , we have that  $|\{v \in \text{vars}(\delta) \mid \theta(v) = a\}| \in K_\delta(a)$ .

The cardinality constraint  $C_2$  from Example 1 can be expressed as an EGC global constraint with description  $\delta$  such that  $K_\delta(1) = \{1\}$ , and  $K_\delta(0) = \{n-1\}$ .

*Example 3 (Clauses).* We can view the disjunctive clauses used to define propositional satisfiability problems as a global constraint type in the following way.

The description  $\delta$  of a clause is simply a list of the literals that it contains, and  $\text{vars}(\delta)$  is the corresponding set of variables. The algorithm for the clause then maps any Boolean assignment  $\theta$  of  $\text{vars}(\delta)$  that satisfies the disjunction of the literals specified by  $\delta$  to 1, and all other assignments to 0.

Note that a clause forbids precisely one assignment to  $\text{vars}(\delta)$  (the one that falsifies all of the literals in the clause). Hence the extension of a clause contains  $2^{|\text{vars}(\delta)|} - 1$  assignments, so the size of the constraint *relation* grows exponentially with the number of variables, but the size of the constraint *description* grows only linearly.

*Example 4 (Table and negative constraints).* A rather degenerate example of a global constraint type is the *table* constraint.

In this case the description  $\delta$  is simply a list of assignments of some fixed set of variables,  $\text{vars}(\delta)$ . The algorithm for a table constraint then decides, for any assignment of  $\text{vars}(\delta)$ , whether it is included in  $\delta$ . This can be done in a time which is linear in the size of  $\delta$  and so meets the polynomial time requirement.

*Negative* constraints are complementary to table constraints, in that they are described by listing *forbidden* assignments. The algorithm for a negative constraint  $e[\delta]$  decides, for any assignment of  $\text{vars}(\delta)$ , whether it is *not* included in  $\delta$ . Observe that the clauses described in Example 3 are a special case of the negative constraint type, as they have exactly one forbidden assignment.

We observe that any global constraint can be rewritten as a table or negative constraint. However, this rewriting will, in general, incur an exponential increase in the size of the description.

**Definition 3 (CSP instance).** *An instance of the constraint satisfaction problem (CSP) is a pair  $\langle V, C \rangle$  where  $V$  is a finite set of variables, and  $C$  is a set of global constraints such that for every  $e[\delta] \in C$ ,  $\text{vars}(\delta) \subseteq V$ . In a CSP instance, we call  $\text{vars}(\delta)$  the scope of the constraint  $e[\delta]$ .*

*A solution to a CSP instance  $\langle V, C \rangle$  is an assignment  $\theta$  of  $V$  which satisfies every global constraint, i.e., for every  $e[\delta] \in C$  we have  $\theta|_{\text{vars}(\delta)} \in e[\delta]$ .*

The general constraint satisfaction problem is clearly NP-complete, so in the remainder of the paper we shall look for more restricted versions of the problem that are *tractable*, that is, solvable in polynomial time.

### 3 Restricted Classes of Constraint Problems

First, we are going to consider restrictions on the way that the constraints in a given instance interact with each other, or, in other words, the way that the constraint scopes overlap; such restrictions are known as *structural* restrictions [11,17,20].

**Definition 4 (Hypergraph).** *A hypergraph  $\langle V, H \rangle$  is a set of vertices  $V$  together with a set of hyperedges  $H \subseteq \mathcal{P}(V)$ .*

*Given a CSP instance  $P = \langle V, C \rangle$ , the hypergraph of  $P$ , denoted  $\text{hyp}(P)$ , has vertex set  $V$  together with a hyperedge  $\text{vars}(\delta)$  for every  $e[\delta] \in C$ .*

One special class of hypergraphs that has received a great deal of attention is the class of *acyclic* hypergraphs [3]. This notion is a generalisation of the idea of tree-structure in a graph, and has been very important in the analysis of relational databases. A hypergraph is said to be acyclic if repeatedly removing all hyperedges contained in other hyperedges, and all vertices contained in only a single hyperedge, eventually deletes all vertices [3].

Solving a CSP instance  $P$  whose constraints are represented extensionally (i.e., as table constraints) is known to be tractable if the hypergraph of  $P$ ,  $\text{hyp}(P)$ , is acyclic [21]. Indeed, this has formed the basis for more general notions of “bounded cyclicity” [21] or “bounded hypertree width” [18], which have also been shown to imply tractability for problems with explicitly represented constraint relations. However, this is no longer true if the constraints are global, not even when we have a fixed, finite domain, as the following examples show.

*Example 5.* Any hypergraph containing only a single edge is clearly acyclic (and therefore has hypertree width one [18]), but the class of CSP instances consisting of a single EGC constraint over an unbounded domain is NP-complete [26].

*Example 6.* The NP-complete problem of 3-colourability [15] is to decide, given a graph  $\langle V, E \rangle$ , whether the vertices  $V$  can be coloured with three colours such that no two adjacent vertices have the same colour.

We may reduce this problem to a CSP with EGC constraints (cf. Example 2) as follows: Let  $V$  be the set of variables for our CSP instance, each with domain  $\{r, g, b\}$ . For every edge  $\langle v, w \rangle \in E$ , we post an EGC constraint with scope  $\{v, w\}$ , parametrised by the function  $K$  such that  $K(r) = K(g) = K(b) = \{0, 1\}$ . Finally, we make the hypergraph of this CSP instance acyclic by adding an EGC constraint with scope  $V$  parametrised by the function  $K'$  such that  $K'(r) = K'(g) = K'(b) = \{0, \dots, |V|\}$ . This reduction clearly takes polynomial time, and the hypergraph of the resulting instance is acyclic.

These examples indicate that when dealing with implicitly represented constraints we cannot hope for tractability using structural restrictions alone. We are therefore led to consider *hybrid* restrictions, which restrict both the nature of the constraints and the structure at the same time.

**Definition 5 (Constraint catalogue).** A constraint catalogue is a set of global constraints. A CSP instance  $\langle V, C \rangle$  is said to be over a constraint catalogue  $\mathcal{C}$  if for every  $e[\delta] \in C$  we have  $e[\delta] \in \mathcal{C}$ .

Previous work on the complexity of constraint problems has restricted the *extensions* of the constraints to a specified set of *relations*, known as a *constraint language* [7]. This is an appropriate form of restriction when all constraints are given explicitly, as table constraints. However, here we work with global constraints where the relations are often implicit, and this can significantly alter the complexity of the corresponding problem classes, as we will illustrate below. Hence we allow a more general form of restriction on the constraints by specifying a constraint catalogue containing all allowed constraints.

**Definition 6 (Restricted CSP class).** Let  $\mathcal{C}$  be a constraint catalogue, and let  $\mathcal{H}$  be a class of hypergraphs. We define  $\text{CSP}(\mathcal{H}, \mathcal{C})$  to be the class of CSP instances over  $\mathcal{C}$  whose hypergraphs are in  $\mathcal{H}$ .

Using Definition 6, we will restate an earlier structural tractability result, which will form the basis for our results in Section 5.

**Definition 7 (Treewidth).** A tree decomposition of a hypergraph  $\langle V, H \rangle$  is a pair  $\langle T, \lambda \rangle$  where  $T$  is a tree and  $\lambda$  is a labelling function from nodes of  $T$  to subsets of  $V$ , such that

1. for every  $v \in V$ , there exists a node  $t$  of  $T$  such that  $v \in \lambda(t)$ ,
2. for every hyperedge  $h \in E$ , there exists a node  $t$  of  $T$  such that  $h \subseteq \lambda(t)$ , and
3. for every  $v \in V$ , the set of nodes  $\{t \mid v \in \lambda(t)\}$  induces a connected subtree of  $T$ .

The width of a tree decomposition is  $\max(\{|\lambda(t)| - 1 \mid t \text{ node of } T\})$ . The treewidth  $\text{tw}(G)$  of a hypergraph  $G$  is the minimum width over all its tree decompositions.

Let  $\mathcal{H}$  be a class of hypergraphs, and define  $\text{tw}(\mathcal{H})$  to be the maximum treewidth over the hypergraphs in  $\mathcal{H}$ . If  $\text{tw}(\mathcal{H})$  is unbounded we write  $\text{tw}(\mathcal{H}) = \infty$ ; otherwise  $\text{tw}(\mathcal{H}) < \infty$ .

We can now restate using the language of global constraints the following result, from Dalmau et al. [12], which builds on several earlier results [13,14].

**Theorem 1 ([12]).** Let  $\mathcal{C}$  be a constraint catalogue and  $\mathcal{H}$  a class of hypergraphs.  $\text{CSP}(\mathcal{H}, \mathcal{C})$  is tractable if  $\text{tw}(\mathcal{H}) < \infty$ .

Observe that the family of constraint problems described in Example 1 is not covered by the above result, because the treewidth of the associated hypergraphs is unbounded.

## 4 Cooperating Constraint Catalogues

Whenever constraint scopes overlap, we may ask whether the possible assignments to the variables in the overlap are essentially different. It may be that some assignments extend to precisely the same satisfying assignments in each of the overlapping constraints. If so, we may as well identify such assignments.

**Definition 8 (Disjoint union of assignments).** Let  $\theta_1$  and  $\theta_2$  be two assignments of disjoint sets of variables  $V_1$  and  $V_2$ , respectively. The disjoint union of  $\theta_1$  and  $\theta_2$ , denoted  $\theta_1 \oplus \theta_2$ , is the assignment of  $V_1 \cup V_2$  such that  $(\theta_1 \oplus \theta_2)(v) = \theta_1(v)$  for all  $v \in V_1$ , and  $(\theta_1 \oplus \theta_2)(v) = \theta_2(v)$  for all  $v \in V_2$ .

**Definition 9 (Projection).** Let  $\Theta$  be a set of assignments of a set of variables  $V$ . The projection of  $\Theta$  onto a set of variables  $X \subseteq V$  is the set of assignments  $\pi_X(\Theta) = \{\theta|_X \mid \theta \in \Theta\}$ .

Note that when  $\Theta = \emptyset$  we have  $\pi_X(\Theta) = \emptyset$  for any set  $X$ , but when  $X = \emptyset$  and  $\Theta \neq \emptyset$ , we have  $\pi_X(\Theta) = \{\perp\}$ .

**Definition 10 (Assignment extension).** Let  $e[\delta]$  be a global constraint, and  $X \subseteq \text{vars}(\delta)$ . For every assignment  $\mu$  of  $X$ , let  $\text{ext}(\mu, e[\delta]) = \pi_{\text{vars}(\delta) - X}(\{\theta \in e[\delta] \mid \theta|_X = \mu\})$ .

In other words, for any assignment  $\mu$  of  $X$ , the set  $\text{ext}(\mu, e[\delta])$  is the set of assignments of  $\text{vars}(\delta) - X$  that extend  $\mu$  to a satisfying assignment for  $e[\delta]$ ; i.e., those assignments  $\theta$  for which  $\mu \oplus \theta \in e[\delta]$ .

**Definition 11 (Extension equivalence).** Let  $e[\delta]$  be a global constraint, and  $X \subseteq \text{vars}(\delta)$ . We say that two assignments  $\theta_1, \theta_2$  to  $X$  are extension equivalent on  $X$  with respect to  $e[\delta]$  if  $\text{ext}(\theta_1, e[\delta]) = \text{ext}(\theta_2, e[\delta])$ . We denote this equivalence relation by  $\text{equiv}[e[\delta], X]$ ; that is,  $\text{equiv}[e[\delta], X](\theta_1, \theta_2)$  holds if and only if  $\theta_1$  and  $\theta_2$  are extension equivalent on  $X$  with respect to  $e[\delta]$ .

In other words, two assignments to some subset of the variables of a constraint  $e[\delta]$  are extension equivalent if every assignment to the rest of the variables combines with both of them to give either two assignments that satisfy  $e[\delta]$ , or two that falsify it.

*Example 7.* Consider the special case of extension equivalence with respect to a clause (cf. Example 3).

Given any clause  $e[\delta]$ , and any non-empty set of variables  $X \subseteq \text{vars}(\delta)$ , any assignment to  $X$  will either satisfy one of the corresponding literals specified by  $\delta$ , or else falsify all of them. If it satisfies at least one of them, then any extension will satisfy the clause, so all such assignments are extension equivalent. If it falsifies all of them, then an extension will satisfy the clauses if and only if it satisfies one of the other literals. Hence the equivalence relation  $\text{equiv}[e[\delta], X]$  has precisely 2 equivalence classes, one containing the single assignment that falsifies all the literals corresponding to  $X$ , and one containing all other assignments.

**Definition 12 (Intersecting variables).** Let  $S$  be a set of global constraints. We write  $\text{iv}(S)$  for the set of variables common to all of their scopes, that is,  $\text{iv}(S) = \bigcap_{e[\delta] \in S} \text{vars}(\delta)$ .

**Definition 13 (Join).** For any set  $S$  of global constraints, we define the join of  $S$ , denoted  $\text{join}(S)$ , to be a global constraint  $e'[\delta']$  with  $\text{vars}(\delta') = \bigcup_{e[\delta] \in S} \text{vars}(\delta)$  such that for any assignment  $\theta$  to  $\text{vars}(\delta')$ , we have  $\theta \in e'[\delta']$  if and only if for every  $e[\delta] \in S$  we have  $\theta|_{\text{vars}(\delta)} \in e[\delta]$ .

The join of a set of global constraints may have no simple compact description, and computing its extension may be computationally expensive. However, we introduce this construct simply in order to describe the combined effect of a set of global constraints in terms of a single constraint.

*Example 8.* Let  $V = \{v_1, \dots, v_n\}$ , for some  $n \geq 3$ , be a set of variables with  $D(v_i) = \{a, b, c\}$ , and let  $S = \{e_1[\delta_1], e_2[\delta_2]\}$  be a set of two global constraints as defined below:

- $e_1[\delta_1]$  is a table constraint with  $\text{vars}(\delta_1) = \{v_1, \dots, v_{n-1}\}$  which enforces *equality*, i.e.,  $\delta_1 = \{\theta_a, \theta_b, \theta_c\}$ , where for each  $x \in D(V)$  and  $v \in \text{vars}(\delta_1)$ ,  $\theta_x(v) = x$ .
- $e_2[\delta_2]$  is a negative constraint with  $\text{vars}(\delta_2) = \{v_2, \dots, v_n\}$  which enforces a *not-all-equal* condition, i.e.,  $\delta_2 = \{\theta_a, \theta_b, \theta_c\}$ , where for each  $x \in D(V)$  and  $v \in \text{vars}(\delta_2)$ ,  $\theta_x(v) = x$ .

We will use substitution notation to write assignments explicitly; thus, an assignment of  $\{v, w\}$  that assigns  $a$  to both variables is written  $\{v/a, w/a\}$ .

We have that  $\text{iv}(S) = \{v_2, \dots, v_{n-1}\}$ . The equivalence classes of assignments to  $\text{iv}(S)$  under  $\text{equiv}[\text{join}(S), \text{iv}(S)]$  are  $\{\{v_2/a, \dots, v_{n-1}/a\}\}$ ,  $\{\{v_2/b, \dots, v_{n-1}/b\}\}$ , and  $\{\{v_2/c, \dots, v_{n-1}/c\}\}$ , each containing the single assignment shown, as well as (for  $n > 3$ ) a final class containing all other assignments, for which we can choose an arbitrary representative assignment,  $\theta_0$ , such as  $\{v_2/a, v_3/b, \dots, v_{n-1}/b\}$ .

Each assignment in the first 3 classes has just 2 possible extensions that satisfy  $\text{join}(S)$ , since the value assigned to  $v_1$  must equal the value assigned to  $v_2, \dots, v_{n-1}$ , and the value assigned to  $v_n$  must be different. The assignment  $\theta_0$  has no extensions, since  $\text{ext}(\theta_0, e_1[\delta_1]) = \emptyset$ .

Hence the number of equivalence classes in  $\text{equiv}[\text{join}(S), \text{iv}(S)]$  is at most 4, even though the total number of possible assignments of  $\text{iv}(S)$  is  $3^{n-2}$ .

**Definition 14 (Cooperating constraint catalogue).** *We say that a constraint catalogue  $\mathcal{C}$  is a cooperating catalogue if for any finite set of global constraints  $S \subseteq \mathcal{C}$ , we can compute a set of assignments of the variables  $\text{iv}(S)$  containing at least one representative of each equivalence class of  $\text{equiv}[\text{join}(S), \text{iv}(S)]$  in polynomial time in the size of  $\text{iv}(S)$  and the total size of the constraints in  $S$ .*

Note that this definition requires two things. First, that the number of equivalence classes in the equivalence relation  $\text{equiv}[\text{join}(S), \text{iv}(S)]$  is bounded by some fixed polynomial in the size of  $\text{iv}(S)$  and the size of the constraints in  $S$ . Secondly, that a suitable set of representatives for these equivalence classes can be computed efficiently from the constraints.

*Example 9.* Consider a constraint catalogue consisting entirely of clauses (of arbitrary arity). It was shown in Example 7 that for any clause  $e[\delta]$  and any non-empty  $X \subseteq \text{vars}(\delta)$  the equivalence relation  $\text{equiv}[e[\delta], X]$  has precisely 2 equivalence classes.

If we consider some finite set,  $S$ , of clauses, then a similar argument shows that the equivalence relation  $\text{equiv}[\text{join}(S), \text{iv}(S)]$  has at most  $|S| + 1$  classes. These are given by the single assignments of the variables in  $\text{iv}(S)$  that falsify the literals corresponding to the variables of  $\text{iv}(S)$  in each clause (there are at most  $|S|$  of these — they may not all be distinct) together with at most one further equivalence class containing all other assignments (which must satisfy at least one literal in each clause of  $S$ ).



Hence the total number of equivalence classes in the equivalence relation  $\text{equiv}[\text{join}(S), \text{iv}(S)]$  increases at most linearly with the number of clauses in  $S$ , and a representative for each class can be easily obtained from the descriptions of these clauses, by projecting the falsifying assignments down to the set of common variables,  $\text{iv}(S)$ , and adding at most one more, arbitrary, assignment.

By same argument, if we consider some finite set,  $S$ , of table constraints, then the equivalence relation  $\text{equiv}[\text{join}(S), \text{iv}(S)]$  has at most one class for each assignment allowed by each table constraint in  $S$ , together with at most one further class containing all other assignments.

In general, arbitrary EGC constraints (cf. Example 2) do not form a cooperating catalogue. However, we will show that if we bound the size of the variable domains, then the resulting EGC constraints do form a cooperating catalogue.

**Definition 15 (Counting function).** *Let  $X$  be a set of variables with domain  $D = \bigcup_{x \in X} D(x)$ . A counting function for  $X$  is any function  $K : D \rightarrow \mathbb{N}$  such that  $\sum_{a \in D} K(a) = |X|$ .*

*Every assignment  $\theta$  to  $X$  defines a corresponding counting function  $K_\theta$  given by  $K_\theta(a) = |\{x \in X \mid \theta(x) = a\}|$  for every  $a \in D$ .*

It is easy to verify that no EGC constraint can distinguish two assignments with the same counting function; for any EGC constraint, either both assignments satisfy it, or they both falsify it. It follows that two assignments with the same counting function are extension equivalent with respect to EGC constraints.

**Definition 16 (Counting constraints).** *A global constraint  $e[\delta]$  is called a counting constraint if, for any two assignments  $\theta_1, \theta_2$  of  $\text{vars}(\delta)$  which have the same counting function, either  $\theta_1, \theta_2 \in e[\delta]$  or  $\theta_1, \theta_2 \notin e[\delta]$ .*

EGC constraints are not the only constraint type with this property. Constraints that require the sum (or the product) of the values of all variables in their scope to take a particular value, and constraints that require the minimum (or maximum) value of the variables in their scope to take a certain value, are also counting constraints.

Another example is given by the NValue constraint type, which requires that the number of distinct domain values taken by an assignment is a member of a specified set of acceptable numbers.

*Example 10 (NValue constraint type  $[4, 6]$ ).* In an NValue constraint,  $e[\delta]$ , the description  $\delta$  specifies a finite set of natural numbers  $L_\delta \subset \mathbb{N}$ . The algorithm  $e$  maps an assignment  $\theta$  to 1 if  $|\{\theta(v) \mid v \in \text{vars}(\delta)\}| \in L_\delta$ .

The reason for introducing counting functions is the following key property, previously noted by Bulatov and Marx [8].

*Property 1.* The number of possible counting functions for a set of variables  $X$  is at most  $\binom{|X|+|D|-1}{|D|-1} = O(|X|^{|D|})$ , where  $D = \bigcup_{x \in X} D(x)$ .

*Proof.* If every variable  $x \in X$  has  $D$  as its set of domain elements, that is,  $D(x) = D$ , then every counting function corresponds to a distinct way of partitioning  $|X|$  variables into at most  $|D|$  boxes. There are  $\binom{|X|+|D|-1}{|D|-1}$  ways of doing so [27, Section 2.3.3]. On the other hand, if there are variables  $x \in X$  such that  $D(x) \subset D$ , then that disallows some counting functions.

**Theorem 2.** *Any constraint catalogue that contains only counting constraints with bounded domain size, table constraints, and negative constraints, is a cooperating catalogue.*

*Proof.* Let  $\mathcal{C}$  be a constraint catalogue containing only global constraints of the specified types, and let  $S \subseteq \mathcal{C}$  be a finite subset of  $\mathcal{C}$ . Partition  $S$  into two subsets:  $S^C$ , containing only counting constraints and  $S^\pm$  containing only table and negative constraints.

Let  $\mathcal{K}$  be a set containing assignments of  $\text{iv}(S)$ , such that for every counting function  $K$  for  $\text{iv}(S)$ , there is some assignment  $\theta_K \in \mathcal{K}$  with  $K_\theta = K$ . By Property 1, the number of counting functions for  $\text{iv}(S)$  is bounded by  $O(|\text{iv}(S)|^d)$ , where  $d$  is the bound on the domain size for the counting constraints in  $\mathcal{C}$ . Hence such a set  $\mathcal{K}$  can be computed in polynomial time in the size of  $\text{iv}(S)$ .

For each constraint in  $S^\pm$  we have that the description is a list of assignments (these are the allowed assignments for the table constraints and the forbidden assignments for the negative constraints, see Example 4).

As we described in Example 9, for each table constraint  $e[\delta] \in S$ , we can obtain a representative for each equivalence class of  $\text{equiv}[e[\delta], \text{iv}(S)]$  by taking the projection onto  $\text{iv}(S)$  of each allowed assignment, which we can denote by  $\pi_{\text{iv}(S)}(\delta)$ , together with at most one further, arbitrary, assignment,  $\theta_0$ , that is not in this set. This set of assignments contains at least one representative for each equivalence class of  $\text{equiv}[e[\delta], \text{iv}(S)]$  (and possibly more than one representative for some of these classes).

Similarly, for each negative constraint  $e[\delta] \in S$ , we can obtain a representative for each equivalence class of  $\text{equiv}[e[\delta], \text{iv}(S)]$ , by taking the projection onto  $\text{iv}(S)$  of each forbidden assignment, which we can again denote by  $\pi_{\text{iv}(S)}(\delta)$ , together with at most one further, arbitrary, assignment,  $\theta_0$ , that is not in this set.

Now consider the set of assignments  $\mathcal{A} = \mathcal{K} \cup \{\theta_0\} \cup \bigcup_{e[\delta] \in S^\pm} \pi_{\text{iv}(S)}(\delta)$ , where  $\theta_0$  is an arbitrary assignment of  $\text{iv}(S)$  which does not occur in  $\pi_{\text{iv}(S)}(\delta)$  for any  $e[\delta] \in S$  (if such an assignment exists). We claim that this set of assignments contains at least one representative for each equivalence class of  $\text{equiv}[\text{join}(S), \text{iv}(S)]$  (and possibly more than one for some classes).

To establish this claim we will show that any assignment  $\theta$  of  $\text{iv}(S)$  that is not in  $\mathcal{A}$  must be extension equivalent to some member of  $\mathcal{A}$ . Let  $\theta$  be an assignment of  $\text{iv}(S)$  that is not in  $\mathcal{A}$  (if such an assignment exists). If  $S^\pm$  contains any positive constraints, then  $\theta$  has an empty set of extensions to these constraints, and hence is extension equivalent to  $\theta_0$ . Otherwise, any extension of  $\theta$  will satisfy all negative constraints in  $S^\pm$ , so the extensions of  $\theta$  that satisfy  $\text{join}(S)$

are completely determined by the counting function  $K_\theta$ . In this case  $\theta$  will be extension equivalent to some element of  $\mathcal{K}$ .

Moreover, the set of assignments  $\mathcal{A}$  can be computed from  $S$  in polynomial time in the size of  $\text{iv}(S)$  and the total size of the descriptions of the constraints in  $S^\pm$ . Therefore,  $\mathcal{C}$  is a cooperating catalogue as described in Definition 14.

*Example 11.* By Theorem 2, the constraints in Example 1 form a cooperating catalogue.

## 5 Polynomial-Time Reductions

In this section, we will show that, for any constraint problem over a cooperating catalogue, a set of variables that all occur in exactly the same set of constraint scopes can be replaced by a single new variable with an appropriate domain, to give a polynomial-time reduction to a smaller problem.

**Definition 17 (Dual of a hypergraph).** Let  $G = \langle V, H \rangle$  be a hypergraph. The dual  $G^*$  of  $G$  is a hypergraph with vertex set  $H$  and a hyperedge  $\{h \in H \mid v \in h\}$  for every  $v \in V$ . For a class  $\mathcal{H}$  of hypergraphs, let  $\mathcal{H}^* = \{G^* \mid G \in \mathcal{H}\}$ .

*Example 12.* Consider the hypergraph  $G$  in Figure 1. The dual,  $G^*$ , of this hypergraph has vertex set  $\{C_1, C_2, C_3, C_4, C_5\}$  and five hyperedges  $\{C_1, C_2\}$ ,  $\{C_1, C_3\}$ ,  $\{C_2, C_4\}$ ,  $\{C_3, C_4\}$  and  $\{C_4, C_5\}$ . This transformation is illustrated in Figure 2.

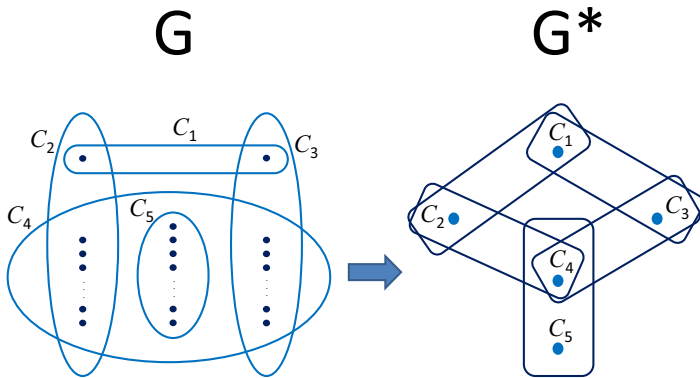


**Fig. 2.**  $G$  and  $G^*$  from Example 12

Note that the dual of the dual of a hypergraph is not necessarily the original hypergraph, since we do not allow multiple identical hyperedges.

*Example 13.* Consider the dual hypergraph  $G^*$  defined in Example 12. Taking the dual of this hypergraph yields  $G^{**}$ , with vertex set  $\{h_1, \dots, h_5\}$  (corresponding to the 5 hyperedges in  $G^*$ ) and 5 distinct hyperedges, as shown in Figure 3.

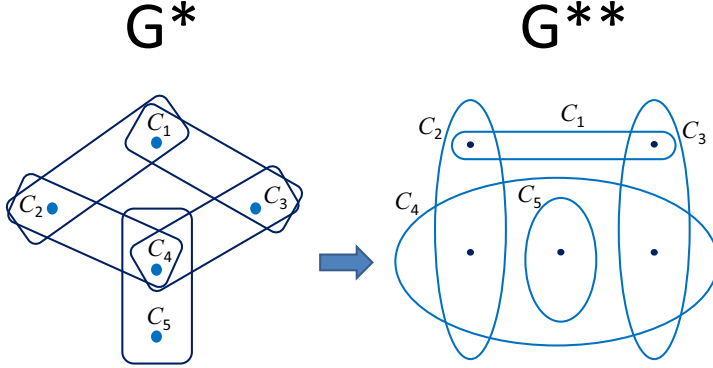


Fig. 3.  $G^*$  and  $G^{**}$  from Example 13

In the example above, taking the dual of a hypergraph twice had the effect of merging precisely those sets of variables that occur in the same set of hyperedges. It is easy to verify that this is true in general: Taking the dual twice equates precisely those variables that occur in the same set of hyperedges.

**Lemma 1.** *For any hypergraph  $G$ , the hypergraph  $G^{**}$  has precisely one vertex corresponding to each maximal subset of vertices of  $G$  that occur in the same set of hyperedges.*

Next, we combine the idea of the dual with the usual notion of treewidth to create a new measure of width.

**Definition 18 (twDD).** *Let  $G$  be a hypergraph. The treewidth of the dual of the dual (twDD) of  $G$  is  $\text{twDD}(G) = \text{tw}(G^{**})$ .*

*For a class of hypergraphs  $\mathcal{H}$ , we define  $\text{twDD}(\mathcal{H}) = \text{tw}(\mathcal{H}^{**})$ .*

*Example 14.* Consider the class  $\mathcal{H}$  of hypergraphs of the family of problems described in Example 1. Whatever the value of  $n$ , the dual hypergraph,  $G^*$  is the same, as shown in Figure 2. Hence for all problems in this family the hypergraph  $G^{**}$  is as shown in Figure 3, and can be shown to have treewidth 3. Hence  $\text{twDD}(\mathcal{H}) = 3$ .

When replacing a set of variables in a CSP instance with a single variable, we will use the following definition.

**Definition 19 (Quotient of a CSP instance).** *Let  $P = \langle V, C \rangle$  be a CSP instance and  $X \subseteq V$  be a non-empty subset of variables that all occur in the scopes of the same set  $S$  of constraints. The quotient of  $P$  with respect to  $X$ , denoted  $P^X$ , is defined as follows.*

- The variables of  $P^X$  are given by  $V^X = (V - X) \cup \{v_X\}$ , where  $v_X$  is a fresh variable, and the domain of  $v_X$  is the set of equivalence classes of  $\text{equiv}[\text{join}(S), X]$ .

- The constraints of  $P^X$  are unchanged, except that each constraint  $e[\delta] \in S$  is replaced by a new constraint  $e^X[\delta^X]$ , where  $\text{vars}(\delta^X) = (\text{vars}(\delta) - X) \cup \{v_X\}$ . For any assignment  $\theta$  of  $\text{vars}(\delta^X)$ , we define  $e^X[\delta^X](\theta)$  to be 1 if and only if  $\theta|_{\text{vars}(\delta)-X} \oplus \mu \in e[\delta]$ , where  $\mu$  is a representative of the equivalence class  $\theta(v_X)$ .

We note that, by Definition 11, the value of  $e^X[\delta^X]$  specified in Definition 19 is well-defined, that is, it does not depend on the specific representative chosen for the equivalence class  $\theta(v_X)$ , since each representative has the same set of possible extensions.

**Lemma 2.** *Let  $P = \langle V, C \rangle$  be a CSP instance and  $X \subseteq V$  be a non-empty subset of variables that all occur in the scopes of the same set of constraints. The instance  $P^X$  has a solution if and only if  $P$  has a solution.*

*Proof (Sketch).* Let  $P = \langle V, C \rangle$  and  $X$  be given, and let  $S \subseteq C$  be the set of constraints  $e[\delta]$  such that  $X \subseteq \text{vars}(\delta)$ .

Construct the instance  $P^X$  as specified in Definition 19. Any solution to  $P$  can be converted into a corresponding solution for  $P^X$ , and vice versa. This conversion process just involves replacing the part of the solution assignment that gives values to the variables in the set  $X$  with an assignment that gives a suitable value to the new variable  $v_X$ .

**Theorem 3.** *Any CSP instance  $P$  can be converted to an instance  $P'$  with  $\text{hyp}(P') = \text{hyp}(P)^{**}$ , such that  $P'$  has a solution if and only if  $P$  does. Moreover, if  $P$  is over a cooperating catalogue, this conversion can be done in polynomial time.*

*Proof.* Let  $P = \langle V, C \rangle$  be a CSP instance. For each variable  $v \in V$  we define  $S(v) = \{e[\delta] \in C \mid v \in \text{vars}(\delta)\}$ . We then partition the vertices of  $P$  into subsets  $X_1, \dots, X_k$ , where each  $X_i$  is a maximal subset of variables  $v$  that share the same value for  $S(v)$ .

We initially set  $P_0 = P$ . Then, for each  $X_i$  in turn, we set  $P_i = (P_{i-1})^{X_i}$ . Finally we set  $P' = P_k$ . By Lemma 1,  $\text{hyp}(P') = \text{hyp}(P)^{**}$ , and by Lemma 2,  $P'$  has a solution if and only if  $P$  has a solution.

Finally, if  $P$  is over a cooperating catalogue, then by Definition 14, we can compute the domains of each new variable introduced in polynomial time in the size of each  $X_i$  and the total size of the constraints. Hence we can compute  $P'$  in polynomial time.

Using Theorem 3, we can immediately get a new tractable CSP class by extending Theorem 1.

**Theorem 4.** *Let  $\mathcal{C}$  be a constraint catalogue and  $\mathcal{H}$  a class of hypergraphs.  $\text{CSP}(\mathcal{H}, \mathcal{C})$  is tractable if  $\mathcal{C}$  is a cooperating catalogue and  $\text{twDD}(\mathcal{H}) < \infty$ .*

*Proof.* Let  $\mathcal{C}$  be a cooperating catalogue,  $\mathcal{H}$  a class of hypergraphs such that  $\text{twDD}(\mathcal{H}) < \infty$ , and  $P \in \text{CSP}(\mathcal{H}, \mathcal{C})$ . Reduce  $P$  to a CSP instance  $P'$  using

Theorem 3. By Definition 18, since  $\text{hyp}(P') = \text{hyp}(P)^{**}$ ,  $\text{tw}(\text{hyp}(P')) < \infty$ , which means that  $P'$  satisfies the conditions of Theorem 1, and hence can be solved in polynomial time.

Recall the family of constraint problems described in Example 1 at the start of this paper. Since the constraints in this problem form a cooperating catalogue (Example 11), and all instances have bounded  $\text{twDD}$  (Example 14), this family of problems is tractable by Theorem 4.

## 6 Summary and Future Work

We have identified a novel tractable class of constraint problems with global constraints. In fact, our results generalize several previously studied classes of problems [12]. Moreover, this is the first representation-independent tractability result for constraint problems with global constraints.

Our new class is defined by restricting both the nature of the constraints and the way that they interact. As demonstrated in Example 5, instances with a single global constraint may already be NP-complete [26], so we cannot hope to achieve tractability by structural restrictions alone. In other words, notions such as bounded degree of cyclicity [21] or bounded hypertree width [18] are not sufficient to ensure tractability in the framework of global constraints, where the arity of individual constraints is unbounded. This led us to introduce the notion of a cooperating constraint catalogue, which is sufficiently restricted to ensure that an individual constraint is always tractable.

However, this restriction on the nature of the constraints is still not enough to ensure tractability on any structure: Example 6 demonstrates that not all structures are tractable even with a cooperating constraint catalogue. In fact, a family of problems with acyclic structure (hypertree width one) over a cooperating constraint catalogue can still be NP-complete. This led us to investigate restrictions on the structure that are sufficient to ensure tractability for all instances over a cooperating catalogue. In particular, we have shown that it is sufficient to ensure that the dual of the dual of the hypergraph of the instance has bounded treewidth.

An intriguing open question is whether there are other restrictions on the nature of the constraints or the structure of the instances that are sufficient to ensure tractability in the framework of global constraints. Very little work has been done on this question, apart from the pioneering work of Bulatov and Marx [8], which considered only a single global cardinality constraint, along with arbitrary table constraints, and of Chen and Dalmau [9] on two specific succinct representations. Almost all other previous work on tractable classes has considered only table constraints. This may be one reason why such work has had little practical impact on the design of constraint solvers, which rely heavily on the use of in-built special-purpose global constraints.

We see this paper as a first step in the development of a more robust and applicable theory of tractability for global constraints.

## References

1. Aschinger, M., Drescher, C., Friedrich, G., Gottlob, G., Jeavons, P., Ryabokon, A., Thorstensen, E.: Optimization methods for the partner units problem. In: Achterberg, T., Beck, J.C. (eds.) CPAIOR 2011. LNCS, vol. 6697, pp. 4–19. Springer, Heidelberg (2011)
2. Aschinger, M., Drescher, C., Gottlob, G., Jeavons, P., Thorstensen, E.: Structural decomposition methods and what they are good for. In: Schwentick, T., Dürr, C. (eds.) Proc. STACS 2011. LIPIcs, vol. 9, pp. 12–28 (2011)
3. Beeri, C., Fagin, R., Maier, D., Yannakakis, M.: On the desirability of acyclic database schemes. *Journal of the ACM* 30, 479–513 (1983)
4. Beldiceanu, N.: Pruning for the minimum constraint family and for the number of distinct values constraint family. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 211–224. Springer, Heidelberg (2001)
5. Bessiere, C., Hebrard, E., Hnich, B., Walsh, T.: The complexity of reasoning with global constraints. *Constraints* 12(2), 239–259 (2007)
6. Bessiere, C., Katsirelos, G., Narodytska, N., Quimper, C.-G., Walsh, T.: Decomposition of the NValue constraint. In: Cohen, D. (ed.) CP 2010. LNCS, vol. 6308, pp. 114–128. Springer, Heidelberg (2010)
7. Bulatov, A., Jeavons, P., Krokhin, A.: Classifying the complexity of constraints using finite algebras. *SIAM Journal on Computing* 34(3), 720–742 (2005)
8. Bulatov, A.A., Marx, D.: The complexity of global cardinality constraints. *Logical Methods in Computer Science* 6(4(4)), 1–27 (2010)
9. Chen, H., Grohe, M.: Constraint satisfaction with succinctly specified relations. *Journal of Computer and System Sciences* 76(8), 847–860 (2010)
10. Cohen, D.A., Green, M.J., Houghton, C.: Constraint representations and structural tractability. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 289–303. Springer, Heidelberg (2009)
11. Cohen, D.A., Jeavons, P., Gyssens, M.: A unified theory of structural tractability for constraint satisfaction problems. *Journal of Computer and System Sciences* 74(5), 721–743 (2008)
12. Dalmau, V., Kolaitis, P.G., Vardi, M.Y.: Constraint satisfaction, bounded treewidth, and finite-variable logics. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 223–254. Springer, Heidelberg (2002)
13. Dechter, R., Pearl, J.: Tree clustering for constraint networks. *Artificial Intelligence* 38(3), 353–366 (1989)
14. Freuder, E.C.: Complexity of k-tree structured constraint satisfaction problems. In: Proc. AAAI, pp. 4–9. AAAI Press / The MIT Press (1990)
15. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman (1979)
16. Gent, I.P., Jefferson, C., Miguel, I.: MINION: A fast, scalable constraint solver. In: Proc. ECAI 2006, pp. 98–102. IOS Press (2006)
17. Gottlob, G., Leone, N., Scarcello, F.: A comparison of structural CSP decomposition methods. *Artificial Intelligence* 124(2), 243–282 (2000)
18. Gottlob, G., Leone, N., Scarcello, F.: Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences* 64(3), 579–627 (2002)
19. Green, M.J., Jefferson, C.: Structural tractability of propagated constraints. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 372–386. Springer, Heidelberg (2008)

20. Grohe, M.: The complexity of homomorphism and constraint satisfaction problems seen from the other side. *Journal of the ACM* 54(1), 1–24 (2007)
21. Gyssens, M., Jeavons, P.G., Cohen, D.A.: Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence* 66(1), 57–89 (1994)
22. Hermenier, F., Demassey, S., Lorca, X.: Bin repacking scheduling in virtualized datacenters. In: Lee, J. (ed.) *CP 2011*. LNCS, vol. 6876, pp. 27–41. Springer, Heidelberg (2011)
23. 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, ch. 6, pp. 169–208. Elsevier (2006)
24. Kutz, M., Elbassioni, K., Katriel, I., Mahajan, M.: Simultaneous matchings: Hardness and approximation. *Journal of Computer and System Sciences* 74(5), 884–897 (2008)
25. Marx, D.: Tractable hypergraph properties for constraint satisfaction and conjunctive queries. In: *Proc. STOC 2010*, pp. 735–744. ACM (2010)
26. Quimper, C.G., López-Ortiz, A., van Beek, P., Golynski, A.: Improved algorithms for the global cardinality constraint. In: Wallace, M. (ed.) *CP 2004*. LNCS, vol. 3258, pp. 542–556. Springer, Heidelberg (2004)
27. Rosen, K.H., Michaels, J.G., Gross, J.L., Grossman, J.W., Shier, D.R. (eds.): *Handbook of Discrete and Combinatorial Mathematics. Discrete Mathematics and Its Applications*. CRC Press (2000)
28. Rossi, F., van Beek, P., Walsh, T. (eds.): *The Handbook of Constraint Programming*. Elsevier (2006)
29. Samer, M., Szeider, S.: Tractable cases of the extended global cardinality constraint. *Constraints* 16(1), 1–24 (2011)
30. Wallace, M.: Practical applications of constraint programming. *Constraints* 1, 139–168 (1996)
31. Wallace, M., Novello, S., Schimpf, J.: ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal* 12(1), 137–158 (1997)



# Postponing Optimization to Speed Up MAXSAT Solving

Jessica Davies<sup>1</sup> and Fahiem Bacchus<sup>2</sup>

<sup>1</sup> MIAT, UR 875, INRA, F-31326 Castanet-Tolosan, France  
`jessica.davies@toulouse.inra.fr`

<sup>2</sup> Department of Computer Science, University of Toronto,  
Toronto, Ontario, Canada, M5S 3H5  
`fbacchus@cs.toronto.edu`

**Abstract.** MAXSAT is an optimization version of SAT that can represent a wide variety of important optimization problems. A recent approach for solving MAXSAT is to exploit both a SAT solver and a Mixed Integer Programming (MIP) solver in a hybrid approach. Each solver generates information used by the other solver in a series of iterations that terminates when an optimal solution is found. Empirical results indicate that a bottleneck in this process is the time required by the MIP solver, arising from the large number of times it is invoked. In this paper we present a modified approach that postpones the calls to the MIP solver. This involves substituting non-optimal solutions for the optimal ones computed by the MIP solver, whenever possible. We describe the new approach and some different instantiations of it. We perform an extensive empirical evaluation comparing the performance of the resulting solvers with other state-of-the-art MAXSAT solvers. We show that the best performing versions of our approach advance the state-of-the-art in MAXSAT solving.

## 1 Introduction

MAXSAT, the optimization version of Satisfiability (SAT), is the problem of finding a minimum cost truth assignment for a set of clauses where a cost is incurred for every falsified clause. It is called MAXSAT since in the simplest case where every clause is equally costly to falsify, a solution will satisfy a maximum number of clauses. In the most general version of MAXSAT some clauses are *hard* incurring an infinite cost if they are falsified, while the other clauses are *soft* incurring some integer cost greater than zero. This most general version of MAXSAT is often called **weighted partial MAXSAT** (WPMS) and is what we address in this paper. Many practical problems can be encoded in MAXSAT, so developing effective ways to solve MAXSAT is an important research topic.

There are two standard methods for solving MAXSAT: using Branch and Bound search (e.g. [9,14]), and using a sequence of decision problems, usually encoded as SAT (e.g. [2,3,5,15,10]). In [6] an alternative algorithm for solving MAXSAT, called MAXHS, was presented. MAXHS also solves a sequence of SAT decision problems, but in contrast to existing approaches the SAT problems do not become more difficult for the SAT solver to solve. This is accomplished via a hybrid approach,

whereby a SAT solver and a Mixed Integer Linear Program (MIP) solver are used to cooperatively solve the MAXSAT problem using an approach similar to Bender's Decomposition [11]. The MIP solver is used to find optimal solutions which the SAT solver then tests for feasibility. If the solution is not feasible the SAT solver computes a new constraint to add to the MIP model and the MIP solver is invoked again to find a new optimal solution that additionally satisfies the new constraint.

In this paper we investigate a new technique for improving the performance of this hybrid approach. In [7] it has already been shown that the hybrid approach is one of the state-of-the-art approaches to solving MAXSAT, and thus improving this approach is one way of advancing the state-of-the-art.

Analyzing the performance of the hybrid approach indicates that the main bottleneck is the time spent by the MIP solver. This time mostly accumulates from the number of times that the MIP solver must be called: it is called every time the SAT solver computes a new feasibility constraint, in order to derive a new optimal solution satisfying this additional constraint. Although these calls often take relatively little time, after hundreds of separate calls the total time becomes quite significant.

Inspired by an idea presented by Moreno-Centeno and Karp [16] we developed a method for delaying the calls to the MIP solver for as long as possible. We accomplish this by recognizing situations where non-optimal solutions can be used in place of the optimal solutions produced by the MIP solver without impacting the algorithm's correctness. The SAT solver can use these non-optimal solutions to compute its feasibility constraints and the iterations of feasibility and "optimization" can continue. However, since the optimization phase is now an approximation that can be computed cheaply, each iteration is much more efficient. Eventually, however, an optimal solution must be computed to ensure correctness. So our technique postpones, rather than removes, optimization.

We show that our new technique yields a significant improvement in the performance of the hybrid algorithm and makes it the most robust current approach to solving MAXSAT. One of the reasons why we obtain such a good performance improvement is that the MIP solver is not perfectly incremental. By using non-optimal solutions we collect many feasibility constraints before having to compute their optimal solution. This means that each additional call to the MIP solver involves a model that has been augmented by many feasibility constraints, whereas in the previous approach the model was only augmented by a single feasibility constraint. Although the MIP solver can take advantage of its previous computations when called again, it is not perfectly incremental. That is, its solving time when given  $k$  new constraints is typically significantly smaller than the sum of its  $k$  solving times when it is given these constraints one at a time and asked to compute an optimal solution each time.

The remainder of the paper is organized as follows. Section 2 provides basic definitions. The MAXHS approach is then reviewed in Section 3 and we show that its main bottleneck is the MIP solving time. Section 4 presents the new algorithm, which addresses this issue by allowing the MIP optimization to be postponed.

An effective additional enhancement, seeding the MIP model with constraints [7], is described in Section 5. The empirical results are reported in Section 6.

## 2 Background

A MAXSAT instance is specified by a set of propositional clauses  $\mathcal{F}$  each having a positive integer or infinite weight  $wt(c)$ ,  $c \in \mathcal{F}$ . Clauses with infinite weight are called hard clauses and are collectively denoted by  $hard(\mathcal{F})$ . The other clauses of  $\mathcal{F}$  all have finite weight and are called soft clauses,  $soft(\mathcal{F})$  ( $\mathcal{F} = hard(\mathcal{F}) \cup soft(\mathcal{F})$ ).

We define the function  $cost$  as follows: (a) if  $H$  is a set of clauses then  $cost(H)$  is the sum of the weights of the clauses in  $H$  ( $cost(H) = \sum_{c \in H} wt(c)$ ); and (b) if  $\pi$  is a truth assignment to the variables of  $\mathcal{F}$  then  $cost(\pi)$  is the sum of the weights of the clauses falsified by  $\pi$  ( $\sum_{\{c \mid \pi \models c\}} wt(c)$ ). A solution for  $\mathcal{F}$  is a truth assignment  $\pi$  to the variables of  $\mathcal{F}$  that satisfies  $hard(\mathcal{F})$  and is of minimum cost. We let  $mincost(\mathcal{F})$  denote the cost of such a solution. If  $hard(\mathcal{F})$  is unsatisfiable, then  $\mathcal{F}$  has no solution. In the remainder of the paper, we assume that  $hard(\mathcal{F})$  is **satisfiable** (this is easy to test in practice and facilitates clarity). A **core**  $\kappa$  of a MAXSAT formula  $\mathcal{F}$  is a subset of  $soft(\mathcal{F})$  such that  $\kappa \cup hard(\mathcal{F})$  is unsatisfiable. Note that since  $hard(\mathcal{F})$  is satisfiable, any solution to  $\mathcal{F}$  must falsify at least one clause in  $\kappa$ .

MAXSAT solvers that solve a sequence of decision problems typically insert blocking variables ( $b$ -variables) into the soft clauses of the MAXSAT instance.

**Definition 1.** *If  $\mathcal{F}$  is a MAXSAT problem, then its  **$b$ -variable relaxation** is a SAT problem  $\mathcal{F}^b = \{(c_i \vee b_i) : c_i \in soft(\mathcal{F})\} \cup hard(\mathcal{F})$  where all clause weights are removed. The  $b$ -variable  $b_i$  appears in the relaxed clause  $(c_i \vee b_i)$  and **no** where else in  $\mathcal{F}^b$ .*

The  $b$ -variable relaxation  $\mathcal{F}^b$  allows cores of the original MAXSAT formula  $\mathcal{F}$  to be computed conveniently, using the **Assumption** mechanism provided by MINISAT [8]. MINISAT can take as input a set of assumptions  $\mathcal{A}$ , specified as a set of literals, along with a CNF formula  $\mathcal{F}$  and then determine if  $\mathcal{F} \wedge \mathcal{A}$  is satisfiable. It will return a satisfying truth assignment for  $\mathcal{F} \wedge \mathcal{A}$  if one exists. Otherwise it will report unsatisfiability and return a learnt clause  $c$  which is a disjunction of negated literals of  $\mathcal{A}$ . This clause has the property that  $\mathcal{F} \models c$ . Thus in order to find a core of MAXSAT instance  $\mathcal{F}$ , we can pass MINISAT the CNF formula  $\mathcal{F}^b$  and the set of all negated  $b$ -variables as the assumptions. If  $\mathcal{F}$  has a core, MINISAT will return UNSAT along with a clause  $c = (b_{i_1} \vee \dots \vee b_{i_k})$  such that  $\mathcal{F}^b \models c$  and  $\kappa = \{c_{i_1}, \dots, c_{i_k}\}$  is a core of  $\mathcal{F}$ . Any clause over positive  $b$ -variables that is entailed by  $\mathcal{F}^b$ , e.g.,  $c = (b_{i_1} \vee \dots \vee b_{i_k})$ , is called a **core constraint**.

Besides supporting the computation of cores, the relaxed formula  $\mathcal{F}^b$  can also be used to find solutions for the original MAXSAT formula  $\mathcal{F}$ . To accomplish this we define an objective function  $bcost(\pi)$  over truth assignments  $\pi$  to the variables of  $\mathcal{F}^b$ , equal to the sum of the costs of the clauses whose  $b$ -variables

are set to *true*:  $bcost(\pi) = \sum_{b_i: \pi \models b_i} wt(c_i)$ . The minimum  $bcost$  models of  $\mathcal{F}^b$  are MAXSAT solutions.<sup>1</sup>

**Proposition 1.**  *$mincost(\mathcal{F}) = \min_{\pi \models \mathcal{F}^b} bcost(\pi)$ . Furthermore, if  $\pi \models \mathcal{F}^b$  achieves a minimum value of  $bcost(\pi)$ , then  $\pi$  restricted to the variables of  $\mathcal{F}$  is a solution for  $\mathcal{F}$ .*

However,  $\mathcal{F}^b$  has many models whose  $bcost$  is greater than necessary. For example, if a model  $\pi$  assigns  $b_i$  to *true* even though it also satisfies the soft clause  $c_i$ , then the  $bcost$  of  $\pi$  could be reduced by instead assigning  $b_i$  to *false*. We can eliminate such models by modifying  $\mathcal{F}^b$  so that the  $b$ -variables are forced to be equivalent to the negation of their corresponding soft clauses.

**Definition 2.** *Let  $\mathcal{F}$  be a MAXSAT formula. Then*

$$\mathcal{F}_{eq}^b = \mathcal{F}^b \cup \bigcup_{c_i \in \text{soft}(\mathcal{F})} \{(\neg b_i \vee \neg \ell) : \ell \in c_i\}$$

*is the relaxation of  $\mathcal{F}$  with  $b$ -variable equivalences.*

Again, the minimum  $bcost$  models of  $\mathcal{F}_{eq}^b$  are MAXSAT solutions.

**Proposition 2.**  *$mincost(\mathcal{F}) = \min_{\pi \models \mathcal{F}_{eq}^b} bcost(\pi)$ . Furthermore, if  $\pi \models \mathcal{F}_{eq}^b$  achieves a minimum value of  $bcost(\pi)$ , then  $\pi$  restricted to the variables of  $\mathcal{F}$  is a solution for  $\mathcal{F}$ .*

Propositions 1 and 2 show that we can solve the MAXSAT problem  $\mathcal{F}$  by searching for a  $bcost$  minimal satisfying assignment to  $\mathcal{F}^b$  or to  $\mathcal{F}_{eq}^b$ . Note also that  $\mathcal{F}_{eq}^b$  is a stronger theory enabling more inferences than  $\mathcal{F}^b$ .

### 3 The MAXHS Approach

In MAXHS [6] a SAT solver is called at every iteration to find a new core of  $\mathcal{F}$ . A MIP solver is then invoked to find a minimum  $bcost$  assignment to the  $b$ -variables that satisfies all of the core constraints found so far. This optimization problem corresponds to a Minimum Cost Hitting Set (MCHS) problem,<sup>2</sup> where the goal is to find a minimum cost set of clauses that hits each of the known cores. The optimal hitting set found by the MIP solver is tested by the SAT solver. If the SAT solver is unable to find another core, it means the MAXSAT solution has been found. Otherwise, the SAT solver returns a new core, the MIP solver re-optimizes, and the iterations continue.

This algorithm is shown in Algorithm 1. The set of cores is initialized on line 2 to the empty set. In the main loop of MAXHS, the MIP solver is invoked to find

<sup>1</sup> Recall we assume  $hard(\mathcal{F})$  is satisfiable, so  $\mathcal{F}^b$  is satisfiable.

<sup>2</sup> An instance of MCHS is given by a universe of weighted elements  $U$  and a collection of subsets of these elements,  $\mathcal{K} = \{\kappa_1, \dots, \kappa_m\}$ ,  $\kappa_i \subseteq U$ . The goal is to find a minimum weight set of elements  $hs \subseteq U$  such that  $hs \cap \kappa_i \neq \emptyset$  for all  $\kappa_i \in \mathcal{K}$ .

---

**Algorithm 1.** The MAXHS algorithm for solving MAXSAT.

---

```

1 MaxHS ( $\mathcal{F}$ )
2  $\mathcal{K} = \emptyset$ 
3 while true do
4    $hs = \text{FindMinCostHittingSet}(\mathcal{K})$ 
5    $(\text{sat?}, \kappa) = \text{SatSolver}(\mathcal{F} \setminus hs)$ 
      // If SAT,  $\kappa$  contains the satisfying truth assignment.
      // If UNSAT,  $\kappa$  is a new core.
6   if  $\text{sat?}$  then
7     break // Exit While Loop,  $\kappa$  is a MAXSAT solution.
8    $\mathcal{K} = \mathcal{K} \cup \{\kappa\}$ 
9 return  $(\kappa, \text{cost}(\kappa))$ 

```

---

a minimum cost hitting set  $hs$  of  $\mathcal{K}$ , on line 4. If removing  $hs$  from  $\mathcal{F}$  results in a satisfiable formula (tested by the SAT solver on line 5), we break out of the loop on line 7 and return the satisfying assignment as the MAXSAT solution on line 9. Otherwise, the SAT solver will return a new core,  $\kappa$  to add to  $\mathcal{K}$  on line 8 and the loop repeats.

The correctness of Algorithm 1 is established by the following theorem.

**Theorem 1.** [6] *If  $\mathcal{K}$  is a set of cores for the MAXSAT problem  $\mathcal{F}$ ,  $hs$  is a minimum cost hitting set of  $\mathcal{K}$ , and  $\pi$  is a truth assignment satisfying  $\mathcal{F} \setminus hs$  then  $\text{mincost}(\mathcal{F}) = \text{cost}(\pi) = \text{cost}(hs)$ .*

This theorem shows that when Algorithm 1 breaks out of its loop,  $\kappa$  is a MAXSAT solution. The argument that the loop must eventually terminate is based on observing that every time the SAT solver returns a core  $\kappa$ , it must be distinct from all previously returned cores (because a hitting set for all previous cores does not hit  $\kappa$ ). Since there is a finite number of distinct cores, the SAT solver must eventually be unable to find another new core and the loop will terminate.

### 3.1 Behaviour of MAXHS

The behaviour of MAXHS is influenced by three potential sources of exponential complexity. These include the time required by the SAT solver to solve  $\mathcal{F} \setminus hs$ , the time required by the MIP solver to solve the NP-hard MCHS problem, and the number of iterations required. The examples below illustrate that each of these factors can, in the worst case, cause exponential runtime.

*Example 1.* Let  $\mathcal{F}$  be an instance of the Pigeon Hole Principle, where all clauses are considered soft with weight 1. Removing any single clause from  $\mathcal{F}$  will make the remaining clauses satisfiable. Therefore, MAXHS will terminate after the first core is found. So only one MCHS problem will be solved, and it is trivial. However, the time spent by the SAT solver to find a single core will be exponential.

*Example 2.* Let  $\mathcal{K}$  be a MCHS instance. We construct a MAXSAT instance  $\mathcal{F}$  that is equivalent to  $\mathcal{K}$  as follows. For each set  $\kappa \in \mathcal{K}$ , where  $\kappa = \{e_1, \dots, e_k\}$ , there is a hard clause  $(e_1 \vee \dots \vee e_k)$ . Finally, there is a soft clause  $(\neg e)$  with weight  $wt(e)$  for each element  $e \in \bigcup_{\kappa \in \mathcal{K}} \kappa$ . A minimal core is a core such that any proper subset is not a core. It is easy to see that the minimal cores of  $\mathcal{F}$  correspond to the hard clauses of  $\mathcal{F}$  and therefore the total number of minimal cores is equal to  $|\mathcal{K}|$ . The SAT solver can find each of the minimal cores in polynomial time, by using unit propagation alone. The number of minimal cores required by MAXHS is at most  $|\mathcal{K}|$ . So the only possible source of exponential runtime on  $\mathcal{K}$  is solving the MCHS problems. Assuming that  $P \neq NP$ , there must be some MCHS instance  $\mathcal{K}$  on which MAXHS will take exponential time and this must arise when MAXHS solves the MCHS problem.

To show that exponential run time can be generated from the number of iterations required we need the following proposition.

**Proposition 1.** *Let  $n$  be an even number and let  $E = \{e_1, \dots, e_n\}$  be a universe of equally weighted elements. Let  $\mathcal{K}_{n,r} = \{\kappa \subseteq E : |\kappa| = r\}$  be an instance of the MCHS problem where  $r = \frac{n}{2}$ . Let  $\mathcal{K}' = \mathcal{K}_{n,r} \setminus \kappa'$  for some  $\kappa' \in \mathcal{K}_{n,r}$ . Then the MCHS of  $\mathcal{K}'$  is strictly smaller than the MCHS of  $\mathcal{K}_{n,r}$ .*

*Example 3.* Let  $\mathcal{F}$  be a MAXSAT instance with an even number  $n$  of soft unit clauses with weight 1,  $(x_1), \dots, (x_n)$  and let the hard clauses of  $\mathcal{F}$  form a CNF encoding of the cardinality constraint  $\sum_{i=1}^n x_i < n/2$ . On this family of problems, an exponential number of cores will always be required by MAXHS, as we explain next. The solutions to  $\mathcal{F}$  are the truth assignments that set as many of the variables to *true* as possible without violating the hard cardinality constraint. Thus a solution to  $\mathcal{F}$  will set exactly  $\frac{n}{2} - 1$  of the  $x_i$  variables to *true* and the rest to *false*, and  $\frac{n}{2} + 1$  is the optimal cost. Any subset of the  $n$  soft clauses, with size greater than or equal to  $\frac{n}{2}$ , is a core of  $\mathcal{F}$ . Therefore,  $\mathcal{F}$  has at least  $\binom{n}{n/2}$  cores. By Proposition 1, for any number of cores  $k < \binom{n}{n/2}$ , the cost of their MCHS is less than the optimum. Therefore, MAXHS will require at least  $\binom{n}{n/2}$  cores, which is exponential in  $n$ .

However, our empirical observations are much more encouraging. In practice, we find that the SAT solving time is typically small.<sup>3</sup> Instead, the performance of MAXHS is most affected by the number of iterations and the time to solve the MCHS problems. Histograms of the percentage of total runtime spent by the SAT solver and the MIP solver are shown in Figure 1 over a set of 4502 Industrial and Crafted instances (the details of the experimental setup are described in Section 6). In order to study the baseline behaviour of Algorithm 1, the improvements presented in prior work [6] are omitted from this implementation. We observe in Figures 1b and 1d that on instances MAXHS failed to solve within

<sup>3</sup> If a MAXSAT instance is difficult for a state-of-the-art SAT solver to refute, then any MAXSAT solver that uses a sequence of SAT instance approach will be unable to solve it efficiently.

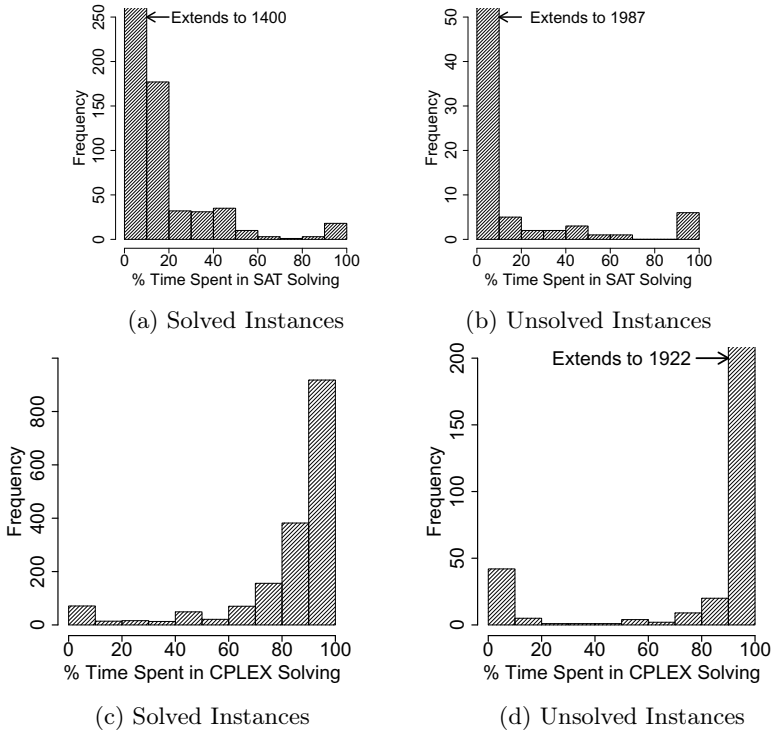


**Fig. 1.** Histograms over 4502 instances of the percentage of runtime spent in SAT solving and in calls to the MIP solver CPLEX, for Algorithm 1

the resource limits, the time spent by CPLEX is a much larger proportion of the total runtime than the time spent on SAT solving. This is true of the solved instances as well, as shown in Figures 1a and 1c. Thus we are motivated to find ways to reduce the time spent solving the MCHS problems, since this has the greatest potential to reduce the total runtime and thus allow more instances to be solved.

## 4 Postponing Optimization

We have seen in the previous section that in practice, the execution time of Algorithm 1 is dominated by its multiple calls to the MIP solver. The MIP solver must optimize an NP-hard problem, Minimum Cost Hitting Set, at each iteration. Therefore, in order to improve the performance of MAXHS it is natural to ask if an approximation to MCHS can ever be used instead. In this section we show that by applying a similar approach to [16] the MAXHS algorithm can be modified to use such approximations, in order to postpone the expensive calls to the MIP solver.

---

**Algorithm 2.** An algorithm for solving MAXSAT that uses non-optimal hitting sets.

---

```

1 MaxHS-nonOPT ( $\mathcal{F}$ )
2  $\mathcal{K} = \text{DisjointCores}(\mathcal{F})$ 
3 while true do
4    $hs = \text{FindMinCostHittingSet}(\mathcal{K})$  /* Find optimal solution */
5    $(\text{sat?}, \kappa) = \text{SatSolver}(\mathcal{F} \setminus hs)$ 
6   // If SAT,  $\kappa$  contains the satisfying truth assignment.
7   // If UNSAT,  $\kappa$  is a new core.
8   if sat? then
9     break // Exit While Loop,  $\kappa$  is a MAXSAT solution.
10     $\kappa = \text{Minimize}(\kappa)$ 
11     $\mathcal{K} = \mathcal{K} \cup \{\kappa\}$ 
12     $\text{nonOptLevel} = 0$ 
13    // Begin a series of non-optimal solutions
14    while true do
15      switch nonOptLevel do
16        case 0
17           $hs = \text{FindIncrementalHittingSet}(\mathcal{K}, \kappa, hs)$ 
18        case 1
19           $hs = \text{FindGreedyHittingSet}(\mathcal{K})$ 
20       $(\text{sat?}, \kappa) = \text{SatSolver}(\mathcal{F} \setminus hs)$ 
21      if sat? then
22        switch nonOptLevel do
23          case 0
24             $\text{nonOptLevel} = 1$ 
25          case 1
26            break /* Exit inner while loop */
27        else
28           $\kappa = \text{Minimize}(\kappa)$ 
29           $\mathcal{K} = \mathcal{K} \cup \{\kappa\}$ 
30           $\text{nonOptLevel} = 0$ 
31    return  $(\kappa, \text{cost}(\kappa))$ 

```

---

The MAXHS algorithm from Algorithm 1 can be modified to use non-optimal hitting set computations, as shown in Algorithm 2. The algorithm operates just like MAXHS in that it terminates at line 7 if  $\mathcal{F}$  is satisfiable after removing from it an **optimal** hitting set (computed at line 4). It varies from Algorithm 1 in that if a new core  $\kappa$  is discovered at line 5, it enters an inner loop where non-optimal hitting sets are used in place of optimal ones.

A simpler version of the algorithm uses only one method for computing approximate hitting sets rather than two. This version, which we explain first, is obtained from the version shown, by (1) replacing the **switch** statement on lines 12–16 by only one of lines 14 or 16 (i.e., we perform a single type of approximate hitting set computation); and (2) replacing the **switch** statement on lines 19–23 with a **break** statement. The resultant simpler version repeatedly finds an approximate hitting set and calls the SAT solver again to find a new core.



Eventually, the SAT solver will fail to find any more cores, the inner loop will be terminated, and we will return to line 4 where an optimal hitting set will then be computed. The simple version thus finds as many cores as possible before optimizing.

The more complex version (as shown) uses two levels of approximation: incremental and greedy. It is assumed that the second level (on line 16) computes a better (i.e., smaller) hitting set than the first. The idea here is that we compute a cheap incremental approximate hitting set until the SAT solver can't find any more cores. Then we compute the more expensive greedy approximate hitting set, which because it can be smaller might allow the SAT solver to find a new core. If a new core is found, we continue with the cheap incremental approximation (on line 27 *nonOptLevel* is reset to zero), until we once again fail to find cores with the SAT solver. If the SAT solver fails to find a new core even when using the more expensive greedy approximate hitting set (line 23), the inner while loop terminates and we finally return to line 4 to compute an optimal hitting set.<sup>4</sup>

Two other improvements to the basic Algorithm 1 are worth mentioning (originally presented in [7]). First, at line 2, we can use the SAT solver to find a set of disjoint cores. This is accomplished by blocking every clause in the cores found so far (by setting the *b*-variables for the cores' clauses to *true*) and finding another core: the new core will not have any clauses in common with the previous cores. This can only be done at the start before the main loop finds other cores. Second, at lines 8 and 25, after each core is found we use the SAT solver to minimize it. This is accomplished by using a simple minimal unsatisfiable core (MUS) algorithm [17]. This results in a stronger constraint for the MIP solver.<sup>5</sup>

In our implementation we use two different methods for computing approximate hitting sets, both of which are very cheap. **FindIncrementalHittingSet** (line 14), simply adds a clause in the newest core to the current hitting set. The chosen clause can be any clause in  $\kappa$ : we choose the clause that appears most frequently in the set  $\mathcal{K}$  of cores found so far. The intuition for this policy is that it takes away clauses that appear in many known cores, so that the next cores found can not use these clauses and thus are more likely to intersect with only a few of the known cores. The second method of computing non-optimal hitting sets, **FindGreedyHittingSet** (line 16), ignores the current hitting set, and instead applies a standard greedy algorithm for the MCHS problem [12].

**Theorem 2.** *Algorithm 2 returns a solution to the MAXSAT problem  $\mathcal{F}$ .*

*Proof.* This proof relies on the same argument as the correctness of Algorithm 1. If the algorithm returns on line 28, it must have broken out of the outer while loop at line 7. In this case,  $\kappa$  is a solution of  $\mathcal{F} \setminus hs$  (by line 5), where  $hs$  is a minimum cost hitting set of  $\mathcal{K}$  (by line 4).  $\mathcal{K}$  is a collection of cores of  $\mathcal{F}$ : it is initialized on line 2, and augmented only on lines 9 and 26 with  $\kappa$ , a core of  $\mathcal{F} \setminus hs$  (thus

<sup>4</sup> As can be seen from the algorithm specification we could add more cases to the **switch** statements if we had multiple approximation algorithms we wished to use.

<sup>5</sup> There are future possibilities for using upper and lower bounds in the algorithm (e.g., at line 17 if we find a satisfying solution its cost is an upper bound).

$\kappa$  is also a core of  $\mathcal{F}$ ). Therefore, if the algorithm returns, by Theorem 1,  $\kappa$  is a MAXSAT solution. It remains to show that the algorithm eventually terminates. Each time SatSolver is called (line 5 or line 17),  $hs$  is a hitting set of all cores in  $\mathcal{K}$ . So if SatSolver returns a core  $\kappa \subseteq \mathcal{F} \setminus hs$  it must be distinct from all cores in  $\mathcal{K}$ . There is a finite number of cores, so SatSolver can not return  $sat? = false$  forever and therefore both while loops must eventually terminate.

## 5 Additional Enhancements

In previous work we also investigated an alternative approach to reduce the time MAXHS spends in MCHS solving, that was based on using more general, **non-core**, constraints [7]. In [7] it is shown that a very effective technique is to **seed** CPLEX with many non-core constraints as a preprocessing step. In this section we show how seeding can also be applied before Algorithm 2 in order to achieve the same benefit.

**Definition 3.** *A non-core constraint for MAXSAT instance  $\mathcal{F}$  is a linear inequality constraint over  $b$ -variables,  $c$ , such that  $\mathcal{F}_{eq}^b \models c$ .*

It is sound to add non-core constraints to the MIP model in the MAXHS algorithm. This is easy to see from Proposition 2, which states that a minimum  $bcost$  solution to  $\mathcal{F}_{eq}^b$  corresponds to a MAXSAT solution, and the fact that the non-core constraints are entailed by  $\mathcal{F}_{eq}^b$ . We obtain a theorem similar to Theorem 1.

**Theorem 3.** *If  $\mathcal{K}$  is a set of core and non-core constraints for the MAXSAT problem  $\mathcal{F}$ ,  $\pi$  is minimum  $bcost$  assignment to the  $b$ -variables that satisfies  $\mathcal{K}$ , and  $\pi'$  is a truth assignment extending  $\pi$  and satisfying  $\mathcal{F}^b$  then  $mincost(\mathcal{F}) = bcost(\pi)$  and  $\pi'$  restricted to the variables of  $\mathcal{F}$  is a MAXSAT solution.*

*Proof.* Since  $\pi$  is a minimum  $bcost$  assignment to the  $b$ -variables that satisfies  $\mathcal{K}$ , and all constraints in  $\mathcal{K}$  are entailed by  $\mathcal{F}_{eq}^b$ , by Proposition 2  $mincost(\mathcal{F}) \geq bcost(\pi)$ . On the other hand,  $\pi'$  extends  $\pi$  to a satisfying assignment of  $\mathcal{F}^b$ , and since  $\pi'$  sets the same  $b$ -variables as  $\pi$ , we have  $bcost(\pi') = bcost(\pi)$ . So by Proposition 1,  $mincost(\mathcal{F}) \leq bcost(\pi') = bcost(\pi)$ . Thus  $mincost(\mathcal{F}) = bcost(\pi)$ . Finally,  $\pi'$  restricted to the variables of  $\mathcal{F}$  is a MAXSAT solution by Proposition 1, since  $\pi'$  is a minimum  $bcost$  satisfying assignment of  $\mathcal{F}^b$ .

This theorem allows us to modify Algorithm 2 by adding a preprocessing step that identifies a collection of non-core constraints  $\mathcal{N}$  as shown in Algorithm 3. Now, when the MIP solver is invoked to find an optimal solution, it no longer solves a pure MCHS problem because it must take into account the seeded non-core constraints  $\mathcal{N}$  in addition to the cores  $\mathcal{K}$ . On line 6, the MIP solver returns an optimal assignment to the  $b$ -variables,  $\mathcal{A}$ , that is then passed as a set of assumptions to the SAT solver on line 7. Note that the SAT solver uses  $\mathcal{F}^b$  as input which allows the settings of the  $b$ -variables in  $\mathcal{A}$  to relax the right set of soft clauses. By Theorem 3, if the SAT solver returns a satisfying assignment it

---

**Algorithm 3.** An algorithm for solving MAXSAT that uses non-optimal hitting sets and seeded non-core constraints.

---

```

1 MaxHS-nonOPT-seed ( $\mathcal{F}$ )
2  $\mathcal{K} = \text{DisjointCores}(\mathcal{F})$ 
3  $\mathcal{N} = \text{NonCoreConstraints}(\mathcal{F}_{eq}^b)$ 
4  $obj = wt(c_i) * b_i + \dots + wt(c_k) * b_k$ 
5 while true do
6    $\mathcal{A} = \text{Optimize}(\mathcal{K} \cup \mathcal{N}, obj)$  /* Find optimal solution */
7    $(sat?, \kappa) = \text{AssumptionSatSolver}(\mathcal{F}^b, \mathcal{A})$ 
   // Subsequent lines identical to Algorithm 2 lines 6–28

```

---

corresponds to the MAXSAT solution. Otherwise, the SAT solver will return a new **core** constraint  $\kappa$ . Thus once the main loop begins, no more non-core constraints will be derived (this differs from the previous work on non-core constraints) and the rest of the algorithm proceeds as before. The argument that the algorithm terminates remains the same as well.

**Theorem 4.** *Algorithm 3 returns a solution to the MAXSAT problem  $\mathcal{F}$ .*

It remains to specify how a collection of non-core constraints are to be found by NonCoreConstraints. We use Eq-Seeding [7] because it was found to be the most effective overall. In Eq-Seeding, we exploit the equivalence between original literals of  $\mathcal{F}$  that appear in soft unit clauses, and their  $b$ -variables. In  $\mathcal{F}_{eq}^b$ ,  $b_i \equiv \neg x$  if there is a unit soft clause  $(x) \in \mathcal{F}$ . So to generate a collection of constraints, we consider each clause  $c$  of  $\mathcal{F}^b$ , and check whether each literal in  $c$  has an equivalent  $b$ -literal (or is itself a  $b$ -literal). If so, we can derive a new  $b$ -variable constraint from  $c$  by replacing every original literal by its equivalent  $b$ -literal. This constraint is a clause over the  $b$ -variables that is entailed by  $\mathcal{F}_{eq}^b$  and it can be added to  $\mathcal{N}$ .

## 6 Experimental Results

We performed an empirical study of ten existing MAXSAT solvers: CPLEX (version 12.2), WPM1 [1], WPM2 (versions 1 and 2 [3]), BINCD [10], WBO [15], MINIMAXSAT [9], SAT4J [5], AKMAXSAT [13], MAXHS-Orig [6], and MAXHS+ [7]. All of these solvers are able to solve MAXSAT in its most general form, i.e., weighted partial MAXSAT, and thus have the widest range of applicability. Our study includes recently developed solvers utilizing a sequence of SAT approach (BINCD, WPM1, WPM2, MAXHS-Orig and MAXHS+), some older solvers (SAT4J and WBO), and two prominent Branch and Bound based solvers (AKMAXSAT and MINIMAXSAT). Also included is the MIP solver CPLEX, which is invoked after applying a standard translation of MAXSAT to MIP [7].

We experiment with three versions of Algorithm 2, that differ by how the non-optimal hitting sets are computed. The first and second versions use the

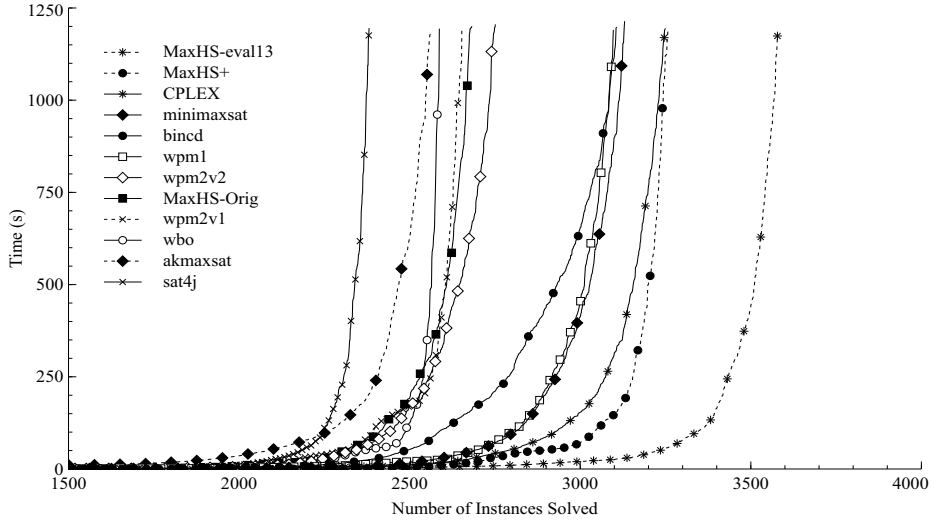


**Fig. 2.** Performance of solvers on all Crafted and Industrial instances

simple form of Algorithm 2 where only one method for computing approximate hitting sets is used. The first version, **MAXHS-incr**, uses `FindIncrementalHittingSet` for this computation, and the second version, **MAXHS-greedy**, uses `FindGreedyHittingSet`. The third version is called **MAXHS-incr-greedy** and it uses the more complex version of Algorithm 2, as specified in the pseudo-code. Finally, we also experiment with the version described in Section 5 that adds Eq-Seeding to **MAXHS-incr-greedy**. This version uses the same algorithm as the solver submitted to the 2013 MAXSAT Evaluation, and will be called **MAXHS-eval13**.<sup>6</sup>

We obtained **all** problems from the previous seven MAXSAT evaluations [4], discarding all instances in the Random category. After removing duplicate problems (as many as we could find) we ended up with 4502 problems divided into 58 families.<sup>7</sup> The family names and the number of instances in each family are shown in Tables 1 and 2. Our experiments were performed on 2.1 GHz AMD Opteron machines with 98GB RAM shared between 24 cores (about 4GB RAM per core). Each problem was run under a 1200 second timeout and with a memory limit of 2.5GB.

The overall results are shown in Figure 2. We see that the earliest **MAXHS-Orig** is a reasonable but not distinguished solver. The improvement presented in [7], **MAXHS+**, is a very good solver being slightly better over all problems than any previous solver. Finally, we see that the best of the versions developed

<sup>6</sup> All versions of **MAXHS** in this study use **MINISAT-2.0** and **CPLEX** version 12.2. The solver submitted to the 2013 Evaluation uses **CPLEX** version 12.5 and it performs slightly better than the version we report on here.

<sup>7</sup> We include results on 17 families within the Crafted category that were omitted in [7].

**Table 1.** Results for the Industrial category instances. Shows the number of instances solved by each solver in each benchmark family. The final row gives the total number of instances solved over both the Industrial and Crafted categories.

| Family               | #    | mini      | CPLEX      | wpm1       | bincd       | MAXHS      |            |            |            |            |             |
|----------------------|------|-----------|------------|------------|-------------|------------|------------|------------|------------|------------|-------------|
|                      |      |           |            |            |             | Orig       | +          | Alg. 2     |            |            | Alg. 3      |
|                      |      |           |            |            |             |            |            | incr       | grdy       | i+g        | eval 13     |
| ms/Safar             | 112  | 3         | 19         | <b>88</b>  | 71          | 75         | 33         | 33         | 34         | 33         | 33          |
| ms/circdebug         | 9    | 0         | 1          | <b>9</b>   | 7           | <b>9</b>   | 3          | 4          | 3          | 4          | 3           |
| pms/bcp-fir          | 59   | 13        | <b>58</b>  | 53         | 55          | 16         | 18         | 32         | 18         | 33         | 33          |
| pms/bcp-msp          | 148  | 108       | 110        | 60         | 117         | 62         | 121        | 90         | 87         | 95         | <b>123</b>  |
| pms/bcp-mtg          | 215  | 208       | 193        | <b>215</b> | <b>215</b>  | 150        | 212        | <b>215</b> | 214        | <b>215</b> | <b>215</b>  |
| pms/bcp-syn          | 74   | 27        | <b>71</b>  | 40         | 45          | 65         | <b>71</b>  | 69         | 69         | 69         | <b>71</b>   |
| pms/pb/logic-syn     | 17   | 2         | <b>16</b>  | 7          | 7           | <b>16</b>  | <b>16</b>  | <b>16</b>  | <b>16</b>  | <b>16</b>  | <b>16</b>   |
| pms/pbo-rout         | 15   | 14        | 14         | <b>15</b>  | <b>15</b>   | 10         | 13         | 12         | 10         | 10         | 13          |
| pms/pseudo/rout      | 15   | 14        | <b>15</b>  | <b>15</b>  | <b>15</b>   | 7          | <b>15</b>  | 11         | 11         | 10         | 12          |
| pms/circracecomp     | 4    | 1         | 0          | 3          | <b>4</b>    | 0          | 0          | 0          | 1          | 1          | 1           |
| pms/pb/primes        | 86   | 76        | 78         | 46         | 76          | 59         | <b>80</b>  | 77         | 74         | 78         | <b>81</b>   |
| pms/pb-nencdr        | 128  | 64        | 23         | 69         | 116         | 48         | 104        | 118        | <b>128</b> | <b>128</b> | 127         |
| pms/pb-nlogencdr     | 128  | 103       | 24         | 88         | <b>128</b>  | 78         | 111        | <b>128</b> | <b>128</b> | <b>128</b> | <b>128</b>  |
| pms/aes              | 7    | 1         | 2          | 0          | 1           | 1          | 2          | 2          | 2          | <b>3</b>   | 2           |
| pms/hap-asmby        | 6    | 0         | 2          | 4          | 0           | <b>5</b>   | <b>5</b>   | <b>5</b>   | <b>5</b>   | <b>5</b>   | <b>5</b>    |
| pms/bcp-hipp         | 1183 | 982       | 962        | 1154       | <b>1164</b> | 1125       | 1142       | 1141       | 1138       | 1137       | 1140        |
| wpms/haplo-ped       | 100  | 0         | 9          | <b>91</b>  | 23          | 27         | 28         | 26         | 33         | 25         | 22          |
| pms/protein-ins      | 12   | <b>11</b> | 1          | 1          | 2           | 1          | 1          | 2          | 1          | 2          | 2           |
| wpms/protein-ins     | 12   | <b>10</b> | 1          | 1          | 2           | 1          | 2          | 2          | 1          | 2          | 1           |
| wpms/timetabling     | 32   | 0         | 0          | <b>13</b>  | 12          | 9          | 8          | 7          | 7          | 7          | 6           |
| wpms/upgrade         | 100  | 0         | <b>100</b> | <b>100</b> | 97          | <b>100</b> | <b>100</b> | <b>100</b> | <b>100</b> | <b>100</b> | <b>100</b>  |
| wpms/up-u98          | 80   | 0         | <b>80</b>  | <b>80</b>  | 79          | <b>80</b>  | <b>80</b>  | <b>80</b>  | <b>80</b>  | <b>80</b>  | <b>80</b>   |
| Total                | 2542 | 1637      | 1779       | 2152       | <b>2251</b> | 1944       | 2165       | 2170       | 2160       | 2181       | 2214        |
| Indust + Craft Total | 4502 | 3130      | 3249       | 3097       | 3106        | 2682       | 3257       | 3288       | 3297       | 3419       | <b>3578</b> |

in this paper, MAXHS-eval13 achieves a significant performance improvement. Although not shown on the plot, the other versions we develop here all improve over MAXHS+, but are not as good as MAXHS-eval13 (see Tables 1 and 2).

The results are broken down by benchmark family in Tables 1 and 2. Included in the tables are the four competing solvers that performed best overall (as shown in Figure 2): CPLEX, MINIMAXSAT, BINCD and WPM1. Observe that all versions of MAXHS that use non-optimal hitting sets (Alg. 2 and Alg. 3) outperform MAXHS+, which uses non-core constraints more extensively than MAXHS-eval13. Yet when the technique of seeding the MIP model with non-core constraints is added to the use of non-optimal hitting sets, performance is improved (see “Alg. 2 i+g” vs. “Alg. 3 eval 13”). We see that on the industrial problems (Table 1) there is still quite a lot of variance in performance between the different solvers across the different families; that MAXHS-eval13 has fairly robust good performance across the different families; and that the MAXHS approach can be significantly better than just using CPLEX alone, indicating the value of our hybrid approach. On the crafted problems (Table 2) we see that the Branch and Bound approach of MINIMAXSAT is most effective. These problems tend to be smaller than the industrial problems and have

**Table 2.** Results for the Crafted category instances. Shows the number of instances solved by each solver in each benchmark family. The benchmark families with an asterisk (\*) are those we classify as having “random” structure [7]. The final row gives the total number of instances solved over both the Industrial and Crafted categories.

| Family               | #    | bincd      | wpm1      | CPLEX      | mini        | MAXHS      |            |            |            |            |             |
|----------------------|------|------------|-----------|------------|-------------|------------|------------|------------|------------|------------|-------------|
|                      |      |            |           |            |             | Orig       | +          | Alg. 2     |            |            | Alg. 3      |
|                      |      |            |           |            |             |            |            | incr       | grdy       | i+g        | eval 13     |
| *ms/spin             | 20   | 0          | 0         | 19         | <b>20</b>   | 0          | 0          | 1          | 4          | 10         | 10          |
| *ms/cut/spin         | 5    | 1          | 1         | <b>3</b>   | <b>3</b>    | 0          | 1          | 1          | 2          | 2          | 2           |
| *wms/cut/spin        | 5    | 0          | 0         | <b>4</b>   | <b>4</b>    | 0          | 1          | 1          | 2          | 3          | 3           |
| *pms/frb             | 25   | 0          | 0         | <b>9</b>   | 5           | 0          | 8          | 5          | 0          | 5          | <b>9</b>    |
| *wms/kexu/frb        | 35   | 9          | 5         | <b>20</b>  | 15          | 10         | <b>20</b>  | 16         | 15         | 15         | <b>20</b>   |
| *pms/csp/sprsls      | 20   | <b>20</b>  | <b>20</b> | <b>20</b>  | <b>20</b>   | 19         | 11         | <b>20</b>  | <b>20</b>  | <b>20</b>  | <b>20</b>   |
| *pms/csp/dsls        | 20   | <b>20</b>  | 16        | <b>20</b>  | <b>20</b>   | 5          | 0          | 15         | 15         | 16         | 16          |
| *pms/csp/sprstgt     | 20   | <b>20</b>  | <b>20</b> | <b>20</b>  | <b>20</b>   | 0          | 0          | 15         | 12         | <b>20</b>  | <b>20</b>   |
| *pms/csp/dstgt       | 20   | 19         | 19        | <b>20</b>  | <b>20</b>   | 0          | 0          | 3          | 6          | <b>20</b>  | <b>20</b>   |
| *ms/ramsey           | 48   | 34         | 34        | 34         | <b>35</b>   | 34         | 34         | 34         | 34         | 34         | 34          |
| *wms/ramsey          | 48   | 36         | 34        | 36         | <b>37</b>   | 34         | 35         | 34         | 34         | 34         | 34          |
| *pms/clq/rand        | 96   | 67         | 0         | <b>96</b>  | <b>96</b>   | 4          | <b>96</b>  | 4          | 44         | 59         | <b>96</b>   |
| *ms/bcut-630         | 100  | 0          | 0         | 0          | <b>83</b>   | 0          | 0          | 0          | 0          | 0          | 0           |
| *pms/kbtree          | 54   | 15         | 14        | <b>54</b>  | 22          | 11         | 15         | 12         | 12         | 12         | 14          |
| *pms/max1/3sat       | 80   | <b>80</b>  | 71        | <b>80</b>  | <b>80</b>   | 20         | <b>80</b>  | 45         | 44         | 57         | <b>80</b>   |
| *ms/cut/rand         | 40   | 0          | 0         | 4          | <b>40</b>   | 0          | 0          | 0          | 0          | 0          | 0           |
| *wms/cut/rand        | 40   | 0          | 0         | 12         | <b>40</b>   | 0          | 0          | 0          | 0          | 0          | 0           |
| ms/cut/dimacs        | 62   | 6          | 5         | 20         | <b>48</b>   | 4          | 4          | 4          | 4          | 4          | 3           |
| wms/cut/dimacs       | 62   | 4          | 5         | 22         | <b>55</b>   | 3          | 3          | 4          | 5          | 8          | 9           |
| pms/max1/struc       | 60   | 59         | 30        | 52         | <b>60</b>   | 5          | <b>60</b>  | 54         | 57         | <b>60</b>  | <b>60</b>   |
| pms/cliique/struc    | 62   | 18         | 8         | 32         | <b>36</b>   | 10         | 29         | 12         | 17         | 17         | 34          |
| pms/queens           | 7    | <b>7</b>   | <b>7</b>  | <b>7</b>   | <b>7</b>    | 2          | 3          | 5          | 4          | 5          | 5           |
| wpms/QCP             | 25   | <b>25</b>  | <b>25</b> | <b>25</b>  | 20          | <b>25</b>  | <b>25</b>  | <b>25</b>  | <b>25</b>  | <b>25</b>  | 23          |
| pms/pb/gardn         | 7    | 5          | 5         | <b>6</b>   | 5           | 5          | <b>6</b>   | 5          | <b>6</b>   | <b>6</b>   | <b>6</b>    |
| wpms/pb/mip          | 16   | <b>7</b>   | 6         | 6          | 5           | 6          | <b>7</b>   | <b>7</b>   | <b>7</b>   | <b>7</b>   | 5           |
| wpms/pb/factor       | 186  | <b>186</b> | 168       | <b>186</b> | <b>186</b>  | <b>186</b> | <b>186</b> | <b>186</b> | <b>186</b> | <b>186</b> | 172         |
| wpms/KnotPip         | 350  | 0          | 161       | 245        | 117         | 57         | 52         | <b>290</b> | 260        | <b>290</b> | 289         |
| wpms/spot5log        | 21   | 11         | <b>12</b> | 6          | 4           | 6          | 6          | 6          | 6          | 6          | 6           |
| wpms/spot5dir        | 21   | 11         | 10        | <b>17</b>  | 3           | 6          | 6          | 6          | 6          | 6          | 6           |
| pms/jobshop          | 4    | <b>4</b>   | 3         | 0          | 2           | 4          | 3          | <b>4</b>   | <b>4</b>   | <b>4</b>   | <b>4</b>    |
| wpms/plan            | 71   | 65         | 64        | 70         | <b>71</b>   | 46         | <b>71</b>  | <b>71</b>  | <b>71</b>  | <b>71</b>  | 61          |
| wpms/aucreg          | 84   | 6          | 0         | <b>84</b>  | <b>84</b>   | 34         | <b>84</b>  | 4          | 13         | 2          | 76          |
| wpms/aucsch          | 84   | 66         | <b>84</b> | <b>84</b>  | <b>84</b>   | 82         | <b>84</b>  | 76         | 77         | 78         | 75          |
| wpms/aucpath         | 88   | 0          | 52        | <b>88</b>  | <b>88</b>   | <b>88</b>  | <b>88</b>  | <b>88</b>  | <b>88</b>  | <b>88</b>  | 78          |
| wpms/ware            | 18   | 1          | 14        | <b>18</b>  | 2           | 1          | <b>18</b>  | 9          | 1          | 12         | <b>18</b>   |
| wpms/plan            | 56   | 53         | 52        | 51         | <b>56</b>   | 31         | <b>56</b>  | <b>56</b>  | <b>56</b>  | <b>56</b>  | <b>56</b>   |
| Total                | 1960 | 855        | 945       | 1470       | <b>1493</b> | 738        | 1092       | 1118       | 1137       | 1238       | 1364        |
| Indust + Craft Total | 4502 | 3106       | 3097      | 3249       | 3130        | 2682       | 3257       | 3288       | 3297       | 3419       | <b>3578</b> |

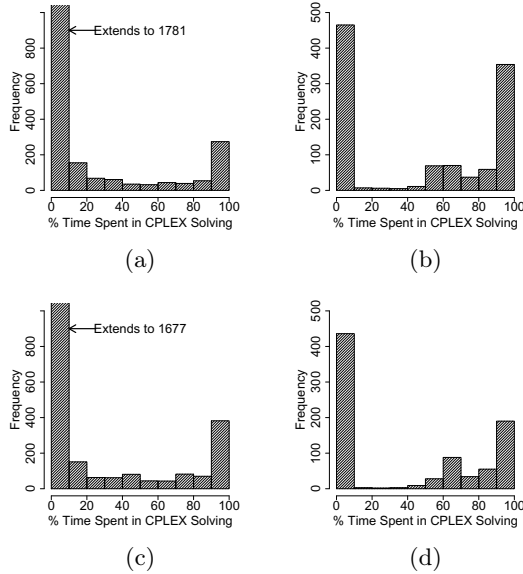


**Fig. 3.** Histograms over all instances for the percentage of runtime spent in calls to CPLEX for Algorithms 2 and 3. (a) Alg. 2, unsolved instances; (b) Alg. 2, solved instances; (c) Alg. 3, solved instances; (d) Alg. 3, unsolved instances.

tightly interacting variables yielding cores containing a large fraction of the total clauses. The data also shows that although the traditional sequence of SAT solvers BINCD and WPM1 do not perform particularly well on these problems, CPLEX is quite effective, as is the hybrid approach of MAXHS.

The good performance of MAXHS-incr-greedy and MAXHS-eval13, implementing Algorithms 2 and 3 respectively, appears to be due to a significant reduction in the total time spent by CPLEX. In Figure 3 we show the percentage of the total runtime that was spent in calls to CPLEX. Comparing these histograms to those in Figure 1, we observe that the time spent solving the MCHS problems to optimality is now almost always a low percentage of the total runtime. The number of calls to CPLEX generally decreases when we use non-optimal hitting sets, as expected. On average (over all 4502 instances), each run of MAXHS-incr-greedy gave a total of 5419 constraints to CPLEX but solved the optimization problem only 14 times. In contrast, each run of Algorithm 1 gave on average only 972 cores to CPLEX, thus having to solve the MCHS problem 972 times.

In conclusion, we have presented a technique for improving the hybrid MAXHS approach to solving MAXSAT. Our method yields an improvement in the state-of-the-art for MAXSAT solving. Although our method successfully shifts the balance of the runtime away from the MIP solver, a promising avenue for future work is to examine the structure of the constraints given to the MIP solver to see if they could be made more effective.

**Acknowledgements.** This work has been partly funded by the “Agence nationale de la Recherche”, reference ANR-10-BLA-0214.

## References

1. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving SAT-based weighted MaxSAT solvers. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 86–101. Springer, Heidelberg (2012)
2. Ansótegui, C., Bonet, M.L., Levy, J.: Solving (weighted) partial MaxSAT through satisfiability testing. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 427–440. Springer, Heidelberg (2009)
3. Ansótegui, C., Bonet, M.L., Levy, J.: A new algorithm for weighted partial maxsat. In: Proceedings of the AAAI National Conference (AAAI), pp. 3–8 (2010)
4. Argelich, J., Li, C.M., Manyà, F., Planes, J.: The maxSAT evaluations (2007–2011), <http://www.maxsat.udl.cat>
5. Berre, D.L., Parrain, A.: The sat4j library, release 2.2. JSAT 7(2-3), 59–64 (2010)
6. 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)
7. Davies, J., Bacchus, F.: Exploiting the power of MIP solvers in MAXSAT. In: Järvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 166–181. Springer, Heidelberg (2013)
8. 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)
9. Heras, F., Larrosa, J., Oliveras, A.: Minimaxsat: An efficient weighted max-sat solver. *Journal of Artificial Intelligence Research (JAIR)* 31, 1–32 (2008)
10. Heras, F., Morgado, A., Marques-Silva, J.: Core-guided binary search algorithms for maximum satisfiability. In: Proceedings of the AAAI National Conference (AAAI), pp. 36–41 (2011)
11. Hooker, J.N.: Planning and scheduling by logic-based benders decomposition. *Operations Research* 55(3), 588–602 (2007)
12. Johnson, D.S.: Approximation algorithms for combinatorial problems. In: Symposium on Theory of Computing, pp. 38–49 (1973)
13. Kügel, A.: Improved exact solver for the weighted Max-SAT problem. In: Workshop on the Pragmatics of SAT (2010)
14. Li, C.M., Manyà, F., Mohamedou, N.O., Planes, J.: Resolution-based lower bounds in maxsat. *Constraints* 15(4), 456–484 (2010)
15. Manquinho, V., Marques-Silva, J., Planes, J.: Algorithms for weighted boolean optimization. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 495–508. Springer, Heidelberg (2009)
16. Moreno-Centeno, E., Karp, R.M.: The implicit hitting set approach to solve combinatorial optimization problems with an application to multigenome alignment. *Operations Research* 61(2), 453–468 (2013)
17. Marques-Silva, J., Lynce, I.: On improving MUS extraction algorithms. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 159–173. Springer, Heidelberg (2011)



# Dead-End Elimination for Weighted CSP

Simon de Givry<sup>1,2,\*</sup>, Steven D. Prestwich<sup>2</sup>, and Barry O’Sullivan<sup>2</sup>

<sup>1</sup> MIA-T, UR 875, INRA, F-31320 Castanet Tolosan, France

<sup>2</sup> Cork Constraint Computation Centre, University College Cork, Ireland  
degivry@toulouse.inra.fr, {s.prestwich,b.osullivan}@cs.ucc.ie

**Abstract.** Soft neighborhood substitutability (SNS) is a powerful technique to automatically detect and prune dominated solutions in combinatorial optimization. Recently, it has been shown in [26] that enforcing partial SNS (PSNS<sup>r</sup>) during search can be worthwhile in the context of Weighted Constraint Satisfaction Problems (WCSP). However, for some problems, especially with large domains, PSNS<sup>r</sup> is still too costly to enforce due to its worst-case time complexity in  $O(ned^4)$  for binary WCSP. We present a simplified dominance breaking constraint, called *restricted dead-end elimination* (DEE<sup>r</sup>), the worst-case time complexity of which is in  $O(ned^2)$ . Dead-end elimination was introduced in the context of computational biology as a preprocessing technique to reduce the search space [13, 14, 16, 17, 28, 30]. Our restriction involves testing only one pair of values per variable instead of all the pairs, with the possibility to prune several values at the same time. We further improve the original dead-end elimination criterion, keeping the same time and space complexity as DEE<sup>r</sup>. Our results show that maintaining DEE<sup>r</sup> during a depth-first branch and bound (DFBB) search is often faster than maintaining PSNS<sup>r</sup> and always faster than or similar to DFBB alone.

**Keywords:** combinatorial optimization, dominance rule, weighted constraint satisfaction problem, soft neighborhood substitutability.

## 1 Introduction

Pruning by dominance in the context of combinatorial optimization involves reducing the solution space of a problem by adding new constraints to it [19]. We study dominance rules that reduce the domains of variables based on optimality considerations (in relation to the optimization of an objective function). The idea is to automatically detect values in the domain of a variable that are *dominated* by another *dominant* value of the domain such that any solution using the dominant value instead of the dominated ones has a better score. Various dominance rules have been studied recently by the Constraint Programming community [5, 6, 26]. In particular, *soft neighborhood substitutability* (SNS) [3, 26] allows us to detect dominated values in polynomial time under specific conditions for Weighted Constraint Satisfaction Problems (WCSP). In a different community,

---

\* This work has been partly funded by the “Agence nationale de la Recherche”, reference ANR-10-BLA-0214 and the European Union, reference FP7 *ePolicy* 288147.

similar dominance rules and others, called *dead-end elimination (DEE) criteria*, have been studied for many years in the context of *computational protein design* [1, 13, 14, 16, 17, 28, 30]. However to the best of our knowledge, these criteria have never been used during search, possibly due to their high computational cost. Following the work done in [26] showing the interest of maintaining such dominance rule during search, we propose a faster pruning by dominance algorithm combining SNS and DEE in a partial and optimistic way.

## 2 Weighted Constraint Satisfaction Problems

A Weighted Constraint Satisfaction Problem (WCSP)  $P$  is a triplet  $P = (X, F, k)$  where  $X$  is a set of  $n$  variables and  $F$  a set of  $e$  cost functions. Each variable  $x \in X$  has a finite domain,  $\text{domain}(x)$ , of values that can be assigned to it. The maximum domain size is denoted by  $d$ . For a set of variables  $S \subseteq X$ ,  $l(S)$  denotes the set of all labelings of  $S$ , i.e., the Cartesian product of the domain of the variables in  $S$ . For a given tuple of values  $t$ ,  $t[S]$  denotes the projection of  $t$  over  $S$ . A cost function  $f_S \in F$ , with scope  $S \subseteq X$ , is a function  $f_S : l(S) \mapsto [0, k]$  where  $k$  is a maximum integer cost used for forbidden assignments. A cost function over one (resp. zero) variable is called a *unary* (resp. *nullary*, i.e., a constant cost paid by any assignment) cost function, denoted either by  $f_{\{x\}}$  or  $f_x$  (resp. by  $f_\emptyset$ ). We denote by  $\Gamma(x)$  the set of cost functions on variable  $x$ , i.e.,  $\Gamma(x) = \{f_S \in F \mid \{x\} \subseteq S\}$ .

The Weighted Constraint Satisfaction Problem consists in finding a complete assignment  $t$  minimizing the combined (*sum*) cost function  $\sum_{f_S \in F} f_S(t[S])$ . This optimization problem has an associated NP-complete decision problem.

Enforcing a given local consistency property on a problem  $P$  involves transforming  $P = (X, F, k)$  into a problem  $P' = (X, F', k)$  that is equivalent to  $P$  (all complete assignments keep the same cost) and that satisfies the considered local consistency property. This enforcing may increase  $f_\emptyset$  and provide an improved lower bound on the optimal cost. It is achieved using *Equivalence Preserving Transformations* (EPTs) that move costs between different scopes [8–12, 21, 22, 24, 31]. In particular, node consistency [21] (NC) satisfies  $\forall x \in X, \min_{a \in \text{domain}(x)} f_x(a) = 0, \forall a \in \text{domain}(x), f_\emptyset + f_x(a) < k$ . Soft arc consistency (AC\*) [21, 31] satisfies NC and  $\forall f_S \in F, \forall x \in S, \forall a \in \text{domain}(x), \min_{t \in l(S \setminus \{x\})} f_S(t \cup \{(x, a)\}) = 0$ .

## 3 Dead-End Elimination

The original dead-end elimination criterion is [14]:

$$\sum_{f_S \in \Gamma(x)} \max_{t \in l(S \setminus \{x\})} f_S(t \cup \{(x, a)\}) \leq \sum_{f_S \in \Gamma(x)} \min_{t \in l(S \setminus \{x\})} f_S(t \cup \{(x, b)\}). \quad (1)$$

This condition implies that value  $b$  can be safely removed from the domain of  $x$  since the total cost of all the cost functions on  $x$  taking their best assignment with  $x$  assigned  $b$  is still worse than that produced by their worst assignment with  $x$  assigned  $a$ . This condition was further improved in [17]:

$$\sum_{f_S \in \Gamma(x)} \max_{t \in l(S \setminus \{x\})} f_S(t \cup \{(x, a)\}) - f_S(t \cup \{(x, b)\}) \leq 0. \quad (2)$$

where the best and worst-cases are replaced by the worst difference in costs for any labeling of the remaining variables in the scope of each cost function. It is easy to see that this condition is always stronger than the previous one.

More recently, the authors in [26] reformulated Equation 2 in the specific context of WCSP with bounded cost addition  $a \oplus b = \min(k, a + b)$  proving that the reformulated criterion<sup>1</sup> is equivalent to soft neighborhood substitutability when  $\Gamma(x)$  is separable (i.e.,  $\forall f_S, f_{S'} \in \Gamma(x) \times \Gamma(x), S \cap S' = \{x\}$ ) and  $\alpha < k$ . In practice, testing Eq. 2 or its reformulation will prune the same values.

They also noticed that if the problem is soft arc consistent then the worst-cost differences are always positive. Equation 1 can be further simplified thanks to soft AC because all the best-case terms are precisely equal to zero:

$$\sum_{f_S \in \Gamma(x)} \max_{t \in l(S \setminus \{x\})} f_S(t \cup \{(x, a)\}) \leq f_x(b). \quad (3)$$

We propose a stronger condition than Eq. 2 or Eq. 3 by discarding forbidden partial assignments with  $x$  assigned  $b$  when computing the worst-cost difference:

$$\sum_{f_S \in \Gamma(x)} \max_{t \in l(S \setminus \{x\}) \text{ st. } C(f_S, t \cup \{(x, b)\}) < k} f_S(t \cup \{(x, a)\}) - f_S(t \cup \{(x, b)\}) \leq 0. \quad (4)$$

where  $C(f_S, t) = f_S(t) + \sum_{y \in S, |S| > 1} f_y(t[y]) + f_\emptyset$ . This new condition is equivalent to Eq. 2 except that some tuples have been discarded from the max operation. These discarded tuples  $t$  are forbidden partial assignments when  $x$  is assigned  $b$  because the sum of the associated cost function  $f_S(t \cup \{(x, b)\})$  plus, if  $|S| > 1$ , all the unary costs on the variables in  $S$  assigned by  $t \cup \{(x, b)\}$  plus the current lower bound  $f_\emptyset$  is greater than or equal to the current upper bound  $k$ . Such tuples  $t$  do not need to be considered by the max operation because  $t \cup \{(x, b)\}$  does not belong to any optimal solution, whereas  $t \cup \{(x, a)\}$  can be.

For CSP (i.e.,  $k = 1$ ), Eq. 2 and Eq. 4 are both equivalent to neighborhood substitutability [15]. For Max-SAT, Eq. 3 and Eq. 2 are equivalent if the problem is soft AC, and correspond to the *Dominating 1-clause rule* [29]. In the general case, Eq. 4 is stronger<sup>2</sup> (more domain values can be pruned) than Eq. 2, which is

<sup>1</sup> They replace the maximum of cost differences  $\alpha - \beta$  by the opposite of the minimum of cost pairs  $(\beta, \alpha)$ , ordered by the relation  $(\beta, \alpha) \leq (\beta', \alpha') \equiv \beta - \alpha < \beta' - \alpha' \vee (\beta - \alpha = \beta' - \alpha' \wedge \alpha < \alpha')$ . Equation 2 becomes  $\sum_{f_S \in \Gamma(x) \cup f_x} \min_{t \in l(S \setminus \{x\})} (f_S(t \cup \{(x, b)\}) - f_S(t \cup \{(x, a)\})) \geq 0$  where  $(\beta, \alpha) \geq 0$  if  $\beta \geq \alpha$ .

<sup>2</sup> The definition of soft AC on fair VCSPs [12] makes Eq. 4 and Eq. 2 equivalent.

stronger than Eq. 3. More complex dominance criteria have been defined in the context of protein design (*e.g.*, a value being dominated by a set of values instead of a single one, see [30] for an overview), but they all incur higher computational costs. In the next section, we recall how to enforce Eq. 2 in WCSP, as originally shown in [26]. Then, in Section 5, we present a modified version to partially enforce the two conditions, Eq. 4 and 3, with a lower time complexity.

## 4 Enforcing Soft Neighborhood Substitutability

Assuming a soft arc consistent WCSP (see *e.g.*, W-AC\*2001 algorithm in [24]), enforcing partial<sup>3</sup> soft neighborhood substitutability (PSNS<sup>r</sup>) is described by Algorithm 1. For each variable  $x$ , all the pairs of values  $(a, b) \in \text{domain}(x) \times \text{domain}(x)$  with  $a < b$  are checked by the function **DominanceCheck** to see if  $b$  is dominated by  $a$  or, if not, vice versa (line 3). At most one dominated value is added to the value removal queue  $\Delta$  at each inner loop iteration (line 2). Removing dominated values (line 4) can make the problem arc inconsistent, requiring us to enforce soft arc consistency again. We successively enforce soft AC and PSNS<sup>r</sup> until no value removals are made by both enforcing algorithms.

---

**Algorithm 1:** Enforce PSNS<sup>r</sup> [26]

---

```

Procedure PSNSr( $P$ : AC* consistent WCSP)
   $\Delta := \emptyset$ ;
  1 foreach  $x \in \text{variables}(P)$  do
  2   foreach  $(a, b) \in \text{domain}(x) \times \text{domain}(x)$  such that  $a < b$  do
  3      $R := \text{DominanceCheck}(x, a \rightarrow b)$ ;
  4     if  $R = \emptyset$  then  $R := \text{DominanceCheck}(x, b \rightarrow a)$ ;
  5      $\Delta := \Delta \cup R$ ;
  6 foreach  $(x, a) \in \Delta$  do remove  $(x, a)$  from  $\text{domain}(x)$ ;

  /* Check if value  $a$  dominates value  $b$  */
  Function  $\text{DominanceCheck}(x, a \rightarrow b)$ : set of dominated values
  5 if  $f_x(a) > f_x(b)$  then return  $\emptyset$ ;
  6  $\delta_{a \rightarrow b} := f_x(a)$ ;
  7 foreach  $f_s \in F$  such that  $\{x\} \subset S$  do
  8    $\delta := \text{getDifference}(f_s, x, a \rightarrow b)$ ;
  9    $\delta_{a \rightarrow b} := \delta_{a \rightarrow b} + \delta$ ;
  10 if  $\delta_{a \rightarrow b} > f_x(b)$  then return  $\emptyset$ ;
  11 return  $\{(x, b)\}$  /*  $\delta_{a \rightarrow b} \leq f_x(b)$  */;

  /* Compute largest difference in costs when using  $a$  instead of  $b$  */
  Function  $\text{getDifference}(f_s, x, a \rightarrow b)$ : cost
  7  $\delta_{a \rightarrow b} := 0$ ;
  8 foreach  $t \in l(S \setminus \{x\})$  do
  9    $\delta_{a \rightarrow b} := \max(\delta_{a \rightarrow b}, f_s(t \cup \{(x, a)\}) - f_s(t \cup \{(x, b)\}))$ ;
  10 return  $\delta_{a \rightarrow b}$ ;

```

---

Function  $\text{DominanceCheck}(x, a \rightarrow b)$  computes the sum of worst-cost differences as defined by Equation 2 and returns a non-empty set containing value  $b$  if Eq. 2 is true, meaning that  $b$  is dominated by value  $a$ . It exploits early breaks as

<sup>3</sup> Enforcing complete soft neighborhood substitutability is co-NP hard as soon as  $k \neq +\infty$  (*i.e.*, no restriction on  $\alpha$  in the reformulated Equation 2).

soon as Eq. 2 can be falsified (lines 5 and 6). Worst-cost differences are computed by the function `getDifference`( $f_s, x, a \rightarrow b$ ) applied to every cost function related to  $x$ . Worst-cost differences are always positive (line 7) due to soft AC.

The worst-case time complexity of `getDifference` is  $O(d^{r-1})$  for WCSP with maximum arity  $r$ . `DominanceCheck` is  $O(qd^{r-1})$  where  $q = |\Gamma(x)|$ . Thus, the time complexity of one iteration of Algorithm 1 (PSNS<sup>r</sup>) is  $O(nd^2qd^{r-1} + nd) = O(ed^{r+1})$  where  $e = nq$ . Interleaving PSNS<sup>r</sup> and soft AC until a fixed point is reached is done at most  $nd$  times, resulting in a worst-case time complexity of PSNS<sup>r</sup> in  $O(ned^{r+2})$ . Its space complexity is  $O(nd^2)$  when using *residues* [26].

In the following, we always consider PSNS<sup>r</sup> using the better condition given by Equation 4 instead of Eq. 2. This does not change the previous complexities.

## 5 Enforcing Partial SNS and Dead-End Elimination

In order to reduce the time (and space) complexity of pruning by dominance, we test only one pair of values per variable. The new algorithm is described in Algorithm 2. We select the pair  $(a, b) \in \text{domain}(x) \times \text{domain}(x)$  in an optimistic way such that  $a$  is associated with the minimum unary cost and  $b$  to the maximum unary cost (lines 8 and 9). Because arc consistency also implies node consistency, we always have  $f_x(a) = 0$ .<sup>4</sup> When all the unary costs (including the maximum) are null (line 10), we select as  $b$  the maximum domain value (or its minimum if this value is already used by  $a$ ). By doing so, we should favor more pruning on max-closed or submodular subproblems<sup>5</sup>.

Instead of checking the new Equation 4 for the pair  $(a, b)$  alone, we also check Eq. 3 for all the pairs  $(a, u)$  such that  $u \in \text{domain}(x) \setminus \{a\}$ . This is done in the function `MultipleDominanceCheck` (lines 16 and 17). This function computes at the same time the sum of maximum costs  $ub_a$  for value  $a$  (lines 12 and 13) and the sum of worst-cost differences  $\delta_{a \rightarrow b}$  for the pair  $(a, b)$ . The new function `getDifference-Maximum`( $f_s, x, a \rightarrow b$ ) now returns the worst-cost difference, discarding forbidden assignments with  $t \cup \{(x, b)\}$  (line 18), as suggested by Eq. 4, and also the maximum cost in  $f_s$  for  $x$  assigned  $a$ . By construction of the two criteria, we have  $\delta_{a \rightarrow b} \leq ub_a$ , so the stopping condition is unchanged at line 14. When the maximum cost of a value is null for all its cost functions, we can directly remove all the other values in the domain avoiding any extra work (line 15). Finally, if the selected pair  $(a, b)$  *prunes*  $b$ , then a new pair is checked.

Notice that DEE<sup>r</sup> is equivalent to PSNS<sup>r</sup> on problems with Boolean variables, such as Weighted Max-SAT. For problems with non-Boolean domains, DEE<sup>r</sup> is still able to detect and prune several values per variable. Clearly, its time (resp. space) complexity is  $O(ned^r)$  (resp.  $O(n)$ ) using only one residue per variable), reducing by a factor  $d^2$  the time and space complexity compared to PSNS<sup>r</sup>.

<sup>4</sup> In fact, we set the value  $a$  to the *unary support* offered by NC [21] or EDAC [22].

<sup>5</sup> Assuming a problem with two variables  $x$  and  $y$  having the same domain and a single submodular cost function  $f(x, y) = 0$  if  $x \leq y$  else  $x - y$  or a single max-closed constraint  $x < y$ , then DEE<sup>r</sup> assigns  $x = \min(\text{domain}(x))$  and  $y = \max(\text{domain}(y))$ .

**Algorithm 2:** Enforce DEE<sup>r</sup>


---

```

Procedure DEEr( $P$ : AC* consistent WCSP)
   $\Delta := \emptyset$ ;
  foreach  $x \in \text{variables}(P)$  do
    8    $a := \arg \min_{u \in \text{domain}(x)} f_x(u)$ ;
    9    $b := \arg \max_{u \in \text{domain}(x)} f_x(u)$ ;
    10  if  $a = b$  /*  $\forall u \in \text{domain}(x), f_x(u) = 0$  */ then
      if  $a = \max(\text{domain}(x))$  then
         $b := \min(\text{domain}(x))$ ;
      else
         $b := \max(\text{domain}(x))$ ;
     $R := \text{MultipleDominanceCheck}(x, a \rightarrow b)$ ;
    11  if  $R = \emptyset$  then  $R := \text{MultipleDominanceCheck}(x, b \rightarrow a)$ ;
     $\Delta := \Delta \cup R$ ;
  foreach  $(x, a) \in \Delta$  do  $\text{remove}(x, a)$  from  $\text{domain}(x)$ ;

  /* Check if value a dominates value b and possibly other values */
  Function  $\text{MultipleDominanceCheck}(x, a \rightarrow b)$ : set of dominated values
    if  $f_x(a) > f_x(b)$  then return  $\emptyset$ ;
     $\delta_{a \rightarrow b} := f_x(a)$ ;
     $ub_a := f_x(a)$ ;
    12  foreach  $f_s \in F$  such that  $\{x\} \subset S$  do
       $(\delta, ub) := \text{getDifference-Maximum}(f_s, x, a \rightarrow b)$ ;
       $\delta_{a \rightarrow b} := \delta_{a \rightarrow b} + \delta$ ;
       $ub_a := ub_a + ub$ ;
      13  if  $\delta_{a \rightarrow b} > f_x(b)$  then return  $\emptyset$ ;
    if  $ub_a = 0$  then return  $\{(x, u) \mid u \in \text{domain}(x)\} \setminus \{(x, a)\}$ ;
     $R := \{(x, b)\}$  /*  $\delta_{a \rightarrow b} \leq f_x(b)$  */;
    16  foreach  $u \in \text{domain}(x)$  such that  $u \neq a$  do
      17  if  $(f_x(u) \geq ub_a)$  then  $R := R \cup \{(x, u)\}$ ;
    return  $R$ ;

  /* Compute largest cost difference and maximum cost for value */
  Function  $\text{getDifference-Maximum}(f_s, x, a \rightarrow b)$ : pair of costs
     $\delta_{a \rightarrow b} := 0$ ;
     $ub_a := 0$ ;
    foreach  $t \in l(S \setminus \{x\})$  do
      18  if  $f_s(t \cup \{(x, b)\}) + f_\emptyset + f_x(b) + \sum_{y \in S \setminus \{x\}} f_y(t[y]) < k$  then
         $\delta_{a \rightarrow b} := \max(\delta_{a \rightarrow b}, f_s(t \cup \{(x, a)\}) - f_s(t \cup \{(x, b)\}))$ ;
         $ub_a := \max(ub_a, f_s(t \cup \{(x, a)\}))$ ;
    return  $(\delta_{a \rightarrow b}, ub_a)$  /*  $\delta_{a \rightarrow b} \leq ub_a$  */;

```

---

## 6 Experimental Results

We implemented PSNS<sup>r</sup> and DEE<sup>r</sup> in `toulbar2`<sup>6</sup>. All methods use residues and variable queues with timestamps as in [26]. PSNS<sup>r</sup> uses `MultipleDominanceCheck` and `getDifference-Maximum` instead of `DominanceCheck` and `getDifference`. `MultipleDominanceCheck` prunes the dominated values directly instead of queuing them into  $R$ . It speeds-up further dominance checks without assuming soft AC anymore during the process (soft AC being restored at the next iteration until a fixed point is reached for AC and SNS/DEE). We compared PSNS<sup>r</sup> and DEE<sup>r</sup> on a collection of binary WCSP benchmarks (<http://costfunction.org>) (except

<sup>6</sup> C++ solver version 0.9.6 `mulcyber.toulouse.inra.fr/projects/toulbar2/`

for *spot5* using ternary cost functions). The *celar* [4] ( $n \leq 458, d \leq 44$ ) and *computational protein design* [1] ( $n \leq 55, d \leq 148$ ) have been selected as they offer good opportunities for neighborhood substitutability, at least in preprocessing as shown in [14, 20]. We added Max SAT *combinatorial auctions* using the CATS generator [27] with 60 goods and a varied number of bids from 70 to 200 (100 to 230 for *regions*) [23]. Other benchmarks were selected by [26] and include: *DIMACS graph coloring* (minimizing edge violations) ( $n \leq 450, d \leq 9$ ), *optimal planning* [7] ( $n \leq 1433, d \leq 51$ ), *spot5* ( $n \leq 1057, d = 4$ ) [2], and *uncapacitated warehouse location* [22] ( $n \leq 1100, d \leq 300$ ). Experiments were performed on a cluster of AMD Opteron 2.3 GHz under Linux.

In Table 1, we compared a Depth First Branch and Bound algorithm using EDAC [22] alone (*EDAC* column), EDAC and  $DEE^r$  (*EDAC+DEE<sup>r</sup>*), EDAC and  $PSNS^r$  in preprocessing only (*EDAC+PSNS<sup>r</sup><sub>pre</sub>*), EDAC and  $PSNS^r$  in preprocessing and  $DEE^r$  during search (*EDAC+PSNS<sup>r</sup><sub>pre</sub>+DEE<sup>r</sup>*), EDAC and  $PSNS^r$  (*EDAC+PSNS<sup>r</sup>*), and no initial upper bound for all. For each benchmark, we report the number of instances, and for each method, the number of instances optimally solved in less than 1,200 seconds. In parentheses, average CPU time over the solved instances (in seconds), average number of nodes, and average number of value removals per search node are reported where appropriate. First, we used a static lexicographic variable ordering and a binary branching scheme (*toulbar2* options *-nopre -svo -d:*).  $DEE^r$  solved always a greater or equal number of instances compared to EDAC alone, and it performed better than  $PSNS^r$  on *celar*, *planning*, *protein*, and *warehouse* benchmarks, all having large domains. We also give the results, when available, in terms of the number of solved instances by  $PSNS^r$  over the total number of instances solved by at least one method as reported in [26], showing the good performance of our approach. They used the same settings except a cluster of Xeon 3.0 GHz and *max degree* static variable ordering (only identical to our lexicographic ordering for *warehouse*). In addition, we solved the *celar7-sub1* instance with the same *max degree* ordering: *EDAC+DEE<sup>r</sup>* solved in (7.7 seconds, 57,584 nodes, 0.96 removals per node), and *EDAC+PSNS<sup>r</sup>* in (69.5, 39,346, 7.2), or (86.4, 70,896, 6) as reported in [26]. Secondly, we used a dynamic variable ordering combining Weighted Degree with Last Conflict [25] and an initial Limited Discrepancy Search (LDS) phase [18] with a maximum discrepancy of 2 (option *-l=2*, except for *protein* using also *-sortd -d:* as in [1]). This greatly improved the results for all the methods and benchmarks except for *warehouse* where LDS slowed down the methods.  $DEE^r$  remained the best method in terms of the number of solved instances;  $PSNS^r$  in preprocessing and  $DEE^r$  during search being a good alternative, especially on the *protein* benchmark. We compared a subset of our results with the last Max SAT 2012 evaluation (<http://maxsat.ia.udl.cat:81/12>). With roughly the same computation time limit (20 min. with 2.3 GHz instead of 30 min. with AMD Opteron 1.5 GHz), for *auction/paths* and *auction/scheduling*,  $DEE^r$  solved 85+82 instances among 170, being in 3rd position among 11 Max SAT solvers.

**Table 1.** For each method, number of instances optimally solved in less than 1,200 seconds, and in parentheses, average CPU time (in seconds) over the solved instances, average number of search nodes, and average number of value removals per node where appropriate

|                                                                                                           | #inst. | EDAC                     | EDAC+DEE <sup>r</sup>          |                       | EDAC+PSNS <sup>r</sup> <sub>pre</sub> | EDAC+PSNS <sup>r</sup> <sub>pre</sub> +DEE <sup>r</sup> | EDAC+PSNS <sup>r</sup> | [26] |
|-----------------------------------------------------------------------------------------------------------|--------|--------------------------|--------------------------------|-----------------------|---------------------------------------|---------------------------------------------------------|------------------------|------|
| Depth First Branch and Bound with static variable ordering                                                |        |                          |                                |                       |                                       |                                                         |                        |      |
| celar                                                                                                     | 46     | <b>24 (180.6, 954K)</b>  | 24 (187.2, 877K, 0.80)         | 24 (188.4, 945K)      | 24 (187.7, 877K, 0.77)                | 17 (168.0, 100K, 8.33)                                  | 12/16                  |      |
|                                                                                                           | 40     | 19 (47.7, 2.4M)          | 19 (45.7, 2.2M, 0.08)          | 19 (46.9, 2.3M)       | 19 (45.6, 2.2M, 0.08)                 | <b>20 (103.5, 3.7M, 0.96)</b>                           | 8/8                    |      |
|                                                                                                           | 76     | 68 (9.8, 39K)            | 75 (7.2, 32K, 4.46)            | 69 (18.3, 127K)       | <b>75 (6.9, 32K, 4.46)</b>            | 75 (10.5, 31K, 5.27)                                    | 27/27                  |      |
|                                                                                                           | 12     | 9 (34.4, 70K)            | 9 (30.9, 42K, 1.50)            | 9 (26.0, 50K)         | <b>9 (25.7, 40K, 1.32)</b>            | 9 (139.0, 31K, 4.37)                                    |                        |      |
|                                                                                                           | 24     | 4 (0.1, 68)              | 7 (93.7, 2.7M, 0.42)           | 6 (172.7, 3.7M)       | 7 (93.2, 2.7M, 0.39)                  | <b>7 (87.0, 2.5M, 0.42)</b>                             | 3/3                    |      |
|                                                                                                           | 55     | <b>46 (55.6, 709)</b>    | 46 (66.1, 542, 34.34)          | 46 (61.3, 688)        | 46 (58.6, 542, 34.73)                 | 45 (56.3, 429, 75.00)                                   | 29/34                  |      |
|                                                                                                           | 420    | 138 (225.4, 5.9M)        | <b>148 (212.8, 5.2M, 0.06)</b> | 138 (223.5, 5.9M)     | 148 (213.7, 5.2M, 0.06)               | 148 (214.0, 5.2M, 0.06)                                 |                        |      |
|                                                                                                           | 420    | 364 (137.5, 3.3M)        | 404 (98.1, 1.9M, 0.03)         | 373 (131.2, 3.2M)     | 403 (94.3, 1.9M, 0.03)                | <b>405 (100.2, 2M, 0.03)</b>                            |                        |      |
|                                                                                                           | 420    | <b>392 (115.3, 2.3M)</b> | 392 (118.4, 2.3M, 0.00)        | 392 (113.3, 2.3M)     | 391 (115.6, 2.2M, 0.00)               | 390 (114.1, 2.2M, 0.00)                                 |                        |      |
|                                                                                                           | total  | 1513                     | 1064                           | 1124                  | 1076                                  | 1122                                                    | 1116                   |      |
| Depth First Branch and Bound with dynamic variable ordering and initial LDS with maximum discrepancy of 2 |        |                          |                                |                       |                                       |                                                         |                        |      |
| celar                                                                                                     | 46     | 40 (22.7, 45K)           | 40 (24.5, 43K, 1.90)           | <b>40 (19.8, 40K)</b> | 40 (24.9, 38K, 1.80)                  | 38 (114.0, 25K, 10.64)                                  |                        |      |
|                                                                                                           | 40     | 23 (6.6, 167K)           | 24 (39.4, 484K, 0.86)          | 23 (6.7, 167K)        | 24 (38.9, 484K, 0.86)                 | <b>24 (9.1, 162K, 1.19)</b>                             |                        |      |
|                                                                                                           | 76     | 76 (1.3, 1.5K)           | 76 (1.2, 1.4K, 3.05)           | <b>76 (0.8, 1.1K)</b> | 76 (1.3, 1.5K, 3.02)                  | 76 (1.2, 1.3K, 3.34)                                    |                        |      |
|                                                                                                           | 12     | 9 (10.1, 7.7K)           | 9 (10.5, 8K, 1.77)             | 9 (9.0, 10K)          | <b>9 (8.5, 8K, 1.33)</b>              | 9 (55.0, 11K, 5.67)                                     |                        |      |
|                                                                                                           | 24     | 8 (21.7, 669K)           | 8 (14.1, 418K, 0.13)           | 8 (27.1, 841K)        | 8 (16.2, 483K, 0.14)                  | <b>8 (12.3, 350K, 0.19)</b>                             |                        |      |
|                                                                                                           | 55     | <b>45 (67.1, 957)</b>    | 43 (30.7, 630, 17.87)          | 45 (70.8, 949)        | 43 (30.2, 618, 18.65)                 | 42 (8.7, 411, 31.45)                                    |                        |      |
|                                                                                                           | 420    | 345 (139.0, 2.5M)        | <b>356 (137.4, 2.4M, 0.16)</b> | 346 (138.5, 2.5M)     | 356 (137.6, 2.4M, 0.16)               | 355 (139.0, 2.4M, 0.16)                                 |                        |      |
|                                                                                                           | 420    | <b>420 (2.5, 27K)</b>    | <b>420 (2.5, 27K, 0.03)</b>    | <b>420 (2.5, 27K)</b> | <b>420 (2.5, 27K, 0.03)</b>           | <b>420 (2.5, 27K, 0.03)</b>                             |                        |      |
|                                                                                                           | 420    | <b>413 (54.8, 1.5M)</b>  | 413 (57.8, 1.5M, 0.00)         | 413 (55.5, 1.5M)      | 413 (57.8, 1.5M, 0.00)                | 413 (57.8, 1.5M, 0.00)                                  |                        |      |
|                                                                                                           | total  | 1513                     | 1379                           | 1389                  | 1380                                  | 1389                                                    | 1385                   |      |



## 7 Conclusion

We have presented a lightweight algorithm for automatically exploiting a dead-end elimination dominance criterion for WCSPs. Experimental results show that it can lead to significant reductions in search space and run-time on several benchmarks. In future work, we plan to study such dominance criteria applied during search in integer linear programming.

**Acknowledgements.** We thank the Genotoul Bioinformatic platform for the cluster and Seydou Traoré, Isabelle André, and Sophie Barbe for the protein instances.

## References

1. Allouche, D., Traoré, S., André, I., de Givry, S., Katsirelos, G., Barbe, S., Schiex, T.: Computational protein design as a cost function network optimization problem. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 840–849. Springer, Heidelberg (2012)
2. Bensana, E., Lemaître, M., Verfaillie, G.: Earth observation satellite management. *Constraints* 4(3), 293–299 (1999)
3. Bistarelli, S., Faltings, B.V., Neagu, N.: Interchangeability in Soft CSPs. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 31–46. Springer, Heidelberg (2002)
4. Cabon, B., de Givry, S., Lobjois, L., Schiex, T., Warners, J.: Radio link frequency assignment. *Constraints Journal* 4, 79–89 (1999)
5. Chu, G., Banda, M., Stuckey, P.: Exploiting subproblem dominance in constraint programming. *Constraints* 17(1), 1–38 (2012)
6. Chu, G., Stuckey, P.J.: A generic method for identifying and exploiting dominance relations. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 6–22. Springer, Heidelberg (2012)
7. Cooper, M., Cussat-Blanc, S., de Roquemaurel, M., Régnier, P.: Soft arc consistency applied to optimal planning. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 680–684. Springer, Heidelberg (2006)
8. Cooper, M., de Givry, S., Sanchez, M., Schiex, T., Zytnicki, M., Werner, T.: Soft arc consistency revisited. *Artificial Intelligence* 174, 449–478 (2010)
9. Cooper, M., de Givry, S., Sanchez, M., Schiex, T., Zytnicki, M.: Virtual arc consistency for weighted CSP. In: Proc. of AAAI 2008, Chicago, IL (2008)
10. Cooper, M.C.: High-order consistency in Valued Constraint Satisfaction. *Constraints* 10, 283–305 (2005)
11. Cooper, M.C., de Givry, S., Schiex, T.: Optimal soft arc consistency. In: Proc. of IJCAI 2007, Hyderabad, India, pp. 68–73 (January 2007)
12. Cooper, M.C., Schiex, T.: Arc consistency for soft constraints. *Artificial Intelligence* 154(1-2), 199–227 (2004)
13. Dahiyat, B., Mayo, S.: Protein design automation. *Protein Science* 5(5), 895–903 (1996)
14. Desmet, J., Maeyer, M., Hazes, B., Lasters, I.: The dead-end elimination theorem and its use in protein side-chain positioning. *Nature* 356(6369), 539–542 (1992)
15. Freuder, E.C.: Eliminating interchangeable values in constraint satisfaction problems. In: Proc. of AAAI 1991, Anaheim, CA, pp. 227–233 (1991)

16. Georgiev, I., Lilien, R., Donald, B.: Improved pruning algorithms and divide-and-conquer strategies for dead-end elimination, with application to protein design. *Bioinformatics* 22(14), e174–e183 (2006)
17. Goldstein, R.: Efficient rotamer elimination applied to protein side-chains and related spin glasses. *Biophysical Journal* 66(5), 1335–1340 (1994)
18. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. In: *Proc. of the 14th IJCAI*, Montréal, Canada (1995)
19. Jougllet, A., Carlier, J.: Dominance rules in combinatorial optimization problems. *European Journal of Operational Research* 212(3), 433–444 (2011)
20. Koster, A.M.C.A.: Frequency assignment: Models and Algorithms. Ph.D. thesis, University of Maastricht, The Netherlands (November 1999), [www.zib.de/koster/thesis.html](http://www.zib.de/koster/thesis.html)
21. Larrosa, J.: On arc and node consistency in weighted CSP. In: *Proc. AAAI 2002*, Edmondston (CA), pp. 48–53 (2002)
22. Larrosa, J., de Givry, S., Heras, F., Zytnicki, M.: Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In: *Proc. of the 19th IJCAI*, Edinburgh, Scotland, pp. 84–89 (August 2005)
23. Larrosa, J., Heras, F., de Givry, S.: A logical approach to efficient max-sat solving. *Artif. Intell.* 172(2-3), 204–233 (2008)
24. Larrosa, J., Schiex, T.: Solving weighted CSP by maintaining arc consistency. *Artif. Intell.* 159(1-2), 1–26 (2004)
25. Lecoutre, C., Saïs, L., Tabary, S., Vidal, V.: Reasoning from last conflict(s) in constraint programming. *Artificial Intelligence* 173, 1592–1614 (2009)
26. Lecoutre, C., Roussel, O., Dehane, D.E.: WCSP Integration of Soft Neighborhood Substitutability. In: Milano, M. (ed.) *CP 2012*. LNCS, vol. 7514, pp. 406–421. Springer, Heidelberg (2012)
27. Leyton-Brown, K., Pearson, M., Shoham, Y.: Towards a Universal Test Suite for Combinatorial Auction Algorithms. In: *ACM E-Commerce*, pp. 66–76 (2000)
28. Looger, L., Hellinga, H.: Generalized dead-end elimination algorithms make large-scale protein side-chain structure prediction tractable: implications for protein design and structural genomics. *Journal of Molecular Biology* 307(1), 429–445 (2001)
29. Niedermeier, R., Rossmanith, P.: New upper bounds for maximum satisfiability. *J. Algorithms* 36(1), 63–88 (2000)
30. Pierce, N., Spriet, J., Desmet, J., Mayo, S.: Conformational splitting: A more powerful criterion for dead-end elimination. *Journal of Computational Chemistry* 21(11), 999–1009 (2000)
31. Schiex, T.: Arc consistency for soft constraints. In: Dechter, R. (ed.) *CP 2000*. LNCS, vol. 1894, pp. 411–424. Springer, Heidelberg (2000)

# Solving Weighted CSPs by Successive Relaxations

Erin Delisle and Fahiem Bacchus

Department of Computer Science, University of Toronto,  
Toronto, Ontario, Canada, M5S 3H5  
{edelisle,fbacchus}@cs.toronto.edu

**Abstract.** In this paper we present a new algorithm for solving weighted CSPs (WCSP). This involves first creating an ordinary unweighted CSP,  $\mathcal{P}$ , by hardening all soft constraints of the WCSP.  $\mathcal{P}$  has a solution if and only if the WCSP has a cost zero solution. The algorithm then proceeds by solving relaxations of  $\mathcal{P}$  each allowing a particular cost to be incurred. If the relaxation has no solution, a set of its forbidden tuples sufficient to rule out all solutions is computed. From this set of culprit tuples we show how to compute a new relaxation of  $\mathcal{P}$  that can again be tested for a solution. If the new relaxation is optimal, incurring a minimum cost, any solution found will also be an solution to the WCSP.

In contrast with traditional branch and bound algorithms our algorithm is a hybrid approach in which a standard CSP solver is used to solve the relaxation and a mixed integer program solver (MIP) is used to compute optimal new relaxations. Our approach is most closely related to unsatisfiable core techniques that have been developed for solving MAXSAT. However by exploiting the fact that at most one tuple in a constraint can be satisfied by any variable assignment we are able to develop a more compact encoding of the optimization problem used to compute the optimal relaxation. We prove that the algorithm is sound, and provide some preliminary empirical results on its performance.

## 1 Introduction

Many practical problems involve some degree of optimization. That is, typically we are not only interested in finding a solution but in finding low cost solutions, or even optimal solutions, when we can compute them. Weighted CSPs (WCSP) or soft CSPs are a CSP based formalism geared towards representing optimization problems. In this formalism constraints are replaced by cost functions (soft constraints) and instead of aiming to find a solution that satisfies all constraints one aims to find a solution that incurs lowest total cost from the cost functions.

The most prominent methods for solving WCSP employ branch and bound search, e.g., Toulbar [1]. These solvers depend on sophisticated methods for computing lower bounds [2] during search.

A WCSP can be viewed as being the CSP version of weighted MAXSAT. In weighted MAXSAT we have a set of clauses each with a weight and are trying to find a truth assignment that falsifies the lowest total weight of clauses. Research

in MAXSAT has also pursued branch and bound solvers, e.g., [3, 4], however an effective alternative is to solve MAXSAT by solving a sequence of decision problems (typically SAT decision problems), e.g., [5–7]. Empirically, it has been found that on larger problems the sequence of decision problems approach works better. For example in the MAXSAT evaluations [8] branch and bound solvers are not as effective on the larger problems in the industrial category.

In this paper we present a sequence of decision problems approach to solving WCSP. Our approach is based on the ideas presented in [7, 9], but involves some key innovations aimed at better exploiting the additional structure of WCSPs. The approach is a hybrid one in which both a hard-CSP solver and a mixed integer program solver (MIP) solver are used. The idea is similar to the general paradigm of Logic Based Benders Decomposition [10]. In particular, we use the MIP solver to generate candidate optimal solutions and the CSP solver to test their feasibility. If the candidate is feasible we have solved the WCSP.

The novelty of our approach lies in the manner in which we construct the MIP and CSP subproblems: our models are designed to exploit the structure of WCSPs. In the paper we present our approach and prove it to be sound. We close the paper with some preliminary empirical results that indicate the approach has some potential, although more work needs to be done to make it competitive with the far more well developed branch and bound solvers.

## 2 Background

A weighted CSP (WCSP),  $wt\mathcal{P} = (C, V)$  is specified by a set of variables  $V = \{v_1, \dots, v_n\}$ , each with an associated domain of values  $D_i$ , and a set of soft constraints or cost functions  $C = \{c_1, \dots, c_m\}$ . Each  $c \in C$  is a function over a subset  $scope(c)$  of  $V$ , called its scope;  $c$  maps tuples of assignments  $\tau$  over the variables in  $scope(c)$  to positive numbers or infinity. If  $c(\tau) = \infty$  then  $\tau$  is **forbidden** by  $c$ . Otherwise  $c(\tau)$  is the cost incurred by  $\tau$  from  $c$ .

An assignment  $\pi$  is a mapping  $v \rightarrow d \in D_i$  for all  $v \in V$ . A partial assignment is a mapping of some subset of the variables. If an assignment  $\pi$  includes all of the variables in  $scope(c)$  for some cost function  $c$ , then  $cost_c(\pi)$  denotes the value of  $c$  evaluated on those assignments.

We restrict our attention to cost functions specified extensionally as tables that list all assignment tuples over the function's variables that have non-zero cost (the table also specifies the cost of each such tuple). Tuples that have infinite cost are **hard** tuples. Tuples that specify finite non-zero costs are **soft** tuples. Hard constraints are constraints containing only hard tuples. We may harden tuples, constraints, or weighted CSPs increasing the weight of the corresponding soft tuples to infinity.

The cost of a complete assignment  $\pi$  is the sum of the costs it incurs from cost functions:  $cost(\pi) = \sum_{c \in C} cost_c(\pi)$ . An assignment  $\pi$  is a **solution** to  $wt\mathcal{P}$  if it has **finite cost**. A solution is an **optimal solution** if no other solution has lower cost. Solving a WCSP means finding an optimal solution.

One successful method of solving weighted CSP problems is branch and bound search which uses lower bounds to prune the search space. Soft arc consistency

techniques [2] such as EDAC, VAC, and OSAC transform a problem preserving the cost of all solutions, but where as much cost as possible is moved into a special 0-ary cost function  $c_\emptyset$  incurred by all assignments. Hence,  $c_\emptyset$  provides a lower bound on the optimal cost. Applying these techniques during search provides a lower bound on the solutions extending the current partial assignment. EDAC and VAC used in this manner form the basis of the successful Toulbar2 solver.

In MAXSAT solving a successful alternative solution technique consists of solving a sequence of decision problems. In the sequence of decision problems approach all soft costs are hardened and a proof of unsatisfiability is derived using a SAT solver. The proof of unsatisfiability takes the form of a *core*, a set of culprit soft clauses at least one of which must be falsified. Either new constraints are added to the MAXSAT problem based on the core, e.g., [5, 6], or a relaxation of the hard problem is derived from the set of known cores [9]. This repeats until the decision problem becomes satisfiable.

In adapting the sequence of decision problems approach to weighted CSPs we define a new concept of a core specific to weighted CSPs, as well as a formal definition of a relaxation of the hardened WCSP.

**Definition 1.** A *weight vector* for  $wt\mathcal{P} = (C, V)$ , where  $C = \{c_1, \dots, c_m\}$  is a vector  $\langle w_1, \dots, w_m \rangle$  of numbers where  $w_i$  is a cost value that could be assigned by  $c_i$ , or zero, or infinity:  $w_i \in \{0, \infty\} \cup \{v | \exists \tau \text{ s.t. } c_i(\tau) = v\}$ . Note that there are only a finite number of weight vectors for  $wt\mathcal{P}$ .

**Definition 2.** For a cost function  $c$  and weight  $w$ ,  $0 \leq w \leq \infty$ , let  $\text{ceiling}(c, w)$  be the tightened cost function generated by hardening all tuples  $\tau$  in  $c$  with  $c(\tau) \geq w$ , i.e.,  $\text{ceiling}(c, w)(\tau) = \infty$  if  $c(\tau) \geq w$  and  $\text{ceiling}(c, w)(\tau) = c(\tau)$  if  $c(\tau) < w$ . Note that  $\text{ceiling}(c, \infty) = c$  and  $\text{ceiling}(c, 0)$  forbids all tuples.

A *core* of  $wt\mathcal{P} = (C, V)$  is a weight vector  $\mathbf{w}$  such that there is no solution (finite cost complete assignment) of the WCSP  $(\{\text{ceiling}(c_i, w_i) | c_i \in C\}, V)$ . That is, if we tighten each cost function  $c_i$  by  $w_i$ , then all complete assignments have infinite weight.

This differs from cores in a MAXSAT problem. In MAXSAT each tuple in a cost function corresponds to a soft clause. A core containing many soft clauses corresponds to a collection of hardened tuples sufficient to cause the CSP to have no solution. Such cores might, and typically will, contain many tuples from each cost function. In our formulation, in contrast, there is only one element in the core per cost function. These elements correspond to *sets of tuples* of related weight within the cost function. This more compact representation of a core is possible because the tuples of each cost function are mutually exclusive—no assignment can contain more than one tuple of the cost function.

**Definition 3.** A *relaxation* of  $wt\mathcal{P} = (C, V)$  is an unweighted CSP  $\mathcal{P}_{\mathbf{w}}$  generated by a weight vector  $\mathbf{w}$ . The CSP  $\mathcal{P}_{\mathbf{w}}$  is formed by converting each  $c_i \in C$  to a true/false constraint  $c_i^{w_i}$  of  $\mathcal{P}$  with  $c_i^{w_i}(\tau) = \text{true}$  iff  $c_i(\tau) \leq w_i$ . The *cost* of a relaxation  $\mathcal{P}_{\mathbf{w}}$ ,  $\text{cost}(\mathcal{P}_{\mathbf{w}})$ , is  $\sum_i w_i$ .

**Algorithm 1.** Algorithm for solving a WCSP  $wt\mathcal{P}$ 


---

```

1 wtCSP-Solver( $wt\mathcal{P}$ )
2 begin
3   Cores  $\leftarrow \emptyset$ 
4    $\mathbf{w} = \mathbf{0}$ 
5   while true do
6     (solvable?, newcore)  $\leftarrow$  relaxAndSolve( $wt\mathcal{P}$ ,  $\mathbf{w}$ )
7     if solvable? then
8       | return extractLastSolution()
9     cores  $\leftarrow$  cores  $\cup \{\text{newcore}\}$ 
10     $\mathbf{w} \leftarrow$  getOptimalWtVec(cores)

```

---

The relaxation  $\mathcal{P}_0$ , in which all weights are zero, admits only assignments incurring zero cost in the WCSP as solutions, and when the weights are all infinite the relaxation admits all assignments as solutions. In general, for weights  $\mathbf{w}$ ,  $\mathcal{P}_{\mathbf{w}}$  is a weakening of  $\mathcal{P}_0$  that admits as solutions only those assignments that incur a cost  $\leq w_i$  from cost function  $c_i$ . For convenience we have defined  $\mathcal{P}_{\mathbf{w}}$  to be a relaxation of the WCSP, although technically it is a relaxation of the CSP  $\mathcal{P}_0$ .

**Definition 4.** A weight vector  $\mathbf{w}$  satisfies a set of cores  $\mathcal{K}$  if for every core (weight vector)  $\mathbf{v} \in \mathcal{K}$  we have that  $w_i \geq v_i$  at some index  $i$ . The vector  $\mathbf{w}$  is **optimal for**  $\mathcal{K}$  if it satisfies  $\mathcal{K}$  and for all other weight vectors  $\mathbf{w}'$  that satisfy  $\mathcal{K}$  we have  $\sum_i w_i \leq \sum_i w'_i$ .

### 3 Relaxation Based Algorithm for Solving wcp

To solve a weighted CSP as a sequence of decision problems we rely on two operations, extracting cores from a hard CSP instance with no solution and finding an optimal relaxation of a set of cores. This solve and relax approach has a number of similarities with the MAXHS solver for MAXSAT problems [7, 9]. Starting with the zero cost relaxation  $\mathcal{P}_0$  we extract a core from it, calculate an optimal satisfying weight vector  $\mathbf{w}$  for the current set of cores, form the new relaxation  $\mathcal{P}_{\mathbf{w}}$ , and repeat until we find a relaxation that has a solution. We will assume for the sake of clarity that the WCSP has at least one solution.

**Lemma 1.** Let  $\pi$  be any complete assignment for  $wt\mathcal{P}$ , and  $\mathbf{w}_{\pi} = \langle cost_{c_1}(\pi), \dots, cost_{c_m}(\pi) \rangle$ , then  $\mathbf{w}_{\pi}$  satisfies all cores of  $wt\mathcal{P} = (C, V)$ .

*Proof.* Let  $\mathbf{w}$  be any core. By definition of a core  $\pi$  must incur infinite cost on  $(\{ceiling(c_i, w_i) | c_i \in C\}, V)$ . This means that  $cost_{c_i}(\pi) \geq w_i$  for some index  $i$ , and thus  $\mathbf{w}_{\pi}$  satisfies  $\mathbf{w}$ .

**Lemma 2.** Let  $\mathcal{P}_{\mathbf{w}}$  be a relaxation generated by the weight vector  $\mathbf{w}$ . If  $\pi$  is a solution of  $\mathcal{P}_{\mathbf{w}}$ , then  $cost(\pi) \leq \sum_i w_i$ .

*Proof.* For every  $c_i$  all tuples of cost greater than  $w_i$  are forbidden in  $\mathcal{P}_{\mathbf{w}}$ . Therefore,  $cost_{c_i}(\pi) \leq w_i$  for every  $c_i$ .

**Lemma 3.** *Let  $\mathcal{K}$  be a set of cores of  $wt\mathcal{P}$  and  $\mathbf{w}$  be an optimal weight vector satisfying  $\mathcal{K}$ . If  $\pi$  is a solution to the relaxation  $\mathcal{P}_{\mathbf{w}}$  then  $cost(\pi) = \sum_i w_i$ .*

*Proof.* By Lemma 1 the weight vector  $\langle cost_{c_1}(\pi), \dots, cost_{c_m}(\pi) \rangle$  satisfies  $\mathcal{K}$ . Since  $\mathbf{w}$  is optimal for  $\mathcal{K}$  we must have  $cost(\pi) \geq \sum_i w_i$ . By Lemma 2  $cost(\pi) \leq \sum_i w_i$ .

**Theorem 1.** *Let  $\mathcal{K}$  be any set of cores of  $wt\mathcal{P}$ , and  $\mathbf{w}$  be an optimal weight vector satisfying  $\mathcal{K}$ . If  $\pi$  is a solution to the relaxation  $\mathcal{P}_{\mathbf{w}}$  then  $\pi$  is an optimal solution for  $wt\mathcal{P}$ .*

*Proof.* Any solution for  $wt\mathcal{P}$  generates a weight vector satisfying  $\mathcal{K}$  by Lemma 1. Therefore, every solution must have cost at least  $\sum_i w_i$ . By Lemma 3,  $\pi$  has cost equal to  $\sum_i w_i$ , therefore  $\pi$  has optimal cost.

Each relaxation computed by “getOptimalWtVec” satisfies all previous discovered cores, so the same core cannot be discovered again. Since there are only a finite number of cores (there are only a finite number of weight vectors), the algorithm must terminate. Theorem 1 shows that when the algorithm terminates it has found an optimal solution to  $wt\mathcal{P}$ .

### 3.1 Extracting Cores

To extract a core from an unsolvable CSP instance we need to determine a set of culprit tuples sufficient to make the current relaxation  $\mathcal{P}_{\mathbf{w}}$  unsatisfiable. The subroutine “relaxAndSolve” in Algorithm 1 uses the clause learning CSP solver MINICSP [11] operating on the strictest relaxation  $\mathcal{P}_0$ .

In  $\mathcal{P}_0$  each cost function is converted to a constraint that forbids all non-zero cost assignments, and MINICSP encodes each of these forbidden tuples as a clause blocking that set of assignments. If the clause  $C$  arises from the tuple  $\tau$  in cost function  $c_i$  with  $0 < c_i(\tau) < \infty$ , then a special “blocking variable”  $b_i^{c_i(\tau)}$  is added to  $C$ , otherwise  $C$  is unchanged. For each cost function  $c_i$  and finite non-zero weight  $w$  that could be assigned by  $c_i$ , we have a single blocking variable  $b_i^w$ : all clauses arising from tuples with cost  $w$  are “blocked” by the same variable  $b_i^w$ .

We use the assumption mechanism of MINICSP in which a set of literals can be given as assumptions. If the CSP is unsatisfiable a subset of these assumptions sufficient to cause unsatisfiability is returned. In particular, to extract a core from the relaxation  $\mathcal{P}_{\mathbf{w}}$  we use the set of assumptions:  $\{-b_i^v | v > w_i\} \cup \{b_i^v | v \leq w_i\}$ .

When  $b_i^v$  is true all clauses of weight  $v$  from cost function  $c_i$  are “blocked”. That is, these clauses are immediately satisfied and no longer constrain the theory. When  $b_i^v$  is false the clauses of weight  $v$  are enforced. Thus any solution to the problem under this set of assumptions must be a solution of  $wt\mathcal{P}$  that incurs no more than cost  $w_i$  from  $c_i$ , for all  $i$ .

If there is no solution then MINICSP will return a subset of the assumptions causing unsatisfiability. The blocking variables appear only positively in the clauses, so no positive  $b_i^v$  assumption can contribute to unsatisfiability. Hence, the subset returned is a set of negated blocking variables asserting that at least

one of these variables  $b_i^v$  must be made true, i.e., we must incur at least one of these costs in any solution of  $wt\mathcal{P}$ .

Let  $U$  be the set of negated blocking variables causing unsatisfiability returned by MINICSP. We convert this into the core  $\kappa = \{w_i | w_i = \min_v(\{b_i^v \in U\})\}$ . That is, the weight vector is determined by the minimum weight tuple of  $c_i$  contained in  $U$ : to find an optimal cost solution we need only consider  $U$ 's lower bound on the cost that could be incurred from  $c_i$ . If no tuples of  $c_i$  contributed to unsatisfiability, we use  $w_i = \infty$  in the core.

### 3.2 Finding Optimal Relaxations

Finding an optimal relaxation of a set of cores is similar to finding a minimum weight hitting set over the cores with some additional constraints. The set of blocking variables,  $B$ , used to extract the cores are used again as 0/1 variables in this optimization problem, along with the added constraint that higher cost variables imply the lower cost variables of the same cost function. We use the MIP solver CPLEX to solve this optimization problem.

Assume that the blocking variables within a cost function are sorted in increasing order of cost. Define  $\rho(b_i^v)$  for  $b_i^v \in B$  to be the next smaller cost blocking variable for  $c_i$  or  $\emptyset$  if  $v$  is the smallest non-zero cost of  $c_i$ .  $\rho$  will act as a predecessor function, returning the variable which came before it in a cost function. Using  $\rho$  we can define  $\delta$ , a function assigning an adjusted cost to blocking variables.  $\delta$  compensates for the fact that larger cost variables imply smaller cost variables.

$$\delta(b_i^w) = \begin{cases} cost(b_i^w) & \text{if } \rho(b_i^w) = \emptyset \\ cost(b_i^w) - cost(\rho(b_i^w)) & \text{otherwise} \end{cases} \quad (1)$$

The objective function is the sum of the adjusted costs of all blocking variables. The constraints are that one variable from each core must be selected and larger cost variables imply smaller cost variables within a cost function. For a set of blocking variables  $B$  and cores  $\mathcal{K}$  we get the following MIP:

$$\text{minimize } \sum_{b_i^w \in B} \delta(b_i^w) \quad (2)$$

Subject to:

$$\rho(b_i^w) - b_i^w \geq 0 \text{ for } b_i^w \in B \text{ such that } \rho(b_i^w) \neq \emptyset \quad (3)$$

$$\forall \mathbf{w} \in \mathcal{K} : \sum_{w_i \in \mathbf{w} \text{ s.t. } w_i < \infty} b_i^{w_i} \geq 1 \quad (4)$$

The term  $\rho(b_i^w) - b_i^w \geq 0$  in Equation 3 is logically equivalent to  $b_i^w \Rightarrow \rho(b_i^w)$  since  $b_i^w$  and  $\rho(b_i^w)$  are binary variables. These constraints ensure that a higher cost variable in a cost function implies all lower cost variables. Equation 2 combined with Equation 3 will cause all  $cost(b_i^w)$  terms in  $\sum \delta(b_i^w)$  to cancel out except for the largest cost in each cost function. Thus Equation 2 minimizes



the sum of the largest cost incurred by each cost function. Equation 4 states that for each core  $\mathbf{w}$ , a cost greater than or equal to  $w_i$  for at least one cost function  $c_i$  must be incurred. It can never be optimal to incur cost  $w_i$  when  $w_i = \infty$  so they can be omitted from the constraint.

The solution  $S$  to this optimization problem is a setting of the 0/1  $B$  variables. From this setting we generate an optimal weight vector for  $\mathcal{K}$   $\{w_i | w_i = \max_v \{b_i^v = 1 \in S\}\}$ . If in  $S$  we have that  $b_i^v = 0$  for all  $v$ , we set  $w_i = 0$ : the cores can be satisfied without incurring any cost from  $c_i$ .

### 3.3 Improving Cores

We have found that computing an optimal weight vector for the current set of cores is typically the most time consuming part of our approach. Spending additional effort in the CSP solver to produce stronger cores can often yield better results by reducing the overall effort required in the optimization phase [7].

As described in Sec. 3.1 we compute a core for  $\mathcal{P}_{\mathbf{w}}$  by solving  $\mathcal{P}_0$  augmented with blocking variables under the assumptions  $\{-b_i^v | v > w_i\} \cup \{b_i^v | v \leq w_i\}$ . MINICSP returns a set  $U$  of negated blocking variables from which we compute a core.

To produce stronger cores we sort  $U$  so that for each cost function its smaller cost blocking variables appear before its higher cost blocking variables:  $-b_i^v$  appears before  $-b_i^w$  if  $v < w$ . We then test the blocking variables of  $U$  in this order to see if they can be removed.

To test if blocking variable  $b$  can be removed we solve the CSP again under the new set of assumptions  $U - \{b\} \cup \{b_i^w | b_i^w \notin U - \{b\}\}$ . These assumptions block all clauses except those in  $U - \{b\}$ . If we obtain unsatisfiability we remove  $b$  from  $U$  and otherwise we retain it. In either case we move on to the next blocking variable of  $U$  and test it in the same manner.<sup>1</sup> When we find that  $b_i^w$  is necessary for unsatisfiability, we can avoid testing all later variables of the form  $b_i^v$  with  $v > w$ . Intuitively, if we can satisfy the problem by incurring cost  $w$  from cost function  $c_i$  (i.e., the CSP becomes satisfiable) then we can satisfy the problem by incurring an even higher cost  $v$  from  $c_i$ .

Once we have updated  $U$  by trying to remove each of its variables, we form a core from it exactly as described in Sec. 3.1.

## 4 Results

The solver described in this paper was implemented and tested on available weighted CSP instances. A few additional optimizations were made. In the MAXHS approach for MAXSAT [7] it was found that computing a new relaxation for every core was time consuming. A technique proposed by Karp [12] was to use a greedy approach for determining the relaxation so as to acquire multiple cores per optimal calculation of the relaxation. Optimal relaxations are generated after a solution is found to a greedy relaxation. Toulbar2 was also used as a preprocessor to provide virtual arc consistency [2].

<sup>1</sup> If the CSP solver returns a subset of  $U$  we can also reduce  $U$  to this subset.

**Table 1.** Time(s) for selected instances from the Spot5 and Linkage benchmarks

| Problem             | CSP-Seq | Toulbar2 | Problem             | CSP-Seq | Toulbar2 |
|---------------------|---------|----------|---------------------|---------|----------|
| Spot5 404           | 6.14    | 209.17   | Linkage pedigree 18 | 14.61   | 119.15   |
| Spot5 503           | 0.48    | -        | Linkage pedigree 14 | 8.92    | 0.52     |
| Spot5 505           | 562.75  | -        | Linkage pedigree 30 | 27.09   | 240.71   |
| Linkage pedigree 25 | 11.64   | -        | Linkage pedigree 9  | 63.47   | 223.85   |
| Linkage pedigree 39 | 36.14   | 3.32     | Linkage pedigree 20 | 15.04   | 0.76     |
| Linkage pedigree 31 | -       | 779.63   | Linkage pedigree 44 | 145.11  | -        |
| Linkage pedigree 7  | 1.52    | 6.98     | Linkage pedigree 13 | 4.06    | 0.56     |
| Linkage pedigree 41 | 270.22  | 969.88   | Linkage pedigree 33 | 0.76    | 10.58    |
| Linkage pedigree 51 | 19.83   | -        |                     |         |          |

Tests were run on the celar, spot5, and linkage benchmarks. The celar benchmark [13] consists of radio frequency link assignment problems and features primarily soft constraints. The spot5 benchmark [14] is about managing satellites and mixes hard and soft constraints. The linkage benchmark problems are about probabilistic inference for genetic linkage. The entries of Table 1 give time in seconds for our solver (CSP-Seq) and Toulbar2 for problems in the spot5 and linkage benchmarks. Excluded are 16 spot5 problems which both solvers time out on, 3 spot5 benchmarks both solvers completed in under 1s, 2 linkage problems both solvers time out on, and 6 linkage problems both solvers completed in under 3s. Tests were run with a 1800s time limit on a AMD Opteron 2435. A table entry of “-” indicates the solver timed out on that problem instance.

Our solver performed poorly on the celar benchmark, with performance dominated by the Toulbar2 solver. We conjecture that this problem set is more well suited to the techniques used in the Toulbar2 solver, and that our technique will be useful for other types of problems, especially those featuring a significant number of hard constraints.

## 5 Conclusion

We have presented a new method for solving WCSPs based on methods that have been employed in MAXSAT. The main innovation of our approach is to recognize that the notion of a relaxation and a core can be specialized to exploit the specific structure of WCSPs. In particular, in a cost function at most one tuple can be activated by any solution. Thus we can compress the cores down to simply a relaxation weight for each constraint. For future work we are examining a number of methods for improving the performance of the approach. Primary among them is take better advantage of the implication relationships between the  $b$ -variables associated with the same constraint. Currently these are encoded directly as implications in the MIP model, but are exploited only via generic methods employed in the MIP solver.

## References

1. Toulbar2: WCSP solver, <http://mulcyber.toulouse.inra.fr/projects/toulbar2>
2. Cooper, M., de Givry, S., Sanchez, M., Schiex, T., Zytnicki, M., Werner, T.: Soft arc consistency revisited. In: Artificial Intelligence, pp. 449–478 (2010)
3. Heras, F., Larrosa, J., Oliveras, A.: Minimaxsat: An efficient weighted max-sat solver. *Journal of Artificial Intelligence Research (JAIR)* 31, 1–32 (2008)
4. Alsinet, T., Manyà, F., Planès, J.: Improved exact solvers for weighted max-SAT. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 371–377. Springer, Heidelberg (2005)
5. Heras, F., Morgado, A., Marques-Silva, J.: Core-guided binary search algorithms for maximum satisfiability. In: Proceedings of the AAAI National Conference. AAAI (2011)
6. Ansótegui, C., Bonet, M.L., Gabàs, J., Levy, J.: Improving SAT-based weighted MaxSAT solvers. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 86–101. Springer, Heidelberg (2012)
7. Davies, J., Bacchus, F.: Exploiting the power of MIP solvers in MAXSAT. In: Jarvisalo, M., Van Gelder, A. (eds.) SAT 2013. LNCS, vol. 7962, pp. 166–181. Springer, Heidelberg (2013)
8. Argelich, J., Li, C.M., Manyà, F., Planes, J.: The maxsat evaluations (2007–2011), <http://www.maxsat.udl.cat>
9. 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)
10. Hooker, J.N.: Planning and scheduling by logic-based benders decomposition. *Operations Research* 55(3), 588–602 (2007)
11. Katsirelos, G.: MINICSP CSP solver. <http://www7.inra.fr/mia/T/katsirelos/minicsp.html>
12. Moreno-Centeno, E., Karp, R.M.: The implicit hitting set approach to solve combinatorial optimization problems with an application to multigenome alignment. *Operations Research* 61(2), 453–468 (2013)
13. Cabon, B., de Givry, S., Lobjois, L., Schiex, T., Warners, J.: Radio link frequency assignment. *Constraints* 4(1), 79–89 (1999)
14. Bensana, E., Lemaitre, M., Verfaillie, G.: Earth observation satellite management. *Constraints* 4(3), 293–299 (1999)

# Constraint-Based Program Reasoning with Heaps and Separation

Gregory J. Duck, Joxan Jaffar, and Nicolas C.H. Koh

Department of Computer Science, National University of Singapore  
{gregory,joxan}@comp.nus.edu.sg, kchuenho@dso.org.sg

**Abstract.** This paper introduces a constraint language  $\mathcal{H}$  for *finite partial maps* (a.k.a. *heaps*) that incorporates the notion of *separation* from *Separation Logic*. We use  $\mathcal{H}$  to build an extension of *Hoare Logic* for reasoning over *heap manipulating programs* using (constraint-based) *symbolic execution*. We present a sound and complete algorithm for solving quantifier-free (QF)  $\mathcal{H}$ -formulae based on *heap element propagation*. An implementation of the  $\mathcal{H}$ -solver has been integrated into a *Satisfiability Modulo Theories* (SMT) framework. We experimentally evaluate the implementation against *Verification Conditions* (VCs) generated from symbolic execution of large (heap manipulating) programs. In particular, we mitigate the *path explosion problem* using subsumption via interpolation – made possible by the constraint-based encoding.

**Keywords:** Heap Manipulating Programs, Symbolic Execution, Separation Logic, Satisfiability Modulo Theories, Constraint Handling Rules.

## 1 Introduction

An important part of reasoning over *heap manipulating programs* is the ability to specify properties local to *separate* (i.e. non-overlapping) regions of memory. Most modern formalisms, such as Separation Logic [20], Region Logic [2], and (Implicit) Dynamic Frames [16][22], incorporate some encoding of separation. Separation Logic [20] explicates separation between regions of memory through *separating conjunction* (\*). For example, the *Separation Logic formula*  $\mathbf{list}(l) * \mathbf{tree}(t)$  represents a program *heap* comprised of two separate sub-heaps: one containing a *linked-list* and the other a *tree* data-structure.

In this paper we explore a reformulation of Separation Logic in terms of a first-order *constraint language*  $\mathcal{H}$  over *heaps* (i.e. finite partial maps between pointers and values). Under this approach, separating conjunction (\*) is re-encoded as a *constraint*  $H \simeq H_1 * H_2$  between heaps, indicating that: (1) heaps  $H_1$  and  $H_2$  are *separate* (i.e. disjoint domains) and (2)  $H$  is the *heap union* of  $H_1$  and  $H_2$ . We can therefore re-encode the above Separation Logic formula as  $\mathbf{list}(l, L) \wedge \mathbf{tree}(t, T) \wedge \bar{H} \simeq L * T$  where  $\mathbf{list}$  and  $\mathbf{tree}$  are redefined to be predicates over heaps, and the special variable  $\bar{H}$  represents the *global heap* at the program point where it appears. We can also represent a *singleton heap* as a constraint  $\bar{H} \simeq (p \mapsto v)$ .

The motivation behind  $\mathcal{H}$  is to lift some of the benefits of Separation Logic to constraint-based reasoning techniques for heap manipulating programs, such as *constraint-based symbolic execution*. Our method is based on an extension of *Hoare Logic* [11] defined in terms of the constraint language  $\mathcal{H}$ . Whilst Separation Logic guarantees *total correctness* w.r.t. *memory safety* (e.g. no memory errors such as dereferencing dangling pointers, etc.), our reformulation allows for weaker axiomatizations, such as a version that drops the memory-safety requirement. This allows for a *Strongest Post Condition* (SPC) *predicate transformer semantics* [7] to be defined in terms of  $\mathcal{H}$ , which forms the basis of symbolic execution. The resulting *Verification Conditions* (VCs) can then be discharged using a suitable  $\mathcal{H}$ -*constraint solver/theorem prover*. This is illustrated with a simple example:

*Example 1 (Heap Equivalence).* Consider the following Hoare triple:

$$\{H = \bar{\mathcal{H}}\} x := \mathbf{alloc}(); \mathbf{free}(x) \{H = \bar{\mathcal{H}}\} \quad (1)$$

This triple states that the global heap *before* the code fragment is equal to the heap *after* the fragment, i.e. the global heap is *unchanged*. Here  $H$  is a *ghost variable* representing the initial state of the global heap  $\bar{\mathcal{H}}$ . Symbolic execution of the precondition  $P \equiv (H = \bar{\mathcal{H}})$  yields the following  $\mathcal{H}$ -constraints:

$$Q \equiv (H = H_0 \wedge \underline{H_1} \simeq (x \mapsto \_) * H_0 \wedge \underline{H_1} \simeq (x \mapsto \_) * \bar{\mathcal{H}})$$

Here  $H_0$  and  $H_1$  represent the initial and intermediate values for  $\bar{\mathcal{H}}$  respectively. The underlined  $\mathcal{H}$ -constraints encode the  $\mathbf{alloc}()$  and  $\mathbf{free}()$  respectively. Next we can employ an  $\mathcal{H}$ -constraint solver to prove that the postcondition is implied by  $Q$ , i.e. the *Verification Condition* (VC)  $Q \rightarrow H = \bar{\mathcal{H}}$  holds, thereby proving the triple (1) valid.  $\square$

In order to discharge the VCs generated from symbolic execution we need a solver for the resulting  $\mathcal{H}$ -formulae. For this we present a simple decision procedure for *Quantifier Free* (QF)  $\mathcal{H}$ -formulae based on the idea of *heap membership propagation*. We show that the algorithm is both sound and complete, and is readily implementable using *Constraint Handling Rules* (CHR) [10]. We present an implementation of an  $\mathcal{H}$ -solver that has been integrated into a *Satisfiability Modulo Theories* (SMT) framework using SMCHR [8]. Our decision procedure is related to established algorithms for finite sets.

We use the  $\mathcal{H}$ -solver as the basis of a simple program verification tool using symbolic execution. In contrast to Separation Logic-based symbolic execution [4], which is based on a set of *rearrangement rules*, our version is based on constraint solving using the  $\mathcal{H}$ -solver as per Example 1 above. Our encoding allows for some optimization. Namely, we mitigate the *path explosion problem* of symbolic execution by employing *subsumption* via *interpolation* [14][17] techniques.

This paper is organized as follows: Section 2 introduces Hoare and Separation Logic, Section 3 formally introduces the  $\mathcal{H}$ -language, Section 4 introduces an extension of Hoare Logic based on the  $\mathcal{H}$ -language, Section 5 presents an  $\mathcal{H}$ -solver algorithm and implementation, and Section 6 experimentally evaluates the implementation. In summary, the contributions of this paper are the following:

- We define the  $\mathcal{H}$ -language that encodes separation as a constraint between heaps. We show that satisfiability of quantifier-free  $\mathcal{H}$ -formulae is decidable, and present a complete algorithm for solving  $\mathcal{H}$ -formulae.
- We present an extension of Hoare Logic based on the  $\mathcal{H}$ -language. Our extension is similar to Separation Logic, but allows for strongest post conditions, and is therefore suitable for program reasoning via constraint-based symbolic execution.
- We present an implementation of the  $\mathcal{H}$ -solver that has been integrated into an SMT framework. We experimentally evaluate the solver against VCs generated from symbolic execution of heap manipulating programs.

## 2 Preliminaries

This section presents a brief overview of Hoare and Separation Logic.

*Hoare Logic* [11] is a formal system for reasoning about program correctness. Hoare Logic is defined in terms of axioms over *triples* of the form  $\{\phi\} C \{\varphi\}$ , where  $\phi$  is the *pre-condition*,  $\varphi$  is the *post-condition*, and  $C$  is some code fragment. Both  $\phi$  and  $\varphi$  are formulae over the *program variables* in  $C$ . The meaning of the triple is as follows: for all program states  $\sigma_1, \sigma_2$  such that  $\sigma_1 \models \phi$  and executing  $\sigma_1$  through  $C$  derives  $\sigma_2$ , then  $\sigma_2 \models \varphi$ . For example, the triple  $\{x < y\} x := x + 1 \{x \leq y\}$  is *valid*. Note that under this definition, a triple is automatically valid if  $C$  is non-terminating or otherwise has undefined behavior. This is known as *partial correctness*.

*Separation Logic* [20] is a popular extension of Hoare Logic for reasoning over *heap manipulating programs*. Separation Logic extends predicate calculus with new logical connectives (namely *empty heap* (**emp**), *singleton heap* ( $p \mapsto v$ ), and *separating conjunction* ( $H_1 * H_2$ )) such that the structure of assertions reflects the structure of the underlying heap. For example, the pre-condition in the valid Separation Logic triple  $\{x \mapsto \_ * y \mapsto 2\} [x] := [y] + 1 \{x \mapsto 3 * y \mapsto 2\}$  represents a heap comprised of two *disjoint singleton* heaps, indicating that both  $x$  and  $y$  are *allocated* and that location  $y$  points to the value 2. Here the notation  $[p]$  represents pointer dereference. In the post-condition we have that  $x$  points to value 3, as expected. Separation Logic also allows *recursively-defined* heaps for reasoning over data-structures, such as **list**( $l$ ) and **tree**( $t$ ) from Section 1.

Separation Logic triples also have a slightly different meaning versus Hoare triples regarding *memory-safety*. A Separation Logic triple  $\{\phi\} C \{\varphi\}$  additionally guarantees that any state satisfying  $\phi$  will not cause a memory access violation in  $C$ . For example, the triple  $\{\mathbf{emp}\} [x] := 1 \{x \mapsto 1\}$  is invalid since  $x$  is a dangling pointer in any state satisfying the pre-condition.

## 3 Heaps with Separation

This section formally introduces the syntax and semantics of heaps with separation, which we denote by  $\mathcal{H}$ , that encodes some of the logical connectives

of Separation Logic. We assume as given a countably infinite set **Values** denoting *values*, e.g.  $\mathbf{Values} = \mathbb{Z}$ . A *heap* is a *finite partial map* between **Values**, i.e.  $\mathbf{Heaps} = \mathbf{Values} \rightarrow_{\text{fin}} \mathbf{Values}$ . This is the same definition as used by Separation Logic. Given a heap  $h \in \mathbf{Heaps}$  with domain  $D = \text{dom}(h)$ , we sometimes treat  $h$  as the set of pairs  $\{(p, v) \mid p \in D \wedge v = h(p)\}$ .

The  $\mathcal{H}$ -language is the first-order language over heaps defined as follows:

**Definition 1 (Heap Language).** We define the  $\mathcal{H}$ -signature  $\Sigma_{\mathcal{H}}$  as follows:

- *sorts*: **Values**, **Heaps**;
- *constants*: (empty heap)  $\emptyset$  of sort **Heaps**;
- *functions*: (singleton heap)  $(\_ \mapsto \_)$  of sort  $\mathbf{Values} \times \mathbf{Values} \mapsto \mathbf{Heaps}$ .
- *predicates*: (heap constraint)  $(\_ * \dots * \_ \simeq \_ * \dots * \_)$  of sort  $\mathbf{Heaps} \times \dots \times \mathbf{Heaps} \mapsto \{\text{true}, \text{false}\}$ .

The  $\mathcal{H}$ -language is the *first-order language* over  $\Sigma_{\mathcal{H}}$ . □

Example 1 used heap constraints of the form  $H \simeq H_1 * H_2$ , where  $H$ ,  $H_1$ , and  $H_2$  are variables. Throughout this paper we shall use upper-case letters  $H$ ,  $I$ ,  $J$ , etc., to denote *heap variables*, and lower-case letters  $p$ ,  $v$ , etc., for *value variables*.

A *valuation*  $s$  (a.k.a. *variable assignment*) is a function mapping values to  $\mathbf{Values} \cup \mathbf{Heaps}$ . We define the semantics of the  $\mathcal{H}$ -language as follows:

**Definition 2 (Heap Interpretation).** Given a valuation  $s$ , the  $\mathcal{H}$ -interpretation  $\mathcal{I}$  is a  $\Sigma_{\mathcal{H}}$ -interpretation such that:

- $\mathcal{I}(v, s) = s(v)$ , where  $v$  is a variable;
- $\mathcal{I}(\emptyset, s) = \emptyset$  (as a **Heap**);
- $\mathcal{I}(p \mapsto v, s) = \{(q, w) \mid \text{where } q = \mathcal{I}(p, s) \text{ and } w = \mathcal{I}(v, s)\}$ ;
- $\mathcal{I}(H_1 * \dots * H_i \simeq H_{i+1} * \dots * H_n, s) = \text{true}$  iff for  $h_i = \mathcal{I}(H_i, s)$  we have that:
  1.  $\text{dom}(h_1) \cap \dots \cap \text{dom}(h_i) = \emptyset$  and  $\text{dom}(h_{i+1}) \cap \dots \cap \text{dom}(h_n) = \emptyset$ ; and
  2.  $h_1 \cup \dots \cup h_i = h_{i+1} \cup \dots \cup h_n$  □

Note that we treat each configuration of  $(*)$  and  $(\simeq)$  as a distinct predicate. Intuitively, a constraint like  $H \simeq H_1 * H_2$  treats  $(*)$  in essentially the same way as *separating conjunction* from Separation Logic, except that we give a name  $H$  to the conjoined heaps  $H_1 * H_2$ .

We define  $\models_{\mathcal{H}} \dots [s]$  as the *satisfaction relation* such that  $\models_{\mathcal{H}} \phi [s]$  holds iff  $\mathcal{I}(\phi, s) = \text{true}$  for all heap formulae  $\phi$ . We also say that  $\phi$  is *valid* if  $\models_{\mathcal{H}} \phi [s]$  holds for all  $s$ , and *satisfiable* if  $\models_{\mathcal{H}} \phi [s]$  holds for at least one  $s$ .

### 3.1 Normalization

In the absence of quantifiers, we can restrict consideration of  $\mathcal{H}$ -formula to a subset in *normal form* defined as follows:

**Definition 3 (Normal Form).** A quantifier-free (QF)  $\mathcal{H}$ -formula  $\phi$  is in *normal form* if (1) all heap constraints are restricted to three basic forms:

$$H \simeq \emptyset \qquad H \simeq (p \mapsto v) \qquad H \simeq H_1 * H_2$$

where  $p$ ,  $v$ ,  $H$ ,  $H_1$ , and  $H_2$  are distinct variables, and (2) there are no negated heap constraints. □

|                                                                                                                                                                                                                                                                             |                              |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------|
| $H \simeq E_1 * E_2 * S \longrightarrow H' \simeq E_1 * E_2 \wedge H \simeq H' * S$                                                                                                                                                                                         |                              |
| $H \simeq E_1 * E_2 \longrightarrow H' \simeq E_1 \wedge H \simeq H' * E_2$                                                                                                                                                                                                 | $(E_1 \text{ non-variable})$ |
| $H \simeq H_1 * E_2 \longrightarrow H' \simeq E_2 \wedge H \simeq H_1 * H'$                                                                                                                                                                                                 | $(E_2 \text{ non-variable})$ |
| $H_1 \simeq H_2 \longrightarrow H' \simeq \emptyset \wedge H_1 \simeq H_2 * H'$                                                                                                                                                                                             |                              |
| $H \not\simeq E_1 * E_2 * S \longrightarrow \bigvee \begin{cases} E_1 \simeq (s \mapsto t) * H'_1 \wedge E_2 \simeq (s \mapsto u) * H'_2 \\ H' \simeq E_1 * E_2 \wedge H \not\simeq H' * S \end{cases}$                                                                     |                              |
| $H \not\simeq E_1 * E_2 \longrightarrow H' \simeq E_1 \wedge H \not\simeq H' * E_2$                                                                                                                                                                                         | $(E_1 \text{ non-variable})$ |
| $H \not\simeq H_1 * E_2 \longrightarrow H' \simeq E_2 \wedge H \not\simeq H_1 * H'$                                                                                                                                                                                         | $(E_2 \text{ non-variable})$ |
| $H \not\simeq \emptyset \longrightarrow H \simeq (s \mapsto t) * H'$                                                                                                                                                                                                        |                              |
| $H \not\simeq (p \mapsto v) \longrightarrow \bigvee \begin{cases} H \simeq \emptyset \\ H \simeq (s \mapsto t) * H' \wedge (p \neq s \vee v \neq t) \end{cases}$                                                                                                            |                              |
| $H \not\simeq H_1 * H_2 \longrightarrow \bigvee \begin{cases} H_1 \simeq (s \mapsto t) * H'_1 \wedge H_2 \simeq (s \mapsto u) * H'_2 \\ H' \simeq H_1 * H_2 \wedge H \not\simeq H' \end{cases}$                                                                             |                              |
| $H_1 \not\simeq H_2 \longrightarrow \bigvee \begin{cases} H_1 \simeq (s \mapsto t) * H'_1 \wedge H_1 \simeq (s \mapsto u) * H'_2 \wedge t \neq u \\ H_1 \simeq I * H'_1 \wedge H_2 \simeq I * H'_2 \wedge H' \simeq H'_1 * H'_2 \wedge H' \not\simeq \emptyset \end{cases}$ |                              |

**Fig. 1.**  $\mathcal{H}$ -formulae normalization rewrite rules

Any given QF  $\mathcal{H}$ -formula  $\phi$  can be rewritten into *normal form* using the following steps: (1) push negation inwards using De Morgan's laws, and (2) transform the resulting formula using the rewrite rules from Figure 1. Here each rewrite rule is of the form (*head*  $\longrightarrow$  *body*), and  $E_i$  runs over *heap expressions* ( $H, \emptyset, (p \mapsto v)$ ),  $S$  runs over  $(*)$ -sequences of heap expressions ( $E, E * E$ , etc.), and everything else runs over the variable symbols. A variable that appears in a rule body, but not the rule head, is taken to represent a fresh variable symbol that is introduced each time the rule is applied. For brevity we omit some rules, namely: normalizing the RHS of a  $(\simeq)$  to a heap variable (as this mirrors the LHS rules), and making variables unique. The main result for normalization is as follows:

**Proposition 1 (Normal Form).** *For all QF  $\mathcal{H}$ -formulae  $\phi$  there exists a QF  $\mathcal{H}$ -formula  $\varphi$  such that (1)  $\varphi$  is in normal form and (2): for all valuations  $s$  there exists a valuation  $s'$  such that  $\models_{\mathcal{H}} \phi [s]$  iff  $\models_{\mathcal{H}} \varphi [s']$  and  $s(v) = s'(v)$  for all  $v \in \text{vars}(\phi)$ .*

*Proof.* (Sketch) By the correctness of, and induction over, the normalization steps from Figure 1.  $\square$

Proposition 1 means that, at the expense of an increased formula size, we need only consider a limited subset of the  $\mathcal{H}$ -language that lacks negation.



### 3.2 Extensions

We may extend Definitions 1 and 2 to include other kinds of heap constraints, such as:

- *Heap union*  $H \simeq H_1 \sqcup H_2$  holds iff there exists a  $h \in \mathbf{Heaps}$  such that  $h = s(H_1) \cup s(H_2)$  as sets and  $s(H) = h$ .
- *Heap intersection*  $H \simeq H_1 \sqcap H_2$  holds iff  $s(H) = s(H_1) \cap s(H_2)$  as sets.
- *Heap subset*  $H_1 \sqsubseteq H_2$  holds iff  $s(H_1) \subseteq s(H_2)$  as sets.

These constraints can similarly be reduced to the normal form from Definition 3.

For some applications we may extend  $\mathcal{H}$  with ad hoc *user-defined* heap constraints. For this we can use *Constraint Logic Programming* (CLP) [13] over  $\mathcal{H}$ , i.e.  $\text{CLP}(\mathcal{H})$ . For example, the following  $\text{CLP}(\mathcal{H})$  predicate  $\mathbf{list}(l, L)$  specifies a skeleton *list constraint* under the standard *least model semantics* of CLP:

$$\begin{aligned} \mathbf{list}(0, L) &:- L \simeq \emptyset \\ \mathbf{list}(l, L) &:- l \neq 0 \wedge L \simeq (l \mapsto n) * L' \wedge \mathbf{list}(n, L') \end{aligned}$$

We can similarly define predicates for trees and arrays. The inclusion of CLP predicates requires stronger reasoning power in contrast to the base  $\mathcal{H}$ -language. For this we can employ standard (yet incomplete) methods such as [15].

## 4 Program Reasoning with $\mathcal{H}$

The core motivation of the  $\mathcal{H}$ -language is reasoning over heap manipulating programs. For this we consider the following extensions of *Hoare Logic* [11].

### 4.1 Direct Separation Logic Encoding

Separation Logic [20] is itself an extension of Hoare Logic. Given the similarity in the heap representations, we can re-encode the axioms of Separation Logic directly into Hoare axioms over  $\mathcal{H}$ -formulae, as shown in Figure 2(B). Each axiom is defined in terms of one of five auxiliary constraints: namely **allocated**, **access**, **assign**, **alloc**, and **free** defined in Figure 2(A), which are themselves defined in terms of  $\mathcal{H}$ -formulae. The **allocated**( $H, x$ ) constraint represents that pointer  $x$  is *allocated* in heap  $H$ , i.e.  $H \simeq (x \mapsto v) * H'$  for some  $v$  and  $H'$ . The remaining auxiliary constraints encode a *heap manipulation statement* as an  $\mathcal{H}$ -formula. The statements are:

- *heap access* ( $x := [y]$ ) sets  $x$  to be the value pointed to by  $y$ ;
- *heap assignment* ( $[x] := y$ ) sets the value pointed to by  $x$  to be  $y$ ;
- *heap allocation* ( $x := \mathbf{alloc}()$ ) sets  $x$  to point to a freshly allocated heap cell.<sup>1</sup>
- *heap free* **free**( $x$ ) deallocates the cell pointed to by  $x$ .

These axioms manipulate the *global heap* that is represented by a *distinguished heap variable*  $\bar{H}$ . Under this treatment,  $\bar{H}$  is an *implicit program variable*<sup>2</sup> of type

<sup>1</sup> Here we assume the (de)allocation of single heap cells. This can be generalized.

<sup>2</sup> The variable is “implicit” in the sense that it is not explicitly represented in the syntax of the programming language.

|     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
|-----|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| (A) | $\begin{aligned} \text{allocated}(H, p) &\stackrel{\text{def}}{=} \exists H', v : H \simeq (p \mapsto v) * H' \\ \text{access}(H, p, v) &\stackrel{\text{def}}{=} \exists H' : H \simeq (p \mapsto v) * H' \\ \text{assign}(H_{OLD}, p, v, H_{NEW}) &\stackrel{\text{def}}{=} \exists H'', w : \left\{ \begin{array}{l} H_{OLD} \simeq (p \mapsto w) * H'' \\ H_{NEW} \simeq (p \mapsto v) * H'' \end{array} \right. \wedge \\ \text{alloc}(H_{OLD}, p, H_{NEW}) &\stackrel{\text{def}}{=} \exists v : H_{NEW} \simeq (p \mapsto v) * H_{OLD} \\ \text{free}(H_{OLD}, p, H_{NEW}) &\stackrel{\text{def}}{=} \exists v : H_{OLD} \simeq (p \mapsto v) * H_{NEW} \end{aligned}$ |
| (B) | $\frac{}{\{\phi \wedge \text{allocated}(\bar{\mathcal{H}}, y)\} \ x := [y] \ \{\exists x' : \text{access}(\bar{\mathcal{H}}, y, x) \wedge \phi[x'/x]\}}$ $\frac{}{\{\phi \wedge \text{allocated}(\bar{\mathcal{H}}, x)\} \ [x] := y \ \{\exists H' : \text{assign}(H', x, y, \bar{\mathcal{H}}) \wedge \phi[H'/\bar{\mathcal{H}}]\}}$ $\frac{}{\{\phi\} \ x := \text{alloc}() \ \{\exists x', H' : \text{alloc}(H', x, \bar{\mathcal{H}}) \wedge \phi[H'/\bar{\mathcal{H}}, x'/x]\}}$ $\frac{}{\{\phi \wedge \text{allocated}(\bar{\mathcal{H}}, x)\} \ \text{free}(x) \ \{\exists H' : \text{free}(H', x, \bar{\mathcal{H}}) \wedge \phi[H'/\bar{\mathcal{H}}]\}}$           |
| (C) | $\frac{\{p(\bar{\mathcal{H}})\} \ C \ \{q(\bar{\mathcal{H}})\}}{\{\bar{\mathcal{H}} \simeq P * R \wedge p(P) \wedge r(R)\} \ C \ \{\exists Q, R' : \bar{\mathcal{H}} \simeq Q * R' \wedge q(Q) \wedge r(R')\}}$                                                                                                                                                                                                                                                                                                                                                                                                                                                               |

**Fig. 2.** (A) Auxiliary constraint definitions, (B) basic Hoare inference rules, and (C) the Frame Rule

Heap that is assumed to be threaded throughout the program. Other axioms of Separation Logic, such as the *Frame Rule* [20], can similarly be re-encoded, as shown in Figure 2(C).

It is not surprising that Separation Logic can be re-formulated as Hoare axioms over the  $\mathcal{H}$ -language. However, there are some important differences to consider. Notably, the  $\mathcal{H}$ -encoding allows for *explicit heap variables* to express relationships *between* heaps across triples. In Example 1, we use the triple  $\{H \simeq \bar{\mathcal{H}}\} \ C \ \{H \simeq \bar{\mathcal{H}}\}$  to express the property that the code fragment  $C$  does not change the global heap  $\bar{\mathcal{H}}$  through an explicit variable  $H$ . Such a global property would require *second order* Separation Logic, e.g.,  $\forall h : \{h\} \ C \ \{h\}$ . Furthermore, with explicit heap variables, we can strengthen the *Frame Rule* by  $R'$  for  $R$  in the post-condition of Figure 2(C).

The  $\mathcal{H}$ -based encoding tends to be more verbose compared to Separation Logic, which favors more concise formulae. Whilst not so important for *automated* systems, the  $\mathcal{H}$ -based encoding is likely less suitable for *manual* proofs of correctness.

$$\boxed{
 \begin{array}{c}
 \frac{}{\{\phi\} \ x := [y] \ \{\exists x' : \text{access}(\bar{\mathcal{H}}, y, x) \wedge \phi[x'/x]\}} \\
 \\
 \frac{}{\{\phi\} \ [x] := y \ \{\exists H' : \text{assign}(H', x, y, \bar{\mathcal{H}}) \wedge \phi[H'/\bar{\mathcal{H}}]\}} \\
 \\
 \frac{}{\{\phi\} \ \text{free}(x) \ \{\exists H' : \text{free}(H', x, \bar{\mathcal{H}}) \wedge \phi[H'/\bar{\mathcal{H}}]\}}
 \end{array}
 }$$

**Fig. 3.** Alternative Hoare inference rules

## 4.2 Strongest Post-condition Encoding

Separation Logic and the corresponding  $\mathcal{H}$ -encoding from Figure 2 (B) enforces *total correctness* w.r.t. *memory safety*. That is, a valid triple  $\{\phi\} \ C \ \{\varphi\}$  additionally ensures that any state satisfying  $\phi$  will not cause a *memory fault* (e.g. dereferencing a dangling pointer) when executed by  $C$ . This is enforced by the *access*, *assignment*, and *free* axioms of Figure 2 (B) by requiring that the pointer  $x$  be allocated in the global heap  $\mathcal{H}$  in the pre-condition via the  $\text{allocated}(\bar{\mathcal{H}}, x)$  constraint.

Memory safety has implications for *forward reasoning* methods such as *symbolic execution*. For example, to symbolically execute a formula  $\phi$  through an assignment  $[x] := v$ , we must first prove that  $\phi \rightarrow \text{allocated}(\bar{\mathcal{H}}, x)$ . Such a proof can be arbitrarily difficult in general, e.g. for formulae with quantifiers or recursively-defined  $\text{CLP}(\mathcal{H})$  predicates. Furthermore, if memory safety is *not* a property of interest, this extra work is unnecessary.

By decoupling the heap representation ( $\mathcal{H}$ ) from the logic, we can experiment with alternative axiomatizations. One such axiomatization that is *partially correct* modulo memory safety is shown in Figure 3.<sup>3</sup> This version drops the requirement that  $x$  be allocated in  $\bar{\mathcal{H}}$  in the pre-condition, and therefore treats memory errors the same way as *undefined behavior* (or *non-termination*) in classic Hoare Logic.

There are several advantages to the weaker axiomatization of Figure 3. Firstly, the axioms of Figure 3 specify a *Strongest Post Condition* (SPC) *predicate transformer semantics* and is therefore immediately suitable for automated forward based reasoning techniques such as symbolic execution. This is in contrast to symbolic execution in Separation Logic [4] (or the corresponding axioms from Figure 2), where symbolic execution requires the  $\text{allocated}$  condition to be separately proven. The SPC axiomatization allows for weaker, more concise, specifications.

*Example 2 (Double List Reverse).* For example, consider the following triples in the spirit of Example 1:

<sup>3</sup> The axiom for *heap allocation* is the same as Figure 2 (B).

$$\{H \simeq \bar{H}\} \ l := \mathbf{reverse}(\mathbf{reverse}(l)) \ \{H \simeq \bar{H}\} \quad (2)$$

$$\{\bar{H} \simeq L * H' \wedge \mathbf{list}(L, l) \wedge H \simeq \bar{H}\} \ l := \mathbf{reverse}(\mathbf{reverse}(l)) \ \{H \simeq \bar{H}\} \quad (3)$$

Both attempt to state the same property: that double *in-place list-reverse* leaves the global heap  $\bar{H}$  unchanged. Suppose that the only property of interest is the heap equivalence (i.e. not memory safety). Triple (2) is valid under the weaker Figure 3 axiomatization, but not the stronger Figure 2 (B) version which requires memory safety. The latter requires a more complex specification, such as Triple (3), where the recursively defined property  $\mathbf{list}(L, l)$  ensures  $l$  points to a valid allocated list.  $\square$

There are also some disadvantages to consider. For obvious reasons, the SPC axiomatization is unsuitable if memory safety *is* a property of interest. Furthermore, the soundness of Separation Logic’s *Frame Rule* (or Figure 2 (C)) depends on memory safety, and thus is not valid under the new interpretation. Therefore the SPC axiomatization is not suitable for Separation Logic-style *local reasoning* proofs. In essence, this is a trade-off between local reasoning vs. making symbolic execution “easier”, highlighting the flexibility of our overall approach.

## 5 A Solver for $\mathcal{H}$ -Formulae

Automated symbolic execution depends on an  $\mathcal{H}$ -solver to discharge the generated *Verification Conditions* (VCs). In this section we present a simple, yet sound and complete, algorithm for solving the quantifier-free (QF) fragment of the  $\mathcal{H}$ -language.

**Algorithm.** The  $\mathcal{H}$ -solver algorithm is based on the *propagation* of *heap membership* and *(dis)equality* constraints. Heap membership (a.k.a. heap element) is represented by an auxiliary  $\mathbf{in}(H, p, v)$  constraint, which is defined as follows:

**Definition 4 (Heap Membership).** We extend Definitions 1 and 2 to include the *heap membership* constraint  $\mathbf{in}(H, p, v)$  defined as follows:

$$\models_{\mathcal{H}} \mathbf{in}(H, p, v) \ [s] \quad \text{iff} \quad (s(p), s(v)) \in s(H)$$

where  $H$ ,  $p$ , and  $v$  are variables.  $\square$

Heap element  $\mathbf{in}(H, p, v)$  is analogous to set membership  $x \in S$  from set theory. (Dis)equality is propagated via the usual  $x = y$  and  $x \neq y$  constraints.

The  $\mathcal{H}$ -solver operates over conjunctions of *normalized*  $\mathcal{H}$ -constraints as per Definition 3. Arbitrary QF  $\mathcal{H}$ -formula  $\phi$  can be normalized to a  $\varphi$  using the rules from Figure 1, such that the solutions to  $\phi$  and  $\varphi$  correspond as per Proposition 1. The arbitrary Boolean structure of  $\varphi$  can be handled using the *Davis-Putnam-Logemann-Loveland* (DPLL) algorithm [6] modulo the  $\mathcal{H}$ -solver.

|                                                                                                               |     |
|---------------------------------------------------------------------------------------------------------------|-----|
| $\text{in}(H, p, v) \wedge \text{in}(H, p, w) \Longrightarrow v = w$                                          | (1) |
| $H \simeq \emptyset \wedge \text{in}(H, p, v) \Longrightarrow \text{false}$                                   | (2) |
| $H \simeq (p \mapsto v) \Longrightarrow \text{in}(H, p, v)$                                                   | (3) |
| $H \simeq (p \mapsto v) \wedge \text{in}(H, q, w) \Longrightarrow p = q \wedge v = w$                         | (4) |
| $H \simeq H_1 * H_2 \wedge \text{in}(H, p, v) \Longrightarrow \text{in}(H_1, p, v) \vee \text{in}(H_2, p, v)$ | (5) |
| $H \simeq H_1 * H_2 \wedge \text{in}(H_1, p, v) \Longrightarrow \text{in}(H, p, v)$                           | (6) |
| $H \simeq H_1 * H_2 \wedge \text{in}(H_2, p, v) \Longrightarrow \text{in}(H, p, v)$                           | (7) |
| $H \simeq H_1 * H_2 \wedge \text{in}(H_1, p, v) \wedge \text{in}(H_2, q, w) \Longrightarrow p \neq q$         | (8) |

**Fig. 4.**  $\mathcal{H}$ -solver CHR propagation rules

We specify the  $\mathcal{H}$ -solver as a set of *Constraint Handling Rules* [10] with disjunction ( $\text{CHR}^\vee$ ) [1] as shown in Figure 4. Here each rule (*Head*  $\Longrightarrow$  *Body*) encodes *constraint propagation*, where the constraints *Body* are added to the store whenever a matching *Head* is found. Rule (1) encodes the functional dependency for finite partial maps; rules (2)–(4) encode propagation for *heap empty*  $H = \emptyset$  and *heap singleton*  $H \simeq (p \mapsto v)$  constraints; and rules (5)–(8) encode heap membership propagation through *heap separation*  $H \simeq H_1 * H_2$  constraints. Most of these rules are self-explanatory, e.g., rule (6) states that if  $H \simeq H_1 * H_2$  and  $\text{in}(H_1, p, v)$ , then it must be the case that  $\text{in}(H, p, v)$ , since  $H_1$  is a sub-heap of  $H$ . We assume a complete solver for the underlying equality theory ( $x = y$ ,  $x \neq y$ ).

The  $\mathcal{H}$ -solver employs the standard  $\text{CHR}^\vee$  execution algorithm with the rules from Figure 4. We shall present a semi-formal summary below. The input is a *constraint store*  $S$  defined to be a set<sup>4</sup> of constraints (representing a conjunction). Let *Rules* be the rules from Figure 4, then the algorithm  $\text{hsolve}(S)$  is recursively defined as follows:

- (*Propagation Step*) If there exists  $R \in \text{Rules}$  of the form  $(h_1 \wedge \dots \wedge h_n \Longrightarrow \text{Body})$ , a subset  $\{c_1, \dots, c_n\} \subseteq S$  of constraints, a subset  $E \subseteq S$  of *equality* constraints, and a *matching substitution*  $\theta$  such that:  $E \rightarrow (\theta.h_i = c_i)$  for  $i \in 1..n$  then rule  $R$  is *applicable* to the store  $S$ . We *apply* rule  $R$  as follows:
  - If  $\text{Body} = \text{false}$  then return *false*;
  - If  $\text{Body} = d_1 \wedge \dots \wedge d_m$  then return  $\text{hsolve}(S \cup \theta.\{d_1, \dots, d_m\})$ ; else
  - If  $\text{Body} = d_1 \vee \dots \vee d_m$  then let  $S_i := \text{hsolve}(S \cup \theta.\{d_i\})$  for  $i \in 1..m$ . If there exists an  $S_i \neq \text{false}$  then return  $S_i$ , else return *false*.
- Else if no such  $R$  exists, return  $S$ .

Propagation proceeds until failure occurs or a fixed point is reached.

*Example 3* ( $\mathcal{H}$ -Solving). Consider the following goal  $G$ :

$$H \simeq (p \mapsto v) \wedge H \simeq I * J \wedge J \simeq (p \mapsto w) \wedge v \neq w$$

<sup>4</sup> We assume a set-based CHR semantics.

|                                                                                                                                                                                |     |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| $\{H \simeq (p \mapsto v), H \simeq I * J, J \simeq (p \mapsto w), v \neq w\}$                                                                                                 | (3) |
| $\{H \simeq (p \mapsto v), H \simeq I * J, \underline{J \simeq (p \mapsto w)}, v \neq w, \text{in}(H, p, v)\}$                                                                 | (3) |
| $\{H \simeq (p \mapsto v), \underline{H \simeq I * J}, J \simeq (p \mapsto w), v \neq w, \text{in}(H, p, v), \text{in}(J, p, w)\}$                                             | (7) |
| $\{H \simeq (p \mapsto v), H \simeq I * J, J \simeq (p \mapsto w), v \neq w, \underline{\text{in}(H, p, v)}, \underline{\text{in}(J, p, w)}, \underline{\text{in}(H, p, w)}\}$ | (1) |
| $\{H \simeq (p \mapsto v), H \simeq I * J, J \simeq (p \mapsto w), \underline{v \neq w}, \underline{\text{in}(H, p, v)}, \text{in}(J, p, w), \underline{v = w}\}$              | (E) |
| <i>false</i>                                                                                                                                                                   |     |

**Fig. 5.**  $\mathcal{H}$ -solving constraint propagation steps

We wish to show that this goal is unsatisfiable using the  $\mathcal{H}$ -solver from Figure 4. Initially the constraint store contains the initial goal  $G$ . Constraint propagation proceeds as shown in Figure 5. Here we apply rules (3), (3), (7), (1), (E) to the underlined constraint(s) in order, where (E) represents an inference made by the underlying equality solver. Propagation leads to failure, and there are no branches – therefore goal  $G$  is unsatisfiable.  $\square$

Since all the rules from Figure 4 are *propagation rules*, the solving algorithm  $\text{hsolve}(G)$  will always terminate with some final store  $S$ . The  $\mathcal{H}$ -solver is both sound and complete w.r.t. (un)satisfiability.

**Proposition 2 (Soundness).** *For all  $G, S$ , if  $\text{hsolve}(G) = S$ , then for all valuations  $s$ ,  $\models_{\mathcal{H}} G[s]$  iff  $\models_{\mathcal{H}} S[s]$ .*

*Proof.* (Sketch) By the correctness of the rules from Figure 4 w.r.t. Definitions 2 and 4.  $\square$

**Proposition 3 (Completeness).** *For all  $G, S$  such that  $\text{hsolve}(G) = S$ , then  $\not\models_{\mathcal{H}} G[s]$  for all valuations  $s$  (i.e.  $G$  is unsatisfiable) iff  $S = \text{false}$ .*

*Proof.* (Sketch) The “ $\Leftarrow$ ” direction follows from Proposition 2. We consider the “ $\Rightarrow$ ” direction. The rest is proof by contrapositive: assuming  $S \neq \text{false}$  we show that there exists a valuation  $s$  such that  $\models_{\mathcal{H}} G[s]$ . Let  $s_E$  be a valuation for the underlying equality subset of  $S$  over integer variables, then let  $s(v) = s_E(v)$  for integer variables, and

$$s(H) = \{(s_E(p), s_E(v)) \mid \text{in}(H, p, v) \in S\} \quad (4)$$

for all heap variables  $H$ . Assume that  $\not\models_{\mathcal{H}} S[s]$ . By case analysis of Definition 2 we find that a rule must be applicable:

- Case  $s(H) \notin \text{Heaps}$ : Rule (1);
- Case  $H \simeq \emptyset$  and  $s(H) \neq \emptyset$ : Rule (2);
- Case  $H \simeq (p \mapsto v)$  and  $s(H) \neq \{(p, v)\}$ : Rules (3) or (4);
- Case  $H \simeq H_1 * H_2$  and  $s(H) \neq s(H_1) \cup s(H_2)$ : Rules (5), (6), or (7);
- Case  $H \simeq H_1 * H_2$  and  $\text{dom}(s(H_1)) \cap \text{dom}(s(H_2)) \neq \emptyset$ : Rule (8)

This contradicts the assumption that  $S$  is a final store, therefore if  $S \neq \text{false}$  then  $\models_{\mathcal{H}} S[s]$ , and therefore  $\models_{\mathcal{H}} G[s]$  by Proposition 2 completes the proof.  $\square$

|                                                                                                                                                                                                                                                                                                                   |      |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------|
| $H \neq \emptyset \implies \text{in}(H, s, t)$                                                                                                                                                                                                                                                                    | (9)  |
| $H \neq (p \mapsto v) \implies \vee \begin{cases} H \simeq \emptyset \\ \text{in}(H, s, t) \wedge (s \neq p \vee t \neq v) \end{cases}$                                                                                                                                                                           | (10) |
| $H \neq H_1 * H_2 \implies \vee \begin{cases} \text{in}(H, s, t) \wedge \neg \text{in}(H_1, s, t) \wedge \neg \text{in}(H_2, s, t) \\ \text{in}(H_1, s, t) \wedge \neg \text{in}(H, s, t) \\ \text{in}(H_2, s, t) \wedge \neg \text{in}(H, s, t) \\ \text{in}(H_1, s, t) \wedge \text{in}(H_2, s, u) \end{cases}$ | (11) |
| $\text{access}(H, p, v) \iff \text{in}(H, p, v)$                                                                                                                                                                                                                                                                  | (12) |
| $\text{assign}(H_0, p, v, H_1) \implies \text{in}(H_0, p, w) \wedge \text{in}(H_1, p, v)$                                                                                                                                                                                                                         | (13) |
| $\text{assign}(H_0, p, v, H_1) \wedge \text{in}(H_0, q, w) \implies p = q \vee \text{in}(H_1, q, w)$                                                                                                                                                                                                              | (14) |
| $\text{assign}(H_0, p, v, H_1) \wedge \text{in}(H_1, q, w) \implies p = q \vee \text{in}(H_0, q, w)$                                                                                                                                                                                                              | (15) |
| $\text{alloc}(H_0, p, H_1) \implies \text{in}(H_1, p, v)$                                                                                                                                                                                                                                                         | (16) |
| $\text{alloc}(H_0, p, H_1) \wedge \text{in}(H_0, q, w) \implies p \neq q \wedge \text{in}(H_1, q, w)$                                                                                                                                                                                                             | (17) |
| $\text{alloc}(H_0, p, H_1) \wedge \text{in}(H_1, q, w) \implies p = q \vee \text{in}(H_0, q, w)$                                                                                                                                                                                                                  | (18) |
| $\text{free}(H_0, p, H_1) \iff \text{alloc}(H_1, p, H_0)$                                                                                                                                                                                                                                                         | (19) |

**Fig. 6.** Extended  $\mathcal{H}$ -solver propagation rules

The proof for Proposition 3 is constructive; namely, (4) can be used to construct a *solution* for a satisfiable goal  $G$ . Furthermore, we can combine the normalization of Proposition 1 and DPLL(hsolve) to derive a sound and complete algorithm for solving arbitrary QF  $\mathcal{H}$ -formulae  $\phi$ .

## 5.1 Extensions

The propagation rules from Figure 4 define a solver for the base  $\mathcal{H}$ -language. We can use heap membership propagation to define rules for other kinds of  $\mathcal{H}$ -constraints, as shown in Figure 6.

Rules (9)–(11) handle the negations of the base  $\mathcal{H}$ -constraints from Definition 3. These rules are an alternative to the decomposition from Figure 1. We can also define rules for directly handling the auxiliary constraints from Figure 2 (A) for program reasoning. For example, rules (13)–(15) handle the  $\text{assign}(H_0, p, v, H_1)$  constraint. We similarly provide rules for the other auxiliary constraints. Here, variables appearing in a rule body but not in the rule head are interpreted the same way as with Figure 1.

## 6 Experiments

In this section we test an implementation of the  $\mathcal{H}$ -solver against *verification conditions* (VCs) derived from symbolic execution. We compare against Verifast [12]

(version 12.12), a program verification system based on Separation Logic. Our motivation for the comparison is: (1) Verifast is based on forward symbolic execution, and (2) Verifast incorporates the notion of separation (via Separation Logic). That said, the Verifast execution algorithm [12] is very different from the  $\mathcal{H}$ -solver.

We have implemented a version of the  $\mathcal{H}$ -solver as part of the *Satisfiability Modulo Constraint Handling Rules* (SMCHR) [8] system.<sup>5</sup> SMCHR is a *Satisfiability Modulo Theories* (SMT) framework that supports theory (T) solvers implemented in CHR. The SMCHR system also supports several “built-in” theories, such as a linear arithmetic solver based on [9], that can be combined with the  $\mathcal{H}$ -solver to handle the underlying (dis)equality constraints. The SMCHR system has also been extended to support *disjunctive propagators* [19] for rules with disjunctive bodies, such as Rule (5).

For these benchmarks we either restrict ourselves to the fragment of Verifast that is fully automatable, or we provide the minimal annotations where appropriate. For the  $\mathcal{H}$ -solver, we have implemented a prototype symbolic execution tool as a GCC plug-in. Our tool symbolically executes GCC’s internal GIMPLE representation to generate *path constraints*. Given a *safety condition*  $\varphi$ , we generate the corresponding *verification condition*  $(\exists \bar{x} : \phi) \models \varphi$ , which is *valid* iff  $\phi \wedge \neg \varphi$  is *unsatisfiable*. Here  $\bar{x}$  represents existential variables introduced during symbolic execution. Unsatisfiability is tested for using the  $\mathcal{H}$ -solver.

A well-known problem with forward symbolic execution is the so-called *path explosion problem*. The number of paths through a (loop-free) program fragment can easily be exponential. We can mitigate this problem using *subsumption via interpolation* [14][17]. The basic idea is as follows: given a VC  $\phi_1 \models \varphi$  that holds for path  $\phi_1$ , we generate an *interpolant*  $\psi_1$  for  $\phi_1$ , that, by definition, satisfies  $\phi_1 \models \psi_1 \models \varphi$ . As symbolic execution continues, we can prune (subsume) all other paths with constraints  $\phi_2$  such that  $\phi_2 \models \psi_1$ . The key is that this pruning can occur *early*, as we construct the constraint for each path.

Our interpolation algorithm is based on an improved version of the *constraint deletion* idea from [14]. Given a path constraint  $\phi = c_1 \wedge \dots \wedge c_n$  we find a subset  $I \subseteq \{c_1, \dots, c_n\}$  such that  $I \wedge \neg \varphi$  remains *unsatisfiable*. For this we simply re-use the SAT solver’s *Unique Implication Point* UIP algorithm over the implication graph formed by the  $\mathcal{H}$ -solver propagation steps.

We test several programs that exhibit the path explosion problem. These include: `subsets_N` - sum-of-subsets size  $N$ ; `expr_N` - simple virtual machine executing  $N$  instructions; `stack_N` - for all  $M \leq N$ , do  $N$ -pushes, then  $N$ -pops; `filter_N` - filter for TCP/IP packets; `sort_N` - bubble-sort of length  $N$ ; `search234_N` - 234-tree search; `insert234_N` - 234-tree insert. Most of our examples are derived from unrolling loops of smaller programs.

The results are shown in Figure 7. Here **Safety** indicates the safety condition (defined below), **LOC** indicates the number of *lines-of-code*, **type** indicates heap operations used (with **r** = read, **w** = write, and **a** = allocation/deallocation), **#bt** is the number of backtracks for our prototype tool, and **#forks** is the number of

<sup>5</sup> SMCHR is available from <http://www.comp.nus.edu.sg/~gregory/smchr/>



|               |               |     |      | Heaps   |      | Verifast |        |
|---------------|---------------|-----|------|---------|------|----------|--------|
| <i>Bench.</i> | Safety        | LOC | type | time(s) | #bt  | time(s)  | #forks |
| subsets_16    | $F$           | 50  | rw-  | 0.00    | 17   | 10.69    | 65546  |
| expr_2        | $F$           | 69  | rw-  | 0.05    | 124  | 18.38    | 136216 |
| stack_80      | $F$           | 976 | rwa  | 8.66    | 320  | 68.20    | 9963   |
| filter_1      | $F$           | 192 | r--  | 0.03    | 80   | 0.75     | 8134   |
| filter_2      | $F$           | 321 | r--  | 0.11    | 307  | —        | —      |
| sort_6        | $F$           | 178 | rw-  | 0.03    | 54   | 2.66     | 35909  |
| search234_3   | $F$           | 251 | r--  | 0.02    | 46   | 0.67     | 1459   |
| search234_5   | $F$           | 399 | r--  | 0.05    | 76   | 90.65    | 118099 |
| insert234_5   | $F$           | 839 | rwa  | 1.19    | 120  | 52.87    | 36885  |
| expr_2        | $\sqsubseteq$ | 69  | rw-  | 0.20    | 1329 | n.a.     | n.a.   |
| stack_80      | $\sqsubseteq$ | 976 | rwa  | 8.07    | 322  | n.a.     | n.a.   |
| filter_2      | $OP$          | 321 | r--  | 0.00    | 2    | n.a.     | n.a.   |
| stack_80      | $A$           | 976 | rwa  | 8.90    | 320  | 65.68    | 9801   |
| insert234_5   | $A$           | 839 | rwa  | 1.50    | 60   | 40.64    | 55423  |
| subsets_16    | $\emptyset$   | 50  | rw-  | 0.00    | 33   | n.a.     | n.a.   |

**Fig. 7.** Theorem proving and symbolic execution benchmarks

symbolic execution forks for Verifast, and corresponds to the number of paths through the code. All experiments were run on GNU/Linux x86\_64 with a Intel<sup>®</sup> Core<sup>™</sup> i5-2500K CPU clocked at 4GHz. A timeout of 10 minutes is indicated by a dash (—). The safety conditions correspond to (some variant of) the following triples:

- *Framing* ( $F$ ) with  $\{\bar{H} \simeq (p \mapsto v) * F\} C \{\exists F' : \bar{H} \simeq (p \mapsto v) * F'\}$  where  $p$  is outside the footprint of the code  $C$ ;
- *Operations* ( $OP$ ) where  $OP \in \{\sqsubseteq, \sqsupseteq, \simeq\}$  with  $\{H \simeq \bar{H}\} C \{H \text{ } OP \text{ } \bar{H}\}$ ;
- *Allocation* ( $A$ ) with  $\{\dots\} C \{\exists F', v : \bar{H} \simeq (p \mapsto v) * F'\}$  for  $p$  allocated by  $C$ ;
- *Empty* ( $\emptyset$ ) with  $\{\bar{H} \simeq \emptyset\} C \{false\}$ , i.e.  $C$  will always fault on memory.

Some safety conditions, namely ( $OP$ ) and ( $\emptyset$ ), cannot be encoded directly in Separation Logic or Verifast, and are marked by “n.a.”.

Overall our tests exhibit significant search-space pruning thanks to interpolation. In contrast Verifast explores the entire search-space, and thus has exponential runtime behavior. However the *time-per-path* ratio favors Verifast, suggesting that Verifast would perform better on examples that do not have a large search-space, or when interpolation fails to subsume a significant number of branches. Our tool and SMT solver implementation are preliminary and can likely be further optimized.

## 7 Related Work

Several systems [3][5][12] implement Separation Logic-based symbolic execution, as described in [4]. However, due to the memory-safety requirements of Separation Logic, symbolic execution is limited to formulae over the footprint of the code. Our symbolic execution is based on the SPC Hoare Logic extension and

therefore works for arbitrary formulae. This is convenient when memory-safety is not a property of interest, such as Example 2.

Several automatic theorem provers for Separation Logic triples/formulae have been developed, including [4][5][18]. These systems generally rely on a set of *rearrangement rules*, and are usually limited to a subset of all formulae, e.g. those with no non-separating conjunction, etc. In contrast our  $\mathcal{H}$ -solver uses a different algorithm based on heap-membership propagation, and handles any arbitrary QF  $\mathcal{H}$ -formulae.

Other formalisms, such as (Implicit) Dynamic Frames [16][22] and Region Logic [2], also encode separation. The underlying approach is to represent the heap  $H$  as a (possibly implicit) *total map* over all possible addresses, and to represent access or modification rights as sets of addresses  $F$ . Separation is represented as set disjoint-ness, i.e.  $F_1 \cap F_2 = \emptyset$ . One difficulty is that we must relate  $H$  with  $F$ , which can make reasoning comparatively more difficult. For example, consider the following VCs:

$$p \notin F \wedge \mathbf{list}(H, F, l) \wedge \mathbf{assign}'(H, p, v, H') \models \mathbf{list}(H', F, l) \quad (5)$$

$$H \simeq L * R \wedge R \simeq (p \mapsto w) * R' \wedge \mathbf{list}(L, l) \wedge \mathbf{assign}(H, p, v, H') \models L \sqsubseteq H' \quad (6)$$

where  $\mathbf{assign}'$  is a suitable re-encoding of  $\mathbf{assign}$  for total heaps. Both VCs are natural encodings of the same problem: we wish to prove that  $l$  is still a list after writing to a (separate) pointer  $p$ . VC (6) holds independently of the recursively defined  $\mathbf{list}$  relation, and can be trivially disposed of using our  $\mathcal{H}$ -solver. In contrast, VC (5) depends on the recursively-defined  $\mathbf{list}$  predicate as it relates  $H$  with  $F$ , and is therefore more difficult to prove.

Our  $\mathcal{H}$ -solving algorithm is related to analogous algorithms for finite sets, such as [23]. Although formalized differently, the basic idea is similar, i.e. based on the propagation of *set membership*  $x \in S$  constraints. In [21] this idea was adapted into a decision procedure for Region Logic. Our approach works directly with heaps rather than indirectly via sets.

## 8 Future Work and Conclusions

In this paper we presented a reformulation of the key ideas behind Separation Logic as a first-order constraint-language  $\mathcal{H}$  over heaps. Here we express separation as a constraint between heaps. We present an SPC extension of Hoare Logic based on encoding of heap-manipulating statements in terms of  $\mathcal{H}$ -formulae. Our extension is suitable for forward reasoning via constraint-based symbolic execution. We present a sound and complete solver for QF  $\mathcal{H}$ -formulae and have implemented a version as part of an SMT framework. Experimental evaluation yields promising results.

There is significant scope for future work, such as: building theorem provers for recursively-defined properties based on the  $\mathcal{H}$ -solver, or further developing program verification tools using  $\mathcal{H}$ -language-based symbolic execution.

## References

1. Abdennadher, S., Schütz, H.: CHR<sup>∨</sup>: A flexible query language. In: Andreassen, T., Christiansen, H., Larsen, H.L. (eds.) FQAS 1998. LNCS (LNAI), vol. 1495, pp. 1–14. Springer, Heidelberg (1998)
2. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411. Springer, Heidelberg (2008)
3. Berdine, J., Calcagno, C., O’Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
4. Berdine, J., Calcagno, C., O’Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
5. Botinčan, M., Parkinson, M., Schulte, W.: Separation logic verification of c programs with an SMT solver. *Electronic Notes in Theoretical Computer Science* 254, 5–23 (2009)
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. *Communications of the ACM* 5(7), 394–397 (1962)
7. Dijkstra, E.: Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM* 18(8), 453–457 (1975)
8. Duck, G.: SMCHR: Satisfiability modulo constraint handling rules. *Theory and Practice of Logic Programming* 12(4-5), 601–618 (2012); *Proceedings of the 28th international conference on Logic Programming*
9. Dutertre, B., de Moura, L.: A fast linear-arithmetic solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)
10. Frühwirth, T.: Theory and practice of constraint handling rules. *Special Issue on Constraint Logic Programming, Journal of Logic Programming* 37 (October 1998)
11. Hoare, C.: An axiomatic basis for computer programming. *Communications of the ACM* 12(10), 576–580 (1969)
12. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)
13. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. *J. LP* 19(20), 503–581 (1994)
14. Jaffar, J., Santosa, A.E., Voicu, R.: An interpolation method for CLP traversal. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 454–469. Springer, Heidelberg (2009)
15. Jaffar, J., Santosa, A.E., Voicu, R.: A coinduction rule for entailment of recursively defined properties. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 493–508. Springer, Heidelberg (2008)
16. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
17. McMillan, K.L.: Lazy annotation for program testing and verification. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 104–118. Springer, Heidelberg (2010)

18. Nguyen, H.H., David, C., Qin, S., Chin, W.-N.: Automated verification of shape and size properties via separation logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
19. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. *Constraints* 14, 357–391 (2009)
20. Reynolds, J.C.: Separation logic: A logic for shared mutable data objects. In: 17th IEEE Symposium on Logic in Computer Science, pp. 55–74. IEEE Computer Society Press (2002)
21. Rosenberg, S., Banerjee, A., Naumann, D.A.: Decision procedures for region logic. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 379–395. Springer, Heidelberg (2012)
22. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
23. Zarba, C.G.: Combining sets with elements. In: Dershowitz, N. (ed.) *Verification: Theory and Practice*. LNCS, vol. 2772, pp. 762–782. Springer, Heidelberg (2004)

# Model Combinators for Hybrid Optimization

Daniel Fontaine<sup>1</sup>, Laurent Michel<sup>1</sup>, and Pascal Van Hentenryck<sup>2</sup>

<sup>1</sup> University of Connecticut, Storrs, CT 06269-2155

<sup>2</sup> NICTA, University of Melbourne

**Abstract.** In recent years, CML, G12 and SIMPL, have achieved significant progress in automating the generation of hybrid solvers from high-level model specifications. This paper pushes this research direction one step further and introduces the concept of model combinators to provide principled model compositions. These model combinators rely on *runnables* capturing executable models, *runnable signatures* that capture what runnables can produce and consume, and *model hierarchies*, which track relationships among models. These concepts make it possible to enforce the soundness of model compositions and to determine the best model compositions automatically. A prototype of the framework on top of the OBJECTIVE-CP optimization system is presented.

## 1 Introduction

The *Comet Modeling Language* (CML) [6] demonstrated that the burden of writing a suitable solver for hard industrial problems can be greatly mitigated with high-level language abstractions. With CML, models are specified abstractly in a technology agnostic manner, and then *concretized* into one or more models based on Constraint Programming (CP), Integer Programming (IP) or Constraint-Based Local Search (CBLS) technologies. Concretization typically relied on transformation and reformulation based on rewrite rules [4].

The value of CML primarily resides in the ease with which users can manipulate and combine models into complex hybrid solvers. It is clear that systems such as CML are moving in the direction of providing something akin to *combinators* [8] over models. The ultimate aim is to deliver a clean, well defined and semantically sound collection of operators that allow models to be combined in a multitude of ways into hybrids that expose a clean interface for further composition. CML fell short of this goal due to two shortcomings:

1. First, model composition was based on syntax rather than semantics and the system was not endowed with the ability to track model relationships such as relaxations. In essence, semantics were opaque to end-users and correctness hinged on the modeler's ability to write sound composition expressions.
2. Second, CML did not require the user to specify any property for the inputs or outputs of operators. Instead, most operators relied on implicit assumptions about the models capabilities and functioned by plugging models into static templates.

The purpose of this paper is to push beyond CML’s limitations and realize the vision of true *model combinators* with sound semantics and complete compositionality. This introduces two requirements. First, the paper postulates the existence of runnables as first class objects that encapsulate a model and a signature specifying their capabilities. Second, it mandates the introduction of combinators specifying how to derive new runnables from their inputs and what the properties of the derived runnable are. The keystone is based on metadata maintained in the runnables and the leverage it provides to check the soundness of the operators application.

The remainder of the paper is organized as follows. Section 2 first discusses the related work. Section 3 introduces the notion of models and program while Section 4 focuses on runnables. Section 5 turns to combinators with Section 6 discussing implementation and Section 7 presenting empirical results. Section 8 concludes the paper.

## 2 Related Work

Several systems aim to facilitate solver independent modeling, model reformulation and automated hybrid generation. The Comet Modeling Language (CML) [6] is unique in that it strives to provide a full programming language in which models can be specified and easily manipulated and composed in sophisticated ways without the need for annotations. The work in this paper directly builds off of what was done in CML.

*G12* models written with *Zinc* and *mini-Zinc* feature solver independent capabilities and model rewriting, which can be achieved via *Cadmium* [3]. *G12* has been used for column generation and branch-and-price hybrid models [10]. *SIMPL* [14] is a high level modeling language based on search-infer-relax. Users write models at a very high level and rely on constraint-based relaxations and inference rules to assemble a model for a chosen technology. *Essence* [7] is designed for model specification using combinatorial constructs rather than CP specific constructs like global constraints. It has recently been combined with *Conjure* [1] to automate the generation of constraint models from a combinatorial problem specification. The *Z3* SMT [2] engine allows user to specify *tactics* which are used to direct the search procedure. *Tactics* are capable of relaxing parts of the SMT problem and determining whether a particular relaxed subproblem will provide an upper or lower bound for the original problem. These tactics can be queried at runtime to determine how the search should proceed and what to invoke next. Finally, work has already been done in providing a rich language of combinators within the context of search [12] and is revisited in [13].

## 3 Models and Programs

This section reviews the concepts of models and programs from OBJECTIVE-CP which serves as the foundations for OCPMCL.

|            |                                                                                                                                        |
|------------|----------------------------------------------------------------------------------------------------------------------------------------|
| Flatten    | <i>Flattening</i> a model decomposes complex expressions into simpler ones, often adding variables and constraints in the process.     |
| Continuous | Performs a continuous <i>relaxation</i> of a model, replacing integer-valued domain constraints with continuous interval domains.      |
| Linear     | Creates a linear reformulation to replace global constraints and logical constraints with a set of equivalent linear constraints [11]. |

**Fig. 1.** Examples of commonly used model operators

**Definition 1.** A model  $M$  is of the form  $\langle X, C, O \rangle$  where  $X$  is the set of model variables,  $C$  the model constraints and  $O$  the (optional) objective function.

**Definition 2.** A model transformation  $\tau$  transforms a model  $M = \langle X, C, O \rangle$  into another model  $\tau(M) = \langle X_o, C_o, O_o \rangle$  satisfying  $X \subseteq X_o$ .

Examples of model transformations performed by OBJECTIVE-CP are shown in Figure 1. When models are in flattened form (sufficiently decomposed), they can be concretized in an optimization program.

**Definition 3.** A model concretization  $\gamma$  takes a model  $M$  in a flattened form and concretizes  $M$  into a program  $P = \langle M, \gamma \rangle$ , where  $P = \gamma(M)$ . The concretization associates a concrete variable with every model variable, a concrete constraint to every model constraint, and a concrete objective with the model objective.

To obtain an optimization program  $P$  from a model  $M$ , OBJECTIVE-CP performs a series of model transformations followed by a concretization, e.g.,

$$P = \gamma(\tau_{k-1}(\cdots \tau_0(M) \cdots)).$$

Model transformations in OBJECTIVE-CP always extend the set of variables, which is convenient both from a semantic and implementation standpoint. In this paper, we ignore that OBJECTIVE-CP can provide a search procedure since it is not relevant for model combinators. Hence, we define a program as a pair (model, concretization).

**Definition 4 (Program).** A program is a pair  $P = \langle M, \gamma \rangle$ , where  $M$  is a model in flattened form and  $\gamma$  is a concretization.

Model transformations impose a natural partial order between models through the concept of relaxation and tightening. These concepts are critical to define sound combinators [8].

**Definition 5 (Solution Set).** A solution for a model  $M = \langle X, C, O \rangle$  is an assignment of all variables in  $X$  satisfying  $C$ . The set of solutions of model  $M$  is denoted by  $Sol(M)$ .

**Definition 6 (Projection of Solution Sets).** Consider a model  $M = \langle X, C, O \rangle$  along with a solution  $s$  and  $X' \subseteq X$ . Then,  $Sol|_{X'}(s)$  and  $Sol|_{X'}(M)$  denotes the projection of solution  $s$  and the solution set of  $M$  on the variables in  $X'$ , respectively.

We now formalize the concept of relaxation, tightening, and equivalence of transformed models.<sup>1</sup> Without loss of generality, optimization problems are all minimization problems in this paper.

**Definition 7 (Relaxations and Tightenings of Satisfaction Problems).**

Let  $M = \langle X, C \rangle$  and  $\tau$  be a transformation. The model  $\tau(M) = M' = \langle X', C' \rangle$  is a relaxation of  $M$ , denoted by  $M' \triangle M$ , iff  $Sol(M) \subseteq Sol|_X(M')$ . It is a tightening, denoted by  $M' \nabla M$ , iff  $Sol|_X(M') \subseteq Sol(M)$ .  $M$  and  $M'$  are equivalent, denoted by  $M \equiv M'$ , if  $M'$  is both a relaxation and a tightening of  $M$ .

**Definition 8 (Relaxations and Tightenings of Minimization Problems).**

Let  $M = \langle X, C, O \rangle$  and  $\tau$  be a transformation. The model  $\tau(M) = M' = \langle X', C', O' \rangle$  is a relaxation of  $M$ , denoted  $M' \triangle M$ , iff  $\langle X', C' \rangle \triangle \langle X, C \rangle$  and

$$\forall s \in Sol(M), s' \in Sol(M') : Sol|_X(s') = s \Rightarrow O'(s') \leq O(s).$$

$M'$  is a tightening of  $M$ , denoted by  $M' \nabla M$ , if  $\langle X', C' \rangle \nabla \langle X, C \rangle$  and

$$\forall s \in Sol(M), s' \in Sol(M') : Sol|_{X'}(s) = s' \Rightarrow O'(s') \geq O(s).$$

$M$  and  $M'$  are equivalent, denoted by  $M \equiv M'$ , if  $M'$  is both a relaxation and a tightening of  $M$ .

The definitions of these concepts are transitive, reflexive and, for equivalences, commutative. We use  $\triangle^*$  (resp.  $\nabla^*$ ) to denote the reflexive and transitive closure of  $\triangle$  (resp.  $\nabla$ ). We use  $\equiv^*$  to denote the reflexive, commutative, and transitive closure of  $\equiv$ . Our combinators use these relations to enforce pre-conditions and post-conditions on their models.

## 4 Runnables and Runnable Signatures

This section introduces the concept of a *runnable*. Informally, a *runnable* can be thought of as a producer/consumer process that uses a *program* to solve an optimization problem, consuming from a number of *input pipes*, and producing into a number of *output pipes* (see Figure 2(a)). The pipes deal with runnable products that are concepts such as solutions and bounds, as well as streams or sets of these products. A runnable is associated with a signature that specifies its inputs and outputs, i.e., the products that it consumes and produces. The implementation creates pipes for each of these inputs and outputs. If a runnable is executed directly, its input and output pipes are not used; the runnable pipes are only useful when it is combined with other runnables through *combinators* (see Figure 2(b)). Note that stream pipes consume or produce products during the lifetime of a runnable; this is the case when exchanging solutions and bounds during the search.

<sup>1</sup> Note that the model  $M'$  always has at least the same variables as  $M$  since  $M'$  is obtained through a transformation of  $M$ . Tightenings are only obtained by adding constraints, while relaxations can be obtained by adding variables or removing constraints.



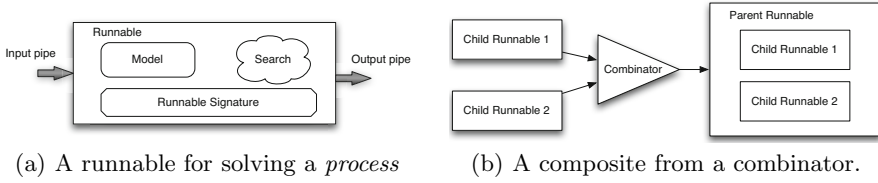


**Fig. 2.** Basic and Composite Runnables

**Definition 9 (Runnable Products).** A runnable product is specified by the grammar

$$\begin{aligned} \langle \text{runnable product} \rangle &::= \langle \text{basic product} \rangle \mid [\langle \text{basic product} \rangle] \mid \{\langle \text{basic product} \rangle\} \\ \langle \text{basic product} \rangle &::= \text{UBD} \mid \text{LBD} \mid \text{COL} \mid \text{CST} \mid \text{SOL} \end{aligned}$$

where the basic products *UBD*, *LBD*, *COL*, *CST*, *SOL* represent upper bounds, lower bounds, columns, constraints and solutions,  $[p]$  represents a stream of products of type  $p$ , and  $\{p\}$  a set of products of type  $p$ .

**Definition 10 (Runnable Signature).** A runnable signature is a pair  $S = \langle I, O \rangle$ , where  $I$  is a set of input runnable products and  $O$  is a set of output runnable products.

**Definition 11 (Runnable).** A runnable is a pair  $R = \langle P, S \rangle$ , where  $P$  is an optimization program and  $S$  is a runnable signature.

We often abuse language and talk about the model of a runnable to denote the model of its program.

**Definition 12 (Pipes of a Runnable).** Let  $R$  be a runnable  $\langle P, \langle I, O \rangle \rangle$ .  $R$  provides the set of input pipes  $\{in(p, R) \mid p \in I\}$  and the set of output pipes  $\{out(p, R) \mid p \in O\}$ .

Our implementation provides a number of primitive runnables. They can be created from a model  $M$ , a flattening, and a concretization. For instance, the `CPRunnable` has a program  $\langle \text{flatten}(M), \gamma_{CP} \rangle$  and a predefined signature.

## 5 Model Combinators

This section describes model combinators. We restrict our attention to binary operators for simplicity but it is easy to generalize our results for non-binary combinators. A model combinator  $R = C(R_1, R_2)$  combines two runnables  $R_1$  and  $R_2$  to produce another runnable. The combinator requires some properties from its runnables, establishes the links between the pipes of its runnables and its own, and specifies how its model relates to the models of its runnables. Figure 3 illustrates the piping intuitively. More precisely, the specification of a model combinator consists of several parts:

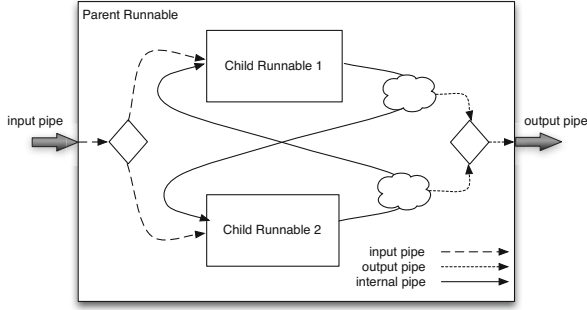


**Fig. 3.** A Composite Runnable and Its Input, Output and Internal Pipes

1. a precondition that specifies the required relationships between the runnable models and the existence of some input/output products;
2. a set of piping rules for linking the input pipes of the combinators to the input pipes of its runnables;
3. a set of piping rules for linking the output pipes of its runnables to its output pipes;
4. a set of piping rules for linking the pipe of the runnables;
5. a relationship between the model of the combinator and the model of the runnables.

A piping rule is an expression of the form  $\pi_1 \rightarrow \pi_2$  which specifies that pipe  $\pi_1$  produces products that are consumed by pipe  $\pi_2$ . For instance, the rule

$$in(SOL, R) \rightarrow in(SOL, R_1)$$

specifies that the input pipe for solutions in  $R$  produces solutions that are consumed by the input pipe for solution in  $R_1$ . If  $p$  is a product, an input pipe rule is of the form

$$in(p, R) \rightarrow in(p, R_i)$$

an output pipe rule is of the form

$$out(p, R_i) \rightarrow out(p, R)$$

and an internal pipe rule is of the form

$$out(p, R_i) \rightarrow in(p, R_j)$$

It is also useful to allow output piping rules with no antecedent, i.e.,

$$\rightarrow out(p, R)$$

for situations where the combinator products are not directly taken from the runnables but computed by the combinator itself.

These piping rules have two main purposes: To establish the plumbing inside the combinators and to synthesize the signature of the combinator. It is important to state that the combinator does not have a static signature. Rather OCPMCL synthesizes the most general signature based on the functionalities of its runnables.

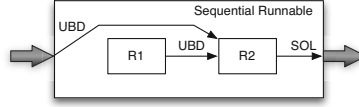


**Fig. 4.** The Runnable of the Sequential Combinator

**Definition 13 (Combinator Specification).** Let  $R_1$  and  $R_2$  be two runnables with signatures  $S_i = \langle I_i, O_i \rangle$  and models  $M_i$  ( $1 \leq i \leq 2$ ). The specification of a combinator  $C(R_1, R_2)$  is a tuple  $\langle \mathcal{P}, \mathcal{I}, \mathcal{O}, \mathcal{E}, \mathcal{M} \rangle$ , where  $\mathcal{P}$  is a precondition on  $M_i$ ,  $I_i$ , and  $O_i$ ,  $\mathcal{I}$ ,  $\mathcal{O}$ ,  $\mathcal{E}$  are sets of input, output, and internal piping rules, and  $\mathcal{M}$  specifies the model relationship.

Obviously, the combinator does not have a model on its own: It combines, sometimes in complex ways, the models of its runnable. Hence, the model relationship specifies the semantics of its products, such as its solutions, its bounds, and streams thereof. For instance, a model relationship  $R \Delta R_1$  specifies that the (implicitly defined) combinator model is a relaxation of the model of  $R_1$ . The new information is propagated through the transitive closures of  $\Delta$  in order to verify preconditions involving  $R$  in subsequent combinations. We are now ready to synthesize the combinator signature.

**Definition 14 (Combinator Signature).** Let  $R_1$  and  $R_2$  be two runnables with signatures  $S_i = \langle I_i, O_i \rangle$  and a combinator  $R = C(R_1, R_2)$  with specification  $\langle \mathcal{P}, \mathcal{I}, \mathcal{O}, \mathcal{E}, \mathcal{M} \rangle$ . The signature of  $R$  is  $\langle I, O \rangle$  where

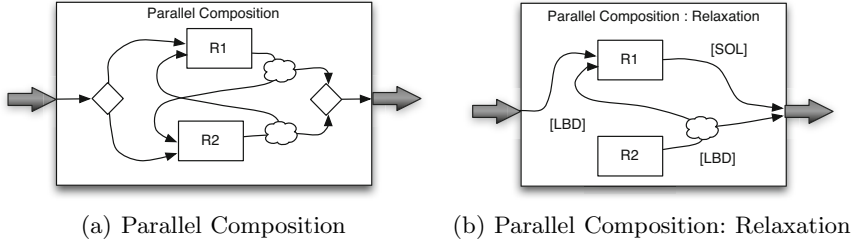
$$\begin{aligned} I &= \{ p \mid \text{in}(p, R) \rightarrow \text{in}(p, R_i) \in \mathcal{I} \wedge p \in I_i \wedge 1 \leq i \leq 2 \} \\ O &= \{ p \mid \text{out}(p, R_i) \rightarrow \text{out}(p, R) \in \mathcal{O} \wedge p \in O_i \wedge 1 \leq i \leq 2 \}. \end{aligned}$$

Observe once again that the definition of input/output is dynamic: The piping rules defines what is possible and the actual input/output definitions are derived from the actual input and output functionalities of the combined runnables. If a runnable does not provide a certain product (e.g., streams of lower bounds), this product is not synthesized in the signature, even if a piping rule was specified. We are now ready to present some combinators.

## 5.1 Sequential Combinator

This section presents a sequential combinator  $R = R_1 \triangleright R_2$  which uses  $R_1$  to compute an upper bound which is then passed as an input to  $R_2$ . This combinator (see Figure 4) is often used in practice when a heuristic search first finds a high-quality upper bound which is then used to seed a systematic search. The combinator specification is as follows:

$$R = R_1 \triangleright R_2$$

**Fig. 5.** Combinators for Parallel Composition

|                                                                                                            |
|------------------------------------------------------------------------------------------------------------|
| $\mathcal{P}_{\triangleright} = R_1 \nabla^* R_2 \wedge UBD \in O_1 \wedge UBD \in I_2 \wedge SOL \in O_2$ |
| $\mathcal{I}_{\triangleright} = \{in(UBD, R) \rightarrow in(UBD, R_2)\}$                                   |
| $\mathcal{O}_{\triangleright} = \{out(SOL, R_2) \rightarrow out(SOL, R)\}$                                 |
| $\mathcal{E}_{\triangleright} = \{out(UBD, R_1) \rightarrow in(UBD, R_2)\}$                                |
| $\mathcal{M}_{\triangleright} = R \equiv R_2$                                                              |

The precondition requires that  $M_1$  be a tightening of  $M_2$  to ensure that the upper bound of  $M_1$  is indeed an upper bound for  $M_2$ . The input piping rule allows the upper bound of  $R$  to be consumed by  $R_2$ ; It cannot be passed to  $R_1$  since  $R_1$  is a tightening of  $M_2$ . The output piping rule allows the solution of  $R_2$  to be produced as a solution to  $R$ . The internal piping rule specifies that the upper bound produced by  $R_1$  can be consumed by  $R_2$ . The model relationship specifies that the resulting combinator is equivalent to  $R_2$ .

The signature that is synthesized here is trivial, since the piping rules are only concerned with required inputs and outputs. It will simply be  $\langle\{UBD\}, \{SOL\}\rangle$ . If the output piping rule

$$out(LBD, R_2) \rightarrow out(LBD, R)$$

had been present, and  $LBD$  would belong to  $O_2$ , the synthesized signature would have been  $\langle\{UBD\}, \{SOL, LBD\}\rangle$ .

## 5.2 Parallel Combinator

We now turn to the parallel composition of two equivalent runnables exchanging solutions (see Figure 5(a)). Its specification is

$$R = R_1 \parallel R_2$$

|                                                                                                                                                                                                                                                                                                                  |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\mathcal{P}_{\parallel} = R_1 \equiv^* R_2 \wedge [SOL] \in I_1 \wedge [SOL] \in O_1 \wedge [SOL] \in I_2 \wedge [SOL] \in O_2$                                                                                                                                                                                 |
| $\mathcal{I}_{\parallel} = \{in([SOL], R) \rightarrow in([SOL], R_1), in([SOL], R) \rightarrow in([SOL], R_2),$<br>$in([UBD], R) \rightarrow in([UBD], R_1), in([UBD], R) \rightarrow in([UBD], R_2),$<br>$in([LBD], R) \rightarrow in([LBD], R_1), in([LBD], R) \rightarrow in([LBD], R_2)\}$                   |
| $\mathcal{O}_{\parallel} = \{out([SOL], R_1) \rightarrow out([SOL], R), out([SOL], R_2) \rightarrow out([SOL], R),$<br>$out([UBD], R_1) \rightarrow out([UBD], R), out([UBD], R_2) \rightarrow out([UBD], R),$<br>$out([LBD], R_1) \rightarrow in([LBD], R), out([LBD], R_2) \rightarrow out([LBD], R)\}$        |
| $\mathcal{E}_{\parallel} = \{out([SOL], R_1) \rightarrow in([SOL], R_2), out([SOL], R_2) \rightarrow in([SOL], R_1),$<br>$out([UBD], R_1) \rightarrow in([UBD], R_2), out([UBD], R_2) \rightarrow in([UBD], R_1),$<br>$out([LBD], R_1) \rightarrow in([LBD], R_2), out([LBD], R_2) \rightarrow in([LBD], R_1)\}$ |
| $\mathcal{M}_{\parallel} = R \equiv R_1$                                                                                                                                                                                                                                                                         |

The precondition  $\mathcal{P}_{\parallel}$  ensures that the two runnables are equivalent and the input and output of both runnables include a stream of solutions. The piping rules are very explicit this time and allow for the exchanges of upper and lower bounds as well. In particular, if the runnables provide lower bounds, the implementation will ensure that the internal piping provides that functionality. Similarly, the input and output piping will synthesize streams of upper and lower bounds if the combined runnables provide these products.

Observe that this combinator can be used for composing three runnables: It suffices to use  $(R_1 \parallel R_2) \parallel R_3$ , since the parallel combinator will satisfy its own precondition. Also, Figure 5(a) shows the flow of solutions within this parallel runnable using black arrows. The small clouds waiting at the outputs of the child runnables represents small blocks of code used by the parent to intercept output solutions coming from the children.

### 5.3 Relaxed Parallel Combinator

The relaxed parallel combinator runs two optimization programs concurrently but one of the models is a relaxation of the other and streams lower bounds. The specification of the combinator is

$$R = R_1 \dashv R_2$$

|                                                                                                                                                                  |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| $\mathcal{P}_{\dashv} = R_2 \triangle^* R_1 \wedge [SOL] \in O_1 \wedge [LBD] \in I_1 \wedge [LBD] \in O_2$                                                      |
| $\mathcal{I}_{\dashv} = \{in([LBD], R) \rightarrow in([LBD], R_1), in([UBD], R) \rightarrow in([UBD], R_1),$<br>$in([SOL], R) \rightarrow in([SOL], R_1)\}$      |
| $\mathcal{O}_{\dashv} = \{out([SOL], R_1) \rightarrow out([SOL], R), out([UBD], R_1) \rightarrow out([UBD], R),$<br>$out([LBD], R_1) \rightarrow in([LBD], R)\}$ |
| $\mathcal{E}_{\dashv} = \{out([LBD], R_2) \rightarrow in([LBD], R_1)\}$                                                                                          |
| $\mathcal{M}_{\dashv} = R \equiv R_1$                                                                                                                            |

The precondition  $\mathcal{P}_{\dashv}$  ensures that  $R_2$  is a relaxation of  $R_1$ ,  $R_1$  produces a stream of solutions, and  $R_2$  produce lower bounds. The input piping rules  $\mathcal{I}_{\dashv}$

states that streams of lower bounds, upper bounds or solutions consumed by  $R_{\neg}$  can be consumed by  $R_1$ . The output piping rules  $\mathcal{O}_{\neg}$  state that  $R_{\neg}$  produces the output streams produced by  $R_1$ . The internal piping rules ensure that the stream of lower bounds produced by  $R_2$  can be consumed by  $R_1$ . The combinator then produces a model equivalent to  $R_1$ . Figure 5(b) illustrates the flow of runnable products through this runnable assuming the children meet only the minimum preconditions for simplicity.

## 5.4 Column-Generation Combinator

Automating column-generation solvers has been done previously in systems such as CML [6] and the G12 Project [9], but the use of runnables allows for a cleaner expression of the semantics as well as a much more compositional interface. The column-generation combines a master problem and a slave problem. The master runnable consumes columns and generates solutions, while the slave runnable consumes solutions and produces columns. The column-generation combinator produces a solution and a stream of upper bounds on its own. An implementation of the combinator copies the master runnable, before starting the column-generation process, in order to allow the master runnable to be reused in other combinators. As a result, the combinator does not use the output of its runnables but generates products on its own. The common terminology for the resulting model is the Restricted Master Problem (RMP), but this paper will refer to it as a *relaxation* of the master model, since column generation adds new columns (i.e., variables). Note that the column-generation combinator is very general: It does not impose how the slave process uses the solution (though the dual values are captured in the solution). As a consequence, it can implement a traditional column-generation algorithm or use a heuristic approach to generate columns based on the problem structure.

$$R = MA \bowtie SL \text{ with } MA = \langle I_m, O_m \rangle \text{ and } SL = \langle I_s, O_s \rangle$$

|                           |                                                                                                |
|---------------------------|------------------------------------------------------------------------------------------------|
| $\mathcal{P}_{\bowtie} =$ | $COL \subseteq I_m \wedge SOL \subseteq O_m \wedge SOL \subseteq I_s \wedge COL \subseteq O_s$ |
| $\mathcal{I}_{\bowtie} =$ | $\{\}$                                                                                         |
| $\mathcal{O}_{\bowtie} =$ | $\{\rightarrow out(SOL, R), \rightarrow out([UBD], R)\}$                                       |
| $\mathcal{E}_{\bowtie} =$ | $\{out(\{COL\}, SL) \rightarrow in(\{COL\}, MA), out(SOL, MA) \rightarrow in(SOL, SL)\}$       |
| $\mathcal{M}_{\bowtie} =$ | $R \triangle MA$                                                                               |

Figure 6 illustrates the combinator.

## 5.5 Logical Benders Decomposition

Consider now a combinator for implementing Logical Benders decomposition. Logical Benders decomposition was not supported in CML but this section highlights that it is in fact the dual of the column-generation combinator and is easily supported in OCPMCL. Informally speaking, in its simplest form, a Benders decomposition features a master that relaxes some of the constraints of an original

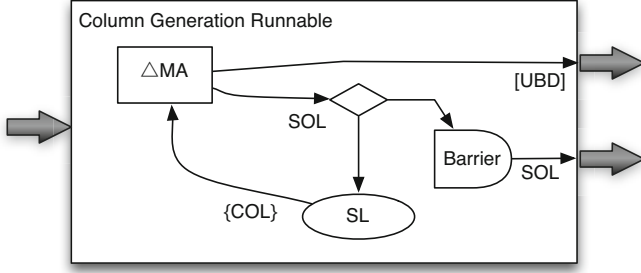


**Fig. 6.** The Column-Generation Combinator

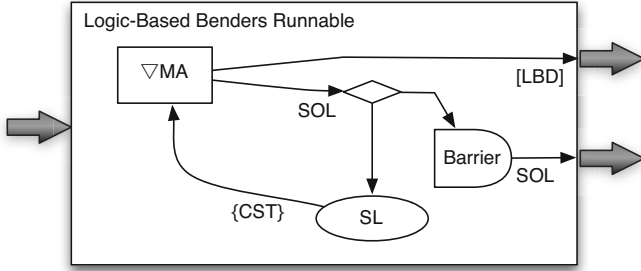


**Fig. 7.** A Combinator for Logical Benders Decomposition

model and a slave that checks if the solution produces by this master are feasible for the relaxed constraints. If these constraints are infeasible, the slave generates new constraints that are added to the master. The process is repeated until a feasible (and optimal) solution is found. Once again, the combinator receives a master and a slave runnable. The master runnable is copied and the combinator implementation keeps adding constraints to the master until an optimal solution is found. The slave receives the solutions to the master and generates new constraints. The combinator produces a stream of lower bounds and a final solution. The model specification closely mirrors the combinator for column generation, with upper bounds being replaced by lower bounds. Moreover, the combinator is now a tightening of the master program since the Benders decomposition adds new constraints. Figure 7 illustrates the combinator.

$$R = MA \otimes SL \text{ with } MA = \langle I_m, O_m \rangle \text{ and } SL = \langle I_s, O_s \rangle$$

|                           |                                                                                                |
|---------------------------|------------------------------------------------------------------------------------------------|
| $\mathcal{P}_{\otimes} =$ | $CST \subseteq I_m \wedge SOL \subseteq O_m \wedge SOL \subseteq I_s \wedge CST \subseteq O_s$ |
| $\mathcal{I}_{\otimes} =$ | $\{\}$                                                                                         |
| $\mathcal{O}_{\otimes} =$ | $\{\rightarrow out(SOL, R), \rightarrow out([LBD], R)\}$                                       |
| $\mathcal{E}_{\otimes} =$ | $\{out(\{CST\}, SL) \rightarrow in(\{CST\}, MA), out(SOL, MA) \rightarrow in(SOL, SL)\}$       |
| $\mathcal{M}_{\otimes} =$ | $R \nabla MA$                                                                                  |

---

```

1 id<ORModel> root = ... // Def. of AP Model
2 id<ORModel> L = [ORFactory linearizeModel: root];
3 id<ORModel> C = [ORFactory continuousRelax: root];
4 id<ORRunnable> r0 = [ORFactory CPRunnable: root];
5 id<ORRunnable> r1 = [ORFactory IPRunnable: L];
6 if<ORRunnable> r2 = [ORFactory LPRunnable: C];
7 id<ORRunnable> cmp = [ORCombinator parallel: cp0 with: ip1];
8 id<ORRunnable> rlx = [ORCombinator relaxedParallel: complete with: lp2];
9 [relaxed run];

```

---

**Fig. 8.** A parallel hybrid for the Asymmetric Traveling Salesman Problem

The structure of the benders runnable is shown in figure 7. The precondition  $\mathcal{P}_{\otimes}$  checks that the *master* accepts a pool of constraints and generates a solution. The input relations  $\mathcal{I}_{\otimes}$  states that the master can receive a stream of constraints. The output relations  $\mathcal{O}_{\otimes}$  states that the output pipe of  $R_{\otimes}$  produces a solution taken from the master. Finally, the internal pipe relations ensures that the master outputs a solution to the slave closure. The slave closure will use the solved master problem to generate and run a slave problem before outputting a set of constraints (cuts) which will be injected into the master.

## 6 Implementation

OCPCML offers a combinator model library built atop OBJECTIVE-CP. The library provides protocols for all the concepts including *abstract models* (**ORModel**), *runnables* (**ORRunnable**), *runnable signatures* (**ORSignature**), *runnable specification* (**ORSpecification**) and *combinators* (**ORCombinator**). The delicate part of the implementation is focused on interpreting the signatures to synthesize the pipes connecting components within a combinator.

*Example.* Consider as an example the simple Assignment Problem (AP) given as an abstract model (**ORModel**) that should be solved with the parallel combination of three models: a complete CP model, a complete IP model and a linear relaxation. This can be achieved with the OBJECTIVE-CP code in Figure 8. Line 2 linearizes the root model into **L** while line 3 stores in **C** a continuous relaxation. Lines 4–6 concretize the three models with a CP solver (OBJECTIVE-CP), an integer-programming solver (using GUROBI) and a linear-programming solver (using GUROBI too). Lines 7–8 combines the first two models with the parallel combinator which is fed to the **relaxedParallel** combinator alongside runnable **r2** which holds the linear relaxation. Line 10 executes the resulting code. The excerpt shows that, with only a few lines of code, three models are easily composed in parallel, producing and consuming solutions, upper bounds and lower bounds. The combinators automatically account for the fact that one model is a relaxation of the other two. (see Figure 11).

*Precondition and Signature.* Figure 9 shows the OCPCML code for the precondition of the relaxed parallel combinator. It is simply a function working on the signatures of the runnables and verifying that the required properties hold.



---

```

1 BOOL pre(id<ORRunnable> r1, id<ORRunnable> r2) {
2   return [r2 isRelaxationOf: r1] && [r1.sig acceptsLowerBoundsStream] &&
3     [r1.sig producesSolutionStream] && [r2.sig producesLowerBoundStream];
4 }

```

---

**Fig. 9.** Precondition closure for ORRelaxedParallelCombinator

---

```

1 void internal(id<ORRunnable> parent, id<ORRunnable> r1, id<ORRunnable> r2) {
2   if([[r1 sig] providesSolutionStream] && [[r2 sig] acceptsSolutionStream])
3     [[r1 outSolutionStream] wheneverNotifiedDo: ^void(id<ORSolution> s) {
4       [[r2 inSolutionStream] notifyWith: s];
5     }];
6   ...
7   if([[r1 sig] providesSolutionStream])
8     [[r1 outSolutionStream] wheneverNotifiedDo: ^void(id<ORSolution> s) {
9       [[parent outSolutionStream] notifyWith: s];
10    }];
11 }

```

---

**Fig. 10.** Internal pipe closure for ORCompleteParallelCombinator

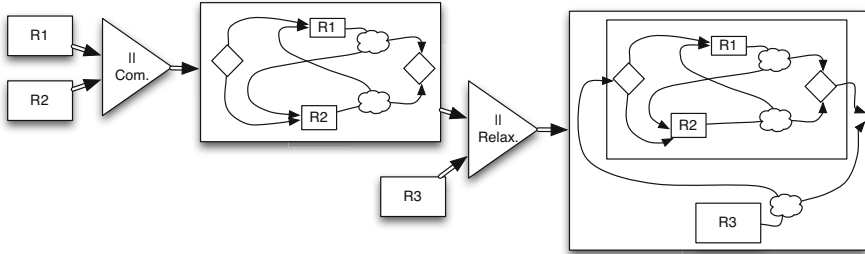
*The Piping Infrastructure.* Communication relies heavily on an event infrastructure provided by OBJECTIVE-CP and is similar to COMET events. In OBJECTIVE-CP, an `ORInformer` object embodies a thread-safe event delivery mechanism and meshes with Objective-C closures. Two operations are available on an informer. One can `notify` the occurrence of the event it represents and one can register a closure to *listen and respond* to an event occurrence. The arguments of the closure offer a simple way to hand-off data. In OCPMCL, runnables provide informers to represent the input and output pipes of their signature. Combinators then generate the right producer/consumer glue to transmit the solutions.

Figure 10 highlights fragments responsible for setting up the plumbing infrastructure for a parallel combinator. Lines 3–5 setup the pipelining for the output solution pipe from  $r_1$  into the input solution pipe of  $r_2$ . This is realized with a simple closure relaying solutions from one informer to the other. The plumbing code is only generated when needed (line 2). Naturally, the other plumbing code (from  $r_2$  to  $r_1$ ) is generated similarly. Lines 7–10 install the output pipe connecting  $r_1$ 's output solution stream to the output solution stream of the parent. This is realized with a simple closure that listens to inbound solutions from  $r_1$  and relays them. The other connections are similar but not shown for brevity.

## 7 Empirical Results

This section presents benchmark results to assess the practicality of OCPMCL. The goal is not to give comprehensive results on a wide variety of benchmarks but to give preliminary evidence that OCPMCL is a promising approach to ease the building of hybrid optimization algorithms.

The first benchmark is the Location-Allocation Problem implemented using the logical Benders approach in [5]. It will allow us to compare the efficiency of an OCPMCL model with a hand-crafted implementation. The experiments feature 6

**Fig. 11.** Nested Parallel composition of two equivalent models with a linear relaxation

| Instance #  | min   | max    | avg   |
|-------------|-------|--------|-------|
| 1           | 19.69 | 21.23  | 20.23 |
| 2           | 22.53 | 26.66  | 23.82 |
| 3           | 11.26 | 13.30  | 11.92 |
| 4           | 5.81  | 7.61   | 6.37  |
| 5           | 94.81 | 110.13 | 99.31 |
| 6           | 67.28 | 79.73  | 70.91 |
| overall avg | 38.76 |        |       |

(a) Logic-Based Bender's

| Inst          | min   | max   | avg   | min  | max  | avg  |
|---------------|-------|-------|-------|------|------|------|
| $8 \times 20$ | 2.2   | 2.5   | 2.3   | 1.8  | 2.3  | 2.1  |
| $8 \times 30$ | 6.1   | 6.5   | 6.3   | 2.5  | 4.3  | 3.5  |
| $8 \times 40$ | 7.9   | 16.2  | 9.7   | 4.5  | 9.5  | 5.7  |
| $9 \times 20$ | 72.0  | 89.8  | 75.3  | 48.6 | 62.3 | 55.5 |
| $9 \times 30$ | 27.6  | 30.2  | 28.3  | 21.8 | 27.9 | 24.1 |
| $9 \times 40$ | 155.2 | 174.5 | 165.6 | 72.6 | 89.5 | 80.2 |

(b) Assignment Problem

**Fig. 12.** Benchmarks for OBJECTIVE-CP runnables

instances from the original paper, namely, the *uncorrelated*  $20 \times 10$  (20 clients, 10 facilities) instances<sup>2</sup>. The authors reported an average running time of 33 seconds for these instances. The OCPMCL results are based on 20 runs of each instance and are given in Table 12(a). The Bender's runnable runs in about 39 seconds on average which is remarkably close to the results for the hand-written model. The experiment was carried out on a 2.13 Ghz Intel Core 2 Duo with 4 GB of RAM running Mac OS X (10.8) which is comparable to the machines in the original paper (Duo Core AMD 270 CPU, 4 GB Ram, Red Hat Linux). Table 13 reports the results of an instrumentation of code to measure the time spent in the master, in the slave, and otherwise, considering that the remainder of time was attributed to the combinator. This is an overestimate of the true combinator cost as any other overhead is attributed to the combinator. The Master columns report the total time spent in the master. The % column report the fraction of the total that this represents. The same applies for the Slave and Combinator columns. For any row, the percentages add up to 100%. Overall, the combinator overhead never exceeds  $\frac{1}{2}\%$  of the runtime and demonstrates that the approach is competitive. This should be contrasted with the brief OCPMCL-based implementation which weighs in at 90 lines of Objective-C code (without data reading) to create the models and setup the Bender's combinator.

The second benchmark is a simple Assignment Problem (AP) in which we run a standard CP implementation [6] in parallel with a CP linear reformulation using the Complete Parallel Combinator. Note that there are better approaches to solving the AP, we only aim to show the benefit of using OCPMCL combinators

<sup>2</sup> A detailed description of the problem can be found in [5].

|           | Master |          |       | Slave |          |      | Combinator |          |      |
|-----------|--------|----------|-------|-------|----------|------|------------|----------|------|
| Instance# | $\mu$  | $\sigma$ | %     | $\mu$ | $\sigma$ | %    | $\mu$      | $\sigma$ | %    |
| 1         | 20.12  | 0.40     | 99.41 | 0.09  | 0.02     | 0.42 | 0.12       | 0.40     | 0.17 |
| 2         | 23.72  | 0.90     | 99.57 | 0.06  | 0.02     | 0.25 | 0.10       | 0.90     | 0.18 |
| 3         | 11.84  | 0.46     | 99.37 | 0.04  | 0.01     | 0.31 | 0.08       | 0.47     | 0.32 |
| 4         | 6.31   | 0.57     | 99.10 | 0.03  | 0.01     | 0.43 | 0.05       | 0.57     | 0.47 |
| 5         | 99.14  | 3.68     | 99.82 | 0.14  | 0.05     | 0.14 | 0.18       | 3.71     | 0.04 |
| 6         | 70.80  | 3.10     | 99.84 | 0.07  | 0.03     | 0.12 | 0.11       | 3.10     | 0.04 |

**Fig. 13.** Time allocation between Master/Slave/Combinator

to generate a parallel runnable. The linear reformulation is substantially slower (particularly as a CP model). The table gives the running time of solving the linear model alone and within the parallel runnable. Results are based on random instances with sizes  $n \times m$  where  $n$  is the number of agents/tasks and  $m$  is the maximum allowed cost (cost range  $\in [1, m]$ ). Columns *min*, *max*, *avg* in Table 12(b) represent the minimum maximum and average running time (in secs) of the standalone linear CP problem, while *min* ||, *max* ||, *avg* || refer to the parallel runnable.

## 8 Conclusion

This paper proposes the concept of model combinators, and its implementation in OCPMCL, to simplify the design of hybrid optimization algorithms and provide a foundation for combining complex models. In earlier work, such as in CML, semantic ambiguity prevented models from being truly composable, as they lacked systematic mechanisms for synthesizing input/output interfaces and verifying preconditions on models. To address these shortcomings, this paper introduces a number of concepts:

1. The definition of relationships between models, tightenings and relaxations, which are derived through model transformations. The transitive closures of these relationships enables OCPMCL to verify preconditions on the models. Combinators can also specify their relationship with the underlying models.
2. The concept of runnable and runnable signatures that specify the functionalities supported by an optimization program and its model.
3. The concept of model specifications, including input/output/internal pipes and pipe rules, that enables the synthesis of the signature of the combinators from the signature of their components.

These concepts were used to specify a number of model combinators, including sequential and parallel composition, column generation, and Benders decomposition. These high-level concepts are implemented in OCPMCL using thread-safe informers, an event mechanism provided by OBJECTIVE-CP. Preliminary experimental results on logical Benders decomposition algorithms presented in the literature and artificial benchmarks indicates that the approach promises to be practical.

## References

1. Akgun, O., Miguel, I., Jefferson, C., Frisch, A., Hnich, B.: Extensible automated constraint modelling (2011)
2. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. *Commun. ACM* 54(9), 69–77 (2011)
3. Duck, G.J., De Koninck, L., Stuckey, P.J.: Cadmium: An implementation of ACD term rewriting. In: Garcia de la Banda, M., Pontelli, E. (eds.) *ICLP 2008*. LNCS, vol. 5366, pp. 531–545. Springer, Heidelberg (2008)
4. Duck, G., Stuckey, P., Brand, S.: Acd term rewriting. In: Etalle, S., Truszczyński, M. (eds.) *ICLP 2006*. LNCS, vol. 4079, pp. 117–131. Springer, Heidelberg (2006)
5. Fazel-Zarandi, M.M., Beck, J.C.: Solving a Location-Allocation Problem with Logic-Based Benders’ Decomposition. In: Gent, I.P. (ed.) *CP 2009*. LNCS, vol. 5732, pp. 344–351. Springer, Heidelberg (2009)
6. Fontaine, D., Michel, L.: A high level language for solver independent model manipulation and generation of hybrid solvers. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) *CPAIOR 2012*. LNCS, vol. 7298, pp. 180–194. Springer, Heidelberg (2012)
7. Frisch, A., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. *Constraints* 13, 268–306 (2008)
8. Seldin, J.P., Roger Hindley, J.: *Lambda-Calculus and Combinators An Introduction*, 2nd edn. Cambridge University Press (2008)
9. Puchinger, J., Stuckey, P.J., Wallace, M., Brand, S.: From high-level model to branch-and-price solution in g12 (2008)
10. Puchinger, J., Stuckey, P.J., Wallace, M.G., Brand, S.: Dantzig-wolfe decomposition and branch-and-price solving in g12. *Constraints* 16(1), 77–99 (2011)
11. Refalo, P.: Linear Formulation of Constraint Programming Models and Hybrid Solvers. In: Dechter, R. (ed.) *CP 2000*. LNCS, vol. 1894, pp. 369–383. Springer, Heidelberg (2000)
12. Schrijvers, T., Tack, G., Wuille, P., Samulowitz, H., Stuckey, P.J.: Search combinators. In: Lee, J. (ed.) *CP 2011*. LNCS, vol. 6876, pp. 774–788. Springer, Heidelberg (2011)
13. Hentenryck, P.V., Michel, L.: Search = Continuations + Controllers. In: Schulte, C. (ed.) *CP 2013*. LNCS, vol. 8124, Springer, Heidelberg (2013)
14. Yunes, T., Aron, I.D., Hooker, J.N.: An integrated solver for optimization problems. *Oper. Res.* 58(2), 342–356 (2010)

# Modelling Destructive Assignments

Kathryn Francis<sup>1,2</sup>, Jorge Navas<sup>2</sup>, and Peter J. Stuckey<sup>1,2</sup>

<sup>1</sup> National ICT Australia, Victoria Research Laboratory

<sup>2</sup> The University of Melbourne, Victoria 3010, Australia

**Abstract.** Translating procedural object oriented code into constraints is required for many processes that reason about the execution of this code. The most obvious is for *symbolic execution* of the code, where the code is executed without necessarily knowing the concrete values. In this paper, we discuss translations from procedural object oriented code to constraints in the context of solving optimisation problems defined via simulation. A key difficulty arising in the translation is the modelling of state changes. We introduce a new technique for modelling destructive assignments that outperforms previous approaches. Our results show that the optimisation models generated by our technique can be as efficient as equivalent hand written models.

## 1 Introduction

*Symbolic reasoning* has been the crux of many software applications such as verifiers, test-case generation tools, and bug finders since the seminal papers of Floyd and Hoare [6,8] in program verification and King [10] in symbolic execution for testing. Common to these applications is their translation of the program or some abstraction of it into equivalent *constraints* which are then fed into a *constraint solver* to be checked for (un)satisfiability.

The principal challenge for this translation is effective handling of destructive state changes. These both influence and depend on the flow of control, making it necessary to reason disjunctively across possible execution paths. In object oriented languages with field assignments, the disjunctive nature of the problem is further compounded by potential aliasing between object variables.

In this paper we introduce a new, *demand-driven* technique for modelling destructive assignments, designed specifically to be effective for the difficult case of field assignments. The key idea is to view the value stored in a variable not as a function of the current state, but as a function of the *relevant assignment statements*. This allows us to avoid maintaining a representation of the entire program state, instead only producing constraints for expressions which are actually required.

The particular application we consider for our new technique is the tool introduced in [7], which aims to provide Java programmers with more convenient access to optimisation technology. The tool allows an optimisation problem to be expressed in simulation form, as Java code which computes the objective value given the decisions. This code could be used directly to solve the optimisation problem by searching over possible combinations of decisions and comparing the computed results, but this is likely to be very inefficient. Instead, the tool in [7]

translates the simulation code into constraints, and then uses an *off-the-shelf* CP solver to find a set of decisions resulting in the optimal return value.

Experimental results using examples from this tool demonstrate that our new technique for modelling destructive assignments is superior to previous approaches, and can produce optimisation models comparable in efficiency to a simple hand written model for the same problem.

## 1.1 Running Example

As a running example throughout the paper we consider a smartphone app for group pizza ordering. Each member of the group nominates a number of slices and some ingredient preferences. The app automatically generates a joint order of minimum cost which provides sufficient pizza for the group, assuming that a person will only eat pizza with at least one ingredient they like and no ingredients they dislike. After approval from the user, the order is placed electronically.

Our focus is on the optimisation aspect of the application: finding the cheapest acceptable order. We assume that for each type of pizza both a price per pizza and a price per slice is specified. The order may include surplus pizza if it is cheaper to buy a whole pizza than the required number of individual slices.

Figure 1 shows a Java method defining this optimisation problem, called **buildOrder**. The problem parameters are the contents of the **people** list and the details stored in the **menu** object when **buildOrder** is called. Each call to the method **choiceMaker.chooseFrom** indicates a decision to be made, where the possible options are the **OrderItem** objects included in the list **pizzas** (the **Order** constructor creates an **OrderItem** for each pizza on the menu, all initially for 0 slices). The objective is to minimise the return value, which is the total cost of the order.

## 1.2 Translating Code into Constraints

To evaluate different possible translations from procedural code to constraints we use examples from the tool in [7]. This tool actually performs the translation on demand at run-time (not as a compile time operation), which complicates the translation process somewhat. For the purpose of this paper we will ignore such implementation details, using the following abstraction to simplify the description of the different translations.

We consider the translation to be split into two phases. In the first phase the code is flattened into a linear sequence of assignment statements, each of which has some conditions attached. We describe this transformation briefly in Section 2. In the second phase, which is the main focus of the paper, the flattened sequence of assignments is translated into constraints.

## 2 Flattening

In the Java programming language only the assignment statement changes the state of the program. All other constructs simply influence which other statements will be executed. It is therefore possible to emulate the effect of a piece of Java code using a sequence of assignment statements, each with an attached set

```

int buildOrder() {
    order = new Order(menu);
    for(Person person : people) {
        // Narrow down acceptable pizzas
        pizzas.clear();
        for(OrderItem item : order.items)
            if(person.willEat(item))
                pizzas.add(item);
        // Choose from these for each slice
        for(int i = 0; i < person.slices; i++) {
            OrderItem pizza =
                choiceMaker.chooseFrom(pizzas);
            pizza.addSlice();
        }
    }
    return order.totalCost();
}

class Order {
    List<OrderItem> items;
    int totalCost() {
        int totalcost = 0;
        for(OrderItem item : items)
            totalcost += item.getCost();
        return totalcost;
    }
}

class OrderItem
{
    int pizzaPrice;
    int slicePrice;
    int fullPizzas = 0;
    int numSlices = 0;

    void addSlice() {
        numSlices = numSlices + 1;
        if(numSlices == slicesPerPizza) {
            numSlices = 0;
            fullPizzas = fullPizzas + 1;
        }
    }

    int getCost() {
        int cost = fullPizzas * pizzaPrice;
        if(numSlices > 0) {
            int slicesCost =
                numSlices * slicePrice;
            if(slicesCost > pizzaPrice)
                slicesCost = pizzaPrice;
            cost = cost + slicesCost;
        }
    }
}

```

**Fig. 1.** A Java simulation of a pizza ordering optimisation problem

of conditions controlling whether or not it should be executed. The conditions reflect the circumstances under which this statement would be reached during the execution of the original code.

The flattening process involves unrolling loops<sup>1</sup>, substituting method bodies for method calls, and removing control flow statements after adding appropriate execution conditions for the child statements. As an example, consider the method `getCost` shown in Figure 1. To flatten an `if` statement we simply add the `if` condition to the execution conditions of every statement within the `then` part. The body of `getCost` can be flattened into the following sequence of conditional assignment statements.

| Conditions                                                     | Variable                  | Assigned Value                          |
|----------------------------------------------------------------|---------------------------|-----------------------------------------|
| 1.                                                             | <code>cost</code>         | <code>:= fullPizzas × pizzaPrice</code> |
| 2. <code>(numSlices &gt; 0)</code>                             | <code>: slicesCost</code> | <code>:= numSlices × slicePrice</code>  |
| 3. <code>(numSlices &gt; 0, slicesCost &gt; pizzaPrice)</code> | <code>: slicesCost</code> | <code>:= pizzaPrice</code>              |
| 4. <code>(numSlices &gt; 0)</code>                             | <code>:</code>            | <code>cost := cost + slicesCost</code>  |

<sup>1</sup> The tool in [7] only supports loops with exit conditions unaffected by the decisions, or iteration over bounded collections. This means the number of loop iterations is always bounded. For unbounded loops partial unrolling can be performed, and the final model will be an under-approximation of the behaviour of the program.

Note that each assignment statement applies to a specific variable. This may be a local variable identified by name (as above), or an object field  $\mathbf{o.f}$  where  $\mathbf{o}$  is a variable storing an object, and  $\mathbf{f}$  is a field identifier. We call an assignment to an object field a *field assignment*. The value of the object variable  $\mathbf{o}$  may depend on the decisions, so the concrete object whose field is updated by a field assignment is not necessarily known.

An important optimisation is to consider the declaration scope of variables. For example, if a variable is declared inside the **then** part of an if statement (as is the case for the `slicesCost` variable above), assignments to that variable need not depend on the if condition. In any execution of the original code where the if condition does not hold, this variable would not be created, and therefore its value is irrelevant. This means assignments 2 and 3 above do not need the condition `numSlices > 0`.

We also need to record the initial program state. For variables which exist outside the scope of the code being analysed, we add an unconditional assignment at the beginning of the list setting the variable to its initial value. We call this an *initialising assignment*. For object fields we add an initialising assignment for each concrete object.

Figure 2 shows the sequence of assignments produced by flattening our example function `buildOrder` for an instance with two people and three pizza types. Note that calls to `ChoiceMaker` methods are left untouched (these represent the creation of new decision variables), and expressions which do not depend on the decisions are calculated upfront. For example, the code used to find acceptable order items for each person does not depend on any decisions, so rather than including assignments originating in this part of the code in the flattened sequence, we simply calculate these lists and then use them as constants. Where these expressions are used as if conditions or loop exit conditions we exclude from the translation any unreachable code.

In the following sections, we assume our input is this flattened list of conditional assignment statements. We also use the notation  $Dom(v, i)$  to refer to the set of possible values for variable  $v$  at (just before) assignment  $i$ . This is easily calculated from the list of assignments. A conditional assignment adds values to the domain of the assigned-to variable, while an unconditional assignment replaces the domain.

### 3 Modelling Assignments: Existing Techniques

Using the flattening transformation described above and a straightforward translation of mathematical and logical expressions, we reduce the problem of representing Java code by constraints to that of modelling (conditional) assignment statements. In this section we describe two existing approaches to this, while in the next section we introduce a new proposed approach.

#### 3.1 Typical CP Approach

One obvious technique for modelling assignments, and that used in [7,2,4], is to create a new version of the assigned-to variable for each assignment, and then



| Cond                                                                                  | Object   | Field/Var    | Assigned Value                        |
|---------------------------------------------------------------------------------------|----------|--------------|---------------------------------------|
| 1.                                                                                    | Veg      | . fullPizzas | := 0                                  |
| 2.                                                                                    | Marg     | . fullPizzas | := 0                                  |
| 3.                                                                                    | Mush     | . fullPizzas | := 0                                  |
| 4-12. <i>other initialisation assignments (for numSlices, pizzaPrice, slicePrice)</i> |          |              |                                       |
| 13.                                                                                   |          | pizzas1      | := [Veg,Marg]                         |
| 14.                                                                                   |          | pizza1       | := chooseFrom(pizzas1)                |
| 15.                                                                                   | pizza1   | . numSlices  | := pizza1.numSlices + 1               |
| 16.                                                                                   |          | b1           | := pizza1.numSlices == slicesPerPizza |
| 17. (b1)                                                                              | : pizza1 | . numSlices  | := 0                                  |
| 18. (b1)                                                                              | : pizza1 | . fullPizzas | := pizza1.fullPizzas + 1              |
| 19-23. <i>repeat assignments 14-18 for 2nd slice (using vars pizza2 and b2)</i>       |          |              |                                       |
| 24.                                                                                   |          | pizzas2      | := [Marg,Mush]                        |
| 25.                                                                                   |          | pizza3       | := chooseFrom(pizzas2)                |
| 26.                                                                                   | pizza3   | . numSlices  | := pizza3.numSlices + 1               |
| 27.                                                                                   |          | b3           | := pizza3.numSlices == slicesPerPizza |
| 28. (b3)                                                                              | : pizza3 | . numSlices  | := 0                                  |
| 29. (b3)                                                                              | : pizza3 | . fullPizzas | := pizza3.fullPizzas + 1              |
| 30-34. <i>repeat assignments 25-29 for 2nd slice (using vars pizza4 and b4)</i>       |          |              |                                       |
| 35-39. <i>repeat assignments 25-29 for 3rd slice (using vars pizza5 and b5)</i>       |          |              |                                       |
| 40.                                                                                   |          | totalcost    | := 0                                  |
| 41.                                                                                   |          | cost1        | := Veg.fullPizzas × Veg.pizzaPrice    |
| 42.                                                                                   |          | b6           | := Veg.numSlices > 0                  |
| 43.                                                                                   |          | slicesCost1  | := Veg.numSlices × Veg.slicePrice     |
| 44.                                                                                   |          | b7           | := slicesCost1 > Veg.pizzaPrice       |
| 45. (b7)                                                                              | :        | slicesCost1  | := Veg.pizzaPrice                     |
| 46. (b6)                                                                              | :        | cost1        | := cost1 + slicesCost1                |
| 47.                                                                                   |          | totalcost    | := totalcost + cost1                  |
| 48-54. <i>repeat assignments 41-47 for 2nd order item (Marg)</i>                      |          |              |                                       |
| 55-61. <i>repeat assignments 41-47 for 3rd order item (Mush)</i>                      |          |              |                                       |
| 62.                                                                                   |          | objective    | := totalcost                          |

**Fig. 2.** Flattened version of `buildOrder` method. We assume an instance where the `menu` lists three different types of pizza (vegetarian, margharita and mushroom), meaning the order will contain three `OrderItems` [Veg, Marg, Mush], and where the people list contains two `Person` objects, the first willing to eat vegetarian or margharita and requiring two slices, and the second willing to eat margharita or mushroom and requiring three slices. The `b` variables have been introduced to store branching conditions. Variables from methods called more than once and those used as the iteration variable in a loop are numbered to distinguish between the different versions.

use the latest version whenever a variable is referred to as part of an expression. If the assignment has some conditions, the new version of the variable can be constrained to equal either the assigned value or the previous version, depending on whether or not the conditions hold. This is easily achieved using a pair of implications, or alternatively using an `element` constraint with the condition as the index. The `element` constraint has the advantage that some propagation is possible before the condition is fixed, so we will use this translation.

The constraint arising from a local variable assignment is shown below, where `localvar0` is the latest version of `localvar` before the assignment, and `localvar1` is

the new variable that results from the assignment, which will become the new latest version of `localvar`. Note that we assume arrays in element constraints are indexed from 1. For convenience, in the rest of the paper we use a simplified syntax for these constraints (also shown below).

**assignment:** condition : localvar := expression  
**constraint:** element(bool2int(condition)+1, [localvar0, expression], localvar1)  
**simple syntax:** localvar1 = [localvar0, expression][condition]

This translation is only correct for local variables. Field assignments are more difficult to handle due to the possibility of aliasing between objects. However, if the set of concrete objects which may be referred to by an object variable is finite (which is the case for our application), then it is possible to convert all field assignments into equivalent assignments over local variables, after which the translation above can be applied.

For each concrete object, a local variable is created to hold the value of each of its fields. In the following we name these variables using the object name and the field name separated by an underscore. Then every field assignment is replaced by a sequence of local variable assignments, one for each of the possibly affected concrete objects. These new assignments retain the original conditions, and each also has one further condition: that its corresponding concrete object is the one referred to by the object variable. Where necessary to avoid duplication, an intermediate variable is created to hold the assigned expression.

An example of this conversion is shown below, where we assume the assignment is on line  $n$  and  $Dom(objectvar, n) = \{Obj1, Obj2, Obj3\}$ .

**field assignment:** condition : objectvar.field := expression  
**assignments:** condition  $\wedge$  (objectvar = Obj1) : Obj1.field := expression  
condition  $\wedge$  (objectvar = Obj2) : Obj2.field := expression  
condition  $\wedge$  (objectvar = Obj3) : Obj3.field := expression

The final requirement is to handle references to object fields. We need to look up the field value for the concrete object corresponding to the current value of the object variable. To achieve this we use a pair of element constraints sharing an index as shown below, where `fieldrefvar` is an intermediate variable representing the retrieved value. We assume the same domain for `objectvar`.

**field reference:** objectvar.field  
**constraints:** element(indexvar, [Obj1, Obj2, Obj3], objectvar)  
element(indexvar, [Obj1.field, Obj2.field, Obj3.field], fieldrefvar)

In summary, this approach involves two steps. First the list of assignments is modified to replace field assignments with equivalent local variable assignments, introducing new variables as required. Then the new list (now containing only local variable assignments) is translated into constraints, with special handling for field references. This approach is quite simple, but can result in a very large model if fields are used extensively. To see the result of applying this translation to a portion of our running example, see Figure 3(a).

### 3.2 Typical SMT Approach

One of the main reasons for the significant advances in program symbolic reasoning (e.g. verification and testing) during the last decade has been the remarkable progress in modern SMT solvers (we refer the reader to [1] for details).

When using SMT, local variable assignments can be translated in the same way as for the CP approach (adding a new version of the variable for each assignment), but using an if-then-else construct (*ite* below) instead of an *element* constraint.

**assignment:**                    condition : localvar := expression  
**formula:**                    localvar1 = ite(condition, expression, localvar0)

For field assignments, it is more convenient to use the theory of arrays. This theory extends the theory of uninterpreted functions with two interpreted functions *read* and *write*. McCarthy proposed [11] the main axiom for arrays:

$$\begin{aligned} \forall a, i, j, x \text{ (where } a \text{ is an array, } i \text{ and } j \text{ are indices and } x \text{ is a value )} \\ i = j \rightarrow \text{read}(\text{write}(a, i, x), j) = x \\ i \neq j \rightarrow \text{read}(\text{write}(a, i, x), j) = \text{read}(a, j) \end{aligned}$$

Note that since we are not interested in equalities between arrays we only focus on the non-extensional fragment.

Following the key idea of Burstall [3] and using the theory of arrays, we define one array variable for each object field. Conceptually, this array contains the value of the field for every object, indexed by object. Note however that there are no explicit variables for the elements.

An assignment to a field is modelled as a write to the array for that field, using the object variable as the index. The result is a new array variable representing the new state of the field for all objects. This is much more concise and efficient than creating an explicit new variable for each concrete object.

We still need to handle assignments with conditions. If the condition does not hold all field values should remain the same, so we can simply use an *ite* to ensure that in this case the new array variable is equal to the previous version.

**field assignment:**                    cond : objectvar.field := expression  
**formula:**                    field1 = ite(cond, write(field0, objectvar, expression), field0)

A reference to an object field is represented as a read of the latest version of the field array, using the object variable as the lookup index.

**field reference:**                    objectvar.field  
**formula:**                    read(field0, objectvar)

For a more complete example, see Figure 3(b). This example clearly demonstrates that the SMT formula can be much more concise than the CP model arising from the translation discussed in the previous section. Its weakness is its inability to reason over disjunction (compared to *element* in the CP approach). The approaches are compared further in Section 4.3.

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> pizza1 in {Veg, Marg} element(index1, [Veg,Marg], pizza1) element(index1, [Veg_numSlices0,Marg_numSlices0], pizza1_numSlices0) temp1 = pizza1_numSlices0 + 1 Marg_numSlices1 = [Marg_numSlices0,temp1][pizza1 = Marg] Veg_numSlices1 = [Veg_numSlices0,temp1][pizza1 = Veg] element(index2, [Veg,Marg], pizza1) element(index2, [Veg_numSlices1,Marg_numSlices1], pizza1_numSlices1) b1 = (pizza1_numSlices1 == slicesPerPizza) Marg_numSlices2 = [Marg_numSlices1,0][b1 ∧ pizza1 = Marg] Veg_numSlices2 = [Veg_numSlices1,0][b1 ∧ pizza1 = Veg] </pre> <p>(a) CP Translation: Constraints</p> |
| <pre> (pizza1 = Marg) ∨ (pizza1 = Veg) numSlicesArray1 = write(numSlicesArray0, pizza1, read(numSlicesArray0,pizza1)+1) b1 = ( read(numSlicesArray1,pizza1) = slicesPerPizza ) numSlicesArray2 = ite(b1, write(numSlicesArray1,pizza1,0), numSlicesArray1) </pre> <p>(b) SMT Translation: Formula</p>                                                                                                                                                                                                                                                                                                |

**Fig. 3.** Translation of assignments 14-17 of the running example (Figure 2) using (a) the obvious CP approach, and (b) the SMT approach

## 4 A New Approach to Modelling Assignments

The main problem with the CP approach presented earlier is the excessive number of variables created to store new field values for every object possibly affected by a field assignment. Essentially we maintain a representation of the complete state of the program after each execution step.

This is not actually necessary. Our only real requirement is to ensure that the values retrieved by variable references are correctly determined by the assignment statements. Maintaining the entire state is a very inefficient way of achieving this, since we may make several assignments to a field using different object variables before ever referring to the value of that field for a particular concrete object. To take advantage of this observation, we move away from the state-based representation, instead simply creating a variable for each field reference, and constraining this to be consistent with the relevant assignments.

### 4.1 The General Case

We first need to define which assignment statements are *relevant* (i.e. may affect the retrieved value) for a given variable reference. Let  $a_i$  be the assignment on line  $i$  of the flattened list, and  $o_i$ ,  $f_i$  and  $c_i$  be the object, field identifier and set of conditions for this assignment. For a reference to variable  $\text{obj.field}$  occurring on line  $n$ , assignment  $a_j$  is *relevant* iff the following conditions hold.

$$\begin{array}{ll}
 j < n \text{ and } f_j = \text{field} & (\text{occurs before the reference, uses the correct field}) \\
 \text{Dom}(\text{obj}, n) \cap \text{Dom}(o_j, j) \neq \emptyset & (\text{assigns to an object which may equal obj}) \\
 \nexists u : o_u = \text{obj}, f_u = \text{field}, c_u = \emptyset, j < u < n & (\text{not overwritten by an unconditional assignment})
 \end{array}$$

As an example, consider the reference to `Veg.fullPizzas` on line 41 in Figure 2. Of the eight assignments to the `fullPizzas` field (all of which occur before this reference), the following three are relevant. The others cannot affect the retrieved value as they use object variables (e.g. `pizza3`) whose domains do not include `Veg`.

1. `Veg . fullPizzas := 0`
18. `(b1) : pizza1 . fullPizzas := pizza1.fullPizzas + 1`
23. `(b2) : pizza2 . fullPizzas := pizza2.fullPizzas + 1`

For a correct model we need constraints ensuring that the retrieved value (`Veg.fullPizzas`) corresponds to the most recent assignment which updated the read variable. To achieve this we introduce a new integer variable `indexvar` whose value indicates which of the relevant assignments this is. We use three element constraints to ensure that the selected assignment applies to the correct object, has true execution conditions, and assigns a value equal to the result.

```
element(indexvar, [Veg.pizza1,pizza2], Veg)
element(indexvar, [true,b1,b2], true)
element(indexvar, [0, pizza1_fullPizzas + 1, pizza2_fullPizzas + 1], Veg_fullPizzas)
```

Note that `pizza1_fullPizzas` and `pizza2_fullPizzas` are the variables introduced for the field references used as part of the assigned values. These would be constrained using their own list of relevant assignments.

The only remaining requirement is that we must choose the *latest* applicable assignment. Using the natural order for the arrays this corresponds to the greatest index. We therefore add constraints stating that if at index  $i$  the object variables are equal and the execution condition is true, then the selected index must be no less than  $i$ .

- $$(b1 \wedge \text{pizza1} = \text{Veg}) \rightarrow \text{indexvar} \geq 2$$
- $$(b2 \wedge \text{pizza2} = \text{Veg}) \rightarrow \text{indexvar} \geq 3$$

The general form of the constraints used for field references is shown below. References to local variables are treated as field references where the object variable is the same as that used for all relevant assignments. When this is the case the first element constraint is not required (as it is trivially satisfied), and the implications can be simplified. Other obvious simplifications are also applied.

|                              |                                                                                                                                                                                                                                                                                                                                                                                                |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>field reference:</b>      | <code>queryobj.field</code>                                                                                                                                                                                                                                                                                                                                                                    |
| <b>relevant assignments:</b> | <code>cond1 : obj1.field := expr1</code><br>...                                                                                                                                                                                                                                                                                                                                                |
|                              | <code>condn : objn.field := exprn</code>                                                                                                                                                                                                                                                                                                                                                       |
| <b>constraints:</b>          | <code>element(indexvar, [obj1, ..., objn], queryobj)</code><br><code>element(indexvar, [cond1, ..., condn], true)</code><br><code>element(indexvar, [expr1, ..., exprn], queryobj_field)</code><br>$(\text{cond2} \wedge \text{queryobj} = \text{obj2}) \rightarrow \text{indexvar} \geq 2$<br>...<br>$(\text{condn} \wedge \text{queryobj} = \text{objn}) \rightarrow \text{indexvar} \geq n$ |

As an optimisation, when the code contains more than one reference to some variable  $v$ , we insert an unconditional assignment to  $v$  at the time of the earlier read, using the read result as the assigned value. This will become the

earliest relevant assignment for the later read, which helps to avoid duplication of expressions and constraints.

There is some similarity between our extraction of relevant assignments and the dynamic slicing technique proposed in [9]. Note however that slicing is used only to reduce the number of statements to be translated into constraints. The actual translation is still based on the standard CP approach.

## 4.2 Special Cases

In some cases, we can detect a pattern to the relevant assignments which allows us to use a more specialised constraint. The three special cases we look for are Boolean variables, sequences of assignments representing a sum calculation, and sequences of assignments representing a maximum or minimum calculation. The translation of assignments automatically detects the cases described below and uses the more efficient translation.

**Boolean Variables.** When the referenced variable is of type `bool`, we can define a Boolean expression for the retrieved value rather than using `element` constraints and implications. The expression (shown below) is true if some assignment with a true value applies and no later assignment with a false value applies.

*field reference:*  $q.\text{field}$

*assignments:*  $c_i : o_i.\text{field} := e_i \quad i \in 1..n$

*expression:* 
$$\bigvee_{i \in 1..n} \left( c_i \wedge e_i \wedge (o_i = q) \wedge \bigwedge_{j \in i+1..n} (e_j \vee \neg c_j \vee (o_j \neq q)) \right)$$

Local variables are handled in the same way except without the object equalities. Using this constraint instead of the generic constraint eliminates the need to introduce an index variable, and allows simplifications to be performed when objects are known to be equal, an assigned value is fixed, or an assignment is unconditional.

**Sum Calculations.** Computations often involve taking the sum of a set of numbers. In a procedural language, sums are commonly calculated by iteratively adding each number to a variable representing the total. This coding pattern results in a sequence of writes where each written value is an addition of the previous value of this variable and some other number. When this pattern is detected, we can replace the usual constraints with a `sum` constraint.

*relevant assignments:*  $\text{total} := 0$

$\text{total} := \text{total} + \text{value1}$

$\text{total} := \text{total} + \text{value2}$

$\text{total} := \text{total} + \text{value3}$

*constraint:*  $\text{finaltotal} = \text{sum}([0, \text{value1}, \text{value2}, \text{value3}])$

In the example above all assignments were unconditional and used exactly the same variable (the local variable `total`). It is also possible to use a general form

of this constraint for sequences of assignments which do not represent a pure sum but a related calculation, such as counting the objects which satisfy some condition. Consider again the relevant writes for the reference to `Veg.fullPizzas` discussed earlier. A better constraint for `Veg.fullPizzas` is shown below.

```
assignments:      Veg . fullPizzas := 0
                   (b1) : pizza1 . fullPizzas := pizza1.fullPizzas + 1
                   (b2) : pizza2 . fullPizzas := pizza2.fullPizzas + 1
constraint:      Veg.fullPizzas =
                   sum([0, bool2int(b1  $\wedge$  pizza1=Veg), bool2int(b2  $\wedge$  pizza2=Veg)])
```

A major advantage of this constraint is that it removes the need to create and constrain the variables `pizza1.fullPizzas` and `pizza2.fullPizzas`. These can be excluded from the model entirely as they were only used to re-assign to the same variable, and are now not required to define the values retrieved from this field.

The general form of the alternative constraint used for sums is given below. The first assignment gives the initial value of `qobj.field`. If not already present it is created from the initialisation assignments for objects in the domain of `qobj`.

```
field reference:  qobj.field
assignments:    qobj.field := init
                   condi : obji.field := obji.field + expri      i  $\in$  1..n
constraint:      qfield = sum([init, expri  $\times$  bool2int(condi  $\wedge$  obji = qobj) | i  $\in$  1..n])
```

**Max/Min Calculations.** Another common coding pattern is to calculate a maximum or minimum by iterating through a list of values overwriting a variable each time a smaller/larger value is found. As with `sum`, we can detect this pattern when building the constraints for the final read of the variable. This time in order for the alternative constraint to apply, every non-initialisation assignment must have a condition which compares the current value of the variable with the assigned value. When there are no other assignment conditions, and the variable is a local variable (or the assigned-to object is known to equal the read object for all relevant assignments), we can use a `max/min` constraint as shown below.

```
assignments:      max := init
                   (value1 > max) : max := value1
                   (value2 > max) : max := value2
constraint:      finalmax = max([init, value1, value2])
```

We can again extend this to apply to field assignments and assignments with additional conditions. When extra conditions are present, we are calculating the maximum or minimum value for which these additional conditions hold. For a maximum, we constrain the result to be no less than any value for which the extra conditions hold, and to equal one of the values for which the conditions hold. Minimum is handled equivalently.

```
field reference:  queryobj.field
assignments:    (condi  $\wedge$  valuei > obji.field) : obji.field := valuei      i  $\in$  1..n
constraints:       $\bigvee_{i \in 1..n} \text{cond}_i \wedge (\text{obj}_i = \text{queryobj}) \wedge (\text{queryobj\_field} = \text{value}_i)$ 
                    $\bigwedge_{i \in 1..n} (\text{cond}_i \wedge \text{obj}_i = \text{queryobj}) \rightarrow \text{queryobj\_field} \geq \text{value}_i$ 
```

**Table 1.** Comparing three approaches to modelling destructive assignments

| Problem |     | smt                 | orig                | Time (secs)         |                     |                     | hand               | Failures (000s) |       |     |      |      |
|---------|-----|---------------------|---------------------|---------------------|---------------------|---------------------|--------------------|-----------------|-------|-----|------|------|
|         |     |                     |                     | orig+               | new                 | new+                |                    | orig            | orig+ | new | new+ | hand |
| proj1   | 200 | 2.2                 | 23.0                | 0.1                 | 12.1                | 0.1                 | 0.1                | 56              | 0     | 34  | 0    | 0    |
|         | 225 | 2.4                 | 3.2                 | 0.1                 | 1.5                 | 0.1                 | 0.1                | 9               | 0     | 4   | 0    | 0    |
|         | 250 | 1.6                 | 61.9 <sub>3</sub>   | 0.1                 | 61.7 <sub>3</sub>   | 0.1                 | 0.1                | 99              | 0     | 127 | 0    | 0    |
| proj2   | 22  | 115.6 <sub>2</sub>  | 84.8 <sub>1</sub>   | 42.7 <sub>1</sub>   | 51.7 <sub>2</sub>   | 23.7 <sub>1</sub>   | 7.6                | 39              | 31    | 110 | 35   | 22   |
|         | 24  | 221.1 <sub>7</sub>  | 286.9 <sub>9</sub>  | 167.6 <sub>5</sub>  | 170.9 <sub>6</sub>  | 129.2 <sub>4</sub>  | 92.2 <sub>4</sub>  | 92              | 89    | 368 | 239  | 280  |
|         | 26  | 262.7 <sub>8</sub>  | 376.2 <sub>16</sub> | 293.3 <sub>10</sub> | 255.9 <sub>11</sub> | 137.9 <sub>6</sub>  | 128.9 <sub>5</sub> | 120             | 144   | 583 | 251  | 452  |
| pizza   | 3   | 56.0                | 37.4 <sub>1</sub>   | 25.1                | 7.0                 | 3.1                 | 2.0                | 175             | 118   | 30  | 14   | 0    |
|         | 4   | 226.4 <sub>8</sub>  | 180.9 <sub>7</sub>  | 175.7 <sub>7</sub>  | 138.0 <sub>4</sub>  | 79.3 <sub>2</sub>   | 2.1                | 544             | 541   | 377 | 252  | 1    |
|         | 5   | 480.9 <sub>22</sub> | 411.8 <sub>18</sub> | 407.5 <sub>18</sub> | 343.4 <sub>13</sub> | 298.3 <sub>12</sub> | 2.2                | 1170            | 1216  | 865 | 945  | 7    |

### 4.3 Comparison with Earlier Approaches

We compared the three presented translation techniques experimentally, using the pizza ordering example plus two benchmarks used in [7] (the other benchmarks require support for collection operations, as discussed in the next section). We used 30 instances for each of several different sizes to evaluate scaling behaviour. For the **original** and **new** CP approaches we show the effect of adding special cases (**orig+** and **new+**). Special cases can be detected in the original method, but only for local variables. Using the new translation makes these cases also recognisable for fields. As a reference we also include a fairly naive **hand** written model for each problem. The Java code defining the problems and all compared constraint models are available online at [www.cs.mu.oz.au/~pjs/optmodel](http://www.cs.mu.oz.au/~pjs/optmodel).

The CP models were solved using the lazy clause generation solver Chuffed. The SMT versions were solved using Z3 [5]. Z3, like most SMT solvers, does not have built-in support for optimisation. We used a technique similar to [12] to perform optimisation using SMT: in incremental mode, we repeatedly ask for a solution with a better objective value until a not satisfiable result is returned.

Table 1 shows average time to solve and failures for the different models. The small number next to the time indicates the number of timeouts ( $> 600$ s). These were included in the average calculations. The results show that while the SMT approach does compete with the original approach, with special case treatment it does not. The **new** approach is quite superior and in fact has a synergy with special cases (since more of them are visible). **new+** competes with **hand** except for **pizza** where it appears that the treatment of the relationship between slices and full pizzas used in **hand** is massively more efficient than the iterative approach in the simulation.

## 5 Collection Operations

The code for the pizza ordering example makes use of collection classes from the Java Standard Library: **Set**, **List** and **Map**. In this case no special handling is required as all collection operations are independent of the decisions, but often it is more natural to write code where that is not the case. For example, say



we wished to extend our application to choose between several possible pizza outlets, each with a different menu. We could do this by adding one extra line at the beginning of the `buildOrder` function.

```
menu = chooseFrom(availableMenus);
```

This change means the contents of the `OrderItem` list in the `Order` class will depend on the decisions, so the `for` loop iterating over this list (in `buildOrder`) will perform an unknown (though bounded) number of iterations, and the result of any query operation on this list will also depend on the decisions.

In [7], collection operations were supported by introducing appropriately constrained variables representing the state of each collection after each update operation (e.g. `List.add`). Query operations (e.g. `List.get`) were represented as a function of the current state of the relevant collections. This is analogous to the way field assignments were handled, with the same drawbacks.

Fortunately our new technique can also be extended to apply to collection operations, resulting in a much more efficient representation. Where previously the flattened list of state changing operations contained only assignments, we now also include collection update operations. Then every query operation on a collection is treated analogously to a field reference. That is, a new variable is created to hold the returned value, and constraints are added to ensure that this value is consistent with the relevant update operations.

Below we provide details of the constraints used for `List` operations. `Set` and `Map` operations are treated similarly; a detailed description is omitted for brevity. We then give experimental results using collection-related benchmarks from [7].

## 5.1 Example: List

For the `List` class we support update operations `add` (at end of list) and `replace` (item at index), and query operations `get` (item at index) and `size`.

A code snippet containing one of each operation type is shown below. Also shown are the assumed possible variable values and initial list contents, and the flattened list of collection update operations. Each update operation has an associated condition, list, index and item. For the `add` operation, the index is a variable `size1` holding the current size of `list1`. The first three operations in the table reflect the original contents of the lists.

| <pre>if(cond) {   list1.add(A);   list1.replace(0, item); } if(list2.size() &gt; ind)   item = list2.get(ind);</pre> | <pre>list1 ∈ {L1,L2,L3} ind ∈ {0,1,2} list2 ∈ {L1,L2,L3} item ∈ {A,B,C} cond ∈ {true,false} L1:[A,B], L2:[C], L3:[]</pre> | <table><tr><th>Cond</th><th>List</th><th>Index</th><th>Item</th></tr><tr><td></td><td>L1</td><td>[0]</td><td>:= A (<i>add</i>)</td></tr><tr><td></td><td>L1</td><td>[1]</td><td>:= B (<i>add</i>)</td></tr><tr><td></td><td>L2</td><td>[0]</td><td>:= C (<i>add</i>)</td></tr><tr><td>cond : list1</td><td>[size1]</td><td></td><td>:= A (<i>add</i>)</td></tr><tr><td>cond : list1</td><td>[0]</td><td></td><td>:= item (<i>repl</i>)</td></tr></table> | Cond                    | List | Index | Item |  | L1 | [0] | := A ( <i>add</i> ) |  | L1 | [1] | := B ( <i>add</i> ) |  | L2 | [0] | := C ( <i>add</i> ) | cond : list1 | [size1] |  | := A ( <i>add</i> ) | cond : list1 | [0] |  | := item ( <i>repl</i> ) |
|----------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------|------|-------|------|--|----|-----|---------------------|--|----|-----|---------------------|--|----|-----|---------------------|--------------|---------|--|---------------------|--------------|-----|--|-------------------------|
| Cond                                                                                                                 | List                                                                                                                      | Index                                                                                                                                                                                                                                                                                                                                                                                                                                                    | Item                    |      |       |      |  |    |     |                     |  |    |     |                     |  |    |     |                     |              |         |  |                     |              |     |  |                         |
|                                                                                                                      | L1                                                                                                                        | [0]                                                                                                                                                                                                                                                                                                                                                                                                                                                      | := A ( <i>add</i> )     |      |       |      |  |    |     |                     |  |    |     |                     |  |    |     |                     |              |         |  |                     |              |     |  |                         |
|                                                                                                                      | L1                                                                                                                        | [1]                                                                                                                                                                                                                                                                                                                                                                                                                                                      | := B ( <i>add</i> )     |      |       |      |  |    |     |                     |  |    |     |                     |  |    |     |                     |              |         |  |                     |              |     |  |                         |
|                                                                                                                      | L2                                                                                                                        | [0]                                                                                                                                                                                                                                                                                                                                                                                                                                                      | := C ( <i>add</i> )     |      |       |      |  |    |     |                     |  |    |     |                     |  |    |     |                     |              |         |  |                     |              |     |  |                         |
| cond : list1                                                                                                         | [size1]                                                                                                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                          | := A ( <i>add</i> )     |      |       |      |  |    |     |                     |  |    |     |                     |  |    |     |                     |              |         |  |                     |              |     |  |                         |
| cond : list1                                                                                                         | [0]                                                                                                                       |                                                                                                                                                                                                                                                                                                                                                                                                                                                          | := item ( <i>repl</i> ) |      |       |      |  |    |     |                     |  |    |     |                     |  |    |     |                     |              |         |  |                     |              |     |  |                         |
| (a) Code                                                                                                             | (b) Variables                                                                                                             | (c) Update Operations                                                                                                                                                                                                                                                                                                                                                                                                                                    |                         |      |       |      |  |    |     |                     |  |    |     |                     |  |    |     |                     |              |         |  |                     |              |     |  |                         |

With our limited set of supported update operations (which is nevertheless sufficient to cover all code used in the benchmarks from [7]), the size of a list is



failure when list2 is empty. We fix the result rather than leaving it unconstrained to avoid searching over its possible values.

```

query:      getresult := list2.get(ind)
constraints: element(indexvar, [L1,L1,L2,list1,list1,list2], list2)
                element(indexvar, [0,1,0,size1,0,ind], ind)
                element(indexvar, [true,true,true,cond,cond,¬(sizeresult>ind)], true)
                element(indexvar, [A,B,C,A,item,A], getresult)
                (list2=L1) ∧ (ind=1)           → indexvar ≥ 2
                (list2=L2) ∧ (ind=0)           → indexvar ≥ 3
                (list2=list1) ∧ (ind=size1) ∧ cond → indexvar ≥ 4
                (list2=list1) ∧ (ind=0) ∧ cond   → indexvar ≥ 5
                ¬(sizeresult>ind)                → indexvar ≥ 6

```

## 5.2 Comparison on Benchmarks with Collections

Table 2 compares the various translation approaches (excluding **smt** and **orig** which were shown to be not competitive in Table 1) and hand written models, using problems involving collections from [7]. It is clear that the new translation substantially improves on the old in most cases, and is never very much worse (**bins,knap1**). With the addition of special case treatment the new translation is often comparable to the hand written model, though certainly not always (**proj3,route**). In a few instances it is superior (**bins,golf**), this may be because it uses a sequential search based on the order decisions are made in the Java code, or indeed that the intermediate variables it generates give more scope for reusable nogood learning.

## 6 Conclusion

Effective modelling of destructive assignment is essential for any form of reasoning about procedural code. We have developed a new encoding of assignment and state that gives effective propagation of state-related information. We demonstrate the effectiveness of this encoding for the automatic generation of optimisation models from simulation code, showing that the resulting model can be comparable in efficiency to a hand-written optimization model.

In the future we will investigate the use of this encoding for applications such as test generation. The main difference is the lack of a known initial state. This will require the creation of variables to represent unknown initial field values, with constraints ensuring that if a pair of object variables are equal then their corresponding initial field variables are also equal. Uncertainty about the initial state will also affect the number of relevant assignments for field references. For a query object with unbounded domain all assignments to the same field occurring prior to the read are relevant, unless one of these is an unconditional assignment using this exact variable. These differences may mean that redundant constraints relating reads to each other (which we have not discussed due to their lack of impact for our application) become more important for effective propagation.

**Acknowledgments.** NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

## References

1. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T.: Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (February 2009)
2. Brodsky, A., Nash, H.: CoJava: Optimization modeling by nondeterministic simulation. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 91–106. Springer, Heidelberg (2006)
3. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. Machine Intelligence 7, 23–50 (1972)
4. Collavizza, H., Rueher, M., Van Hentenryck, P.: CPBPV: a constraint-programming framework for bounded program verification. Constraints 15(2), 238–264 (2010)
5. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. Floyd, R.W.: Assigning meanings to programs. In: Proceedings of the American Mathematical Society Symposia on Applied Mathematics, vol. 19, pp. 19–31 (1967)
7. Francis, K., Brand, S., Stuckey, P.J.: Optimisation modelling for software developers. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 274–289. Springer, Heidelberg (2012)
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12(10), 576–580 (1969)
9. Hofer, B., Wotawa, F.: Combining slicing and constraint solving for better debugging: The CONBAS approach. Advances in Software Engineering 2012, Article ID 628571 (2012)
10. King, J.C.: Symbolic Execution and Program Testing. Com. ACM, 385–394 (1976)
11. McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress, pp. 21–28 (1962)
12. Sebastiani, R., Tomasi, S.: Optimization in SMT with  $LA(Q)$  cost functions. In: IJCAR, pp. 484–498 (2012)

# An Improved Search Algorithm for Min-Perturbation

Alex Fukunaga

The University of Tokyo

**Abstract.** In many scheduling and resource assignment problems, it is necessary to find a solution which is as similar as possible to a given, initial assignment. We propose a new algorithm for this minimal perturbation problem which searches a space of variable commitments and uses a lower bound function based on the minimal vertex covering of a constraint violation graph. An empirical evaluation on random CSPs show that our algorithm significantly outperforms previous algorithms, including the recent two-phased, hybrid algorithm proposed by Zivan, Grubshtein, and Meisels.

## 1 Introduction

In many CP applications it is necessary to find solutions that are as similar as possible to a given, initial assignment of values to variables. For example, in a meeting scheduling problem or resource scheduling problem, constraints can change unexpectedly after a solution has been generated. This is a type of dynamic constraint satisfaction problem. Similarly, there are situations where there is an “ideal” (but possibly infeasible) assignment of values to variables for a CSP, and the goal is to find an assignment which differs as little as possible from the target. Another scenario where a solution similar to a given initial state is desired occurs in staff scheduling. Employees express preferences regarding when they want to work, but their preferences must be balanced against the staffing demands and constraints of the business, requiring a schedule that satisfies staffing requirements while deviating minimally from employee preferences.

This paper considers search algorithms for this class of *minimal perturbation problem* (MPP) for CSPs, where we seek a solution that minimizes the number of variables whose values differ from a target assignment, or equivalently, the minimal number of variable changes that are necessary to a CSP solution when some of the constraints change unexpectedly. In particular, we focus on minimal perturbation for binary CSPs. Previously, Ran et al. proposed an iterated deepening algorithm for the MPP that searches the space of variable assignments that differ from the target/initial assignment by at most  $d$  assignments, where  $d$  is the iterative deepening bound [9]. More recently, Zivan, Grubshtein, and Meisels proposed a two-phased algorithm that interleaves the problem of bounding the number of necessary perturbations from the initial assignment, and the problem of testing if such an assignment is possible [11].

We propose a new search algorithm for the MPP, where the main features are (1) a search space where nodes represent a set of *committed* variable assignments, (2) a lower bound based on the minimal vertex covering of the current set of violated constraints, which dominates the lower bound by Zivan et al. This generalizes an earlier, domain-specific MPP algorithm proposed in [4].

## 2 Problem Definition and Preliminaries

The *Minimal Perturbation Problem* (MPP) is defined as follows: Let  $C = (V, D, C)$  be a CSP, where  $V = v_1, \dots, v_n$  is a set of variables,  $D = D_1, \dots, D_n$  is a set of domains where  $D_i$  is a finite discrete set of possible values for variable  $v_i$ , and  $C = c_1, \dots, c_m$  is a set of constraints which restricts the set of values that the variables can be simultaneously assigned.

Let  $I$  be a complete assignment for  $C$ . The objective of the MPP is to find an assignment  $A$  such that all of the constraints are satisfied, and the *number* of variables in  $A$  whose value differs from  $I$  is minimized. Following [11], the value of variable  $v$  in the original assignment is called the *Starting Variable Assignment* of  $v$ , or its SVA.

While previous work [9,11] defined the MPP more generally, i.e., a general distance function, and a partial initial assignment, the lower bound functions used in the previous work assume the definition above, and the actual experimental evaluations of the previous algorithms were performed on binary CSPs based on this definition.

## 3 Previous Algorithms for the MPP

The first algorithm which specifically addressed the MPP defined in Sec 2 was the *Repair-Based algorithm with Arc-Consistency (RB-AC)*, by Ran et al. [9]. Given an initial variable assignment  $I = \{x_1 = v_1, \dots, x_n = v_n\}$ , let  $D_i$  be the set of states which have exactly  $i$  variables whose value are different from that of the initial state  $I$ . We call the set  $D = D_1 \cup \dots \cup D_n$  the *difference space*, or *D-space*. The root node of this search space is  $I$ . Nodes at depth  $d$  of the search tree contain variable assignments which differ by  $d$  assignments from  $I$ . Each edge in the tree changes the value of one variable which has not yet been changed by any ancestor. RB-AC searches D-space using a depth-first iterative deepening strategy, IDA\* [7]. The  $d$ -th iteration of IDA\* explores the subset of the depth-first branch-and-bound D-space search tree where at each node, the sum  $f = g + h \leq d$ , where  $g$  is the number of differences from the initial state in the current solution, and  $h$  is the lower bound on the additional number of differences required to find a conflict-free solution. RB-AC uses a simple lower bound,  $L_1$ , which is the number of variables that do not have the SVA in its domain.

Zivan, Grubshtein, and Meisels proposed HS\_MPP, a “hybrid” search algorithm for the MPP [11]. Their algorithm consists of two, interleaved phases: The first phase performs branch-and-bound on a binary search tree where each node represents a variable, and the branches correspond to a decision regarding whether to assign the variable to the same value as in the initial assignment. At each node, HS\_MPP-Phase1 computes a lower bound on the number of perturbations, and prunes the search if this exceeds or equals the current upper bound. Then,  $v$ , variable such that  $SVA(v) \in dom(v)$  is selected. If there is no such variable (i.e., all remaining variables must be perturbed), then HS\_PP-Phase2, described below, is called to test for feasibility. Otherwise, HS\_MPP branches: The left branch assigns  $v$  its SVA and recursively searches the remaining variables; the right branch of the binary search tree, HS\_MPP eliminates the SVA from the domain of  $v$ , and recursively searches the remaining variables.

The HS\_MPP algorithm uses a lower bound, which we denote  $L_Z$ , to prune the branch-and-bound tree in HS\_MPP-Phase1. This bound improves upon  $L_1$  by exploiting

the fact that if there is a pair of variables which have the SVA in the domain, but the SVAs conflict with each other, then one of these variables must be assigned a non-SVA, so the bound can be increased relative to  $L_1$  by accounting for such pairs (see [11]).

After each decision in Phase 1, the following, limited filtering function is applied: For each remaining variable  $v$ ,  $SVA(v)$  is removed from  $domain(v)$  if  $SVA(v)$  is inconsistent with the current assignment of SVAs.

HS\_MPP-Phase2 applies a standard MAC (maintaining-arc-consistency) algorithm to the remaining variables (i.e., variables which do not have the SVA in the domain and must be perturbed). If the MAC algorithm finds a satisfying assignment of values to  $v_r$ , then this is a solution to the MPP.

Finally, a third previous approach is by Hebrard, O'Sullivan and Walsh, who proposed a GAC for distance constraints [6]. Zivan et al compared HS\_MPP to this GAC method and showed that HS\_MPP performed significantly better on random binary CSPs (30-40 variables) and meeting rescheduling problems.

## Related Work

Other previous work has addressed problems that are related to (but different from) the MPP formulation treated in this paper. A Dynamic CSP is a sequence of constraint satisfaction problems where each instance is derived from the previous instance by modifying some constraints [2]. Verfaillie and Schiex solved Dynamic CSPs by repairing the solution to the previous CSP instance [10]. They proposed a depth-first backtracking algorithm in D-space. Since the goal is to solve the Dynamic CSP instance, there is no mechanism to guarantee minimal perturbation, although they incorporate variable ordering heuristics that tend to bias the search towards a minimal perturbation solution. El Sakkout and Wallace [3] investigated a minimal cost repair problem for scheduling. They consider difference functions that can be expressed linearly (our MPP difference count objective is nonlinear). Their probe backtracking algorithm does not explicitly consider the initial schedule, and reschedules from scratch [3]. Barták et al. investigated overconstrained CSPs for which there is likely to be no feasible solution without violated constraints [1], and studied methods to seek a maximal assignment of consistent variables which also differs minimally from an initial state. They also studied an iterative repair (local search) algorithm biased to seek minimal perturbation solutions for course timetabling [8].

## 4 A Commitment-Space Search Algorithm for the MPP

We now describe our algorithm for the MPP. Unlike RB-AC, which searches D-space, and HS\_MPP, which searches a 2-phase search in the space of variable assignments, our algorithm searches the space of variable commitments.

In a *commitment-based search space* (C-space) for the MPP, each node in the search tree represents a complete assignment of values to variables, where some subset of the variables are committed to their current value. Edges in the search tree represent a decision to commit a variable to some value. For each variable, we represent its current value, as well as whether a commitment has been made to the value. The root node of

this search space is the initial assignment  $I$ . We say that a variable  $x$  is *committed* to value  $v$  at node  $N$  if  $x$  is assigned to  $v$  at  $N$  and every descendant of  $N$ , and *uncommitted* otherwise.

Each node represents the result of committing some variable to a particular value. Thus, this search space has a branching factor of  $d$ , the domain size, and a maximum depth of  $n$ , the number of variables. We originally proposed C-space for minimal perturbation in [4]. However that previous work focused on a specific type of MPP (bin packing constraint repair e.g., virtual machine reassignment in data centers), and C-space has not been evaluated for standard, domain-independent binary CSPs. C-space has a narrower structure (smaller branching factor) compared to D-space, at the cost of some redundancy. See [4] for an analysis, as well as a figure illustrating example search trees.

We evaluated both a standard depth-first branch-and-bound strategy, as well as an iterative deepening (IDA\*) strategy [7] for C-space. Although iterative deepening can repeatedly visit the same state, in cases where the minimal perturbation solution is close to the initial solution, the IDA\* search strategy would be expected to be faster than depth-first branch-and-bound.

For both of these strategies, a standard, most-constrained variable ordering is used, and a min-conflicts (with respect to the original values in the initial assignment) value ordering is used. At each node, arc consistency (AC-3) is applied for filtering. The depth-first branch-and-bound version is shown in Algorithm 1.

## Lower Bound

The new lower bound function is based on a constrained vertex covering of a constraint violation graph. At every node in the search tree, there is a non-empty set of violated constraints. Given the set of all violated constraints, we construct a *constraint violation graph*  $G$  where each variable corresponds to a vertex in  $G$ , and there is an edge between vertex  $v_i$  and  $v_j$  if a constraint between variables  $x_i$  and  $x_j$  is violated. A vertex cover (VC) of a graph is a subset  $vc \subset V$  of the vertices such that for every edge  $e = (v_a, v_b)$  in  $G$ , either  $v_a \in vc$  or  $v_b \in vc$ . A minimal vertex cover of  $G$  is a covering of  $G$  which has minimal cardinality.

The minimal VC of a constraint violation graph is clearly a relaxation of the MPP. The minimal VC identifies a subset of variables that could possibly eliminate all constraint violations, without identifying the actual values that must be assigned. The covering has one additional constraint: variables which no longer have the SVA in the domain are forced to be included in the covering. Thus, the cardinality of the (constrained) minimal VC is a lower bound on the number of perturbations required to result in a conflict-free assignment of values to variables. It is easy to see that this bound dominates the  $L_Z$  bound [11].

Although computing a minimal vertex cover is NP-complete [5], computing the minimal VC of a graph is much easier than solving the MPP (the search space is a binary tree with depth =  $\#vars$ , as opposed to a tree with branch factor  $|Domain|$  for the MPP C-space search), so the minimal VC can be used as the basis for a lower bound. Our current implementation performs a straightforward branch-and-bound search where each node determines whether a variable is included or excluded from the cover. A simple filtering/pruning rule is used: for every edge  $(v_a, v_b)$ , if  $v_a$  is excluded from the covering,



then  $v_b$  must be included; conversely, if  $v_b$  is excluded, then  $v_a$  must be included. No other lower-bounding techniques or optimizations are used in the minimal VC computation, but as shown below, this simple implementation suffices in practice.

---

**Algorithm 1.** C-space Search Algorithm
 

---

```

mpp_search(uncommittedVars, committedVars, numChanges)
if get_conflicts(uncommittedVars, committedVars) == 0 then
    if count_num_perturbations(committedVars) < minimalChanges then
        minimalChanges = count_num_perturbations(committedVars) {replace best-so-far solution}
    return s success
if lowerbound(uncommittedVars, committedVars) > minimalChanges then
    return f failure {pruning based on lower bound}
V = select(uncommittedVars)
for all val in Order(domain(V)) do
    commit(V, val) {commitment also applies filtering (arc-consistency)}
    r = mpp_search(uncommittedVars \ V, committedVars ∪ V)
    if r == success then
        return s success
return f failure
  
```

---

## 5 Experimental Evaluation

We evaluated the performance of the MPP algorithms using problems derived from standard, randomly generated binary CSPs. The classes of MPPs used in the experiments are defined by 5 parameters  $(n, k, p_1, p_2, \delta)$ . The first 4 parameters are used to first generate a random, uniform binary CSP,  $C$ , where  $n$  is the number of variables,  $k$  is domain size of all variables,  $p_1$  is the constraint density (probability that any 2 variables have a constraint), and  $p_2$  is the tightness (probability that any 2 values in a pair of constraint variables are a nogood). Then,  $C$  is solved using a standard CSP solver. If  $C$  is unsatisfiable, then it is discarded. If  $C$  is satisfiable, then the solution that is found is used as  $I$ , the initial assignment for the MPP. Then,  $C$  is perturbed by replacing some fraction  $\delta$  of the constraints, resulting in a perturbed CSP  $C'$ , and the MPP instance is  $(C', I)$ . For  $n=30$  and 40 variables, we generated 30 candidate binary CSPs each for all combinations of  $p_1, p_2, \delta$ , where  $p_1 \in 0.3, 0.4, 0.5, 0.6, 0.7$ ,  $p_2 \in 0.3, 0.4, 0.5, 0.6, 0.7$ , and  $\delta \in 0.05, 0.10, 0.25, 0.50, 0.75, 1.00$ . All of these were tested for solvability using a standard CSP solver (i.e., whether there is any satisfying assignment, regardless of distance from the initial configuration  $I$ ). Of these, 2676 of the 30-variable instances and 1578 of the 40-variable instances were satisfiable MPPs. Similarly, for  $n=50$  variables, we generated 30 candidate MPPs for all combinations of  $p_1, p_2$  taken from  $p_1 \in 0.3, 0.4, 0.5$ ,  $p_2 \in 0.3, 0.4, 0.5$ ,  $\delta \in 0.05, 0.10, 0.25, 0.50, 0.75, 1.00$ . and 936 were satisfiable.

In the experiments below, we compare the algorithms in such a way that only these solvable instances matter, i.e., comparisons of the time to find solutions, with a time limit of 900 seconds. This is because unsolvable instances can be detected by running a standard CSP solver much more quickly than any of the MPP algorithms (clearly, checking satisfiability is a simpler problem than seeking a minimal perturbation). Our

new algorithm is at least as fast as the previous algorithms in detecting unsatisfiability. In practice, the best strategy would be to first run a standard CSP to check for satisfiability, then run a dedicated MPP solver to minimize the perturbations.

We compared the following algorithms:

- C-space/ $L_{vc}$  - our C-space search algorithm using the new  $L_{vc}$  lower bound and depth-first branch-and-bound.
- C-space/ $L_{vc}/ID$  - Iterative Deepening C-space search algorithm using the  $L_{vc}$  lower bound.
- HS\_MPP - The hybrid algorithm by Zivan et al [11].
- RB-AC/ $L_{vc}$  - A modified version of RB-AC algorithm by Ran et al [9], which uses our  $L_{VC}$  lower bound instead of the  $L_1$  bound [9] and searches D-space using iterative-deepening.
- C-space/ $L_Z$  - C-space search algorithm using the  $L_Z$  lower bound [11]. This comparison isolates the effect of the lower bound function  $L_{vc}$  compared to  $L_Z$ .

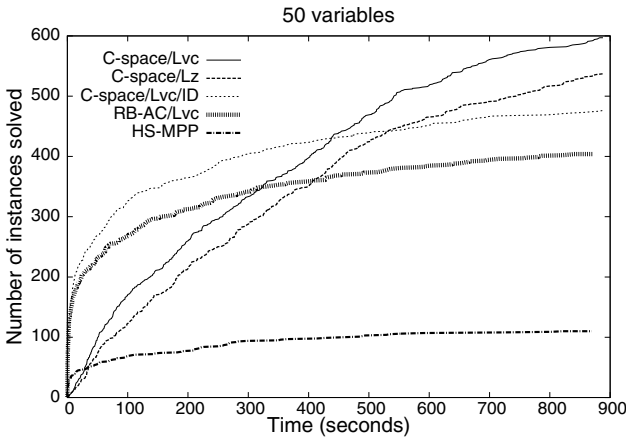


**Fig. 1.**  $n=50$ , cumulative number of problems that can be solved after a given time

Each algorithm was executed on each of the 30, 40, and 50-variable random binary MPP instances, with a 900 second time limit per run. Note that although we focus on runtime due to space restrictions, comparisons of the number of backtracks and constraint checks are qualitatively similar to the runtime results.

Figure 1 shows an overall comparison of the MPP algorithms, and plots the cumulative number of problems solved (y-axis) as the amount of time increases (x-axis) by each algorithm for the 50-variable problems. For example, the C-space/ $L_{vc}$  algorithm solved around 500 instances within 500 seconds. Overall, C-space/ $L_{vc}$  performed best on the hardest instances (which require  $> 400$  seconds), while C-space/ $L_{vc}/ID$  performed best on problems requiring less than 400 seconds. Another interesting result is that RB-AC with the  $L_{vc}$  bound performs significantly better than HS\_MPP, suggesting that the success of HS\_MPP compared to the original RB-AC algorithm was due much more to the lower bound than to the hybrid search strategy.

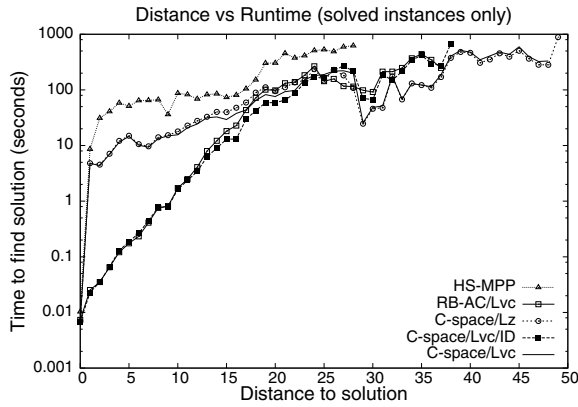


**Fig. 2.** Effect of Distance from Initial/Target State

While results for 30 and 40 variable problems are not shown due to space, they look similar, except that C-space/ $L_{vc}$ /ID performs relatively better with fewer variables.

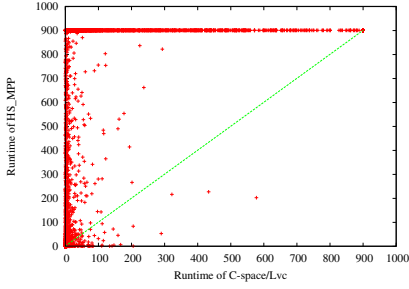
Figure 2 plots average runtime required to solve instances as a function of the distance of the solution found to the initial assignment. This only includes successful runs and excludes runs that timed out, so some of the lines appear truncated (otherwise, for the less successful algorithms, it is difficult to see the impact of distance because there were so many failed runs). Overall, if the distance to a solution is within 10-15 variable assignment changes (i.e., the amount of repair required is small), the faster algorithms such as C-space/ $L_{vc}$  can solve the problems within 10 seconds (if at all).

Figure 3 compares key pairs of MPP algorithms on all of the 30, 40, and 50 variable problems. Each figure plots the runtimes for all instances on a pair of algorithms  $A_1$ ,  $A_2$ , where the x-coordinate is the runtime of  $A_1$  on the instance, and the y-coordinate is the runtime of  $A_2$ . An x or y value of 900 indicates failure to solve the instance. The straight diagonal line is  $(x=y)$ , i.e., points above the line indicate that C-space/ $L_{vc}$  performed better, while points below the line indicate that the other algorithm performed better.

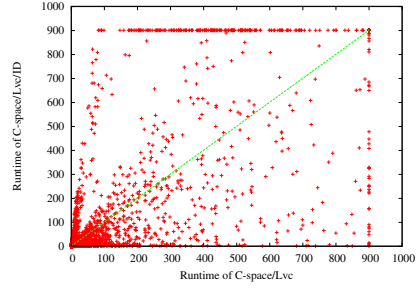
Figure 3a shows that C-space/ $L_{vc}$  clearly outperforms HS\_MPP, the previous state-of-the-art algorithm. The average ratio of runtimes for HS\_MPP vs C-space/ $L_{vc}$  is 86.48 for all problems that were solved by at least 1 of these solvers, and 160.35 for problems that took more than 60 seconds for the faster solver on each instance.

Figure 3d compares C-space/ $L_{vc}$ /ID and RB-AC/ $L_{vc}$ . These two algorithms, which both use iterative deepening search and the same lower bound ( $L_{vc}$ ) differ mainly in the choice of search space (C-space and D-space, respectively). Figure 3d shows that C-space/ $L_{vc}$ /ID clearly outperforms RB-AC/ $L_{vc}$  on almost every problem instance, suggesting that C-space is better structured for search than D-space. However, the advantage of C-space over D-space seems to be less pronounced for this class of benchmarks compared to the virtual machine reassignment problem in [4].

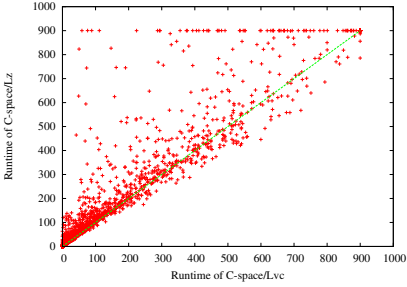
Figure 3b compares C-space/ $L_{vc}$  and C-space/ $L_Z$ . Combined with Figures. 1, and 2 the results show that iterative deepening is a good strategy for quickly solving relatively easy problems (problems where the distance from  $I$  to a solution is small); however,



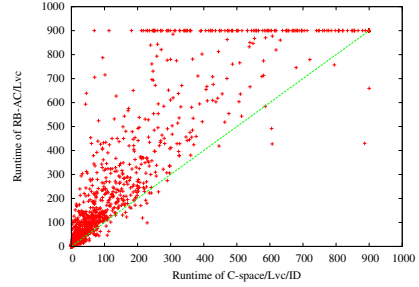
(a) Comparison with previous state-of-the-art (C-space/ $L_{vc}$  vs. HS\_MPP)



(b) Effect of iterative deepening (C-space/ $L_{vc}$  vs. C-space/ $L_{vc}/ID$ )



(c) Impact of new lower bound  $L_{vc}$  (C-space/ $L_{vc}$  vs. C-space/ $L_z$ )



(d) Impact of search space (C-space/ $L_{vc}/ID$  vs. RB-AC/ $L_{vc}$ )

**Fig. 3.** Pairwise comparison of MPP algorithms (includes all 30,40, and 50 variable problems)

for harder problems (where the distance from  $I$  to a solution is large), straightforward depth-first branch-and-bound seems to be a more robust choice.

Figure 3c compares C-space/ $L_{vc}$  and C-space/ $L_z$ . The results show that the new vertex-cover based lower bound  $L_{vc}$  clearly outperforms the previous lower bound  $L_z$  by Zivan et al [11]. The average ratio of runtimes using lower bound  $L_z$  vs  $L_{vc}$  is 1.73 for all instances solved by at least one solver, and 2.10 for instances that required 60 seconds or more for the faster solver.

## 6 Discussion and Conclusions

We proposed a search algorithm for optimal solutions to the min-perturbation problem. Our main contributions are: (1) We showed that our new CSpace/ $L_{vc}$  algorithm significantly improves upon the previous state of the art (HS\_MPP) for random binary CSPs generated with a wide range of parameters. (2) We showed that both  $L_{vc}$ , the new lower bound for the MPP based on vertex covering of the constraint graph, as well as the C-space search space contribute significantly to the performance of the new algorithm (Fig. 3). Future work includes evaluation on applications such as employee shift rescheduling and meeting rescheduling.

## References

1. Barták, R., Müller, T., Rudová, H.: A new approach to modeling and solving minimal perturbation problems. In: Apt, K.R., Fages, F., Rossi, F., Szeredi, P., Váncza, J. (eds.) CSCP 2003. LNCS (LNAI), vol. 3010, pp. 233–249. Springer, Heidelberg (2004)
2. Dechter, R., Dechter, A.: Belief maintenance in dynamic constraint networks. In: Proc. AAAI, pp. 37–42 (1988)
3. El-Sakkout, H., Wallace, M.: Probe backtrack search for minimal perturbation in dynamic scheduling. *Constraints* 5, 359–388 (2000)
4. Fukunaga, A.S.: Search spaces for min-perturbation repair. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 383–390. Springer, Heidelberg (2009)
5. Garey, M., Johnson, D.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company (1979)
6. Hebrard, E., O’Sullivan, B., Walsh, T.: Distance constraints in constraint satisfaction. In: Proc. IJCAI, pp. 106–111 (2007)
7. Korf, R.: Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence* 27(1), 97–109 (1985)
8. Müller, T., Rudová, H., Barták, R.: Minimal perturbation problem in course timetabling. In: Burke, E.K., Trick, M.A. (eds.) PATAT 2004. LNCS, vol. 3616, pp. 126–146. Springer, Heidelberg (2005)
9. Ran, Y., Roos, N., van den Herik, H.: Approaches to find a near-minimal change solution for dynamic CSPs. In: Proc. CP-AI-OR, pp. 378–387 (2002)
10. Verfaillie, G., Schiex, T.: Solution reuse in dynamic constraint satisfaction problems. In: Proc. AAAI, Seattle, Washington, pp. 307–312 (1994)
11. Zivan, R., Grubshtein, A., Meisels, A.: Hybrid search for minimal perturbation in dynamic CSPs. *Constraints* 16, 228–249 (2011)

# Explaining Propagators for Edge-Valued Decision Diagrams

Graeme Gange<sup>1</sup>, Peter J. Stuckey<sup>1,2</sup>, and Pascal Van Hentenryck<sup>1,2</sup>

<sup>1</sup> National ICT Australia, Victoria Laboratory

<sup>2</sup> Department of Computer Science and Software Engineering

The University of Melbourne, Vic. 3010, Australia

ggange@csse.unimelb.edu.au,

{peter.stuckey,pvh}@nicta.com.au

**Abstract.** Propagators that combine reasoning about satisfiability and reasoning about the cost of a solution, such as weighted all-different, or global cardinality with costs, can be much more effective than reasoning separately about satisfiability and cost. The COST-MDD constraint is a generic propagator for reasoning about reachability in a multi-decision diagram with costs attached to edges (a generalization of COST-REGULAR). Previous work has demonstrated that adding nogood learning for MDD propagators substantially increases the size and complexity of problems that can be handled by state-of-the-art solvers. In this paper we show how to add explanation to the COST-MDD propagator. We demonstrate on scheduling benchmarks the advantages of a learning COST-MDD global propagator, over both decompositions of COST-MDD and MDD with a separate objective constraint using learning.

## 1 Introduction

Optimization constraints merge the checking of feasibility and optimization conditions into a single propagator. A propagator for an optimization constraint filters decisions for variables which cannot take part in a solution which is better than the best known solution. They also propagate the bounds on the cost variable to keep track of its lower bound, and hence allow fathoming of the search, when no better solution can be found. There is a significant body of work on optimization constraints including: *weighted alldifferent* [1] and *global cardinality with costs* [2]. In this paper we examine the COST-MDD optimization constraint which is a generalization of the COST-REGULAR [3] constraint.

Previous work has explored the use of *Boolean Decision Diagrams* (BDDs) [4,5] and *Multi-valued Decision Diagrams* (MDDs) [6] for automatically constructing efficient global propagators. But these propagators do not handle costs. And adding a separate objective function constraint to encode the costs, leads to significantly weaker propagation.

COST-MDD is a generic constraint that can be used to encode many problems where the feasibility of a sequence of decisions is represented by an MDD, and the costs of the sequence of decisions is given by the sum of the weights on the

edges taken in this MDD. COST-REGULAR [3] is encoded as a particular form of COST-MDD where the set of states at each level is uniform, and the transition from one level to another is uniform. The WEIGHTED-GRAMMAR constraint [7] is a similar optimization constraint which permits a more concise encoding of some constraints than COST-MDD, but is less convenient to construct and manipulate.

In this paper we investigate how to incorporate COST-MDD global propagators into a lazy clause generation [8] based constraint solver. The principle challenge is to be able to explain propagations as concisely as possible, in order that the nogoods learnt are as reusable as possible. We give experimental evidence that explaining COST-MDD propagators outperform both decompositions of COST-MDD and previous MDD-based propagators.

## 2 Preliminaries

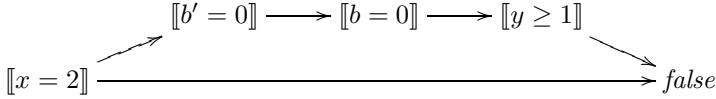
Constraint programming solves constraint satisfaction problems by interleaving propagation, which remove impossible values of variables from the domain, with search, which guesses values. All propagators are repeatedly executed until no change in domain is possible, then a new search decision is made. If propagation determines there is no solution then search undoes the last decision and replaces it with the opposite choice. If all variables are fixed then the system has found a solution to the problem. For more details see e.g. [9].

We assume we are solving a constraint satisfaction problem over set of variables  $x \in \mathcal{V}$ , each of which takes values from a given initial finite set of values or *domain*  $D_{init}(x)$ . The domain  $D$  keeps track of the current set of possible values  $D(x)$  for a variable  $x$ . Define  $D \sqsubseteq D'$  iff  $D(x) \subseteq D'(x), \forall x \in \mathcal{V}$ . We let  $\text{lb}_D(x) = \min D(x)$  and  $\text{ub}_D(x) = \max D(x)$ , and will omit the  $D$  subscript when  $D$  is clear from the context. The constraints of the problem are represented by propagators  $f$  which are functions from domains to domains which are monotonically decreasing  $f(D) \sqsubseteq f(D')$  whenever  $D \sqsubseteq D'$ , and contracting  $f(D) \sqsubseteq D$ .

We make use of constraint programming with learning using the lazy clause generation [8] approach. Learning keeps track of what caused changes in domain to occur, and on failure computes a *nogood* which records the reason for failure. The nogood prevents search making the same incorrect set of decisions later.

In a lazy clause generation solver integer domains are also represented using Boolean variables. Each variable  $x$  with initial domain  $D_{init}(x) = [l..u]$  is represented by two sets of Boolean variables  $\llbracket x = d \rrbracket, l \leq d \leq u$  and  $\llbracket x \leq d \rrbracket, l \leq d < u$  which define which values are in  $D(x)$ . We use  $\llbracket x \neq d \rrbracket$  as shorthand for  $\neg \llbracket x = d \rrbracket$ , and  $\llbracket x \geq d \rrbracket$  as shorthand for  $\neg \llbracket x \leq d - 1 \rrbracket$ . A lazy clause generation solver keeps the two representations of the domain in sync. For example if variable  $x$  has initial domain  $[0..5]$  and at some later stage  $D(x) = \{1, 3\}$  then the literals  $\llbracket x \leq 3 \rrbracket, \llbracket x \leq 4 \rrbracket, \neg \llbracket x \leq 0 \rrbracket, \neg \llbracket x = 0 \rrbracket, \neg \llbracket x = 2 \rrbracket, \neg \llbracket x = 4 \rrbracket, \neg \llbracket x = 5 \rrbracket$  will hold. Explanations are defined by clauses over this Boolean representation of the variables.

*Example 1.* Consider a simple constraint satisfaction problem with constraints  $b \leftrightarrow x + y \leq 2$ ,  $x + y \leq 2$ ,  $b' \leftrightarrow x \leq 1$ ,  $b \rightarrow b'$ , with initial domains  $D_{init}(b) = D_{init}(b') = \{0, 1\}$ , and  $D_{init}(x) = D_{init}(y) = \{0, 1, 2\}$ . There is no initial propagation. Setting  $x = 2$  makes the third constraint propagate  $D(b') = \{0\}$  with explanation  $\llbracket x = 2 \rrbracket \rightarrow \llbracket b' = 0 \rrbracket$ , this makes the last constraint propagate  $D(b) = \{0\}$  with explanation  $\llbracket b' = 0 \rrbracket \rightarrow \llbracket b = 0 \rrbracket$ . The first constraint propagates that  $D(y) = \{1, 2\}$  with explanation  $\llbracket b = 0 \rrbracket \rightarrow \llbracket y \geq 1 \rrbracket$  and the second constraint determines failure with explanation  $\llbracket x = 2 \rrbracket \wedge \llbracket y \geq 1 \rrbracket \rightarrow false$ . The graph of the implications is



Any cut separating the decision  $\llbracket x = 2 \rrbracket$  from *false* gives a nogood. The simplest one is  $\llbracket x = 2 \rrbracket \rightarrow false$  or equivalently  $\llbracket x \neq 2 \rrbracket$ .  $\square$

## 2.1 Edge-Valued Decision Diagrams

A *Multi-valued Decision Diagram (MDD)* encodes a propositional formula as a directed acyclic graph with a single terminal  $\mathcal{T}$  representing *true* (the *false* terminal is typically omitted for MDDs). In an MDD  $G$ , each internal node  $n = node(x_i, [(v_1, n_1), (v_2, n_2), \dots, (v_k, n_k)])$  is labelled with a variable  $x_i$ , and outgoing edges consisting of a value  $v_j$  and destination node  $n_j$ . Each node represents the formula

$$\langle n \rangle \Leftrightarrow \bigvee_{j=1}^k (x = v_j \wedge \langle n_j \rangle)$$

where  $\langle n \rangle$  is a Boolean representing the reachability of node  $n$ , and  $\langle \mathcal{T} \rangle = true$ . The MDD constraint enforces  $\langle G.root \rangle = true$  where  $G.root$  is the root of the MDD.

In this paper we restrict ourselves to *layered* MDDs. In a layered MDD  $G$  each node  $n$  is assigned to a layer  $k$  and all its child nodes must be at layer  $k + 1$ . Each node at layer  $k$  is labelled with the *same* variable  $x_k$ , and the root node  $G.root$  is at layer 1. This encodes an ordered MDD with no *long edges*, which typically propagate faster than MDDs with long edges [6]. Each assignment satisfying the constraint represented by  $G$  corresponds to a path from the root  $G.root$  to the terminal  $\mathcal{T}$ . If, at the  $i$ -th layer, the path follows an edge with value  $v_j$ , the corresponding assignment has  $x_i = v_j$ .

An *Edge-valued MDD (EVMDD)*  $G$  is a (layered) MDD with a weight attached to each edge. Hence nodes are of the form

$$n = node(x_i, [(v_1, w_1, n_1), (v_2, w_2, n_2), \dots, (v_k, w_k, n_k)]),$$

where  $w_j$  is the weight of the  $j^{th}$  outgoing edge. The cost of a solution  $\theta = [x_1 = d_1, x_2 = d_2, \dots, x_n = d_n]$  which defines a path from the root of  $G$  to  $\mathcal{T}$  is given



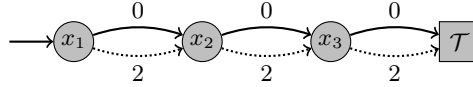


Fig. 1. A simple EVMDD with only paths of even cost

by the sum of the weights along the corresponding path in the EVMDD. Each node  $n$  enforces the constraint:

$$\langle\langle n \rangle\rangle = \begin{cases} 0 & n = \mathcal{T} \\ \min\{w_j + \langle\langle n_j \rangle\rangle \mid j = 1, \dots, k \wedge x_i = v_j\} & \text{otherwise} \end{cases}$$

where  $\langle\langle n \rangle\rangle$  holds the cost of the minimal weight path from  $n$  to  $\mathcal{T}$ .

For convenience, we denote edges by 4-tuples  $(n, x_i = v_j, w_j, n_j)$ , representing the edge with source  $n$  (in layer  $i$ ), destination  $n_j$  (in layer  $i + 1$ ) and weight  $w_j$  corresponding to the value  $v_j$ . We will refer to the components as  $(e.begin, e.var = e.val, e.weight, e.end)$ .

We use  $s.out\_edges$  to refer to all the edges of the form  $(s, -, -, -)$ , i.e. those leaving node  $s$ , and  $d.in\_edges$  to refer to edges of the form  $(-, -, -, d)$ , i.e. those entering node  $d$ . We use  $G.edges(x_i, v_j)$  to record the set of edges of the form  $(-, x_i = v_j, -, -)$  in EVMDD  $G$ .

The COST-MDD constraint  $COST-MDD(G, [x_1, \dots, x_n], \bowtie, C)$  requires that  $\langle\langle G.root \rangle\rangle \bowtie C$  where  $\bowtie \in \{\leq, =, \geq\}$ . Note that the constraint (except the  $\geq$  incarnation) enforces satisfiability, i.e., that there is a path from  $G.root$  to  $\mathcal{T}$ , since otherwise  $\langle\langle G.root \rangle\rangle = \infty$ . The COST-MDD constraint can represent COST-REGULAR as well as other constraints representable by automata with counters.

Our definition of EVMDDs differs from the standard treatment of edge-valued BDDs [10], apart from the extension from Boolean variables to finite-domain variables. We do not require the graph to be deterministic; a single node may have multiple edges annotated with the same value. Also, we do not require the edge weights to be normalized; normalization may reduce the size of the graph by inducing additional sharing, but does not affect propagation or explanation.

### 3 EVMDD Propagation

An incremental algorithm for propagating COST-REGULAR constraints was described in [3]. This algorithm essentially converts the COST-REGULAR constraint into a COST-MDD constraint where  $\bowtie$  is  $=$ , then performs propagation on this transformed representation. This algorithm operates by incrementally maintaining the distance of the shortest  $up[n]$  and longest  $lup[n]$  path from the root to each node  $n$ , and the distance of the shortest  $dn[n]$  and longest  $ldn[n]$  path from each node  $n$  to  $\mathcal{T}$ . Given a constraint  $COST-MDD(G, [x_1, \dots, x_n], =, C)$ , an edge  $e$  may be used to build a path from  $G.root$  to  $\mathcal{T}$  only if  $up[e.start] + e.weight + dn[e.end] \leq ub(C)$  and  $lup[e.start] + e.weight + ldn[e.end] \geq lb(C)$ .

The description in [3] does not mention how changes to the bounds of  $C$  are handled. When the upper bound of  $C$  is reduced, the lengths of all shortest

paths remain the same; however, the domains of variables  $x_i$  may change, if the shortest path through  $x_i = v_j$  is longer than the updated bound.

*Example 2.* Consider the EVMDD (EVBDD)  $G$  shown in Figure 1 where edges for value 0 are shown dotted, and edges for value 1 are shown full. The constraint  $\text{COST-MDD}(G, [x_1, x_2, x_3], =, C)$  encodes the equation  $2x_1 + 2x_2 + 2x_3 = C$ . If we initially have  $D(C) = [0..2]$ , no values may be eliminated, as every edge can occur on a path of cost at most 2. However, if we reduce  $\text{ub}(C)$  to 1, we must eliminate  $x_i = 1$  from the domain of each variable.

The authors claim that their propagation algorithm enforces domain consistency on the  $x$  variables in a COST-MDD constraint. This statement is not correct.

*Example 3.* Consider again the EVMDD  $G$  shown in Figure 1. The algorithm of [3] makes no propagation for the constraint  $\text{COST-MDD}(G, [x_1, x_2, x_3], =, C)$  when  $D(C) = \{3\}$ . This is because every edge can take part in a path which is both longer (length 4) or shorter (length 2) than the bounds of  $C$ . But there is no support for any value of  $x_i$  since there is no path of length exactly 3.  $\square$

In fact even bounds propagation is NP-hard for COST-MDD where  $\bowtie$  is  $=$ , using any applicable definition of bounds consistency [11].

**Theorem 1.** *Domain propagation,  $\text{bounds}(\mathcal{Z})$  or  $\text{bounds}(\mathcal{D})$  consistent propagation for  $\text{COST-MDD}(G, [x_1, \dots, x_n], =, C)$  is NP-hard*

*Proof.* We map SUBSETSUM to COST-MDD propagation. Given a set  $S = \{s_1, \dots, s_m\}$  of numbers and target  $T$  we build an EVBDD with  $m$  0-1 variables  $x_1, \dots, x_m$  and  $m$  nodes  $n_1, \dots, n_m$  ( $n_{m+1} = \mathcal{T}$ ) with  $2m$  edges ( $n_i, x_i = 0, 0, n_{i+1}$ ) and ( $n_i, x_i = 1, s_i, n_{i+1}$ ). Enforcing domain (or equivalently in this case  $\text{bounds}(\mathcal{Z})$  or  $\text{bounds}(\mathcal{D})$ ) consistency on  $\text{COST-MDD}(G, [x_1, \dots, x_n], =, C)$  with  $D(C) = \{T\}$  generates a false domain unless the SUBSETSUM holds.  $\square$

In this paper we restrict consideration to the COST-MDD constraint of the form  $\text{COST-MDD}(G, [x_1, \dots, x_n], \leq, C)$ . This is the critical form of the constraint when we are trying to minimize costs. Treatment of  $\text{COST-MDD}(G, [x_1, \dots, x_n], \geq, C)$  is identical by negating each edge weight and the cost variable; the treatment of  $\text{COST-MDD}(G, [x_1, \dots, x_n], =, C)$  in [3] is effectively combining propagators for each of  $\text{COST-MDD}(G, [x_1, \dots, x_n], \leq, C)$  and  $\text{COST-MDD}(G, [x_1, \dots, x_n], \geq, C)$ .

We give a non-incremental propagation algorithm for the constraint  $\text{COST-MDD}(G, [x_1, \dots, x_n], \leq, C)$  in Figure 2.<sup>1</sup> `evmdd.prop` first records the shortest path (given the current domain  $D$ ) from each node  $n$  to  $\mathcal{T}$  in  $dn[n]$  using `mark_paths`. It returns the shortest path from  $G.\text{root}$  to  $\mathcal{T}$ . It then visits using `infer` all the edges reachable from  $G.\text{root}$  that appear on paths of length less than  $\text{ub}(C)$ . Initially the negation of all edge labels are placed in *inferences*. When an edge that appears on a path of length less than or equal to  $\text{ub}(C)$  is discovered, the negation of its label is removed from *inferences*. The algorithm returns the a lower bound of  $C$  (which may not be new) and any new inferences on  $x_i$  variables.

<sup>1</sup> This is not novel with respect to [3] but they don't formally define their algorithm.

*Example 4.* Consider the propagation that occurs with the EVMDD of Figure 1 with  $C \leq 2$  when we set  $x_1 \neq 1$  ( $x_1 = 0$ ) and  $x_2 \neq 0$  ( $x_2 = 1$ ). `mark_paths` sets  $dn[\mathcal{T}] = 0$ ,  $dn[x_3] = 0$  (using the variable name for the node name),  $dn[x_2] = 2$  and  $dn[x_1] = 2$  and returns 2. `infer` initially starts with  $inferences = \{\llbracket x_1 \neq 0 \rrbracket, \llbracket x_1 \neq 1 \rrbracket, \llbracket x_3 \neq 0 \rrbracket, \llbracket x_3 \neq 1 \rrbracket\}$ . It sets  $up[x_1] = 0$  then removes  $\llbracket x_1 \neq 0 \rrbracket$  from  $inferences$  setting  $up[x_2] = 0$ . It then removes  $\llbracket x_2 \neq 2 \rrbracket$  from  $inferences$  setting  $up[x_3] = 2$ . It removes  $\llbracket x_3 \neq 0 \rrbracket$  from  $inferences$ , but then when examining the full edge from  $x_3$  the distance test fails. Hence it returns  $\{\llbracket x_3 \neq 1 \rrbracket\}$ . The final inferences are  $\{\llbracket C \geq 2 \rrbracket, \llbracket x_3 \neq 1 \rrbracket\}$ .

**Proposition 1.** *evmdd\_prop maintains domain consistency for  $\text{COST-MDD}(G, [x_1, \dots, x_n], \leq, C)$ .*

*Proof.* After `evmdd_prop` finishes if  $v_j \in D(x_i)$  then there is an edge  $(s, x_i = v_j, w, d)$  in  $G$  where  $up[s] + w + dn[d] \leq \text{ub}(C)$ . Hence there is a path of edges from  $G.\text{root}$  to  $s$  of length  $up[s]$  and a path of edges from  $d$  to  $\mathcal{T}$  of length  $dn[d]$ . If we set each variable to the value given on this path and  $C = \text{ub}(C)$  we have constructed a solution supporting  $x_i = v_j$ . Similarly, given  $l = \text{lb}(C)$  then after `evmdd_prop` finishes there is a path from  $G.\text{root}$  to  $\mathcal{T}$  of length  $l$ . If we set each variable to the value given on this path, and  $C$  to any value  $d \in D(C)$  domain we have constructed a solution supporting  $C = d$ .  $\square$

It is straightforward to make the above algorithm incremental in changes in  $x$  variables. A removed edge  $e = (s, x = v, w, d)$  forces the recalculation of  $dn[s]$  which may propagate upward, and  $up[s]$  which may propagate downwards. If a change reaches  $G.\text{root}$  or  $\mathcal{T}$  then the lower bound on  $C$  may change. When the upper bound of  $C$  changes, we simply scan the edges for each value until we find one that is still feasible (infeasible edges are not checked on later calls).

## 4 Explaining EVMDD Propagation

A nogood learning solver, upon reaching a conflict, analyses the inference graph to determine some subset of assignments that results in a conflict. This subset is then added to the solver as a *nogood* constraint, preventing the solver from making the same set of assignments again, and reducing the search space. In order to be incorporated in a nogood learning solver, the EVMDD propagator must be able to explain its inferences.

### 4.1 Minimal Explanation

The explanation algorithm is similar in concept to that used for BDDs and MDDs. To explain  $\llbracket x \neq v \rrbracket$  we assume  $\llbracket x = v \rrbracket$  and hence make the EVMDD unsatisfiable. A correct explanation is (the negation of) all the values for other variables which are currently false. We then progressively remove assignments (unfix literals) from this explanation while ensuring the constraint as a whole remains unsatisfiable. We are guaranteed to create a *minimal explanation* (but

```

evmdd_prop( $G, [x_1, \dots, x_n], C, D$ )
   $\hat{c} := \text{mark\_paths}(G, D)$ 
   $L := \text{infer}(G, [x_1, \dots, x_n], D, \text{ub}(C))$ 
  return  $\{\llbracket C \geq \hat{c} \rrbracket\} \cup L$ 

mark_paths( $G, D$ )
  for ( $n \in G.\text{nodes}$ )  $dn[n] := \infty$ 
   $dn[\mathcal{T}] := 0$ ;  $queue := \{\mathcal{T}\}$ 
  while ( $queue \neq \emptyset$ )
     $nqueue := \{\}$  % Record nodes of interest on the next level.
    for ( $node$  in  $queue$ )
      for ( $e$  in  $node.\text{in\_edges}$ )
        if ( $e.\text{val} \in D(e.\text{var})$ )
           $dn[e.\text{begin}] := \min(dn[s.\text{begin}], e.\text{weight} + dn[node])$ 
           $nqueue \cup = \{e.\text{begin}\}$ 
     $queue := nqueue$ 
  return  $dn[G.\text{root}]$ 

infer( $G, [x_1, \dots, x_n], D, u$ )
   $\text{inferences} := \{\llbracket x_i \neq v_j \rrbracket \mid 1 \leq i \leq n, v_j \in D(x_i)\}$ 
  for ( $n \in G.\text{nodes}$ )  $up[n] := \infty$ 
   $up[G.\text{root}] := 0$ ;  $queue := \{G.\text{root}\}$ 
  while ( $queue \neq \emptyset$ )
     $nqueue := \{\}$  % Record nodes of interest on the next level.
    for ( $node$  in  $queue$ )
      for ( $e \in node.\text{out\_edges}$ )
        if ( $e.\text{val} \in D(e.\text{var})$ )
          if ( $up[node] + e.\text{weight} + dn[e.\text{end}] \leq u$ )
             $\text{inferences} := \text{inferences} - \{\llbracket e.\text{var} \neq e.\text{val} \rrbracket\}$ 
             $up[e.\text{end}] := \min(up[e.\text{end}], e.\text{weight} + up[node])$ 
             $nqueue \cup = \{e.\text{end}\}$ 
     $queue := nqueue$ 
  return  $\text{inferences}$ 

```

**Fig. 2.** Algorithm for inferring newly propagated literals

not the smallest minimal explanation)  $\bigwedge_{l \in \text{expln}} l \rightarrow \llbracket x \neq v \rrbracket$  since removing any literal  $l'$  from the  $\text{expln}$  would mean  $\text{COST-MDD}(G, [x_1, \dots, x_n], \leq, C) \wedge \bigwedge_{l \in \text{expln} - \{l'\}} l \wedge \llbracket x = v \rrbracket$  is satisfiable. Constructing a smallest minimal explanation for an EVMDD is NP-hard just as for BDDs [12].

We adapt the minimal MDD explanation algorithm used in [6] to COST-MDD constraints. The propagator conflicts when the shortest path from  $G.\text{root}$  to  $\mathcal{T}$  (under the current domain) is longer than  $\text{ub}(C)$ . To construct a minimal explanation, we begin with the set of values that have been removed from variable domains, and progressively restore any values which would not re-introduce a path of length  $\leq \text{ub}(C)$ .

The minimal explanation algorithm is illustrated in Figure 3. To explain  $\llbracket x \neq v \rrbracket$  under current domain  $D$ , we first create the domain  $D'$  where  $D'(x) =$

```

evmdd_explain( $G, C, D, \llbracket x \neq v \rrbracket$ )
   $D' := D$  with  $D(x)$  replaced by  $D'(x) = \{v\}$ 
   $\hat{c} := \text{mark\_paths}(G, D')$ 
  if ( $\hat{c} < \infty$  or choice)  $u := \text{ub}(C) + 1$ 
  else  $u := \infty$ 
  return  $\llbracket x \neq v \rrbracket \leftarrow \text{collect\_expln}(G, C, x, v, u)$ 

evmdd_explain_lb( $G, C, D, \llbracket C \geq l \rrbracket$ )
   $\text{mark\_paths}(G, D)$  % unnecessary if just run evmdd_prop
  return  $\llbracket C \geq l \rrbracket \leftarrow \text{collect\_expln}(G, C, \perp, \perp, l) - \{\llbracket C \leq l - 1 \rrbracket\}$ 

collect_expln( $G, C, x, v, u$ )
   $\text{queue} := \{G.\text{root}\}; \text{up}[G.\text{root}] := 0; s := \infty$ 
  while ( $\text{queue} \neq \emptyset$ )
    for ( $\text{node}$  in  $\text{queue}$ )
      for ( $e \in \text{node.out\_edges}$ )
         $\text{up}[e.\text{end}] := \infty$ 
        if ( $e.\text{var} \neq x$  and  $\text{up}[\text{node}] + e.\text{weight} + \text{dn}[e.\text{end}] < u$ )
           $\text{explanation} \cup= \llbracket e.\text{var} \neq e.\text{val} \rrbracket$ 
        else  $s := \min(s, \text{up}[\text{node}] + e.\text{weight} + \text{dn}[e.\text{end}])$ 
       $n\text{queue} := \{\}$  % Record nodes of interest on the next level.
      for ( $\text{node}$  in  $\text{queue}$ )
        for ( $e \in \text{node.out\_edges}$ )
          if (( $e.\text{var} = x$  and  $e.\text{val} = v$ )
              or ( $e.\text{var} \neq x$  and  $\llbracket e.\text{var} \neq e.\text{val} \rrbracket \notin \text{explanation}$ ))
             $n\text{queue} \cup= \{e.\text{end}\}$ 
           $\text{up}[e.\text{end}] := \min(\text{up}[e.\text{end}], \text{up}[\text{node}] + e.\text{weight})$ 
       $\text{queue} := n\text{queue}$ 
  return  $\text{explanation} \cup \llbracket C \leq s - 1 \rrbracket$ 

```

**Fig. 3.** Algorithms for computing a minimal explanation

$\{v\}$  and otherwise  $D'$  agrees with  $D$ . With this domain the constraint is unsatisfiable. We use `mark_paths` to compute the shortest path from each node  $n$  to  $\mathcal{T}$  and store this in  $\text{dn}[n]$ . It returns the shortest path  $\hat{c}$  from root to  $\mathcal{T}$ . If  $\hat{c}$  is finite, or we choose to (by setting global *choice* true) we use an upper bound of  $C$  in the explanation, by setting  $u \neq \infty$ . `collect_expln` traverses the EVMDD from the root, building an explanation of literals which if not true would cause a path of length  $< u$  to be created in the EVMDD. The algorithm examines all reachable nodes on a level (initially just the root) and if adding an edge would create a path shorter than  $u$  then the (negation of) the label on the edge is added to the explanation, if not then we update  $s$  which records the shortest path found from root to  $\mathcal{T}$  with length  $\geq u$ . The algorithm then adds all the nodes of the next level which are still reachable, and updates the shortest path from the root to each such node  $n$  storing this in  $\text{up}[n]$ . This continues while there are still some reachable nodes. At the end the algorithm returns the collected explanation, plus the relaxed upper bound literal  $\llbracket C \leq s - 1 \rrbracket$ , which ensures that none of the paths found from root to  $\mathcal{T}$  can be traversed.

Since the procedures `mark_paths` and `collect_expln` perform one and two breadth-first traversals of the graph, respectively, the explanation requires  $O(|G|)$  time.

**Proposition 2.** *`evmdd_explain`( $G, C, D, \llbracket x \neq v \rrbracket$ ) returns a correct minimal explanation for  $\llbracket x \neq v \rrbracket$ .*

*Proof. (Sketch)* The algorithm implicitly maintains the invariant that there is no path in  $G$  through an edge labelled  $x = v$  of length less than or equal to  $\text{lb}(C)$  which does not make use of an edge in  $DE$ . Initially  $DE$  is the set of edges  $e$  where  $e.\text{var} \neq x$  and  $e.\text{val} \notin D(e.\text{var})$ . The base case holds using the correctness of `evmdd_prop`. During `collect_expl` we remove processed edges from this implicit set  $DE$ , except those kept in explanation. Whenever we remove an edge from  $DE$  the shortest path through the edge that uses an edge labeled  $x = v$  and none of the edges in  $DE$  is  $> \text{ub}(C)$ . This demonstrates the correctness of the algorithm, since the explanation literals force that  $\llbracket x \neq v \rrbracket$  holds since there is no feasible path through any edges labelled  $x = v$ .

For minimality we can reason that if we remove any literal from the explanation, then we would have added a path that was too short passing through an edge labelled  $x = v$ . The minimality of the bound constraint  $\llbracket C \leq s - 1 \rrbracket$  follows since if we relax it we will allow a path of length  $s$  through  $x = v$ .  $\square$

Explaining a new lower bound  $l$  for  $C$  is similarly defined by `evmdd_explain_lb`. We compute  $dn[n]$  for each reachable node using `mark_paths` with the current domain  $D$ , then choose a set of literals to ensure no shorter paths are allowed. In this case `collect_expln` will always return  $\llbracket C \leq l - 1 \rrbracket$  in the explanation which we can safely omit. Explaining failure of the whole constraint is identical to explaining why  $C \geq \infty$ .

*Example 5.* Consider the constraint defined by the EVMDD shown in Figure 4, which encodes a simple scheduling constraint requiring shifts to be of even length.

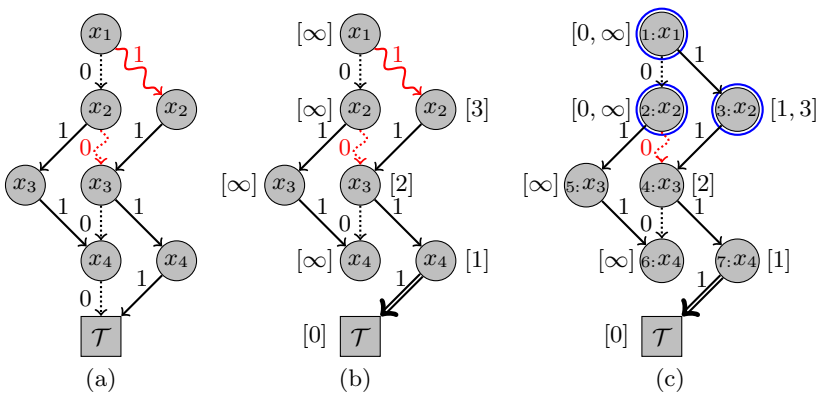


**Fig. 4.** (a) An EVMDD which requires shifts to be assigned in blocks of two. (b) We compute the shortest path from each node to  $\mathcal{T}$ . (c) Enqueued nodes are shown circled, and have been annotated with the shortest path from  $n_1$  under the current assignment.

Assume the solver first propagates  $C \leq 2$ , then fixes  $\llbracket x_1 \neq 1 \rrbracket$  and  $\llbracket x_2 \neq 0 \rrbracket$ . The only satisfying assignment is then  $[x_1, x_2, x_3, x_4] = [0, 1, 1, 0]$ .

If we are asked to explain the inference  $x_4 \neq 1$ , we first compute the shortest paths from each node to  $\mathcal{T}$  through  $x_4 = 1$ , using `mark_paths`. This is shown in Figure 4(b). Notice that the cost at the root node is  $\infty$ . This indicates that, even without a cost bound, there is no feasible path through  $x_4 = 1$ . We have the choice of either omitting the cost bound (obtaining an explanation not dependent on  $C$ ) or including it and possibly obtaining a smaller explanation.

Whether or not we include a bound on  $C$ , we proceed by sweeping down level-by-level from the  $r_1$ . Assuming we include the bound  $\llbracket C \leq 2 \rrbracket$ , so  $u = 3$ , we first check if any of the outgoing edges would introduce a path of length less than 3. We find that the edge from  $n_1$  to  $n_3$  can safely be restored, since  $up[n_1] + 1 + dn[n_3] = 4 \geq u = 3$ . We update  $s = 4$ . As no edges introduce a feasible path, we update  $up$  for both  $n_2$  and  $n_3$ , and add them to the queue for the next level.

At the second level, we discover that restoring the edge from  $n_2$  to  $n_4$  would introduce a feasible path, as  $up[n_2] + 0 + dn[n_4] = 2 < u = 3$ . The literal  $\llbracket x_2 \neq 0 \rrbracket$  must then be added to the explanation. Since  $n_4$  is still reachable via  $n_3$ , both  $n_4$  and  $n_5$  are added to the queue for the next level; however,  $up[n_4]$  is only updated by the edge from  $n_3$ , and not from  $n_2$ . This process continues until no further nodes remain. At the end  $s = 4$  so we didn't need the bound on paths to be  $ub(C)$  it could have been looser. Hence we add  $\llbracket C \leq 3 \rrbracket$  to the explanation. The explanation returned is  $\llbracket x_4 \neq 1 \rrbracket \leftarrow \llbracket C \leq 3 \rrbracket \wedge \llbracket x_2 \neq 0 \rrbracket$ . This is a minimal explanation.

If we omit the cost bound, then we cannot restore the edge from  $n_1$  to  $n_3$ ; so we construct the alternate explanation  $\llbracket x_4 \neq 1 \rrbracket \leftarrow \llbracket x_1 \neq 1 \rrbracket \wedge \llbracket x_2 \neq 0 \rrbracket$  which is also minimal. Note we omit the redundant literal  $\llbracket C \leq \infty - 1 \rrbracket$  created by `collect_expl`.  $\square$

## 4.2 Incremental Explanation

*Example 6.* Unfortunately, on large EVMDDs, constructing a minimal explanation can be expensive since explaining each inference may involve exploring the entire EVMDD. For these cases, we present a greedy algorithm for constructing valid, but not necessarily minimal, explanations in an incremental manner, often only examining a small part of the EVMDD.

We adapt the incremental MDD algorithm of [6] to `COST-MDD`. As in the MDD case, we explain  $\llbracket x \neq v \rrbracket$  beginning from the set of edges corresponding to  $x = v$ . For all such edges  $e = (s, x = v, w, d)$ , we know that  $up[s] + w + dn[d] > ub(C)$ . If we have  $up[s] + w + dn[d] = ub(C) + 1$ , then there is no flexibility in the bounds; we must select an explanation which ensures the shortest path from  $G.root$  to  $s$  has cost  $up[s]$ , and the shortest path from  $e$  to  $\mathcal{T}$  has cost  $dn[d]$ . We record the amount of cost that needs to be explained on all paths to  $s$ ; this is denoted by  $upe[s]$ . We then sweep upwards, level-by-level, collecting an explanation which guarantees this minimum cost. At each level, we maintain the set of edges which need to be explained. If for some edge we have  $up[s] + w < upe[d]$ , then  $\llbracket x \neq v \rrbracket$

```

mdd_inc_explain( $G, x, v, u$ )
  for( $n \in G.nodes$ )  $upe[n] := \infty$ 
  for( $n \in G.nodes$ )  $dne[n] := \infty$ 
   $kfa := \{\}$  % edges killed from above
   $kfb := \{\}$  % edges killed from below
  for( $e$  in  $G.edges(x, v)$ )
    % Split possible supports
    Assign  $p_{up}, p_{dn}$  subject to:
       $p_{up} + e.weight + p_{dn} \geq u \wedge p_{up} \leq up[e.begin] \wedge p_{dn} \leq dn[e.end]$ 
    if( $p_{up} > up_0[e.begin]$ )
       $kfa \cup = \{edge\}$ 
       $upe[e.begin] := \max(upe[e.begin], p_{up})$ 
    if( $p_{dn} > dn_0[e.end]$ )
       $kfb \cup = \{edge\}$ 
       $dne[e.end] := \max(dne[e.end], p_{dn})$ 
  % Explain all those killed from below
  return explain_down( $kfb$ )
  % And all those killed from above
   $\cup$  explain_up( $kfa$ )

```

Fig. 5. Top-level wrapper for incremental explanation

must be added to the current explanation; otherwise, a feasible path would be introduced. We perform an initial pass over the edges at the current level to determine which values must be included in the explanation; during the second pass, we update  $upe$  for the source node of each edge that hasn't been excluded, and enqueue the set of incoming edges to be processed at the next level. If at any point we have  $upe[s]$  is no greater than  $up_0[s]$  (the shortest path to  $s$  under the initial variable domains), then we don't need to enqueue the incoming edges, as an empty explanation is sufficient.

If  $up[s] + w + dn[d] > ub(C) + 1$ , then we can potentially relax the generated explanation. Obviously, the amount by which we relax  $up[s]$  affects the amount of slack available to  $dn[d]$ . To relax the bounds as far as possible, we would initially allocate as much slack as possible to  $up[s]$ , and collect the corresponding explanation. Before performing the downward pass, we would then propagate the newly reduced path lengths back to the current layer, to determine how much slack remains for the explanation of  $d$ .

Instead, we determine *a priori* how the slack is allocated in the explanation. If either  $up[s]$  or  $dn[d]$  is  $\infty$ , then we build the explanation in only that direction (if both, we arbitrarily explain upwards). Otherwise, we explain as much as possible in the upward pass, and allocate all possible slack to the downwards pass. Alternative strategies for relaxing the bounds is interesting future work.

Consider again the case described in Example 5. During incremental propagation, we maintain  $up$  and  $dn$  for each node. These are shown in Figure 7(a). To explain  $\llbracket x_4 \neq 1 \rrbracket$ , we need to eliminate some set of values which ensures that  $up[n_7] + 1 + dn[\mathcal{T}] \geq 3$ .



```

explain_down(kfb)
  reason = {}
  % Traverse the MDD downwards, breadth first
  while( $\neg$ is_empty(kfb))
    % Scan the current level for edges that will need explaining.
    pending = {}
    for(e in kfb)
      % For each edge requiring explanation
      if(e.val  $\notin$  D(e.var) and
        e.weight + dn[e.end] < dne[e.begin])
        % There is no later explanation,
        % so add  $\llbracket e.var \neq e.val \rrbracket$  to the reason.
        reason  $\cup$ = { $\llbracket e.var \neq e.val \rrbracket$ }
      else
        pending  $\cup$ = {e}
    next = {}
    % Collect the edges that haven't been explained at this level.
    for(e in pending)
      if( $\llbracket e.var \neq e.val \rrbracket \notin$  reason and e.weight + dn0[e.end] < dne[e.begin])
        % If e is not explained already collect its outgoing edges
        next  $\cup$ = e.end.out_edges
        dne[e.end] := max(dne[e.end], dne[e.begin] - e.weight)
      % Continue with the next layer of edges.
    kfb = next
  return reason

```

**Fig. 6.** Pseudo-code for incremental explanation of EVMDDs. *explain\_up* acts in exactly the same fashion as *explain\_down*, but in the opposite direction.

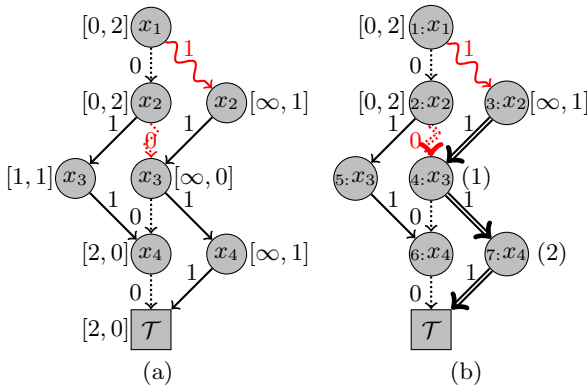


**Fig. 7.** The EVMDD from Example 5. (a) Values of  $[up, dn]$  for each node. (b) Edges enqueued while explaining  $\llbracket x_4 \neq 1 \rrbracket$ .

Under the current assignment,  $up[n_7] = \infty$ . However, as our current cost bound is 2, we only need to ensure  $up[n_7] + 1 \geq 3$ . We set  $upe[n_7] = 2$ , the amount of cost that must be guaranteed from above, and add  $n_7$  to the queue. Expanding  $n_7$ , we find that it has only one parent, which is the edge from  $n_4$  with weight 1. This edge cannot be eliminated, so we set  $upe[n_4] = upe[n_7] - 1 = 1$ , and enqueue  $n_4$ .

$n_4$  has 2 incoming edges, so we first check both edges to determine if any values must be added to the explanation. Examining the edge from  $n_2$  to  $n_4$ , we have  $upe[n_4] - 0 = 1 > up[n_2]$ . This indicates that, if the edge from  $n_2$  to  $n_4$  were restored, a path of length 2 would be introduced.  $\llbracket x_2 \neq 0 \rrbracket$  is therefore added to the explanation. The edge from  $n_3$  to  $n_4$  is safe, as  $up[n_3] = \infty \geq upe[n_3] - 1$ .

We then make a second pass through the edges, to determine which new nodes must be enqueued. As  $\llbracket x_2 \neq 0 \rrbracket$  is in the explanation, we don't need to expand the node from  $n_2$  to  $n_4$ . The edge from  $n_3$  to  $n_4$  is traversable, so we update  $upe[n_3]$ . However, since  $upe[n_3] = 0$ , and the base cost to reach  $n_3$  is 1 (that is,  $upe[n_3] \leq up_0[n_3]$ ), we don't need to enqueue  $n_3$ , since the cost to  $n_3$  will always be at least 0. Since we have no nodes enqueued, the upwards pass is finished. Since there are no nodes which must be propagated downwards, this yields the final explanation  $\llbracket x_4 \neq 1 \rrbracket \leftarrow \llbracket C \leq 2 \rrbracket \wedge \llbracket x_2 \neq 0 \rrbracket$ .

Observe that this is not minimal, since the explanation is still valid if we replace  $\llbracket C \leq 2 \rrbracket$  with  $\llbracket C \leq 3 \rrbracket$ .  $\square$

## 5 Experimental Results

Experiments were conducted on a 3.00GHz Core2 Duo with 4 Gb of RAM running Ubuntu GNU/Linux 10.04. The propagators were implemented in **chuffed**, a state-of-the-art lazy-clause generation [8] based constraint solver. All experiments were run with a 10 minute time limit. For the minimal explanation algorithm, we always selected to use upper bounds in the explanation if possible.

We evaluate the COST-MDD constraints on a standard set of shift scheduling benchmarks. For the experiments, **dec** denotes propagation using a decomposition of COST-MDD like that of [13] but introducing a cost variable per layer of the EVMDD and summing them to compute cost, and **dec<sub>mdd</sub>** uses the domain-consistent Boolean decomposition described in [14] (or equivalently in [6]) and a separate cost constraint. **mdd** denotes using a separate MDD propagator [6] and cost constraint, **ev-mdd** denotes COST-MDD using incremental propagation and minimal explanations, **ev-mdd<sub>I</sub>** denotes COST-MDD using incremental propagation and greedy explanations. We also tried a domain consistent decomposition of COST-MDD based on [7] but it failed to solve any of the shown instances, and is omitted.

### 5.1 Shift Scheduling

*Shift scheduling*, a problem introduced in [3], allocates  $n$  workers to shifts such that (a) each of  $k$  activities has a minimum number of workers scheduled at

**Table 1.** Comparison of different methods on shift scheduling problems

| Inst.  | dec    |        | dec <sub>mdd</sub> |        | mdd    |           | ev-mdd        |          | ev-mdd <sub>l</sub> |          |
|--------|--------|--------|--------------------|--------|--------|-----------|---------------|----------|---------------------|----------|
|        | time   | fails  | time               | fails  | time   | fails     | time          | fails    | time                | fails    |
| 1,2,4  | —      | —      | 14.51              | 39700  | 3.49   | 21888     | 0.20          | 607      | <b>0.17</b>         | 635      |
| 1,3,6  | —      | —      | 11.25              | 40675  | 19.00  | 76348     | <b>0.87</b>   | 4045     | 0.91                | 4156     |
| 1,4,6  | 36.48  | 86762  | 2.62               | 7582   | 0.69   | 3518      | <b>0.11</b>   | 350      | 0.27                | 1077     |
| 1,5,5  | 5.64   | 32817  | 0.41               | 1585   | 0.52   | 3955      | 0.07          | 239      | <b>0.06</b>         | 238      |
| 1,6,6  | 7.32   | 35064  | 0.40               | 1412   | 0.21   | 1161      | <b>0.08</b>   | 249      | 0.11                | 413      |
| 1,7,8  | 27.58  | 77757  | 4.03               | 13149  | 2.43   | 12046     | <b>0.73</b>   | 3838     | 0.83                | 4279     |
| 1,8,3  | 67.74  | 126779 | 0.85               | 5002   | 0.39   | 3606      | <b>0.06</b>   | 219      | 0.07                | 262      |
| 1,10,9 | 321.44 | 441884 | 17.55              | 44222  | 19.77  | 68688     | <b>1.23</b>   | 5046     | 1.31                | 7419     |
| 2,1,5  | 1.29   | 12520  | 0.14               | 691    | 0.24   | 1490      | 0.02          | 78       | <b>0.01</b>         | 45       |
| 2,2,10 | —      | —      | —                  | —      | 131.29 | 286747    | <b>43.62</b>  | 99583    | 49.05               | 100958   |
| 2,3,6  | —      | —      | 188.77             | 187760 | 144.99 | 289568    | <b>2.39</b>   | 6443     | 5.94                | 13695    |
| 2,4,11 | —      | —      | —                  | —      | 391.59 | 918438    | <b>42.38</b>  | 111567   | 92.89               | 220568   |
| 2,5,4  | —      | —      | 25.85              | 59635  | 12.18  | 50340     | 0.65          | 1545     | <b>0.48</b>         | 1541     |
| 2,6,5  | —      | —      | 83.78              | 104911 | 30.27  | 80046     | <b>6.18</b>   | 12100    | 7.63                | 16074    |
| 2,8,5  | —      | —      | 90.28              | 153331 | 34.69  | 110917    | <b>4.99</b>   | 15507    | 10.02               | 26565    |
| 2,9,3  | —      | —      | 6.10               | 20472  | 9.17   | 42105     | 0.86          | 1898     | <b>0.47</b>         | 1593     |
| 2,10,8 | —      | —      | 349.88             | 303227 | 95.61  | 168720    | <b>8.85</b>   | 26331    | 17.22               | 37356    |
| Total  | —      | —      | —                  | —      | 896.53 | 2139581   | <b>113.29</b> | 289645   | 187.44              | 436874   |
| Mean   | —      | —      | —                  | —      | 52.74  | 125857.71 | <b>6.66</b>   | 17037.94 | 11.03               | 25698.47 |
| Geom.  | —      | —      | —                  | —      | 7.92   | 31465.82  | <b>0.86</b>   | 2667.65  | 1.03                | 3428.35  |

any given time, and (b) the overall cost of the schedule is minimised, without violating any of the additional constraints:

- An employee must work on a task ( $A_i$ ) for at least one hour, and cannot switch tasks without a break ( $b$ ).
- A part-time employee ( $P$ ) must work between 3 and 5.75 hours, plus a 15 minute break.
- A full-time employee ( $F$ ) must work between 6 and 8 hours, plus 1 hour for lunch ( $L$ ), and 15 minute breaks before and after.
- An employee can only be rostered while the business is open.

These constraints can be formulated as a **grammar** constraint as follows:

$$\begin{aligned}
 S &\rightarrow RP^{[13,24]}R \mid RF^{[30,38]}R \\
 F &\rightarrow PLP \quad P \rightarrow WbW \\
 W &\rightarrow A_i^{[4,\dots]} \quad A_i \rightarrow a_i A_i \mid a_i \\
 L &\rightarrow lll \quad R \rightarrow rR \mid r
 \end{aligned}$$

We convert the **grammar** constraint into a Boolean formula, as described in [13]; however, we convert the formula directly into an MDD, rather than a **s-DNNF** circuit; the MDD and cost-MDD propagators, as well as the decompositions, are all constructed from this MDD. This process is similar to the reformulation described in [15]. Note that some of the productions for  $P$ ,  $F$  and  $A_i$  are annotated with restricted intervals – while this is no longer strictly

context-free, it can be integrated into the graph construction with no additional cost.

The coverage constraints and objective function are implemented using the monotone BDD decomposition described in [16].

The model using MDD is substantially better than the COST-MDD decomposition, and also superior to the MDD decomposition. It already improves upon the best published CP/SAT models for these problems<sup>2</sup> in [15]. The results for `ev-mdd` show that modelling the problem using COST-MDD is substantially better than separately modelling cost and an MDD constraint. Incremental greedy explanation can improve on minimal explanations, but the results demonstrate that minimal explanations are preferable. This contrasts with results for explaining MDD [6] where greedy incremental explanations were almost always superior. This may be because the presence of path costs in EVMDDs means that decisions higher in the graph have a greater impact on explanations further down (whereas for MDDs, the explanation only changes if a node is rendered completely unreachable).

## 6 Conclusion

In this paper we have defined how to explain the propagation of an EVMDD. Interestingly we have a trade-off between using cost bounds or literals on  $x$  to explain the same propagation. We define non-incremental minimal and incremental non-minimal explanation algorithms for EVMDDs. Using EVMDD with explanation to define a COST-MDD constraint, we are able to substantially improve on other modelling approaches for solving problems with COST-MDD with explanation.

**Acknowledgments.** NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

## References

1. Focacci, F., Lodi, A., Milano, M.: Cost-based domain filtering. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 189–203. Springer, Heidelberg (1999)
2. Régin, J.-C.: Arc consistency for global cardinality constraints with costs. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 390–404. Springer, Heidelberg (1999)
3. Demassey, S., Pesant, G., Rousseau, L.-M.: A cost-regular based hybrid column generation approach. *Constraints* 11(4), 315–333 (2006)
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)

---

<sup>2</sup> The best results for these problems use dynamic programming as a column generator in a branch-and-price solution [17].

5. Gange, G., Stuckey, P., Lagoon, V.: Fast set bounds propagation using a BDD-SAT hybrid. *Journal of Artificial Intelligence Research* 38, 307–338 (2010)
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.: The weighted grammar constraint. *Annals OR* 184(1), 179–207 (2011)
8. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. *Constraints* 14(3), 357–391 (2009)
9. Schulte, C., Stuckey, P.: Efficient constraint propagation engines. *ACM Transactions on Programming Languages and Systems* 31(1), Article No. 2 (2008)
10. Vrudhula, S.B., Pedram, M., Lai, Y.-T.: Edge valued binary decision diagrams. In: *Representations of Discrete Functions*, pp. 109–132. Springer (1996)
11. Choi, C.W., Harvey, W., Lee, J.H.M., Stuckey, P.J.: Finite domain bounds consistency revisited. In: Sattar, A., Kang, B.-H. (eds.) *AI 2006. LNCS (LNAI)*, vol. 4304, pp. 49–58. Springer, Heidelberg (2006)
12. Subbarayan, S.: Efficient reasoning for nogoods in constraint solvers with bDDs. In: Hudak, P., Warren, D.S. (eds.) *PADL 2008. LNCS*, vol. 4902, pp. 53–67. Springer, Heidelberg (2008)
13. Quimper, C.G., Walsh, T.: Global grammar constraints. In: Benhamou, F. (ed.) *CP 2006. LNCS*, vol. 4204, pp. 751–755. Springer, Heidelberg (2006)
14. Jung, J.C., Barahona, P., Katsirelos, G., Walsh, T.: Two encodings of DNNF theories. In: *ECAI Workshop on Inference Methods Based on Graphical Structures of Knowledge* (2008)
15. Katsirelos, G., Narodytska, N., Walsh, T.: Reformulating global grammar constraints. In: van Hoeve, W.-J., Hooker, J.N. (eds.) *CPAIOR 2009. LNCS*, vol. 5547, pp. 132–147. Springer, Heidelberg (2009)
16. Abío, I., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: BDDs for pseudo-boolean constraints – revisited. In: Sakallah, K.A., Simon, L. (eds.) *SAT 2011. LNCS*, vol. 6695, pp. 61–75. Springer, Heidelberg (2011)
17. Côté, M.C., Gendron, B., Rousseau, L.M.: Grammar-based integer programming models for multiactivity shift scheduling. *Management Science* 57(1), 151–163 (2011)

# A Simple and Effective Decomposition for the Multidimensional Binpacking Constraint

Stefano Gualandi<sup>1</sup> and Michele Lombardi<sup>2</sup>

<sup>1</sup> Università di Pavia, Dipartimento di Matematica

<sup>2</sup> Università di Bologna, Dipartimento di Informatica: Scienza ed Ingegneria  
 stefano.gualandi@unipv.it, michele.lombardi2@unibo.it

**Abstract.** The **multibin\_packing** constraint captures a fundamental substructure of many assignment problems, where a set of items, each with a fixed number of dimensions, must be assigned to a number of bins with limited capacities. In this work we propose a simple decomposition for **multibin\_packing** that uses a **bin\_packing** constraint for each dimension, a set of **all\_different** constraints automatically derived from a conflict graph, plus two alternative symmetry breaking approaches. Despite its simplicity, the proposed decomposition is very effective on a number of instances recently proposed in the literature.

## 1 Introduction

Given a set  $I = \{1, \dots, n\}$  of items and a set  $K = \{1, \dots, k\}$  of dimensions, where each item  $i$  has a weight  $w_{i,l}$  for every dimension  $l$ , and given a set  $B = \{1, \dots, m\}$  of bins  $j$  with a capacity  $c_{j,l}$  for every dimension  $l$ , the **multibin\_packing** constraint states that every item must be packed into a single bin while the sum of weights for each bin and for each dimension cannot exceed the corresponding bin capacity. In particular, we extend the formulation of the **bin\_packing** constraint from [10] by using the following signature:

$$\text{multibin\_packing}([y_{j,l}], [x_i], [w_{i,l}]) \quad (1)$$

where  $x_i = j$  if item  $i$  is assigned to bin  $j$  and  $y_{j,l}$  is a load variable ranging in  $[0, c_{j,l}]$  that represents the total weight packed on bin  $j$  for dimension  $l$ . Indeed, the semantic of constraint (1) is equivalent to the following relations:

$$\sum_{i \in I: x_i = j} w_{i,l} = y_{j,l} \leq c_{j,l} \quad \forall j \in B, \forall l \in K. \quad (2)$$

The case of a single dimension  $k = 1$  reduces (1) to the well-known **bin\_packing** constraint. Therefore, a natural decomposition of (1) is to use a single **bin\_packing** for each dimension as follows:

$$\text{bin\_packing}([y_{1,l}, \dots, y_{m,l}], [x_i], [w_{i,l}]) \quad \forall l \in K. \quad (3)$$

This decomposition is used for instance in [6] to formulate the Machine Reassignment problem proposed in the Roadeff Google Challenge 2012<sup>1</sup>. In the Machine Reassignment problem, each item represents an application to be (re)assigned to a server, and each dimension represents a resource consumed by an application, such as, for instance, CPU time, memory, and bandwidth. Since every server has a limited capacity for each resource, the **multibin\_packing** constraint captures a fundamental substructure of the Machine Reassignment problem.

In [2], the authors propose a constraint based on Multivalued Decisions Diagrams (MDD) for multidimensional bin packing problems and they show that the approach is very effective on a set of randomly generated instances. The MDD approach definitely outperforms a basic CP model based on decomposition (3) and solved with a basic first-unassigned min-value branching strategy. In several cases, the model with the MDD constraint outperforms a Mixed Integer Programming approach as well. In particular, the MDD approach is attractive for hard instances on the transition phases from infeasible to feasible, likely due to the ability of MDD to handle symmetries. While their approach is very interesting, it is hard to embed into existing CP solvers.

The first filtering algorithm for the **bin\_packing** constraint was presented by Shaw in [10], where the author combined the algorithm for the Knapsack constraint introduced in [11] with a well-known lower bound on the minimum bin packing problem (e.g., see Chapter 10 in [5]). Shaw’s algorithm is implemented in several CP solvers, it is listed in the global constraint catalog [1], and it is useful for several industrial applications (e.g., see [9]).

The contribution of this paper is to propose a simple decomposition for the **multibin\_packing** constraint that, in addition to constraints (3), uses a collection of **all\_different** constraints automatically derived from a conflict graph, and that posts symmetry breaking constraints. The benefits of the **all\_different** constraints are twofold: they perform additional filtering and they help the branching to identify “most-conflicting” variables. We show experimentally that our approach is very effective on the instances recently presented in [2]. Since the proposed **multibin\_packing** decomposition is based on existing constraints, it can be easily implemented in any existing CP solver.

The outline of the paper is as follows: Section 2 introduces the problem decomposition, while Section 3 presents two alternative symmetry breaking strategies. Section 4 reports computational results and concludes the paper.

## 2 Constraint Decomposition

The **multibin\_packing** constraint naturally decomposes into  $k$  independent **bin\_packing** constraints. However, this basic decomposition does not account for the different dimensions of each item: while two items may fit in the same bin while considering a given dimension  $l_1$ , they may be in conflict while considering a different dimension  $l_2$ , since the sum of their weights might exceed the

<sup>1</sup> <http://challenge.roadeff.org/2012/en/>, last visited April, 24-th, 2013.

bin capacity for  $l_2$ . However, the interdependencies among the item dimensions across the bins can be exploited systematically via a conflict graph.

Given an instance of the **multibin\_packing** constraint, we build an undirected *conflict* graph  $G = (V, E)$  by looking for pairs of conflicting items. The conflict graph  $G$  is constructed as follows. First, for each item  $i$  in  $I$ , we add a vertex to  $V$ : we have a one-to-one mapping between items and vertices of  $G$ . Second, we add an edge  $\{i_1, i_2\}$  to  $E$  for each pair of items with  $i_1, i_2 \in I$  and  $i_1 < i_2$ , and such that the following relation holds:

$$\forall j \in B. \quad \exists l \in K : \quad w_{i_1, l} + w_{i_2, l} > c_{j, l} \quad (4)$$

Indeed, two items  $i_1$  and  $i_2$  are in conflict if for each bin there exists at least a dimension  $l$  such that the sum of the two item weights exceed the bin capacity for dimension  $l$ .

We show next how we use the conflict graph. Recall that a *clique* is complete subgraph of  $G$ , and that a clique is *maximal* if it is not a subset of any other clique. Given a subset of items  $J \subseteq I$ , we denote by  $x_J$  the subset of the  $[x_i]$  variables corresponding to the items in  $J$ .

**Proposition 1.** *Given an instance of **multibin\_packing**, the cardinality  $\omega(G)$  of the maximum clique, denoted by  $C^*$ , of the conflict graph gives a lower bound on the number of bins necessary in any feasible assignment of items to bins.*

Clearly, every pair of items corresponding to vertices in  $C^*$  must be assigned to different bins, therefore we need at least  $\omega(G)$  bins to have a feasible assignment. However, we do not really need a maximum clique in order to detect infeasibility, as shown next.

**Proposition 2.** *Given an instance of **multibin\_packing**, if the conflict graph  $G$  contains a clique  $C$  of cardinality strictly greater than the number of available bins, that is  $|C| > m$ , then the **multibin\_packing** constraint cannot hold.*

Therefore, to declare infeasibility of a given instance of **multibin\_packing**, we are interested in a finding clique of cardinality equal to  $m + 1$ . Despite finding a clique of a given size is an NP-complete problem [3], we have that  $m \ll n$  and that the conflict graph is sparse (otherwise the **multibin\_packing** instance would be likely infeasible). In practice, exact algorithms (e.g., see [7]) are extremely efficient in checking if a clique of size  $m + 1$  exists in the **multibin\_packing** instances taken from the literature. For easy instances, the set of edges  $E$  may even be empty.

It is possible to exploit the cliques in  $G$  for more than pure consistency checking. Specifically, since for any clique  $C$  with  $1 < |C| \leq m$ , it is possible to post an **all\_different** constraint on the item variables corresponding to the vertices in the clique. More formally:

**Proposition 3.** *Given an instance of **multibin\_packing**, every subset  $J \subset I$  of items corresponding to vertices of any (maximal) clique  $C$  of  $G$ , with  $1 < |C| \leq m$ , must be assigned to different bins, that is the corresponding subset of item variables  $x_J$  must take a different value: **all\_different** $([x_J])$  must hold.*



The proposition holds for any clique, but it is better to consider maximal cliques only, since they are fewer in number.

While the number of maximal cliques of  $G$  is, in theory, exponential in the number of vertices, in many practical cases their number is definitely reasonable. In addition, we are not forced to list every maximal cliques, and it is easy to devise heuristics that limit the number of cliques considered. For instance, it is possible to look for every vertex  $i$  of  $G$  a maximal clique that contains vertex  $i$ . Simple greedy heuristics for finding maximal cliques have worst-case time complexity of  $O(n^2)$ .

### 3 Symmetry Breaking

In case the bins have identical capacities for every dimension, that is whenever  $c_{jl} = c_l$  for all  $j$  in  $B$ , and in case there is not other special constraint nor costs on the assignment of items to bins, then the **multibin\_packing** constraint admits several symmetric assignments, since every possible permutation of bin assignment gives equivalent solutions. For instance, given a feasible assignment, if we exchange all the items assigned to the first bin with all the items assigned to the second bin, we get another feasible solution. This kind of situation seems to occur quite often in practice.

#### 3.1 Symmetry Breaking by Variable Fixing

In order to partially break symmetries, we can again exploit the conflict graph  $G$ . In practice, we can take the largest clique  $C$  with cardinality smaller than or equal to  $m$  (otherwise the constraint is infeasible) and we can fix every item variable corresponding to vertices in  $C$  to a different value from its own domain. Let  $dom(x_i)$  be the domain of variable  $x_i$ ; initially, every variables has the same domain. We take the first variable in  $C$ , and we assign it the minimum value in its domain; then we take the second variable in  $C$  and we assign it the second value in its domain; and so on. With a small abuse of notation, we write:

$$x_i \leftarrow \min\{dom(x_i)\} \quad \text{for } i \in C. \quad (5)$$

where after each assignment, propagation reduces the other variable domains.

Note that whenever it is possible to use this variable fixing technique, then, once the variables have been fixed and all other problem constraints have propagated, it is possible to construct a new conflict graph  $G$  using the residual bin capacity obtained by considering the fixing in (5). The new conflict graph can again be used for consistency checking and, likely, to post additional **all\_different** constraints.

#### 3.2 Symmetry Breaking Constraints

If the assignment of items to bins is symmetrical, as discussed previously, a different but standard way to break symmetries during search consists in posting

an additional constraint that invalidates equivalent feasible assignments (e.g., see Chapter 10 in [8]).

If we consider a single dimension  $l$ , for instance the first dimension  $l = 1$ , we can post an ordering constraint on the corresponding subset of load variables:

$$y_{j,1} \geq y_{(j+1),1} \quad \forall j \in \{1, \dots, m-1\}. \quad (6)$$

However, no ordering constraint can be posted on the other load variables.

Note that constraint (6) reduces the number of equivalent feasible assignments of items to bins, and therefore on *easy* instances of **multibin\_packing** it might have a negative effect when looking for a single feasible solution. We will discuss this issue in the computational results section.

The symmetry breaking constraints based on (6) are only applicable when every bin has the same capacity for every dimension. If we perform an item-bin variable fixing for statically break some symmetries, as described in Section 3.1, we are in practice modifying the problem and, as a consequence, the bins have no longer an identical capacity for each dimension. For this reason, the two symmetry breaking strategies are incompatible.

## 4 Computational Results

In order to evaluate our approach, we have implemented the proposed decomposition of **multibin\_packing** within the Gecode constraint system v.3.7.3 [4]. Note that Gecode has a very efficient implementation of the Shaw's Bin Packing constraint [10]. In order to list every maximal cliques of the conflict graph we used **cliquer-1.21** that is the state-of-the-art exact clique finder for sparse graphs [7]. Everything was compiled using the gnu-gcc v4.7.2 compiler. All the tests were run on standard computer with Linux as operating system, with 4GB of RAM and an AMD Opteron 2.4GHz CPU, but using a single thread and limiting the process memory size to 1GB. Our implementation of the constraint decomposition is available online<sup>2</sup>.

We have run experiments using combinations of the following decomposition:

$$\begin{aligned} \text{multibin\_packing}([y_{j,l}], [x_i], [w_{i,l}]) = \\ \text{bin\_packing}([y_{1,l}, \dots, y_{m,l}], [x_i], [w_{i,l}]) \quad \forall l \in K \end{aligned} \quad (7)$$

$$\text{all\_different}([x_C]) \quad \forall C \in \mathcal{C} \quad (8)$$

$$y_{j,1} \geq y_{(j+1),1} \quad \forall j \in \{1, \dots, m-1\} \quad (9)$$

where  $\mathcal{C}$  is the collection of all maximal cliques of the conflict graph defined for the given instance of **multibin\_packing**.

We denote by:

- (A) the decomposition using only constraints (7)
- (B) the decomposition using constraints (7) and (8)

---

<sup>2</sup> <http://github.com/stegua/binpacking/tree/master/release>

- (C) the decomposition using constraints (7) and (9)
- (D) the decomposition using constraints (7), (8), and (9)
- (E) the decomposition using constraints (7), (8), and (5)

The rationale for studying decompositions (B) and (C) is to assess the impact of the additional constraints (7) and (8), and for decompositions (D) and (E) is to compare the two alternative symmetry breaking strategies.

In order to make a fair comparison between the different decomposition strategies, in our computational results, we did not update the conflict graph for strategy (E). In practice, constraints (8) are posted on the same collection of maximal clique for method (B), (D), and (E).

We consider a set of instances taken from [2] that corresponds to randomly generated instances with 18 items, 6 dimensions, and 6 bins. In order to have a range of instances from “easy infeasible” to “easy feasible” while passing through a “hard” phase transition, the instances differ on the tightness of the bin capacities (i.e. the bin slacks), according to a parameter  $\beta$  (for the details on the instances generation see [2]). The values of  $\beta$  range in  $\{0, \dots, 35\}$ , and for each value of  $\beta$  there are 52 instances, for a total of 1872. The phase transition happens for  $\beta \in \{16, \dots, 24\}$ .

Figures (1.a)–(1.d) show the average computation times in seconds (vertical axis) in function of the percentage of bin slacks, that is, for value of  $\beta$  from 0 to 35. Plot (1.a) clearly shows that the hard instances correspond to values of  $\beta$  ranging from 15 to 25, while the plots (1.b)–(1.d) show that:

- For the hard instances (1.b) and the easy infeasible instances (1.c) the decomposition (E) that exploits both the **all\_different** constraints and the variable fixing on the maximum clique, outperforms the other decompositions by a large margin: the average computation time is below 1 second also for the instances with  $\beta = 20$ , and, hence, it is a *simple* and *effective* alternative to the **multibin\_packing** constraint based on Multivalues Decision Diagrams proposed in [2].
- For the easy infeasible (1.c) and the easy feasible instances (1.d), the symmetry breaking constraint (9) does not always pay off, while it plays an important role on the hard instances (1.b), since decomposition (C) and (D) are always more efficient than (A) and (B).
- As expected, for the easy feasible instances (1.d), the simplest decomposition (A) is quite efficient, since in this case the filtering algorithms play a minor role.

Figures (2.a) and (2.b) give a different view of the same results by showing the empirical cumulative distributions of the fraction of instances solved as a function of the run times. Figure (2.a) are the distributions for all the 1872 instances proposed in [2], while Figure (2.b) considers only the instances with  $\beta = 20$ , i.e. among the hardest instances. Again decomposition (E) outperforms all the others by a large margin. The decomposition (B) based on the **all\_different** constraints outperforms the simple decomposition (A) for a large number of instances, but not for all of them. The decomposition (C) and (D) that exploit

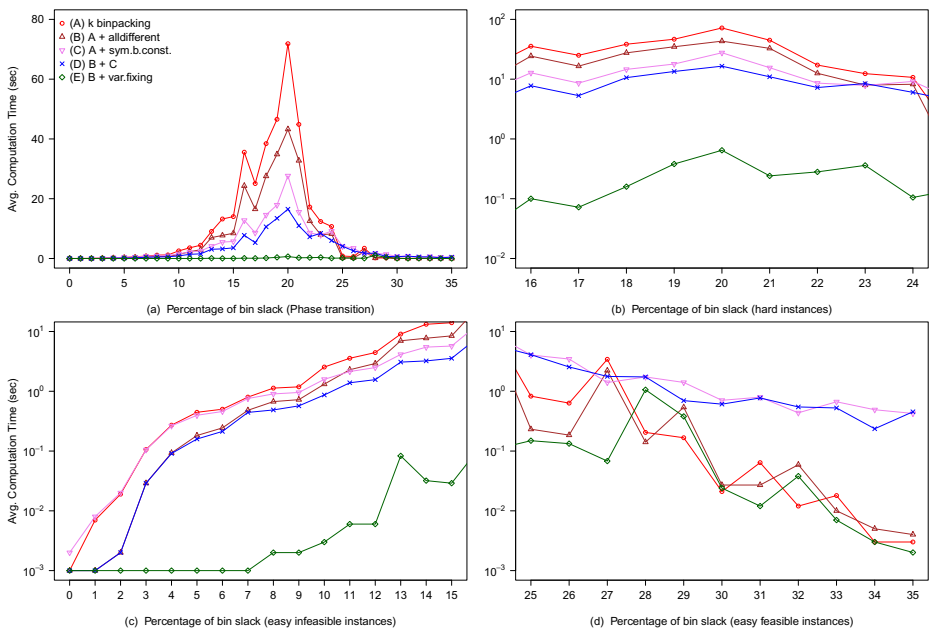


**Fig. 1.** Average Computation Time vs. Percentage of Bin Slack

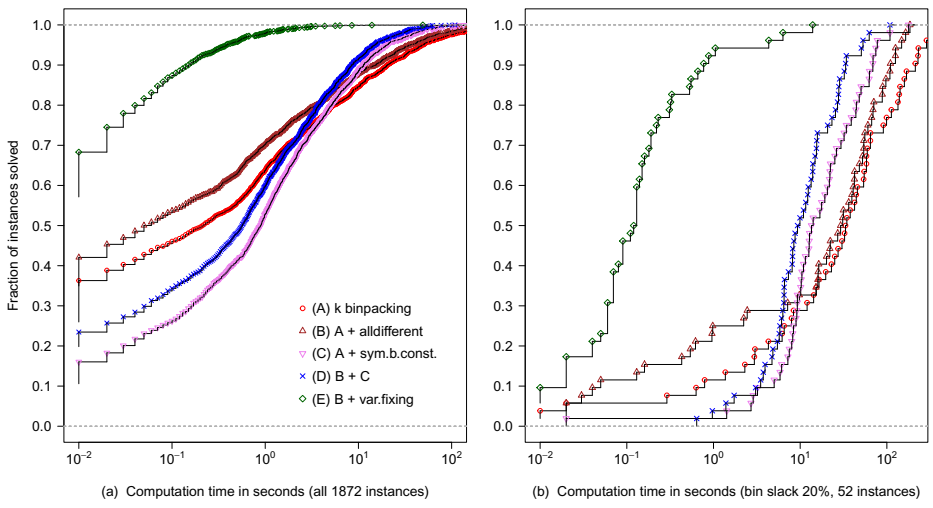


**Fig. 2.** Empirical Cumulative Distribution of the fraction of instances solved as a function of computation times

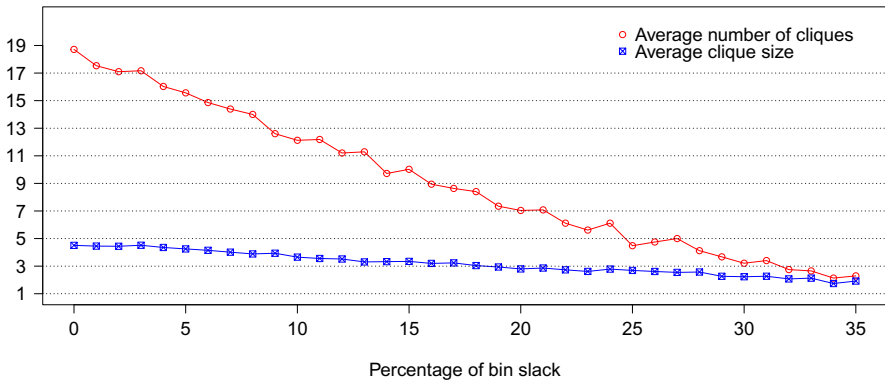


**Fig. 3.** Average number of cliques and average clique size vs. bin slacks

the symmetry breaking constraints are useful on the hard instances, as shown in Figure (2.b), but in some cases the overhead of constraints (9) does not pay off.

Finally, since the main contribution of our approach is based on detecting maximal cliques in the conflict graph and on posting an `all_different` constraint for every clique found, Figure 3 characterizes the instances in terms of average clique number and average clique size as a function of the bin slack. Note that for the easy infeasible instances (i.e.  $\beta < 15$ ), there are many and large cliques, while for the easy feasible instances (i.e.  $\beta > 25$ ) the cliques are few and rather small (sometimes are not even present). Clearly, our decomposition approach is better suited for instances with several large cliques.

## References

1. Beldiceanu, N., Carlsson, M., Rampon, J.X.: Global constraint catalog. SICS Research Report (2005)
2. Kell, B., van Hoeve, W.-J.: An MDD approach to multidimensional bin packing. In: Gomes, C., Sellmann, M. (eds.) CPAIOR 2013. LNCS, vol. 7874, pp. 128–143. Springer, Heidelberg (2013)
3. Garey, M.R., Johnson, D.S.: Computers and intractability. Freeman, New York (1979)
4. Gecode Team. Gecode: Generic constraint development environment (2013), <http://www.gecode.org>
5. Martello, S., Toth, P.: Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Inc. (1990)
6. Mehta, D., O’Sullivan, B., Simonis, H.: Comparing solution methods for the machine reassignment problem. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 782–797. Springer, Heidelberg (2012)
7. Östergård, P.R.J.: A fast algorithm for the maximum clique problem. Discrete Applied Mathematics 120(1), 197–207 (2002)

8. Rossi, F., Van Beek, P., Walsh, T.: Handbook of constraint programming. Elsevier Science (2006)
9. Schaus, P., Régin, J.-C., Van Schaeren, R., Dullaert, W., Raa, B.: Cardinality reasoning for bin-packing constraint: Application to a tank allocation problem. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 815–822. Springer, Heidelberg (2012)
10. Shaw, P.: A constraint for bin packing. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 648–662. Springer, Heidelberg (2004)
11. Trick, M.A.: A dynamic programming approach for consistency and propagation for knapsack constraints. *Annals of Operations Research* 118(1), 73–84 (2003)

# Maintaining Soft Arc Consistencies in BnB-ADOPT<sup>+</sup> during Search<sup>\*</sup>

Patricia Gutierrez<sup>1</sup>, Jimmy H.M. Lee<sup>2</sup>, Ka Man Lei<sup>2</sup>,  
Terrence W.K. Mak<sup>3</sup>, and Pedro Meseguer<sup>1</sup>

<sup>1</sup> IIIA - CSIC, Universitat Autònoma de Barcelona, 08193 Bellaterra Spain  
{patricia,pedro}@iiia.csic.es

<sup>2</sup> Department of Computer Science and Engineering,  
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong  
{jlee, kmlei}@cse.cuhk.edu.hk

<sup>3</sup> NICTA Victoria Laboratory & University of Melbourne, VIC 3010, Australia  
Terrence.Mak@nicta.com.au

**Abstract.** Gutierrez and Meseguer show how to enforce consistency in BnB-ADOPT<sup>+</sup> for distributed constraint optimization, but they consider unconditional deletions only. However, during search, more values can be pruned conditionally according to variable instantiations that define subproblems. Enforcing consistency in these subproblems can cause further search space reduction. We introduce efficient methods to maintain soft arc consistencies in every subproblem during search, a non trivial task due to asynchronicity and induced overheads. Experimental results show substantial benefits on three different benchmarks.

## 1 Introduction

Distributed Constraint Optimization Problems (DCOPs) have been applied in modeling and solving a substantial number of multiagent coordination problems, such as meeting scheduling [1], sensor networks [2] and traffic control [3]. Several distributed algorithms for optimal DCOP solving have been proposed: ADOPT [4], DPOP [5], BnB-ADOPT [6], NCBB [7] and others.

BnB-ADOPT<sup>+</sup>-AC/FDAC [8] incorporate consistency enforcement during search into BnB-ADOPT<sup>+</sup> [9], obtaining substantial efficiency improvements. Enforcing consistency allows to prune some values, making the search space smaller. This previous work considers unconditional deletions only so as to avoid overhead in handling assignments and backtracking. However, values that could be deleted conditioned to some assignments will not be pruned with this strategy, so that search space reduction opportunities are missed. In this paper, we propose an efficient way to maintain soft

---

<sup>\*</sup> We are grateful to the anonymous referees for their constructive comments. The work of Lei and Lee was generously supported by grants CUHK413808, CUHK413710 and CUHK413713 from the Research Grants Council of Hong Kong SAR. The work of Gutierrez and Meseguer was partially supported by the Spanish project TIN2009-13591-C02-02 and Generalitat de Catalunya 2009-SGR-1434. The work of Gutierrez, Lee and Meseguer was also jointly supported by the CSIC/RGC Joint Research Scheme grants S-HK003/12 and 2011HK0017. The work of Mak was performed while he was at CUHK.

arc consistencies, considering any kind of deletions resulting from enforcing consistency in asynchronous distributed constraint solving, something that—to the best of our knowledge—has not been explored before.

A search-based constraint solving algorithm forms subproblems of the original problem by assignments. We maintain soft arc consistencies in each subproblem, so that variable assignments during search are also considered in consistency enforcement. As a result, we can explore more value pruning opportunities and thus further reduce the search space. Gutierrez and Meseguer introduce an extra copy of cost functions in each agent, so that search and consistency enforcement are done asynchronously. Our contribution goes further maintaining soft arc consistencies in each subproblem during search, so that (i) search and consistency enforcement are done asynchronously, introducing some extra copies of cost functions; (ii) the induced overhead caused by backtracking and undoing assignments and deletions is minimized. The asynchronicity requirement and different cost measurements require us to introduce novel techniques over those used in centralized CP. Experimentally, we show the benefits of our proposal on benchmarks usually unamenable to solvers without consistency.

## 2 Preliminaries

**DCOP.** A DCOP is defined by  $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \alpha \rangle$ , 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 for  $\mathcal{X}$ ;  $\mathcal{C}$  is a set of cost functions;  $\mathcal{A} = \{1, \dots, n\}$  is a set of  $n$  agents and  $\alpha : \mathcal{X} \rightarrow \mathcal{A}$  maps each variable to one agent. We use binary and unary cost functions only, which produce non-negative costs. The cost of a complete assignment is the sum of all unary and binary cost functions evaluated on it. An *optimal solution* is a complete assignment with minimum cost. Each agent holds exactly one variable, so that variables and agents can be used interchangeably. Agents communicate through messages, which are never lost and delivered in the order they were sent, for any agent pair.

DCOPs can be arranged in a *pseudo-tree*, where nodes correspond to variables and edges correspond to binary cost functions. There is a subset of edges, called *tree-edges*, that form a rooted tree. The remaining edges are called *back-edges*. Variables involved in the same cost function appear in the same branch. Tree edges connect parent-child nodes. Back-edges connect a node with its pseudo-parents and pseudo-children.

**BnB-ADOPT and BnB-ADOPT<sup>+</sup>.** BnB-ADOPT [6] is an algorithm for optimal DCOP solving. It uses the communication framework of ADOPT [4] (agents are arranged in a pseudo-tree), but it changes the search strategy to depth first branch-and-bound. It shows improvements over ADOPT. Each agent holds a context, as a set of assignments involving some of the agent's ancestors that is updated with message exchanges. Message types are: VALUE, COST and TERMINATE. A BnB-ADOPT agent executes this loop: it reads and processes all incoming messages and assigns its value. Then, it sends a VALUE to each child or pseudochild and a COST to its parent. BnB-ADOPT<sup>+</sup> [9] is a version of BnB-ADOPT that prevents from sending most redundant messages, keeping optimality and termination. It substantially reduces communication.

**Soft Arc Consistency.** Let  $(i, a)$  represents  $x_i$  taking value  $a$ ,  $\top$  is the lowest unacceptable cost,  $C_{ij}$  is the binary cost function between  $x_i$  and  $x_j$ ,  $C_i$  is the unary cost



function on  $x_i$  values,  $C_\phi$  is a zero-ary cost function (lower bound of the cost of any solution). We consider the following local consistencies [10,11]:

- *Node Consistency* (NC):  $(i, a)$  is NC if  $C_\phi + C_i(a) < \top$ ;  $x_i$  is NC if all its values are NC and  $\exists b \in D_i$  s.t.  $C_i(b) = 0$ .  $P$  is NC if every variable is NC.
- *Arc Consistency* (AC):  $(i, a)$  is AC w.r.t.  $C_{ij}$  if  $\exists b \in D_j$  s.t.  $C_{ij}(a, b) = 0$ ;  $b$  is a *support* of  $a$ ;  $x_i$  is AC if all its values are AC w.r.t. every binary cost function involving  $x_i$ ;  $P$  is AC if every variable is AC and NC.
- *Directional Arc Consistency* (DAC):  $(i, a)$  is DAC w.r.t.  $C_{ij}$ ,  $j > i$ , if  $\exists b \in D_j$  s.t.  $C_{ij}(a, b) + C_j(b) = 0$ ;  $b$  is a *full support* of  $a$ ;  $x_i$  is DAC if all its values are DAC w.r.t. every  $C_{ij}$ ;  $P$  is DAC if every variable is DAC and NC.
- *Full DAC* (FDAC):  $P$  is FDAC if it is DAC and AC.

AC/DAC can be reached by forcing supports/full supports to NC values and pruning values that are not NC. Supports can be forced by projecting the minimum cost from its binary cost functions to its unary costs, and then projecting the minimum unary cost into  $C_\phi$ . Full supports can be forced in the same way, but first it is needed to extend from the unary costs of neighbors to the binary cost functions the minimum cost required to perform in the next step the projection over the value. The systematic application of *projection* and *extension* does not change the optimum cost [10,11]. When we prune a value from  $x_i$ , we need to recheck AC/DAC on every variable that  $x_i$  is constrained with, since the deleted value could be the support/full support of a value of a neighbor variable. So, a deleted value in one variable might cause further deletions in others. The AC/DAC check must be done until no further values are deleted.

**BnB-ADOPT<sup>+</sup> and Soft Arc Consistencies.** BnB-ADOPT<sup>+</sup> has been combined with AC and FDAC [8]. Search is based on BnB-ADOPT<sup>+</sup>, maintaining the same data and communication structure. Soft arc consistencies are enforced on a copy of the original cost functions, limited to unconditional deletions. This combination has caused a number of modifications in the original algorithm, both in messages and in computation.

Regarding messages, (i) COST messages include *subtreeContr* that aggregates the costs of unary projections to  $C_\phi$  made on every agent; (ii) VALUE messages include  $\top$  and  $C_\phi$ ; (iii) a new DEL message is added to inform of value deletions; when received, neighbors recheck AC/FDAC, which may lead to further deletions; (iv) a new UCO message is added when FDAC is enforced, to inform the unary costs needed for enforcing DAC; when received, agents enforce DAC with any other higher constrained agents and recheck FDAC, which may lead to further deletions.

Regarding computation, each agent holds one copy of constrained agents' domains and related binary cost functions for consistency enforcement. Handling value deletions require some extra effort. Only the agent owner of a variable can modify its domain.

### 3 Maintaining Soft Arc Consistencies

We enforce AC and FDAC asynchronously in all subproblems during search by utilizing additional copies of variable domains and cost functions in each agent. To explain our *Maintaining AC* (MAC) and *Maintaining FDAC* (MFDAC) algorithms, we first outline

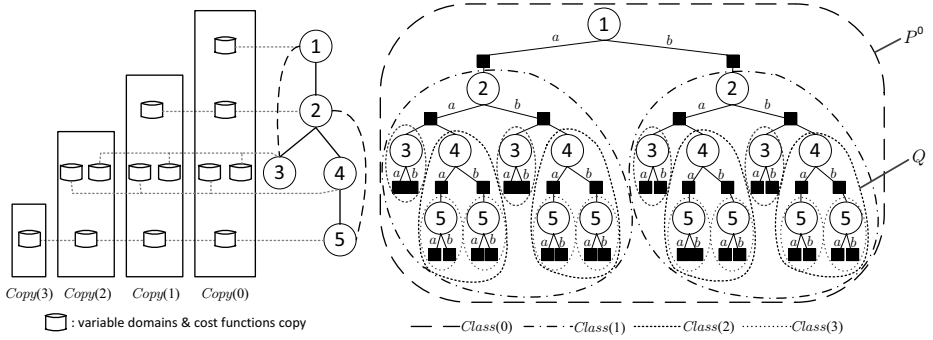


**Fig. 1.** Left: The pseudo-tree of a DCOP with five variables, the variable domains and cost functions copies they maintain. Right: Search tree ( $a/b$  domains), subproblems and classes of subproblems. Subproblems at the same depth belong to same class.

an agent classing scheme based on the position of an agent in the problem structure. The scheme governs the required number of copies of variable domains and cost functions. Second, we provide the information of messages in our methods and an overview of the changes in the overall message handling mechanism after adopting our new methods in BnB-ADOPT<sup>+</sup>. Third, we provide methods for reinitializing variable domains and cost functions copies in an agent when the context of a subproblem changes. Such reinitialization is needed since conditional deletions are no longer valid. Thus, consistency enforcement has to start from scratch again using the new context. Fourth, we propose a new message type and the handling mechanism for backtracking, when an agent arrives at the empty domain within a subproblem. This means that the assignments of some ancestor agents cannot lead to the optimal solution and should be pruned. Fifth, we reduce costs by transferring deletions from subproblems to inner subproblems. Sixth, we present an ordering scheme and asynchronous messaging mechanism to ensure that the two separate copies of the same cost function stored in the two constrained agents are identical even in the presence of simultaneous consistency operations. Finally, we describe how we ensure optimality and termination after introducing the new methods.

### 3.1 Classes of Subproblems

In BnB-ADOPT<sup>+</sup> [9], all agents are organized in a pseudo-tree (Fig. 1 Left). The variable ordering of the corresponding AND-OR search tree [6] (Fig. 1 Right) follows the (partial) order defined in the pseudo-tree. When an agent is assigned a value, the descendant agents together with the current assignments form a subproblem. Notations:  $P^0$  is the original DCOP;  $P$  is a subproblem of  $P^0$ ;  $T^0$  is a pseudo-tree that defines the variable ordering in  $P^0$ ;  $d_j$  is the *depth* of agent  $j$  in  $T^0$  as the distance from the root node to  $j$  excluding back-edges;  $vars(P)$  is the set of variables of  $P$ ;  $depth(P)$  is the smallest depth among all variables in  $vars(P)$ ;  $ancestors(P)$  is the set of *ancestor variables* satisfying (1) they are in  $vars(P^0)$  but not in  $vars(P)$ , (2) they have depths smaller than  $depth(P)$ , and (3) they are constrained with at least one variable in  $vars(P)$ ;  $context(P)$  is the variable assignments of  $ancestors(P)$ ;  $context_j$ , the

*context* of agent  $j$ , is the set  $\{(i, a, t) | i = j \text{ or } i \text{ is an ancestor of } j, a \text{ is agent } i \text{ value which is assigned to } i \text{ at timestamp } t\}$ . Two contexts are *compatible* if no agent takes on different values in the two contexts. Every *subproblem*  $P$  of  $P^0$  is uniquely identified by  $(\text{depth}(P), \text{ancestors}(P), \text{context}(P), \text{vars}(P))$ .

Fig. 1 Right illustrates the search tree and subproblems of a DCOP with 5 agents. Each circular node is the root node of a subproblem and there are 19 such subproblems (including the original problem) in the example. The original problem  $P^0$  is  $(0, \emptyset, \emptyset, \{1, 2, 3, 4, 5\})$  and  $(2, \{1, 2\}, \{(1, b), (2, b)\}, \{4, 5\})$  (labeled  $Q$  in the figure) is the subproblem of  $P^0$  after instantiating agent 1 and agent 2 to value  $b$ . We define a class of subproblems as follows. A subproblem  $P$  of  $P^0$  is of *Class*  $d$  if  $\text{depth}(P) = d$ . We further define  $\text{Class}(d) = \{P | \text{depth}(P) = d\}$ .

Fig. 1 Right also illustrates the classes of subproblems of the DCOP. There are four classes of subproblems:  $\text{Class}(0)$  involves the original problem only.  $\text{Class}(1)$  includes two subproblems  $(0, \{1\}, \{(1, a)\}, \{2, 3, 4, 5\})$  and  $(0, \{1\}, \{(1, b)\}, \{2, 3, 4, 5\})$ ,  $\text{Class}(2)$  includes eight subproblems in which four are rooted at node 3 and the other four are rooted at node 4. All  $\text{Class}(2)$  subproblems hold the assignment information of agents 1 and 2 (their context).  $\text{Class}(3)$  includes eight subproblems which are all rooted at node 5 and hold assignment information of agents 1, 2 and 4.

In BnB-ADOPT<sup>+</sup>-AC/FDAC [8], search and consistency enforcement are done asynchronously: an extra copy of each cost function is used for consistency enforcement and they do not interfere with the original copy used for search. We use the same idea for MAC and MFDAC: we include extra copies of variable domains and cost functions for enforcing consistency in different subproblems, but not a copy for each subproblem. Each agent  $i$  of depth  $d_i$  will hold one copy  $\text{Copy}(d)$  for each class  $\text{Class}(d)$  of subproblems where  $d \leq d_i$ . For instance, in Fig. 1 Left, agents keep the following copies of cost functions and domains: agent 1 one copy, agent 2 two copies, agent 3 and 4 three copies, and agent 5 four copies. Then, each agent  $i$  will hold  $d_i + 1$  copies of variable domains and cost functions and the space complexity of each agent is  $O(dhm^2)$  where  $d$  is the agent's depth,  $h$  is the pseudo-tree's height and  $m$  is the maximum domain size of agents. These copies will play a key role in reinitializing domains and cost functions when conditional deletions are no longer valid in a context change.

### 3.2 Maintaining Consistencies in All Subproblems: An Overview

To maintain soft arc consistencies in every subproblem, extra operations and information exchanges are needed. The major additional operations include (1) reinitialization, (2) backtracking to the culprit when an empty domain is detected and (3) transferring

**Table 1.** Messages of AC, FDAC, MAC and MFDAC. New fields are underlined. DEL messages contain ACC or DACC depending on the AC or FDAC consistency level enforced.

| AC/FDAC                                                      | MAC/MFDAC                                                                       |
|--------------------------------------------------------------|---------------------------------------------------------------------------------|
| VALUE( <u>src,dest,value,threshold</u> , $\top$ , $C_\phi$ ) | VALUE( <u>src,dest,value,threshold</u> , $\top$ , $C_\phi$ [], <u>context</u> ) |
| COST( <u>src,dest,lb,ub,reducedContext,subtreeContr</u> )    | COST( <u>src,dest,lb,ub,context,subtreeContr</u> [])                            |
| DEL( <u>src,dest,value,ACC</u>  DACC)                        | DEL( <u>src,dest,depth,values</u> [], <u>context,ACC</u> []) DACC[])            |
| UCO*( <u>src,dest,vectorOfExtensions,ACC</u> )               | UCO**( <u>src,dest,depth,vectorOfExtensions,context,ACC</u> )                   |
|                                                              | BTK( <u>src,dest,targetDepth,context</u> )                                      |
| * Only in FDAC                                               | ** Only in MFDAC                                                                |

```

procedure ProcessVALUE(msg)
  do the work as in BnB-ADOPT+
  Reinitialize(msg.src,msg.context)
  update  $\top$  and  $C_\phi$  if applicable

procedure ProcessCOST(msg)
  do the work as in BnB-ADOPT+
  Reinitialize(msg.src,msg.context)
  aggregate  $C_\phi$  from msg.subtreeContr
  update  $C_\phi$  if applicable

procedure ProcessDEL(msg)
  Reinitialize(msg.src,msg.context)
   $d \leftarrow msg.depth$ 
  vars  $\leftarrow$  set of variables  $i$  in  $context_{self}$  where  $d_i \in [0, msg.depth - 1]$ 
  if values of vars in msg.context are compatible with those in  $context_{self}$  then
    for  $d' = d \rightarrow d_{self}$  do
      delete msg.values[] from msg.src's domain in  $Copy(d')$ 
      undo disordered operations in  $Copy(d')$  if necessary
      perform projection in  $Copy(d')$ 
      update ACC counter if necessary

procedure ProcessUCO(msg)
  Reinitialize(msg.src,msg.context)
   $d \leftarrow msg.depth$ 
  vars  $\leftarrow$  set of variables  $i$  in  $context_{self}$  where  $d_i \in [0, msg.depth - 1]$ 
  if values of vars in msg.context are compatible with those in  $context_{self}$  then
    if  $ACC_{self \rightarrow msg.src} = msg.ACC$  then
      perform extension in  $Copy(d)$ 
      update DAC counter if necessary

```

**Fig. 2.** Pseudocode for handling VALUE, COST, DEL and UCO messages

deletions to subproblems. Reinitialization is needed for ensuring the correctness of the algorithm. Backtracking to the culprit and transferring deletions to subproblems are not necessary for correctness but they can improve performance. Besides, to ensure the agents maintain the same cost functions in each copy, Gutierrez and Meseguer [12] proposed to include two new messages to synchronize deletions. However, these messages introduce an extra overhead and slow down the consistency enforcement. We propose a new method to allow agents to undo and reorder some of their operations in order to ensure identical cost functions copies.

Consistency enforcement in each subproblem is similar to that of Gutierrez and Meseguer [8], in which consistency is only enforced in the copy for the original problem. In our case, consistency is enforced in the copy for every class of subproblems at the same time. Extra information is embedded in the existing messages (TERMINATE message same as the one in BnB-ADOPT<sup>+</sup> and UCO only in FDAC and MFDAC) and only one new message type (BTK) is added. Table 1 summarizes the information per type. Fig. 2 shows the pseudocode for handling these messages (pseudocode for BTK appears in Section 3.4).

When an agent receives a VALUE or COST message, it first performs the BnB-ADOPT<sup>+</sup> process, and then it checks for reinitialization. When an agent receives a DEL message, it does the following steps (1) reinitialization checking, (2) compatibility checking, (3) value deletions, (4) maintaining identical cost function copies, (5) projections and (6) update projection counter. Similarly, when an agent receives an UCO message, it checks for reinitialization first and then performs the extension and extension counter update. After an agent  $i$  has processed a VALUE, COST, DEL or UCO message, AC/DAC may be re-enforced in  $Copy(d)$  where  $d \in [1, d_i]$  if (1)  $Copy(d)$  is

reinitialized, (2) a better  $\top$  or  $C_\phi$  is found in  $Copy(d)$ , (3) some values are deleted in  $Copy(d)$ , and (4) some unary costs are increased in  $Copy(d)$  (only apply for DAC).

### 3.3 Reinitialization

When enforcing consistencies in a subproblem  $P$  (excluding the original problem), the conditional deletions generated depend on the variable assignments information ( $context(P)$ ) that  $P$  holds. These conditional deletions may not occur in other subproblems in  $Class(depth(P))$ , when the variable assignments have changed. Therefore, conditionally deleted values have to be recovered when values of ancestor agents change. When an agent  $i \in ancestors(P)$  of a subproblem  $P$  in  $Class(d)$  changes its value, the context no longer matches that of  $P$ . Search should be now switched to another  $P' \in Class(d)$  such that  $context(P')$  matches the new value of agent  $i$  and other existing assignments. In addition, the copies of cost functions owned by the agents in  $vars(P')$  should be reset using the corresponding copies from upper classes and updated with the variable assignments in  $context(P')$ . Otherwise, the search algorithm will search for solution based on obsolete value pruning information and may result in suboptimal solution. This rationale justifies our next rule.

**Rule 1.** When an agent  $i$  changes its value, all agents  $j \in vars(P)$  where  $P \in \{P' | i \in ancestors(P')\}$  should reinitialize  $Copy(d)$  where  $d_i < d \leq d_j$  to be the corresponding subproblem based on the updated context. The reinitialization in  $j$  is done in a top-down sequence as follows. For  $d = d_i + 1$  to  $d_j$ : (1)  $Copy(d) = Copy(d - 1)$ ; (2) Transform each binary cost function  $C_{jk}$  where  $k \in ancestors(P')$  to unary cost functions  $C_j$  by assigning each  $k$  to value  $a$  where  $(k, a) \in context(P')$ .

Next we describe how to implement Rule 1 in BnB-ADOPT<sup>+</sup> with MAC and MF-DAC. Rule 1 affects an agent when there is a context change. We use VALUE, COST, DEL and UCO messages to carry the context information.

Receiving a VALUE message always signifies a context change in  $i$ 's parent or pseudo-parent. Thus, agent  $i$  always performs reinitialization before deciding whether to change its own value. Receiving a COST message from any of its children may cause a context change. Agent  $i$  should always first compare the timestamps of the child's context and  $i$ 's own context. If the child's context is older than or equal to  $i$ 's context,  $i$  performs nothing. Otherwise, there is a context change and  $i$  will perform reinitialization before performing other BnB-ADOPT<sup>+</sup> operations. When receiving a DEL or UCO message, an agent performs similar checking before proceeding to consistency enforcement operations—if any—. Strictly speaking, reinitialization in an agent is needed only when handling VALUE and COST messages to ensure correctness of the solving result; skipping the reinitialization step for DEL and UCO messages will only miss pruning opportunities, and thus losing efficiency.

Agents need to have the context of *all* ancestors to check for context changes and to do reinitialization. In our VALUE, COST, DEL and UCO messages we include the context of the sender agent, instead of the agent's reduced context as used in BnB-ADOPT<sup>+</sup>, which does not necessarily contain the information of all ancestors.

Fig. 3 shows how to reinitialize, and  $Reinitialize(src, context_{src})$  is called whenever an agent *self* receives a VALUE, COST, DEL or UCO message from *src*

```

procedure Reinitialize(src, contextsrc)
  mindepth =  $\infty$ 
  for  $d = d_{src} \rightarrow 0$  do
    if  $d \geq d_{self}$  then continue
    var  $\leftarrow$  variable of depth  $d$  in contextself
    if Time(contextsrc, var) > Time(contextself, var) then
      contextself(var)  $\leftarrow$  contextsrc(var)
      mindepth  $\leftarrow$   $d + 1$ 
  if mindepth  $\neq \infty$  then
    for  $d' = mindepth \rightarrow d_{self}$  do
      Copy( $d'$ )  $\leftarrow$  Copy( $d' - 1$ )
      var'  $\leftarrow$  variable of depth ( $d' - 1$ ) in contextself
      TransformBinaryToUnary(Copy( $d'$ ), var', contextself)

function Timestamp(context, var) return  $t$  where  $(var, a, t) \in context$ 

procedure TransformBinaryToUnary(Copy, var, context)
  if self is constrained with var and  $(var, a, t) \in context$  then
    for each  $b \in D_{self}$  do
      Copy.Cself( $b$ )  $\leftarrow$  Copy.Cself( $b$ ) + Copy.Cself,var( $b, a$ )

```

**Fig. 3.** Pseudocode for performing reinitialization

(as shown in Fig. 2 and 3). When *self* receives *context<sub>src</sub>*, it first checks whether the variable assignments that *src* holds are the latest information by comparing the timestamps of each variable assignment in *context<sub>src</sub>* and *context<sub>self</sub>*. If the information in *context<sub>src</sub>* is more updated, *self* updates *context<sub>self</sub>* according to *context<sub>src</sub>*. If *self*'s context is updated, it has to perform reinitialization starting from the class of subproblems *Class*(*mindepth*) where *mindepth* is the smallest depth  $d_i$  and agent  $i$ 's context has been changed in *context<sub>self</sub>*. The operations for reinitializing *Copy*( $d$ ) are described in Rule 1. We do not reinitialize the subproblem from the original problem but by duplicating from *Copy*( $d - 1$ ). Thus the works done in the current subproblem of *Class*( $d - 1$ ) will not have to be repeated in *Class*( $d$ ).

### 3.4 Backtracking

Enforcing consistencies in a subproblem  $P$  can lead to an empty domain in some agent of  $P$ . In this case, *context*( $P$ ) is inconsistent and it should be changed. Upon backtracking, the current assigned value  $a$  to the parent, say  $j$ , of the root of  $P$  should be changed: value  $a$  is removed from  $D_j$ , and agent  $j$  can then pick another value from  $D_j$ . This justifies our next rule.

**Rule 2.** If an agent  $i$  obtains an empty domain in the subproblem  $P$  during consistency enforcement, the agent  $j \in ancestors(P)$  with  $d_j = depth(P) - 1$  can delete its value  $a$  from its domain in *Copy*( $d_j$ ), where  $(j, a) \in context(P)$ , provided that *context<sub>j</sub>* is compatible with *context*( $P$ ).

We add a new message BTK to notify backtrackings. When agent  $i$  obtains an empty domain in  $P$ ,  $i$  sends a message BTK( $i, k, depth(P) - 1, context(P)$ ) to its parent  $k$ . The BTK message is sent to the parent agent for propagation because agents can only communicate with constrained agents but the targeted agent may not be a constrained agent. Therefore, this message is propagated up the pseudo-tree until it reaches agent  $j \in ancestors(P)$  where  $d_j = depth(P) - 1$ .

Fig. 4 shows how to handle an incoming BTK message. When an agent other than  $j$  receives a BTK message, it forwards the message to its parent. When  $j$  receives that

```

procedure ProcessBTK(msg)
  if  $d_{self} \neq msg.targetDepth$  then
    sendMsg: (BTK, self, parent, msg.targetDepth, msg.context)
  else
    if msg.context is compatible with  $context_{self}$  then
      DeleteValue(Copy(msg.targetDepth), a) where  $(self, a, t) \in msg.context$ 

```

**Fig. 4.** Pseudocode for handling the BTK message

message,  $j$  checks whether the attached context is compatible with its own context. If yes, it knows that its current assignment  $(j, a) \in context(P)$  will not lead to an optimal solution and it deletes  $a$  from  $Copy(d_j)$ . Otherwise,  $j$  ignores the message.

### 3.5 Transferring Deletions to Subproblems

Redundant deletions may appear in embedded subproblems. It is easy to see that if  $P'$  is a subproblem of  $P$ , the values deleted in  $P$  can also be deleted in  $P'$ . We can transfer the deletions in  $P$  to  $P'$  and no need to send out redundant information of these deletions for  $P'$ . Transferring deletions to subproblems not just avoid redundant DEL messages, it may also increase the chance of reducing more search space. Since the consistency enforcement in different subproblem is different, the suboptimal values found in  $P$  may not be found in  $P'$ . If we transfer these suboptimal values from  $P$  to  $P'$ , more pruning opportunities may be found in  $P'$ .

When an agent deletes values in subproblem  $P$ ,  $depth(P) = d$ , it can also apply the deletions to subproblems  $P'$  where  $depth(P') > d$ . A DEL message is labeled by  $d$ . When other agents receive that message, they apply the deletions to all the subproblems  $P'$  s.t.  $depth(P') \geq d$ . The pseudocode of transferring deletions to subproblems when receiving a DEL message is covered in the ProcessDEL() procedure in Fig. 2.

### 3.6 Keeping Cost Functions Copies Identical

Each of the two agents constrained by a cost function holds a separate copy of the cost function for consistency enforcement. It is thus of paramount importance to ensure the two copies being identical but this task is made difficult by the asynchronous nature of the search algorithm. Fig. 5 gives a simple example of simultaneous deletions [12] in constrained agents  $i$  and  $j$ , which cause projections from  $C_{ij}$  to  $C_i$  in agent  $j$  and  $C_j$  in agent  $i$  respectively. The asynchronous nature of message exchanges can result in the projections/extensions performed in different order and thus different  $C_{ij}$  copies in agents  $i$  and  $j$  respectively.

Gutierrez and Meseguer [12] propose to include two new messages to synchronize deletions but the overhead is high. by allowing one of the two agents to undo and reorder the operations. With this *Undo Mechanism* we keep the asynchronicity and avoid extra messages. We give preference to one of the two agents. The operations will be done in the order of the preferred agent, while the non-preferred one must undo the operations that do not follow that order.

Let us consider two constrained agents  $i$  and  $j$ , and the cost function between them  $C_{ij}$ ;  $i$  and  $j$  each holds a copy of it, denoted by  $C_{ij}^i$  and  $C_{ij}^j$  respectively. Both agents maintain AC. The projection from  $C_{ij}$  to  $C_i$  has to be done on both  $C_{ij}^i$  and  $C_{ij}^j$ .  $C_{ij}^i \Rightarrow$

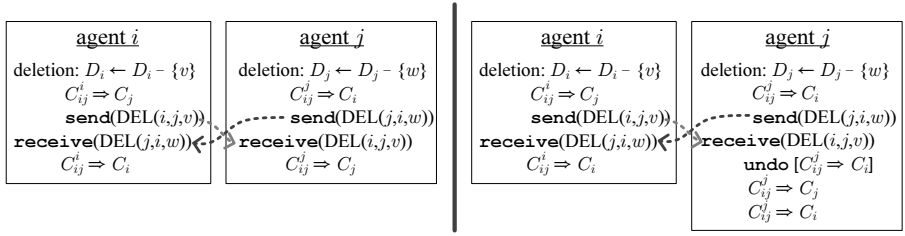


**Fig. 5.** Left: issue by two simultaneous deletions Right: proposed solution when maintaining AC

$C_i$  represents *i* performs projection from  $C_{ij}$  to  $C_i$  on *i*'s copy and  $C_{ij}^j \Rightarrow C_i$  represents *j* performs projection from  $C_{ij}$  to  $C_i$  on *j*'s copy. If value  $v$  is deleted from  $D_i$  and value  $w$  from  $D_j$  simultaneously, both *i* and *j* will process these deletions (which imply each agent projecting from  $C_{ij}$  to each other) and they will send DEL messages to each other (Fig. 5 Left). If *i* is the preferred agent, upon receipt of the DEL message from *j*, it performs  $C_{ij}^i \Rightarrow C_i$  and updates  $C_{ij}^i$ . However, when *j* receives the DEL message from *i*, if *j* realizes that it has done more projections  $C_{ij}^j \Rightarrow C_i$  than the agent *i*, then it has to undo some of these projections, until both have done the same number of projections. The proposed solution appear in Fig. 5 Right. The same ordering of operations in both agents is achieved as follows. Agent *i* keeps a counter  $ACC_{j \rightarrow i}$  to record the number of projections  $C_{ij}^i \Rightarrow C_i$  (and  $DACC_{j \rightarrow i}$  to record the number of extensions from agent *j* to *i* in FDAC/MFDAC cases). These counter and stack are stored in the copy of each class of subproblems. Agent *j* keeps a stack  $P_{j \rightarrow i}^j$  that records each projection operation  $C_{ij}^j \Rightarrow C_i$ . The operations of the Undo Mechanism on  $C_{ij}$  between agents *i* and *j* for AC and MAC are:

Agent *i*:

- When there is a value deletion, perform projection  $C_{ij}^i \Rightarrow C_j$ . Attach  $ACC_{j \rightarrow i}^i$  in a DEL message and send it to *j*. Then, reset  $ACC_{j \rightarrow i}^i$  to zero.
- When *i* receives a DEL message from *j*, perform projection  $C_{ij}^i \Rightarrow C_i$  and increment  $ACC_{j \rightarrow i}^i$  by 1.

Agent *j*:

- When there is a value deletion, perform projection  $C_{ij}^j \Rightarrow C_i$ . Push this projection in the stack  $P_{j \rightarrow i}^j$ . Send the DEL message to *i*.
- When *j* receives a DEL message from *i*, pop and undo  $|P_{j \rightarrow i}^j| - ACC_{j \rightarrow i}^i$  number of projection records from the stack  $P_{j \rightarrow i}^j$ , where  $|P_{j \rightarrow i}^j|$  is the size of the stack  $P_{j \rightarrow i}^j$ , and clear the stack. Then, the DEL message is processed, projecting  $C_{ij}^j \Rightarrow C_j$ . If there is at least one pop/undo performed, then perform projection  $C_{ij}^j \Rightarrow C_i$ .

To maintain FDAC between two constrained agents *i* and *j*, DAC is maintained in one direction (e.g. *j* to *i*) and AC in the other (e.g. *i* to *j*). In FDAC, preference should be given to agent *i* if AC is enforced from  $C_{ij}$  to  $C_j$  since the enforcement of DAC from *j* to *i* is ensured under the assumption that *i* is AC w.r.t.  $C_{ij}$  [10] (in AC, any agent *i* or *j* may be preferred). Due to space limits, we skip the details for FDAC and MFDAC.



### 3.7 Optimality and Termination

Enforcing MAC and MFDAC during BnB-ADOPT<sup>+</sup> search maintains the optimality and termination properties of BnB-ADOPT<sup>+</sup>, as we see next.

Projections and extensions to maintain MAC and MFDAC are done on a copy of the cost functions. In this way, the search process is based on the unmodified original copy of the cost functions. The only changes with respect to the BnB-ADOPT<sup>+</sup> operations come from the fact that inconsistent values discovered by local consistency enforcement are removed from the domain of agents.

Termination is justified as follows. BnB-ADOPT<sup>+</sup> always terminates [6,9] and the only change that BnB-ADOPT<sup>+</sup>-MAC introduces is AC enforcement after variable assignments. AC enforcement terminates, because the number of agents involved is finite and their domains are also finite. When enforcing AC in a particular subproblem, after a finite amount of time all subproblem variables become AC (possibly after some value deletions) reaching a fixpoint.

Optimality is justified as follows. In the case of unconditional deletions, deleted values are suboptimal values which will not be present in the optimal solution, so it is completely legal to remove them. In the case of conditional deletions, deleted values are values proved inconsistent conditioned to the current assignment of ancestor agents. They are properly restored using a reinitialization mechanism when the assignments of ancestors change. Operation is as follows. An agent may change its assigned value, selecting another one from its domain, only after it receives a VALUE or COST message. Reinitialization is done whenever an agent receives a VALUE or COST message and there is context change. Thus, reinitialization is guaranteed to be performed before any agent changes its value, so that no obsolete value deletions will be considered. Then, in both cases all solutions potentially optimal are visited. Next we detail these operations, showing they do not affect optimality and termination.

In MAC (both unconditional and conditional deletions), we perform projections over the cost functions (projections from binary to unary cost functions, and from unary to  $C_\phi$ ). Projection is an equivalence preserving transformation [11]. Its application maintains the optimum cost and the set of optimal solutions. In our approach (distributed context), we assure identical copies of any binary cost function in the two involved agents: cost projections are performed in the same order in the two agents (Section 3.6). Therefore, costs cannot be duplicated when projections are performed inside each agent (equivalence is preserved) or when costs are propagated to other agents. Since each agent contributes to  $C_\phi$  projecting on its unary cost functions, we can conclude that projections of different agents into  $C_\phi$  does not duplicate costs. Proving that a value  $a$  of variable  $x_i$  is not NC involves its unary cost  $C_i(a)$  and  $C_\phi$ . Since we have seen that, neither  $C_i(a)$  nor  $C_\phi$  contains duplicated costs, the NC detection is correct and  $a$ 's deletion is legal. Because of the NC definition, the first found optimal solution can never be pruned, since the cost of their values will never reach  $\top$ .

In the case of conditional deletions, the reinitialization mechanism (Section 3.3) ensures the correctness of values deletions in different copies. For each copy, projections and deletions are performed conditioned to the ancestor assignments. For example, in *Copy*(0), projections are performed contemplating no previous assignments, and only unconditional deletions are detected; in *Copy*(1), inconsistent values are discovered

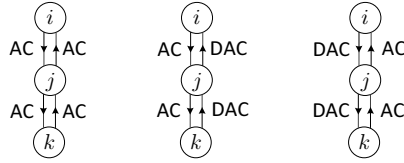


**Fig. 6.** Directions of enforcing AC/FDAC consistencies

and deleted conditioned to the first-level ancestor's assignment; in *Copy(2)*, inconsistent values are deleted conditioned to the first and second level agent's assignment, and so on. Each time an agent of depth  $d$  changes its variable assignment, the *Copy(d)* of descendant agents are restored to *Copy(d - 1)*. In this way, all modifications that were performed according to the previous variable assignment are undone.

Regarding values pruned by backtracking messages, the justification of its correctness is as follows. When an empty domain is found in *Copy(d)* in one agent, we have discovered that the current assignment of the ancestor at depth  $d$  is inconsistent, and so it must be removed. This is implemented by sending a BTK message to that ancestor. Note that only BTK messages containing a compatible context are accepted in the ancestor. In this way, it is assured that the ancestor agent changes its value if the empty domain of the descendant agent was generated considering a compatible context. Otherwise, either the descendant or the ancestor is missing one or several messages that will properly update their contexts. Upon receipt of these messages, proper actions, depending on the missing messages, will be taken by the ancestor/descendant.

Regarding MFDAC, in addition to projections, we have to take into account extensions, another equivalence preserving transformation [10]. Our approach (distributed case) is correct, since each agent can extend its own unary costs only. So no cost duplication may occur. The process is done in such a way that the copies of any binary cost function are kept identical in the two involved agents. From this point on, only projections are done, and arguments from previous paragraphs apply.

## 4 Experimental Results

We evaluate the efficiency of BnB-ADOPT<sup>+</sup>-MAC/MFDAC (abbrev as MAC/MFDAC) by comparing to BnB-ADOPT<sup>+</sup>-AC/FDAC (abbrev as AC/FDAC). For AC and MAC algorithms, AC is enforced in both directions of each binary cost function. The direction of DAC enforcement matters in FDAC and MFDAC algorithms. Fig. 6 shows the direction of AC and DAC enforcement between agents, where  $i$  ( $j$ ) is the parent or pseudo-parent of  $j$  ( $k$ ). For FDAC algorithm, we use the direction as shown in Fig. 6 Middle. DAC is enforced bottom-up so that the unary costs are pushed upward so as to hopefully increase the opportunities of pruning more values in upper agents (pruning values in upper agents is more preferred because BnB-ADOPT<sup>+</sup> is a depth-first search algorithm). For MFDAC, we evaluate both directions: MFDAC1 uses the direction shown in Fig. 6 Middle and MFDAC2 uses the direction in Fig. 6 Right. We evaluate both because of the possible tradeoff between backtracking and direct pruning in upper agents. With MFDAC1, unary costs will float upward and increase the opportunities of pruning values directly in upper agents. However, MFDAC2 pushes the unary

costs downward and increases the opportunities of reaching empty domains in lower agents, which can possibly increase the pruning opportunities in upper agents.

Our simulator runs in cycles, during which every agent reads its incoming messages, performs computation and sends its outgoing messages. Without delays, a message sent in a cycle is delivered in the next cycle. To make a more realistic evaluation, a random delay of  $[0, 50]$  cycles is introduced to each message in our experiments. Besides, we have an extensive number of instances over three benchmarks. Since AC is too slow to generate results for hard or large-scale problems within a reasonable time, we set a  $2 \times 10^8$  NCCCs limit in our simulator. One can expect that setting the NCCCs limit is to our *disadvantage* since MAC/MFDAC can improve even more on harder or larger-scale problems (normally taking bigger effort to solve but some of these problems are skipped because of the NCCCs limit). Three measures of performance are thus compared: (1) the number of messages to evaluate the communication cost, (2) the number of non-concurrent constraint checks (NCCCs) to evaluate the computation effort, and (3) the number of instances that can be solved within the  $2 \times 10^8$  NCCCs limit to evaluate the general efficiency of each algorithm. In addition, we assume that each randomly delayed cycle costs 100 NCCCs and it is counted in the total NCCCs accordingly.

We test our algorithms on three sets of benchmarks: binary random DCOPs [8], Soft Graph Coloring Problems (SGCP) and Radio Link Frequency Assignment Problem (RLFAP) [13]. We run 50 instances for each parameter setting. Results are reported in Tables 2, 3 and 4. The columns show (from left to right) the problem, algorithm, the number of instances that can be solved within limit, the number of commonly solved instances (the number of messages and NCCCs are averaged over this number), total number of messages, number of VALUE, COST, DEL, BTK and UCO messages, and NCCCs. The best results for each measure are highlighted in bold.

Binary random DCOPs [8] are characterized by  $\langle n, d, p \rangle$ , where  $n$  is the number of variables,  $d$  is the domain size and  $p$  is the network connectivity. We have generated random DCOP instances:  $\langle n = 10, d = 10, p \in \{0.3, 0.4, 0.5, 0.6\} \rangle$ . Costs are selected from a uniform cost distribution. Following Guitierrez and Meseguer [8], two types of binary cost functions are used, small and large. Small cost functions randomly extract costs from the set  $\{0, \dots, 10\}$  while large ones randomly extract costs from the set  $\{0, \dots, 1000\}$ . The proportion of large cost functions is 1/4 of the total number of cost functions. Results are reported in Table 2.

Soft Graph Coloring Problems are the softened version of graph coloring problems by allowing the inequalities to return costs from the violation measure  $M^2 - |v_i - v_j|^2$ , where  $M$  is the maximum domain size,  $v_i$  and  $v_j$  are the values of agent  $i$  and  $j$  respectively. Each SGCP is also characterized by  $\langle n, d, p \rangle$ , where  $n$  is the number of variables,  $d$  is the domain size and  $p$  is the network connectivity. We evaluate four sets of instances:  $\langle n \in \{6, 7, 8, 9\}, d = 8, p = 0.4 \rangle$ . Results are shown in Table 3.

We generate the Radio Link Frequency Assignment Problems according to two small but hard CELAR sub-instances [13], which are extracted from CELAR6. All instances are generated with parameters  $\langle i, n, d \rangle$ , where  $i$  is the index of the CELAR sub-instances,  $n$  is an even number of links, and  $d$  is an even number of allowed frequencies. For each instance, we randomly extract a sequence of  $n$  links from the corresponding CELAR sub-instance and fix a domain of  $d$  frequencies. If two links are restricted not to take

**Table 2.** Random DCOPs

| $p$ | Algorithm | #instances solved within NCCCs limit | Avg. over (common instances) | #Msgs         | #VALUE | #COST  | #DEL  | #BTK  | #UCO  | NCCCs             |
|-----|-----------|--------------------------------------|------------------------------|---------------|--------|--------|-------|-------|-------|-------------------|
| 0.3 | AC        | <b>50</b>                            | 50                           | 6,802         | 1,619  | 5,099  | 59    | 0     | 0     | 5,622,762         |
|     | FDAC      | <b>50</b>                            |                              | 4,645         | 1,062  | 3,389  | 117   | 0     | 53    | 3,857,078         |
|     | MAC       | <b>50</b>                            |                              | 5,610         | 1,124  | 3,569  | 760   | 134   | 0     | 4,203,119         |
|     | MFDAC1    | <b>50</b>                            |                              | <b>3,656</b>  | 726    | 2,346  | 338   | 13    | 184   | <b>2,738,511</b>  |
|     | MFDAC2    | <b>50</b>                            |                              | 5,036         | 923    | 2,911  | 495   | 249   | 435   | 3,511,191         |
| 0.4 | AC        | 47                                   | 47                           | 56,632        | 11,581 | 44,946 | 79    | 0     | 0     | 42,210,453        |
|     | FDAC      | 48                                   |                              | 39,560        | 8,043  | 31,188 | 195   | 0     | 105   | 29,477,148        |
|     | MAC       | <b>50</b>                            |                              | 36,309        | 6,692  | 25,564 | 2,399 | 1,628 | 0     | 24,845,040        |
|     | MFDAC1    | <b>50</b>                            |                              | <b>28,493</b> | 5,271  | 20,541 | 1,430 | 236   | 967   | <b>19,236,541</b> |
|     | MFDAC2    | <b>50</b>                            |                              | 29,814        | 5,116  | 20,255 | 1,523 | 1,441 | 1,451 | 19,434,413        |
| 0.5 | AC        | 35                                   | 34                           | 106,194       | 20,796 | 85,260 | 106   | 0     | 0     | 78,603,224        |
|     | FDAC      | 38                                   |                              | 75,074        | 14,412 | 60,231 | 247   | 0     | 152   | 55,129,851        |
|     | MAC       | 43                                   |                              | 63,571        | 11,238 | 46,279 | 2,694 | 3,329 | 0     | 43,949,687        |
|     | MFDAC1    | 44                                   |                              | <b>54,564</b> | 9,490  | 39,791 | 2,926 | 286   | 2,018 | <b>36,699,194</b> |
|     | MFDAC2    | <b>46</b>                            |                              | 57,150        | 9,497  | 39,535 | 2,245 | 3,651 | 2,191 | 37,488,828        |
| 0.6 | AC        | 9                                    | 9                            | 124,222       | 26,839 | 97,268 | 86    | 0     | 0     | 91,145,921        |
|     | FDAC      | 16                                   |                              | 90,850        | 14,867 | 55,465 | 277   | 0     | 211   | 51,437,525        |
|     | MAC       | 20                                   |                              | 47,586        | 8,973  | 35,153 | 2,143 | 1,288 | 0     | 34,059,166        |
|     | MFDAC1    | <b>24</b>                            |                              | <b>37,697</b> | 6,883  | 27,900 | 1,141 | 463   | 1,255 | <b>27,122,566</b> |
|     | MFDAC2    | 20                                   |                              | 45,988        | 8,093  | 31,699 | 2,011 | 2,047 | 2,109 | 31,074,814        |

**Table 3.** Soft Graph Coloring Problems

| $n$ | Algorithm | #instances solved within NCCCs limit | Avg. over (common instances) | #Msgs         | #VALUE | #COST  | #DEL  | #BTK | #UCO  | NCCCs             |
|-----|-----------|--------------------------------------|------------------------------|---------------|--------|--------|-------|------|-------|-------------------|
| 6   | AC        | <b>50</b>                            | 50                           | 459           | 123    | 321    | 8     | 0    | 0     | 572,082           |
|     | FDAC      | <b>50</b>                            |                              | 376           | 91     | 240    | 29    | 0    | 7     | 438,002           |
|     | MAC       | <b>50</b>                            |                              | 358           | 81     | 190    | 67    | 11   | 0     | 361,607           |
|     | MFDAC1    | <b>50</b>                            |                              | <b>287</b>    | 51     | 127    | 54    | 9    | 30    | <b>248,106</b>    |
|     | MFDAC2    | <b>50</b>                            |                              | 367           | 71     | 161    | 70    | 17   | 40    | 333,807           |
| 7   | AC        | <b>50</b>                            | 50                           | 1,349         | 370    | 961    | 9     | 0    | 0     | 1,534,451         |
|     | FDAC      | <b>50</b>                            |                              | 875           | 225    | 594    | 37    | 0    | 8     | 974,678           |
|     | MAC       | <b>50</b>                            |                              | 888           | 213    | 507    | 143   | 14   | 0     | 841,000           |
|     | MFDAC1    | <b>50</b>                            |                              | <b>628</b>    | 127    | 314    | 95    | 20   | 51    | <b>521,659</b>    |
|     | MFDAC2    | <b>50</b>                            |                              | 883           | 185    | 437    | 143   | 27   | 81    | 733,084           |
| 8   | AC        | <b>50</b>                            | 50                           | 8,611         | 2,072  | 6,523  | 5     | 0    | 0     | 8,562,373         |
|     | FDAC      | <b>50</b>                            |                              | 5,764         | 1,359  | 4,354  | 29    | 0    | 11    | 5,727,394         |
|     | MAC       | <b>50</b>                            |                              | 4,955         | 1,044  | 3,166  | 625   | 109  | 0     | 4,261,463         |
|     | MFDAC1    | <b>50</b>                            |                              | <b>4,359</b>  | 905    | 2,942  | 287   | 61   | 138   | 3,799,575         |
|     | MFDAC2    | <b>50</b>                            |                              | 4,695         | 857    | 2,615  | 613   | 163  | 437   | <b>3,553,383</b>  |
| 9   | AC        | 46                                   | 46                           | 39,199        | 8,659  | 30,525 | 3     | 0    | 0     | 32,353,604        |
|     | FDAC      | 46                                   |                              | 30,189        | 6,580  | 23,559 | 23    | 0    | 14    | 24,858,245        |
|     | MAC       | <b>47</b>                            |                              | 23,164        | 4,554  | 15,882 | 2,545 | 170  | 0     | 17,448,119        |
|     | MFDAC1    | <b>47</b>                            |                              | 25,738        | 5,265  | 19,124 | 795   | 69   | 453   | 19,829,021        |
|     | MFDAC2    | <b>47</b>                            |                              | <b>20,219</b> | 3,547  | 12,624 | 2,081 | 493  | 1,461 | <b>13,863,427</b> |

frequencies  $f_i$  and  $f_j$  with distance less than  $t$ , we measure the costs of interference using a binary cost function  $\max(0, t - |f_i - f_j|)$ . Results of evaluating three sets of instances, A(0, 10, 12), B(1, 6, 6), and C(1, 6, 8), are reported in Table 4.

As we see in Tables 2, 3 and 4, MAC, MFDAC1 and MFDAC2 substantially further reduce the total number of messages and NCCCs, and be able to solve the same number or more instances within the NCCCs limit over all three benchmarks. Moreover, MAC outperforms FDAC in almost all cases even when MAC is maintaining a weaker form of consistency than FDAC. Although our methods introduce overhead, i.e., increase in the number of DEL, BTK and UCO messages, the reduction in the number of VALUE

**Table 4.** Radio Link Frequency Assignment Problems

|   | Algorithm | #instances<br>solved within<br>NCCCs limit | Avg. over<br>(common<br>instances) | #Msgs         | #VALUE | #COST  | #DEL  | #BTK  | #UCO  | NCCCs             |
|---|-----------|--------------------------------------------|------------------------------------|---------------|--------|--------|-------|-------|-------|-------------------|
| A | AC        | 50                                         | 50                                 | 28,837        | 5,064  | 23,751 | 0     | 0     | 0     | 24,522,945        |
|   | FDAC      | 50                                         |                                    | 28,894        | 5,069  | 23,790 | 0     | 0     | 13    | 24,621,897        |
|   | MAC       | 50                                         |                                    | 22,840        | ,3802  | 16,540 | 1,954 | 522   | 0     | 17,447,513        |
|   | MFDAC1    | 50                                         |                                    | <b>18,054</b> | 2,937  | 12,000 | 1,606 | 378   | 1,090 | 13,051,121        |
|   | MFDAC2    | 50                                         |                                    | 19,233        | 2,888  | 11,711 | 1,861 | 1,250 | 1,501 | <b>12,845,773</b> |
| B | AC        | 21                                         | 21                                 | 56,943        | 10,466 | 46,455 | 11    | 0     | 0     | 67,658,716        |
|   | FDAC      | 21                                         |                                    | 57,964        | 10,635 | 47,267 | 39    | 0     | 9     | 69,091,598        |
|   | MAC       | 50                                         |                                    | 29,120        | 4,930  | 21,521 | 1,061 | 1,596 | 0     | 37,861,737        |
|   | MFDAC1    | 50                                         |                                    | <b>18,080</b> | 3,228  | 13,900 | 403   | 433   | 100   | <b>20,881,354</b> |
|   | MFDAC2    | 50                                         |                                    | 25,430        | 4,490  | 19,489 | 541   | 702   | 197   | 2,937,7041        |
| C | AC        | 18                                         | 18                                 | 29,385        | 5,505  | 23,853 | 15    | 0     | 0     | 34,158,516        |
|   | FDAC      | 18                                         |                                    | 31,133        | 5,814  | 25,250 | 47    | 0     | 9     | 36,259,302        |
|   | MAC       | 50                                         |                                    | 13,914        | 2,464  | 10,787 | 297   | 356   | 0     | 15,890,040        |
|   | MFDAC1    | 50                                         |                                    | <b>11,964</b> | 2,183  | 9,394  | 177   | 123   | 71    | <b>14,067,760</b> |
|   | MFDAC2    | 50                                         |                                    | 13,454        | 2,431  | 10,496 | 220   | 207   | 89    | 15,731,062        |

and COST messages (and thus search space) outweighs the overhead. Therefore, we conclude that maintaining soft arc consistencies during search is beneficial.

We also observe that the improvement of MFDAC over AC and FDAC in random DCOPs increases as constraint density increases. More constraints in the problem implies more pruning opportunities and thus substantial smaller search space. Similar observations cannot be concluded for Soft Graph Coloring and Radio Link Frequency Assignment Problems since these problems have particular problem structures affecting the efficiency and power of consistency enforcement.

To compare the different directions of DAC enforcement, we can see MFDAC1 outperforms MFDAC2 in some instances while MFDAC2 outperforms MFDAC1 in others. For random DCOPs and Radio Link Frequency Assignment Problem, MFDAC1 performs the best in almost all instances. However, for Soft Graph Coloring Problem, MFDAC2 performs better for instances with  $n = 9$  and MFDAC1 performs better on another three sets of instances. From these results we can see that the directions of DAC enforcement can affect the efficiency and the effects are problem-specific.

## 5 Conclusion

In this paper, we propose methods to maintaining soft arc consistencies in every subproblem during search. In order to preserve the asynchronicities of search and consistency enforcement, we propose to include extra copies (a small number) of variable domains and cost functions. Besides, we minimize the induced overhead caused by backtracking and undoing assignments and deletions by attaching information in the existing messages rather than creating new ones. We present the issues and solutions for maintaining consistencies in subproblems and ensure their correctness: (i) reinitializing variables' domains and cost functions after context changes in subproblems to ensure the search algorithm would not search on values using obsolete value pruning information, (ii) backtracking when an agent arrives at the empty domain within a subproblem so as to prune the value in upper agents which could not lead to an optimal solution, (iii) transferring deletions from subproblems to further subproblems to avoid redundant messages, and (iv) asynchronous methods to ensure identical cost functions copies in

different agents by ensuring the ordering of consistency operations between every two agents. Our experimental results show that our methods can substantially further reduce the communication and computation efforts compared to BnB-ADOPT<sup>+</sup>-AC/FDAC, which only consider unconditional deletions. These results allow us to consider the proposed methods as important steps to maintain consistencies in every subproblems asynchronously during search and improve the efficiency of optimal DCOP solving. As a future work, we may go further to maintain the even stronger Existential Directional Arc Consistency (EDAC) [14] during distributed and asynchronous search, but preserving privacy is a concern [15]. The study of how DAC enforcement directions affect efficiency and the possible heuristics for such ordering is a worthwhile direction.

## References

1. Maheswaran, R.T., Tambe, M., Bowring, E., Pearce, J.P., Varakantham, P.: Taking DCOP to the real world: Efficient complete solutions for distributed multi-event scheduling. In: Proc. AAMAS 2004, pp. 310–317 (2004)
2. Jain, M., Taylor, M., Tambe, M., Yokoo, M.: DCOPs meet the realworld: exploring unknown reward matrices with applications to mobile sensor networks. In: Proc. IJCAI 2009, pp. 181–186 (2009)
3. Junges, R., Bazzan, A.L.C.: Evaluating the performance of DCOP algorithms in a real world, dynamic problem. In: Proc. AAMAS 2008, pp. 599–606 (2008)
4. Modi, P.J., Shen, W.M., Tambe, M., Yokoo, M.: ADOPT: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence* 161, 149–180 (2005)
5. Petcu, A., Faltings, B.: A scalable method for multiagent constraint optimization. In: Proc. IJCAI 2005, pp. 266–271 (2005)
6. Yeoh, W., Felner, A., Koenig, S.: BnB-ADOPT: an asynchronous branch-and-bound DCOP algorithm. *JAIR* 38, 85–133 (2010)
7. Chechotka, A., Sycara, K.: No-commitment branch and bound search for distributed constraint optimization. In: Proc. AAMAS 2006, pp. 1427–1429 (2006)
8. Gutierrez, P., Meseguer, P.: BnB-ADOPT<sup>+</sup> with several soft arc consistency levels. In: Proc. ECAI 2010, pp. 67–72 (2010)
9. Gutierrez, P., Meseguer, P.: Saving redundant messages in BnB-ADOPT. In: Proc. AAAI 2010, pp. 1259–1260 (2010)
10. Larrosa, J., Schiex, T.: In the quest of the best form of local consistency for weighted CSP. In: Proc. IJCAI 2003, pp. 239–244 (2003)
11. Larrosa, J., Schiex, T.: Solving weighted csp by maintaining arc consistency. *Artificial Intelligence* 159(1), 1–26 (2004)
12. Gutierrez, P., Meseguer, P.: Improving BnB-ADOPT<sup>+</sup>-AC. In: Proc. AAMAS 2012, pp. 273–280 (2012)
13. Cabon, B., Givry, S.D., Lobjois, L., Fcabor, L.L., Schiex, T.: Radio link frequency assignment. *Constraints* 4, 79–89 (1999)
14. de Givry, S., Heras, F., Larrosa, J., Zytnicki, M.: Existential arc consistency: getting closer to full arc consistency in weighted CSPs. In: Proc. IJCAI 2005, pp. 84–89 (2005)
15. Gutierrez, P., Meseguer, P.: Enforcing soft local consistency on multiple representations for DCOP solving. In: CP 2010, Workshop: Preferences and Soft Constraints, pp. 98–113 (2010)

# Solving String Constraints: The Case for Constraint Programming

Jun He<sup>1,2</sup>, Pierre Flener<sup>1</sup>, Justin Pearson<sup>1</sup>, and Wei Ming Zhang<sup>2</sup>

<sup>1</sup> Uppsala University, Department of Information Technology, Uppsala, Sweden

<sup>2</sup> National University of Defense Technology,

School of Information System and Management, Changsha, Hunan, China

{Jun.He,Pierre.Flener,Justin.Pearson}@it.uu.se, wmmzhang@nudt.edu.cn

**Abstract.** We improve an existing propagator for the context-free grammar constraint and demonstrate experimentally the practicality of the resulting propagator. The underlying technique could be applied to other existing propagators for this constraint. We argue that constraint programming solvers are more suitable than existing solvers for verification tools that have to solve string constraints, as they have a rich tradition of constraints for membership in formal languages.

## 1 Introduction

For constraint programming (CP) languages, user-level extensibility has been an important goal for over a decade. Global constraints for formal languages are promising for this purpose. The REGULAR constraint [16] requires a sequence of decision variables to belong to a regular language, specified by a deterministic finite automaton (DFA) or a regular expression; the AUTOMATON constraint [2] takes a DFA with counters. The CFG constraint [17,20] requires a sequence of decision variables to belong to a context-free language, specified by a context-free grammar (CFG). For many applications, the length  $n$  of a sequence constrained to belong to some formal language is known in advance. Since every fixed-size language is finite and hence regular, the need for a CFG constraint in such applications depends on the grammar and the complexities of the propagators. It takes  $O(n|A|)$  time to achieve generalised arc consistency (GAC) for a REGULAR constraint with an automaton  $A$ , but  $O(n^3|G|)$  time for a CFG constraint with a grammar  $G$ . In [12], the authors introduce a reformulation of a grammar into an automaton for a fixed length  $n$ , and show that this reformulation is preferable if the resulting automaton is not huge. However, their reformulation itself needs a CFG propagator to achieve domain consistency at the root of the search tree so that the resulting automaton is smaller. In [7], the authors introduce a forklift scheduling problem, where there is no tractable reformulation of a grammar into an automaton as the size of the resulting automaton is exponential in  $n$ . Hence, a CFG propagator is necessary in this case. To the best of our knowledge, no CP solver includes the CFG constraint.

In the analysis, testing, and verification of string-manipulating programs, constraints on sequences (strings) of decision variables arise. Kiežun *et al.* [14] argue

that custom string solvers should not be designed any more, for sustainability reasons, since powerful off-the-shelf solvers are available: their tool, HAMPI, translates a REGULAR or CFG constraint on a fixed-size string into bit-vector constraints so as to solve them using the SMT solver STP [6], much more efficiently than three custom tools and even up to three orders of magnitude faster than the SAT-based CFGAnalyzer tool [1]. The solver KALUZA [19] handles constraints over multiple string variables, unlike the restriction of HAMPI to one such variable, and it also generates bit-vector constraints that are passed to STP. Fu *et al.* [5] argue that it is important to model regular replacement operations, which are not supported by HAMPI and KALUZA, and introduce the custom string solver SUSHI, which models string constraints via automata instead of a bit-vector encoding. So the question arises whether the formal language constraints of CP are competitive with HAMPI, KALUZA, and SUSHI.

In this paper, we revisit the CFG constraint and make the following *contributions*:

- We improve the CFG propagator of [11], which improves the one of [20], by exploiting an idea of [14] for reformulating a grammar into a regular expression for a fixed string length. We conjecture that this idea also applies to the CFG propagators of [7,13,17,18]. (Section 3)
- We implement our CFG propagator for the GECODE [8] open-source CP solver, and demonstrate experimentally its practicality. (Sections 4.1 to 4.3)
- We show that the CP solver GECODE with our CFG propagator (or even its ancestor [11]) systematically beats HAMPI and KALUZA, by up to four orders of magnitude, on HAMPI’s benchmark (Section 4.3). We show that GECODE with the built-in REGULAR propagator systematically beats KALUZA and SUSHI, by a factor up to 130, on SUSHI’s benchmark (Section 4.4).

## 2 Background

We first give some background material on grammars (e.g., see [10]).

### 2.1 Context-Free Grammars

A *CFG* is a tuple  $\langle \Sigma, N, P, S \rangle$ , where  $\Sigma$  is the alphabet and any value  $v \in \Sigma$  is called a terminal,  $N$  is the finite set of non-terminals,  $P \subseteq N \times (\Sigma \cup N)^*$  is the finite set of productions, and  $S \in N$  is the start non-terminal. A CFG is said to be in *Chomsky normal form* (CNF) iff  $P \subseteq N \times (\Sigma \cup N^2)$ . Every CFG can be converted into an equivalent grammar in CNF.

*Example 1.* Consider the CFG  $G_B = \langle \Sigma, N, P, S \rangle$ , where  $\Sigma = \{\ell, r\}$ ,  $N = \{S\}$ , and  $P = \{S \rightarrow \ell r, S \rightarrow SS, S \rightarrow \ell S r\}$ . It defines a language of correctly bracketed expressions (e.g.,  $\ell r \ell r$  and  $\ell \ell r r$ ), with ‘ $\ell$ ’ denoting the left bracket and ‘ $r$ ’ the right one. Its CNF is  $G'_B = \langle \Sigma, N', P', S \rangle$ , where  $N' = \{L, M, R, S\}$  and  $P' = \{S \rightarrow LR, S \rightarrow SS, S \rightarrow MR, M \rightarrow LS, L \rightarrow \ell, R \rightarrow r\}$ .



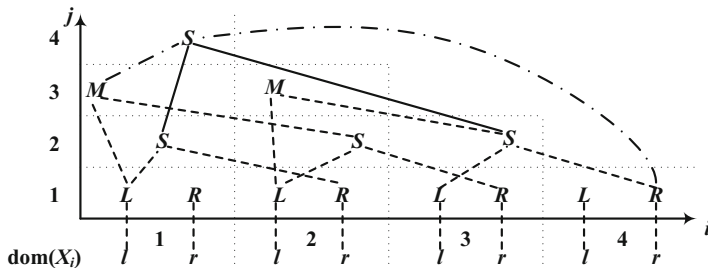


**Fig. 1.** The CYK-based propagator parses a sequence  $\langle X_1, \dots, X_4 \rangle$  of  $n = 4$  decision variables with the same domain  $\{\ell, r\}$  under the CFG  $G'_B$  of Example 1

The Cocke-Younger-Kasami (CYK) algorithm is a parser for CFGs in CNF. We describe it for a sequence of decision variables instead of values. Given a CFG  $\langle \Sigma, N, P, S \rangle$  in CNF and a sequence  $\langle X_1, \dots, X_n \rangle$  of  $n$  decision variables, the CYK parser computes a table  $V$ , where  $V_{i,j}$  (with  $1 \leq j \leq n$  and  $1 \leq i \leq n+1-j$ ) is the set of non-terminals (or at most the start non-terminal  $S$  for  $i = 1$  and  $j = n$ ) that can be parsed using a sequence of  $j$  values in the domains of  $X_i$  to  $X_{i+j-1}$  respectively, using dynamic programming:

$$V_{i,j} = \begin{cases} \{W \mid (W \rightarrow b) \in P \wedge b \in \text{dom}(X_i)\} & \text{if } j = 1 \\ \bigcup_{k=1}^{j-1} \left\{ W \mid \begin{array}{l} (W \rightarrow YZ) \in P \wedge (j < n \vee W = S) \\ \wedge Y \in V_{i,k} \wedge Z \in V_{i+k,j-k} \end{array} \right\} & \text{otherwise} \end{cases}$$

For example, Figure 1 gives the CYK table  $V$  when parsing a sequence  $X$  of 4 decision variables with the same domain  $\{\ell, r\}$  under the grammar  $G'_B$  of Example 1. We have  $V_{1,1} = \{L, R\}$  and  $V_{1,4} = \{S\}$ . Note that we use  $\text{dom}(X_i)$  to denote the domain of the decision variable  $X_i$ .

Given a word  $w \in \Sigma^n$ , let  $w_i$  (with  $1 \leq i \leq n$ ) denote the letter at position  $i$  of  $w$ . If all decision variables  $X_i$  have  $\text{dom}(X_i) = \{w_i\}$ , then  $w$  is *accepted* by  $G$  iff  $V_{1,n} = \{S\}$ .

## 2.2 The CFG Constraint

The CFG constraint is defined as  $\text{CFG}(X, G)$ , where  $X$  is a sequence of decision variables and  $G$  is a grammar. An assignment  $w$  to  $X$  is a solution iff  $w$  is a word accepted by  $G$ .

Given a CFG  $G = \langle \Sigma, N, P, S \rangle$  in CNF and a sequence  $X$  of  $n$  variables, let  $|G| = \sum_{p \in P} |p|$  be the size of  $G$ , and  $|p|$  the number of (non-)terminals in the production  $p$ . The propagator of [11] achieves GAC for the  $\text{CFG}(X, G)$  constraint in  $O(n^3 |G|)$  time with  $O(n^2 |G|)$  space, which is better than the propositional satisfiability (SAT) based propagator of [18], which decomposes and achieves GAC for the CFG constraint in  $O(n^3 |G|)$  time and space. More recently, another SAT-based propagator is introduced in [7], which works similarly to the propagator of [11] and outperforms the propagator of [18].

In this paper, we use the propagator of [11] *as an example* to show how to improve a CFG propagator. We conjecture that the same idea can be used to improve the propagators of [7,13,17,18].

To describe elegantly the propagator of [11] and ours (given in Section 3), we first introduce a novel concept. Informally, given a non-terminal  $W$  in  $V_{i,j}$  of the CYK table, a *low support* for this  $W$ , namely  $(W \rightarrow YZ, k)$ , denotes that two non-terminals lower down in  $V$ , namely  $Y$  in  $V_{i,k}$  and  $Z$  in  $V_{i+k,j-k}$ , support the existence of  $W$  in  $V_{i,j}$ ; and this low support corresponds to two *high supports*, namely  $(W \rightarrow YZ, j)$  of  $Y$  in  $V_{i,k}$  and  $Z$  in  $V_{i+k,j-k}$ . Formally:

**Definition 1 (Support).** *For any  $1 < j \leq n$ ,  $1 \leq i \leq n+1-j$ , and non-terminal  $W$  in  $V_{i,j}$  of the CYK table, the set  $\overline{\text{LS}}_{i,j}(W) = \{(W \rightarrow YZ, k) \mid (W \rightarrow YZ) \in P \wedge 0 < k < j\}$  is called the candidate low-support set for  $W$  in  $V_{i,j}$ . The set  $\text{LS}_{i,j}(W) = \{(W \rightarrow YZ, k) \in \overline{\text{LS}}_{i,j}(W) \mid Y \in V_{i,k} \wedge Z \in V_{i+k,j-k}\}$  is called the low-support set for  $W$  in  $V_{i,j}$ . For  $j = 1$  and any  $1 \leq i \leq n$  and non-terminal  $W$  in  $V_{i,1}$ , we define  $\overline{\text{LS}}_{i,1}(W) = \{(W \rightarrow b) \in P\}$  and  $\text{LS}_{i,1}(W) = \{(W \rightarrow b) \in \overline{\text{LS}}_{i,1}(W) \mid b \in \text{dom}(X_i)\}$ .*

*For any  $1 \leq j < n$ ,  $1 \leq i \leq n+1-j$ , and non-terminal  $W$  in  $V_{i,j}$  of the CYK table, the set  $\overline{\text{HS}}_{i,j}(W) = \{(Y \rightarrow QZ, k) \mid (Y \rightarrow QZ) \in P \wedge (W = Q \vee W = Z) \wedge j < k \leq n\}$  is called the candidate high-support set of  $W$  in  $V_{i,j}$ . The set  $\text{HS}_{i,j}(W) = \{(Y \rightarrow QZ, k) \in \overline{\text{HS}}_{i,j}(W) \mid (W = Q \wedge Y \in V_{i,k} \wedge Z \in V_{i+j,k-j}) \vee (W = Z \wedge Y \in V_{i-j,k} \wedge Q \in V_{i-j,k-j})\}$  is called the high-support set of  $W$  in  $V_{i,j}$ . For any  $1 \leq i \leq n$  and value  $b$  in  $\text{dom}(X_i)$ , we define  $\overline{\text{HS}}_i(b) = \{(W \rightarrow b) \in P\}$  and  $\text{HS}_i(b) = \{(W \rightarrow b) \in \overline{\text{HS}}_i(b) \mid b \in \text{dom}(X_i)\}$ .  $\square$*

For example, in the CYK table  $V$  of Figure 1,  $\overline{\text{LS}}_{1,4}(S) = \{S \rightarrow LR, S \rightarrow SS, S \rightarrow MR\} \times \{1, 2, 3\}$  has 9 candidate low supports; only 2 thereof are low supports for non-terminal  $S$  in  $V_{1,4}$ , namely  $(S \rightarrow SS, 2)$  (depicted by the solid arcs), and  $(S \rightarrow MR, 3)$  (depicted by the dash-dotted arcs). The low support  $(S \rightarrow SS, 2)$  for  $S$  in  $V_{1,4}$  denotes that it is supported by  $S$  in  $V_{1,2}$  and  $V_{3,2}$ , hence the low support corresponds to 2 high supports, namely  $(S \rightarrow SS, 4)$  of  $S$  in  $V_{1,2}$  and  $V_{3,2}$ .

The propagator of [11] achieves GAC for the  $\text{CFG}(X, G)$  constraint as follows: (1) The CYK parser computes the table  $V$ . (2) A bottom-up process finds the *first* low support in every  $\overline{\text{LS}}_{i,j}(W)$ . A top-down process finds the *first* high support in every  $\overline{\text{HS}}_{i,j}(W)$ . All non-terminals  $W$  with no support are removed from  $V$ . (3) The *first* high support in every  $\overline{\text{HS}}_i(b)$  is found, and all values  $b$  in any  $\text{dom}(X_i)$  with no high support are removed from  $\text{dom}(X_i)$ . When a support is found in steps 2 and 3, its position in the candidate support set is recorded. When a support is lost as the domains shrink, the next support is to be found starting *after* the previous support in the candidate support set. The propagator is incremental, and explores all candidate supports at most once.

### 3 An Improved Propagator

Inspired by [14], we present, verify, and analyse an improved version of the propagator of [11] for the CFG constraint.

### 3.1 Motivation and Theoretical Foundation

There are two dependent opportunities for improving the propagator of [11].

*Encoding the Support Sets Space-Efficiently.* The propagator of [11] explores *all* candidate supports once in the worst case, hence its time complexity is bounded by

$$|\overline{\text{LS}}| + |\overline{\text{HS}}| = \sum_{j=1}^n \sum_{i=1}^{n+1-j} \sum_{W \in V_{i,j}} |\overline{\text{LS}}_{i,j}(W)| + |\overline{\text{HS}}_{i,j}(W)| = O(n^3 |G|). \text{ If we can}$$

make the propagator run on the small support sets instead of the large candidate support sets, then the propagator probably runs faster. Consider that  $\overline{\text{LS}}_{i,j}(W) \supseteq \text{LS}_{i,j}(W)$  and  $\overline{\text{HS}}_{i,j}(W) \supseteq \text{HS}_{i,j}(W)$  (from Definition 1), and that the gaps may be huge. For example in Figure 1,  $\text{LS}_{1,4}(S) = \{(S \rightarrow SS, 2), (S \rightarrow MR, 3)\}$  is of size 2, while  $\overline{\text{LS}}_{1,4}(S) = \{S \rightarrow LR, S \rightarrow SS, S \rightarrow MR\} \times \{1, 2, 3\}$  is of size 9;  $\text{HS}_{2,1}(R) = \{(S \rightarrow LR, 2)\}$  is of size 1, while  $\overline{\text{HS}}_{2,1}(R) = \{S \rightarrow LR, S \rightarrow MR\} \times \{2, 3, 4\}$  is of size 6. However, the challenge is to avoid having to pay with space what we save in time.

Given a CFG  $G = \langle \Sigma, N, P, S \rangle$  in CNF and  $n$  decision variables, Kadioğlu and Sellmann [11] claim that storing all support sets takes  $O(n^3 |G|)$  space, which is expensive. Their propagator thus runs on the large *candidate* support sets, which can be encoded very space-efficiently. Two sets  $\text{Out}(W) = \{(W \rightarrow YZ) \in P\}$  and  $\text{In}(W) = \{(Y \rightarrow QZ) \in P \mid W = Q \vee W = Z\}$  are computed for any  $W \in N$ , so that  $\overline{\text{LS}}_{i,j}(W) = \text{Out}(W) \times \{1, \dots, j-1\}$  and  $\overline{\text{HS}}_{i,j}(W) = \text{In}(W) \times \{j+1, \dots, n\}$ . For any  $j$ , the sets  $\{1, \dots, j-1\}$  and  $\{j+1, \dots, n\}$  need not be stored. Hence encoding all candidate support sets only takes  $O(|G|)$  space by storing all  $\text{Out}(W)$  and  $\text{In}(W)$ . As it takes  $O(n^2 |G|)$  space to store the CYK table  $V$ , the overall space complexity is  $O(n^2 |G|)$ .

However, we *can* decrease the space requirement for encoding all low-support sets and a superset of all high-support sets (given in Theorem 2 below) from  $O(n^3 |G|)$  to  $O(n^2 |G|)$ , which is the *same* as the one needed to store the CYK table  $V$ , by using an idea of [14] for reformulating a grammar into a regular expression for a fixed string length  $n$ . In that reformulation, a regular expression is obtained by using the *same* domains:  $\text{dom}(X_i) = \Sigma$  for *all*  $1 \leq i \leq n$ . A regular expression  $E_{1,j}$  for the sub-sequence  $\langle X_1, \dots, X_j \rangle$  is computed and stored as a template for every  $1 \leq j \leq n$ , and then the regular expression  $E_{i,j}$  for the sub-sequence  $\langle X_i, \dots, X_{i+j-1} \rangle$  turns out to be equal to  $E_{1,j}$  for every  $1 < i \leq n+1-j$ . Similarly, in Figure 1, we find that  $V_{i,j} = V_{1,j}$  and every non-terminal in  $V_{i,j}$  has the same low supports as in  $V_{1,j}$ . For example,  $V_{3,2} = V_{2,2} = V_{1,2} = \{S\}$  and  $\text{LS}_{3,2}(S) = \text{LS}_{2,2}(S) = \text{LS}_{1,2}(S) = \{(S \rightarrow LR, 1)\}$ . Based on this observation, we give the following theorem (we show in Section 3.2 how to lift the same-domain restriction):

**Theorem 1.** *Given a CFG  $G = \langle \Sigma, N, P, S \rangle$  in CNF and a sequence  $\langle X_1, \dots, X_n \rangle$  of  $n$  decision variables, if all  $X_i$  have the same domain, then for any  $1 \leq j \leq n$  and  $1 < i \leq n+1-j$ :*

1.  $V_{i,j} = V_{1,j}$
2.  $\forall W \in V_{i,j} : \text{LS}_{i,j}(W) = \text{LS}_{1,j}(W)$

**Proof:** We prove claim 1 by complete induction on  $j$ .

(Base:  $j = 1$ ) For any non-terminal  $W$ , we have  $W \in V_{i,1}$  iff there exists a production  $(W \rightarrow b) \in P$  such that  $b \in \text{dom}(X_i)$ . As  $\text{dom}(X_i) = \text{dom}(X_1)$ , we have  $W \in V_{i,1}$  iff  $W \in V_{1,1}$ .

(Step:  $1 < j \leq n$ ) For any  $1 \leq j' < j$ , the induction hypothesis is  $V_{i,j'} = V_{1,j'}$  for any  $1 < i$ . We want to prove  $V_{i,j} = V_{1,j}$  for any  $1 < i \leq n+1-j$ . For any non-terminal  $W$ , we have  $W \in V_{i,j}$  iff there exists a production  $(W \rightarrow YZ) \in P$  and  $1 \leq k < j$  such that  $Y \in V_{i,k}$  and  $Z \in V_{i+k,j-k}$ . As  $V_{i,k} = V_{1,k}$  and  $V_{i+k,j-k} = V_{1,j-k} = V_{1+k,j-k}$ , we have  $W \in V_{i,j}$  iff  $W \in V_{1,j}$ .

Using this, claim 2 follows from Definition 1.  $\square$

The next theorem enables a space-efficient encoding of the support sets (again, we show in Section 3.2 how to lift the same-domain restriction).

**Theorem 2.** *Given a CFG  $G = \langle \Sigma, N, P, S \rangle$  in CNF and a sequence  $\langle X_1, \dots, X_n \rangle$  of  $n$  decision variables, if all  $X_i$  have the same domain, then it takes  $O(n^2 |G|)$  space to encode the CYK table  $V$  and all support sets.*

**Proof:** For any  $1 \leq j \leq n$  and  $1 < i \leq n+1-j$ :

By Theorem 1, we have  $V_{i,j} = V_{1,j}$ . Hence we obtain the whole CYK table  $V$  by storing all  $V_{1,j}$  in  $\sum_{j=1}^n |V_{1,j}| = O(n|N|) = O(n|G|)$  space, as  $|G| = \sum_{p \in P} |p| > |N|$ .

By Theorem 1, we have  $\text{LS}_{i,j}(W) = \text{LS}_{1,j}(W)$ . Hence we obtain all low supports by storing all  $\text{LS}_{1,j}(W)$  in  $\sum_{j=1}^n \sum_{W \in V_{1,j}} |\text{LS}_{1,j}(W)| \leq \sum_{j=1}^n |P \times \{k \mid 1 \leq k < j\}|$   
 $= \sum_{j=1}^n O(n|G|) = O(n^2 |G|)$  space, as  $|G| = \sum_{p \in P} |p| > |P|$  and each low support takes constant space.

Considering the high-support set  $\text{HS}_{i,j}(W)$ , it takes  $O(n^3 |G|)$  space to store all  $\text{HS}_{i,j}(W)$  as  $\text{HS}_{i,j}(W) = \text{HS}_{1,j}(W)$  is not true for all  $1 \leq j \leq n$  and  $i > 1$ . For example in Figure 1, we have  $\text{HS}_{2,1}(R) = \{(S \rightarrow LR, 2)\}$ , while  $\text{HS}_{1,1}(R) = \emptyset$ . To save space, we compute the set  $\text{HS}'_{i,j}(W) = \bigcup_{k=1}^{n+1-j} \text{HS}_{k,j}(W)$  instead of  $\text{HS}_{i,j}(W)$ , as we can encode  $\text{HS}'_{i,j}(W)$  efficiently. Note that we still have  $\text{HS}'_{i,j}(W) \subseteq \overline{\text{HS}}_{i,j}(W)$  as  $\overline{\text{HS}}_{i,j}(W) = \overline{\text{HS}}_{1,j}(W)$  (its formulation in Definition 1 is independent of  $i$ ) and  $\text{HS}'_{i,j}(W) = \bigcup_{k=1}^{n+1-j} \text{HS}_{k,j}(W) \subseteq \bigcup_{k=1}^{n+1-j} \overline{\text{HS}}_{k,j}(W) = \overline{\text{HS}}_{1,j}(W)$ . Hence we obtain all  $\text{HS}'_{i,j}(W)$  by computing and storing all  $\text{HS}'_{1,j}(W)$  in  $O(n^2 |G|)$  space, as  $\text{HS}'_{i,j}(W) = \text{HS}'_{1,j}(W)$  and  $\sum_{j=1}^n \sum_{W \in V_{1,j}} |\text{HS}'_{1,j}(W)| \leq$

$2 \sum_{j=1}^n \sum_{W \in V_{1,j}} |\text{LS}_{1,j}(W)| = O(n^2 |G|)$  (the definition of  $\text{HS}'_{i,j}(W)$  is independent of  $i$  and one low support corresponds to at most two high supports).

Hence we can encode the CYK table  $V$ , all  $\text{LS}_{i,j}(W)$ , and all  $\text{HS}'_{i,j}(W)$  in  $O(n^2|G|)$  space.  $\square$

Using Theorem 2, it *is* practical to make the propagator run on  $\text{LS}_{i,j}(W)$  and  $\text{HS}'_{i,j}(W)$ , which are subsets of the candidate support sets, with  $O(n^2|G|)$  space. Although Theorem 2 requires all  $\text{dom}(X_i)$  to be the *same*, this is not an obstacle in practice, as shown in Section 3.2 below. Note that  $|\text{LS}| + |\text{HS}'|$  and  $|\overline{\text{LS}}| + |\overline{\text{HS}}|$  are asymptotically the same (as shown in Section 3.3 below), hence we cannot improve the propagator of [11] asymptotically.

*Counting the Supports.* For each non-terminal  $W$  in the CYK table, the propagator of [11], which is based on the arc-consistency (AC) algorithm AC-6 [3], decides whether  $W$  has low and high supports by exhibiting two actual supports (one low and one high). However, this is not necessary. We can simply *count* the supports for  $W$  as in AC-4 [15], and then just decrease the counter by one when a support is lost. Although Bessi re [3] shows that AC-4 is worse than AC-6 for *binary* CSPs given *extensionally* because initialising the counters is expensive, in our case initialisation is much cheaper because we have  $|\text{LS}_{i,j}(W)| = |\text{LS}_{1,j}(W)|$  initially when using our efficient encoding of the support sets. However, by using counters, we do not need complex data structures and operations to trace which non-terminal in the CYK table is currently supporting and supported by which non-terminal(s), as in [11]. Indeed, our experiments (omitted for space reasons, see Appendix C of [9]) show that counting with our efficient encoding of the support sets works better (up to 12 times) than using *only* the latter, which already works better (up to 20 times) than the propagator of [11].

### 3.2 Description and Proof of Our Propagator

Consider a CFG  $G = \langle \Sigma, N, P, S \rangle$  in CNF and a sequence  $X = \langle X_1, \dots, X_n \rangle$  of  $n$  decision variables. We introduce a propagator for the  $\text{CFG}(X, G)$  constraint using the AC-4 framework, which computes all supports and counts them when *posting* the constraint (see Algorithm 1), and then *only* decreases the support counters during propagation (see Algorithm 2), without changing the support sets. Hence, to satisfy the condition of Theorem 2, we *only* need to make all decision variables *temporarily* take the same domain when *posting* the constraint. Our propagator has *no* limitation on the initial domains of the decision variables, as we will show how our propagator lifts the temporary restriction at no asymptotic overhead.

Let  $C_{i,j}^{\text{LS}}(W)$  (or  $C_{i,j}^{\text{HS}}(W)$ ) denote the number of low (or high) supports for (or of) a non-terminal  $W$  in  $V_{i,j}$  of the CYK table during propagation. Similarly, let  $C_i^{\text{LS}}(b)$  (or  $C_i^{\text{HS}}(b)$ ) denote the number of low (or high) supports for (or of) a terminal  $b$  in  $\text{dom}(X_i)$ . Note that every (non-)terminal has two counters and there is no sharing of counters between any two (non-)terminals, as the counters will be changed *independently* during propagation. Using Theorem 2, Algorithm 1 posts the  $\text{CFG}(X, G)$  constraint, encodes the CYK table

and support sets, counts the supports, and achieves GAC. Given all propagator state variables, which are also shared by Algorithm 2, initialised so that  $V_{1,j} = \text{LS}_{1,j}(W) = \text{HS}'_{1,j}(W) = \emptyset$  and  $C_{i,j}^{\text{HS}}(W) = 0$  (lines 2 to 4), Algorithm 1 works as follows. First, it constructs a virtual domain  $\text{Dom}' = \bigcup_{i=1}^n \text{dom}(X_i)$  (line 5), and uses it to post the  $\text{CFG}(X, G)$  constraint, hence the condition of Theorem 2 is satisfied as all domains are now the same. Using the virtual domain may introduce extra solutions, and we show in the last step how to avoid this. Second, it uses a bottom-up process (lines 6 to 17) based on the CYK parser to compute all  $V_{1,j}$ ,  $\text{LS}_{1,j}(W)$ ,  $\text{HS}'_{1,j}(W)$ , and  $C_{i,j}^{\text{LS}}(W)$ . Note that we only need to compute  $V_{1,j}$  by Theorem 1, and any reference to  $V_{i,j}$  is replaced by  $V_{1,j}$ . The same holds for  $\text{LS}_{i,j}(W)$ , and  $\text{HS}'_{i,j}(W)$  (by its definition independently of  $i$  in Theorem 2). If the start non-terminal  $S$  is not in  $V_{1,n}$ , then it fails (line 18; no word from the current domains is accepted by  $G$ , hence no solution exists). Third, it uses a top-down process (lines 19 to 25) to compute all  $C_{i,j}^{\text{HS}}(W)$ . Fourth, it removes all values with no high support from the domains (lines 26 to 28). Finally, it constructs a set  $\Delta$  of all variable-value pairs that are not in the domains of  $X$  but in the virtual domain (line 29), and calls the function `filterFromUpdate` (in Algorithm 2, discussed next) to re-establish GAC after removing all such variable-value pairs (line 30). Hence the side effect of using the virtual domain is lifted; we show in Section 3.3 that calling the function `filterFromUpdate` does not increase the asymptotic complexity of Algorithm 1.

Given a set  $\Delta$  of all recently filtered variable-value pairs by other propagators or a branching of the search tree, the function `filterFromUpdate` in Algorithm 2 incrementally re-establishes GAC for the  $\text{CFG}(X, G)$  constraint as follows. First, it creates two arrays  $Q_{\text{LS}}$  and  $Q_{\text{HS}}$  of initially empty queues (line 2), with  $Q_{\text{LS}}[j]$  (or  $Q_{\text{HS}}[j]$ ) storing all non-terminals  $W$  in the  $j$ -th row of the CYK table with no low (or high) supports due to the domain changes  $\Delta$ . Second, it iterates over all removed values in  $\Delta$ , decreasing the counter  $C_{i,1}^{\text{LS}}(W)$  for all non-terminals  $W$  in the bottom row supported by a removed value, and adding all  $W$  with no low support to the queue  $Q_{\text{LS}}[1]$  (lines 3 to 7). Third, a bottom-up process (lines 8 to 11) calls the procedure `rmNoLS` handling all  $W$  in the queue  $Q_{\text{LS}}[j]$ . Given a non-terminal  $W$  with no low support, `rmNoLS` iterates over each high support of  $W$ , decreasing the three counters related with this lost high support, and enqueueing  $Q_{\text{LS}}[j]$  (or  $Q_{\text{HS}}[j]$ ) whenever a low (or high) support counter is zero (lines 22 to 33). Fourth, a top-down process (lines 12 to 14) calls the procedure `rmNoHS` (omitted for space reasons, see Appendix C of [9]), which works similarly to `rmNoLS`, handling all  $W$  in the queue  $Q_{\text{HS}}[j]$ . Finally, it removes inconsistent values (with no high support) from the domains of  $X$  (lines 15 to 20), and reaches a fixpoint (line 21). Note that Algorithm 2 is a direct usage of the AC-4 framework. Once Algorithm 1 initialises the support sets and counters correctly, the correctness of Algorithm 2 is guaranteed by the AC-4 framework.

**Theorem 3.** *Our propagator achieves GAC for  $\text{CFG}(X, G)$ .*

**Proof:** A value is removed by our propagator from the domains of  $X$  iff it has no high supports, as with the propagator of [11]. Hence the two propagators are equivalent. The result follows from Theorem 2 on page 132 of [11].  $\square$

**Algorithm 1.** An improved propagator for the  $\text{CFG}(X, G)$  constraint, where  $X = \langle X_1, \dots, X_n \rangle$  is a sequence of  $n$  decision variables and  $G = \langle \Sigma, N, P, S \rangle$  is a CFG in CNF

---

```

1: function post( $\text{CFG}(X, G)$ )
2: for all  $W \in N$  and  $j \leftarrow 1$  to  $n$  do
3:    $V_{1,j} \leftarrow \text{LS}_{1,j}(W) \leftarrow \text{HS}'_{1,j}(W) \leftarrow \emptyset$ 
4:   for all  $i \leftarrow 1$  to  $n+1-j$  do  $C_{i,j}^{\text{HS}}(W) \leftarrow 0$ 
5:    $\text{Dom}' \leftarrow \bigcup_{i=1}^n \text{dom}(X_i)$ 
6:    $V_{1,1} \leftarrow \{W \mid (W \rightarrow b) \in P \wedge b \in \text{Dom}'\}$ 
7:    $\text{LS}_{1,1}(W) \leftarrow \{W \rightarrow b \mid (W \rightarrow b) \in P \wedge b \in \text{Dom}'\}$ 
8:    $\text{HS}'_1(b) \leftarrow \{W \rightarrow b \mid (W \rightarrow b) \in P \wedge b \in \text{Dom}'\}$ 
9:   for all  $j \leftarrow 2$  to  $n$  do
10:    for all  $(W \rightarrow YZ) \in P$  and  $k \leftarrow 1$  to  $j-1$  do
11:      if  $Y \in V_{1,k} \wedge Z \in V_{1,j-k} \wedge (j < n \vee W = S)$  then
12:         $V_{1,j} \leftarrow V_{1,j} \cup \{W\}$ 
13:         $\text{LS}_{1,j}(W) \leftarrow \text{LS}_{1,j}(W) \cup \{(W \rightarrow YZ, k)\}$ 
14:         $\text{HS}'_{1,k}(Y) \leftarrow \text{HS}'_{1,k}(Y) \cup \{(W \rightarrow YZ, j)\}$ 
15:         $\text{HS}'_{1,j-k}(Z) \leftarrow \text{HS}'_{1,j-k}(Z) \cup \{(W \rightarrow YZ, j)\}$ 
16:    for all  $j \leftarrow 1$  to  $n$  and  $W \in V_{1,j}$  do
17:      for all  $i \leftarrow 1$  to  $n+1-j$  do  $C_{i,j}^{\text{LS}}(W) \leftarrow |\text{LS}_{1,j}(W)|$ 
18:    if  $S \notin V_{1,n}$  then return failed
19:    for all  $j \leftarrow n$  to  $2$ ,  $W \in V_{1,j}$ , and  $i \leftarrow 1$  to  $n+1-j$  do
20:      if  $C_{i,j}^{\text{HS}}(W) > 0 \vee j = n$  then
21:        for all  $(W \rightarrow YZ, k) \in \text{LS}_{1,j}(W)$  do
22:           $C_{i,k}^{\text{HS}}(Y) ++$ ;  $C_{i+k,j-k}^{\text{HS}}(Z) ++$ 
23:    for all  $W \in V_{1,1}$  and  $i \leftarrow 1$  to  $n$  do
24:      if  $C_{i,1}^{\text{HS}}(W) > 0$  then
25:        for all  $(W \rightarrow b) \in \text{LS}_{1,1}(W)$  do  $C_i^{\text{HS}}(b) ++$ 
26:    for all  $i \leftarrow 1$  to  $n$  and  $b \in \text{dom}(X_i)$  do
27:      if  $C_i^{\text{HS}}(b) = 0$  then  $\text{dom}(X_i) \leftarrow \text{dom}(X_i) \setminus \{b\}$ 
28:      if  $\text{dom}(X_i) = \emptyset$  then return failed
29:     $\Delta \leftarrow \{(X_i, b) \mid X_i \in X \wedge b \in \text{Dom}' \setminus \text{dom}(X_i)\}$ 
30:    return filterFromUpdate( $\text{CFG}(X, G), \Delta$ )

```

---

### 3.3 Complexity Analysis

We first investigate the worst-case *time* complexity of our propagator for the  $\text{CFG}(X, G)$  constraint. In Algorithm 1, the time complexity of lines 2 to 29 is dominated by lines 19 to 25, which explore at most all low-support

sets  $\text{LS}_{i,j}(W)$  (referenced as  $\text{LS}_{1,j}(W)$ ) once in  $\sum_{j=1}^n \sum_{i=1}^{n+1-j} \sum_{W \in V_{1,j}} |\text{LS}_{1,j}(W)| <$

$n \sum_{j=1}^n \sum_{W \in V_{1,j}} |\text{LS}_{1,j}(W)| = O(n^3 |G|)$  time, by Theorem 2; line 30 calls the

function filterFromUpdate in Algorithm 2, which explores once all  $\text{LS}_{i,j}(W)$

**Algorithm 2.** Given a set  $\Delta$  of domain changes, the function `filterFromUpdate` incrementally re-establishes GAC for the  $\text{CFG}(X, G)$  constraint on a sequence  $X = \langle X_1, \dots, X_n \rangle$  of  $n$  decision variables.

---

```

1: function filterFromUpdate( $\text{CFG}(X, G), \Delta$ )
2: for all  $j \leftarrow 1$  to  $n$  do  $Q_{\text{LS}}[j] \leftarrow []$ ;  $Q_{\text{HS}}[j] \leftarrow []$ 
3: for all  $(X_i, b) \in \Delta$  do
4:    $C_i^{\text{HS}}(b) \leftarrow 0$ 
5:   for all  $(W \rightarrow b) \in \text{HS}'_1(b)$  do
6:     if  $C_{i,1}^{\text{LS}}(W) > 0$  then
7:       if  $--C_{i,1}^{\text{LS}}(W) = 0$  then  $Q_{\text{LS}}[1].\text{enqueue}((W, i))$ 
8: for all  $j \leftarrow 1$  to  $n$  do
9:   while  $Q_{\text{LS}}[j] \neq []$  do
10:    if  $j = n$  then return failed as  $S_{1,n}$  has no low support
11:     $(W, i) \leftarrow Q_{\text{LS}}[j].\text{dequeue}()$ ;  $\text{rmNoLS}(W, i, j, Q_{\text{LS}}, Q_{\text{HS}})$ 
12: for all  $j \leftarrow n - 1$  to  $2$  do
13:   while  $Q_{\text{HS}}[j] \neq []$  do
14:     $(W, i) \leftarrow Q_{\text{HS}}[j].\text{dequeue}()$ ;  $\text{rmNoHS}(W, i, j, Q_{\text{LS}}, Q_{\text{HS}})$ 
15: while  $Q_{\text{HS}}[1] \neq []$  do
16:    $(W, i) \leftarrow Q_{\text{HS}}[1].\text{dequeue}()$ 
17:   for all  $(W \rightarrow b) \in \text{LS}_{1,1}(W)$  do
18:     if  $C_i^{\text{HS}}(b) > 0$  then
19:       if  $--C_i^{\text{HS}}(b) = 0$  then  $\text{dom}(X_i) \leftarrow \text{dom}(X_i) \setminus \{b\}$ 
20:       if  $\text{dom}(X_i) = \emptyset$  then return failed
21: return at-fixpoint

22: procedure rmNoLS( $W, i, j, Q_{\text{LS}}, Q_{\text{HS}}$ )
23: if  $C_{i,j}^{\text{HS}}(W) > 0$  then
24:   for all  $(F \rightarrow YZ, k) \in \text{HS}'_{1,j}(W)$  do
25:     if  $W = Y \wedge F \in V_{i,k} \wedge Z \in V_{i+j,k-j}$  then
26:        $(i_F, j_F, B, i_B, j_B) \leftarrow (i, k, Z, i+j, k-j)$ 
27:     else if  $W = Z \wedge F \in V_{i-j,k} \wedge Y \in V_{i-j,k-j}$  then
28:        $(i_F, j_F, B, i_B, j_B) \leftarrow (i-j, k, Y, i-j, k-j)$ 
29:     else skip lines 30 to 33
30:     if  $C_{i_F,j_F}^{\text{LS}}(F) > 0 \wedge C_{i_B,j_B}^{\text{HS}}(B) > 0 \wedge C_{i,j}^{\text{HS}}(W) > 0$  then
31:       if  $--C_{i_F,j_F}^{\text{LS}}(F) = 0$  then  $Q_{\text{LS}}[j_F].\text{enqueue}((F, i_F))$ 
32:       if  $--C_{i_B,j_B}^{\text{HS}}(B) = 0$  then  $Q_{\text{HS}}[j_B].\text{enqueue}((B, i_B))$ 
33:       if  $--C_{i,j}^{\text{HS}}(W) = 0$  then  $Q_{\text{HS}}[j].\text{enqueue}((W, i));$  return

```

---

and  $\text{HS}'_{i,j}(W)$  in the worst case, hence takes  $\sum_{j=1}^n \sum_{i=1}^{n+1-j} \sum_{W \in V_{i,j}} |\text{LS}_{1,j}(W)| + |\text{HS}'_{1,j}(W)| = O(n^3 |G|)$  time, for similar reasons. Hence there is no asymptotic overhead by line 30, and the overall time complexity is  $O(n^3 |G|)$ .

Consider now the worst-case *space* complexity of our propagator. By Theorem 2, encoding the CYK table  $V$ , all  $\text{LS}_{i,j}(W)$ , and all  $\text{HS}'_{i,j}(W)$  takes

$O(n^2 |G|)$  space. There are  $\sum_{j=1}^n \sum_{i=1}^{n+1-j} |V_{i,j}| = \sum_{j=1}^n \sum_{i=1}^{n+1-j} |V_{1,j}| = O(n^2 |N|) =$



$O(n^2 |G|)$  non-terminals in  $V$ , hence storing the support counters for all non-terminals takes  $O(n^2 |G|)$  space. There are  $n |\Sigma|$  terminals in the domains, hence storing the support counters for all terminals takes  $O(n |G|)$  space. The two arrays  $Q_{LS}$  and  $Q_{HS}$  of queues contain at most all non-terminals in  $V$ , hence take  $O(n^2 |G|)$  space. The overall space complexity is thus  $O(n^2 |G|)$ .

Although our propagator has the same *worst*-case time and space complexity as the one of [11], which is probably optimal anyway, our experiments below show that our propagator systematically beats it in practice (by up to two orders of magnitude), which might be confirmed by an *average*-case complexity analysis.

## 4 Experimental Evaluation

We now demonstrate the speed-up of our CFG propagator over its ancestor [11]. We implemented our propagator and the one of [11] in GECODE [8]. Katsirelos *et al.* [12] show how to reformulate a CFG into a DFA for a fixed length, as propagation for the REGULAR constraint is much cheaper than for CFG. This reformulation needs a propagator for the CFG constraint to shrink the initial domains of all decision variables to achieve GAC for all constraints at the root of the search tree, so that the obtained DFA is smaller. Hence this reformulation also benefits from a more efficient propagator for the CFG constraint.

Note that Sections 4.3 and 4.4 demonstrate that CP outperforms some state-of-the-art solvers from the verification literature by orders of magnitude on their own benchmarks. Our experimental results show that those benchmarks are trivial, *but these benchmarks were not known to be trivial* before this paper, and we have neither discarded any non-trivial benchmarks (of HAMPI and SUSHI) nor included the benchmarks that were in the meantime known to be trivial.

We use the GECODE built-in REGULAR propagator. We ran the experiments of Sections 4.1, 4.2, and 4.3 under GECODE 3.7.3, HAMPI 20120213, and Ubuntu Linux 11.10 on 1.8 GHz Intel Core 2 Duo with 3GB RAM; and we ran the experiment of Section 4.4 under GECODE 3.7.3, KALUZA, SUSHI 2.0, and Ubuntu Linux 10.04 with 1GB RAM in Oracle VirtualBox 4.2.4 (recommended by the SUSHI developers) on the same hardware. As our chosen search heuristics do not randomise, all instances of Sections 4.1, 4.2, and 4.3 were run once. However, for Section 4.4, we ran each instance 10 times and recorded the average runtime, as the performance of the virtual machine might vary significantly.

### 4.1 A Shift Scheduling Problem

Demassez *et al.* [4] introduce a real-life shift scheduling problem for staff in a retail store. Let  $w$  be the number of workers,  $p$  the number of periods of the scheduling horizon, and  $a$  the number of work activities. The aim is to construct a  $w \times p$  matrix of values in  $[1, \dots, a + 3]$  (there are 3 non-work activities, namely break, lunch, and rest) to satisfy work regulation constraints, which can be modelled with a CFG constraint for each worker over the  $p$  periods and some global cardinality constraints (GCC).

Katsirelos *et al.* [12] model this problem as an optimisation problem, so that the reformulation of the grammar into a DFA takes only a tiny part of the runtime; they show that this optimisation problem is extremely difficult for CP-based CFG and REGULAR propagators. We are here, like [11], primarily interested in the first solution to the satisfaction version of this problem. We use the search heuristic of [11], namely selecting the second-largest value from the first decision variable with the minimum domain size in the last period with unassigned variables. HAMPI cannot handle multiple variables, while HAMPI, KALUZA, and SUSHI cannot model GCC, so we do not compare with them.

Table 1 gives our results: each row gives the instance, the search tree size, the DFA size after the reformulation of [12] of CFG into REGULAR, and the runtimes of four methods in seconds, namely our propagator (denoted by G++), the one of [11] (denoted by G), and the reformulation, using the two CFG propagators respectively (denoted by DFA<sub>G++</sub> and DFA<sub>G</sub>). We find that G++ always works much better (up to 18 times) than G; DFA<sub>G++</sub> always works much better (up to 10 times) than DFA<sub>G</sub>, as the reformulation of [12] itself needs a CFG propagator to shrink the initial domains at the root of the search tree (the reformulation, which is *instance-dependent*, is here taken on-line and takes about 85% of the total runtime) and as G++ works better than G; overall, G++ wins on 15 instances, and DFA<sub>G++</sub> wins on the other 2 instances. When solving for all or best solutions, DFA<sub>G++</sub> gradually takes over as the best method, as predicted by [12], but G++ continues to dominate G, and DFA<sub>G++</sub> decreasingly dominates DFA<sub>G</sub>, as instances get harder.

## 4.2 A Forklift Scheduling Problem

Gange and Stuckey [7] introduce a forklift scheduling problem. Let  $s$  be the number of stations,  $i$  the number of items, and  $n$  the length of the scheduling horizon. There is a unique forklift and a shipping list giving the initial and final stations of each item. The aim is to construct an array of  $n$  actions, where an action can move the forklift from a station to any other station with a cost of 3, load an item from the current station onto the top of the forklift tray with a cost of 1, unload the item from the top of the forklift tray at the current station with a cost of 1, or do nothing with a cost of 0, so that the shipping list is accomplished with a minimised cost under forklift behaviour constraints, which can be modelled with one CFG constraint and  $i$  REGULAR constraints. We use the first-fail search heuristic, namely selecting the smallest value from the first decision variable with the minimum domain size, to solve this optimisation problem. Since HAMPI, KALUZA, and SUSHI cannot solve optimisation problems, we do not compare with them.

Table 2 gives our results over the instances solvable in one CPU hour: each row specifies the instance and gives the runtimes of two methods in seconds, namely our propagator (denoted by G++) and the one of [11] (denoted by G). We find that G++ always works better (up to 5 times) than G. The reformulation of [12] of the CFG constraint into the REGULAR constraint is not suitable for this problem, as the resulting automaton is of size exponential in  $n$ .

**Table 1.** Runtimes for the shift scheduling problem

| benchmark ( $p = 96$ ) |     |     | search tree size |               |        | DFA   | runtimes of four methods in seconds |                    |                  |        |
|------------------------|-----|-----|------------------|---------------|--------|-------|-------------------------------------|--------------------|------------------|--------|
| instance               | $a$ | $w$ | #nodes           | #propagations | #fails | $ A $ | G++                                 | DFA <sub>G++</sub> | DFA <sub>G</sub> | G      |
| 1_1                    | 1   | 1   | 11               | 438           | 1      | 446   | <b>0.24</b>                         | 0.49               | 4.26             | 3.93   |
| 1_2                    | 1   | 3   | 133              | 2123          | 33     | 998   | <b>0.90</b>                         | 3.78               | 15.38            | 12.87  |
| 1_3                    | 1   | 4   | 349              | 5790          | 137    | 998   | <b>1.68</b>                         | 4.10               | 19.48            | 19.49  |
| 1_4                    | 1   | 5   | 95               | 1836          | 7      | 814   | <b>1.18</b>                         | 2.41               | 21.99            | 20.53  |
| 1_5                    | 1   | 4   | 71               | 1332          | 3      | 722   | <b>0.92</b>                         | 1.75               | 16.95            | 16.32  |
| 1_6                    | 1   | 5   | 76               | 1567          | 3      | 722   | <b>1.17</b>                         | 2.01               | 21.16            | 20.17  |
| 1_7                    | 1   | 6   | 3623             | 56635         | 1773   | 814   | 7.87                                | <b>2.97</b>        | 25.56            | 47.48  |
| 1_8                    | 1   | 2   | 57               | 1005          | 10     | 998   | <b>0.52</b>                         | 3.59               | 10.76            | 8.47   |
| 1_9                    | 1   | 1   | 19               | 460           | 1      | 630   | <b>0.22</b>                         | 0.80               | 4.41             | 3.94   |
| 1_10                   | 1   | 7   | 12699            | 209988        | 6305   | 814   | 23.31                               | <b>4.02</b>        | 30.14            | 100.95 |
| 2_1                    | 2   | 2   | 46               | 1414          | 8      | 984   | <b>0.93</b>                         | 1.69               | 16.76            | 15.97  |
| 2_5                    | 2   | 4   | 83               | 2208          | 20     | 1209  | <b>1.02</b>                         | 3.15               | 18.51            | 16.41  |
| 2_6                    | 2   | 5   | 89               | 1801          | 12     | 1207  | <b>1.35</b>                         | 2.94               | 23.03            | 21.57  |
| 2_7                    | 2   | 6   | 258              | 5847          | 104    | 944   | <b>1.97</b>                         | 2.63               | 32.22            | 32.03  |
| 2_8                    | 2   | 2   | 1046             | 28691         | 500    | 1774  | <b>2.86</b>                         | 7.75               | 23.09            | 24.09  |
| 2_9                    | 2   | 1   | 35               | 1249          | 8      | 1460  | <b>0.63</b>                         | 4.11               | 14.21            | 11.03  |
| 2_10                   | 2   | 7   | 4690             | 100007        | 2302   | 1506  | <b>7.64</b>                         | 7.82               | 43.24            | 53.90  |

**Table 2.** Runtimes for the forklift scheduling problem

| instance |     |     | runtimes in seconds |         | instance |     |     | runtimes in seconds |         |
|----------|-----|-----|---------------------|---------|----------|-----|-----|---------------------|---------|
| $s$      | $i$ | $n$ | G++                 | G       | $s$      | $i$ | $n$ | G++                 | G       |
| 3        | 4   | 15  | <b>4.35</b>         | 20.02   | 3        | 4   | 16  | <b>22.64</b>        | 103.75  |
| 3        | 4   | 17  | <b>20.98</b>        | 100.48  | 3        | 4   | 18  | <b>76.77</b>        | 382.31  |
| 3        | 4   | 19  | <b>72.66</b>        | 338.69  | 3        | 4   | 20  | <b>197.98</b>       | 1013.78 |
| 3        | 5   | 16  | <b>67.54</b>        | 297.55  | 3        | 5   | 17  | <b>81.67</b>        | 368.65  |
| 3        | 5   | 18  | <b>200.91</b>       | 1058.17 | 3        | 6   | 18  | <b>1134.58</b>      | 5008.90 |
| 4        | 5   | 17  | <b>388.92</b>       | 1631.94 | 4        | 5   | 18  | <b>819.82</b>       | 3876.87 |

### 4.3 Intersection of Two Context-Free Languages

HAMPI [14] selects a subset of 100 CFG pairs (from the benchmark of CFGAnalyzer [1]), where a string of length  $1 \leq n \leq 50$  accepted by both CFGs in each pair is to be found (8 instances are satisfiable and 92 are unsatisfiable; disjointness of two context-free languages is undecidable). The CFGs of this benchmark have 10 to 600 productions in CNF and up to 18 alphabet symbols. This problem can also be solved using tools from automata theory. On this benchmark, HAMPI beats CFGAnalyzer by a large margin. HAMPI also beats other ad hoc solvers on other benchmarks, which are too easy (HAMPI solves them in one second), hence any improvements might be subject to runtime measurement errors.

Instead of running each CFG pair 50 times with the  $n$ -th run to find a string of length  $n$  accepted by both CFGs, we search once, namely for the first solution string of length up to 50 for each pair. Given a CFG  $G = \langle \Sigma, N, P, S \rangle$ , we create a new CFG  $G' = \langle \Sigma', N', P', S' \rangle$  with  $\Sigma' = \Sigma \cup \{\#\}$  (let  $\# \notin \Sigma$  denote a dummy symbol),  $N' = N \cup \{S'\}$ , and  $P' = P \cup \{S' \rightarrow S \mid S \# \}$ . If a string  $s'$  of length

$n$  is accepted by  $G'$ , then the string  $s$  obtained by removing all ‘#’ at the end of  $s'$  has a length up to  $n$  and is accepted by  $G$ .

Given a CFG pair  $(G_1, G_2)$ , our model is  $\text{CFG}(X, G'_1) \wedge \text{CFG}(X, G'_2)$ , where  $X$  is a sequence of  $n$  decision variables with  $\text{dom}(X_i) = \Sigma'_1 \cup \Sigma'_2$ . Our search heuristic is to select the first value from the last unassigned variable. Figure 2 gives the runtimes of HAMPI and the two CFG propagators for the 55 instances where HAMPI takes at least one second. Each ‘ $\times$ ’ (or ‘+’) denotes the comparison between our propagator (or the one of [11]) and HAMPI; each ‘ $\Delta$ ’ denotes the solving time of the bit-vector solver STP. For all 100 instances, the two propagators always work much better (up to 9000 times) than HAMPI, and even always work much better than STP when the fixed-sizing of the grammar into a regular expression and the transformation into bit-vector constraints are taken off-line; our propagator always works much better (up to 250 times) than the one of [11]. As 97 instances turn out to be solvable at the root of the search tree, the reformulation of [12] of the CFG constraint into the REGULAR constraint has similar results; for the other 3 instances, our CFG propagator is 3 to 5 orders of magnitude faster (details omitted for space reasons, see Appendix C of [9]). The two CFG propagators always beat HAMPI for all  $n < 50$  (up to 380 times even with  $n = 10$ ), and whether run on the CFG pair  $(G'_1, G'_2)$  or the original pair  $(G_1, G_2)$ . We get similar speed-ups (details omitted for space reasons, see Appendix C of [9]) over 99% of the CFG pairs even with the first-fail search heuristic. Note that KALUZA uses HAMPI’s functionality to solve the CFG constraint, hence KALUZA has the same performance as HAMPI on this benchmark.

#### 4.4 Solving String Equations

Fu *et al.* [5] introduce just one benchmark of 5 string equations with a parameter  $1 \leq n \leq 37$  to demonstrate the practicality of their string solver SUSHI against KALUZA. SUSHI handles string variables of *unbounded* length. Like KALUZA, we expect a user-given parameter  $\bar{n}$  and look for the first solution string of up to  $\bar{n}$  symbols. Unlike KALUZA, which tries all lengths until  $\bar{n}$ , we allow strings to end with dummy symbols ‘#’ (as in Section 4.3) and add length constraints. For a sequence  $X = \langle X_1, \dots, X_{\bar{n}} \rangle$ , let decision variable  $n_X$  with  $\text{dom}(n_X) = \{0, \dots, \bar{n}\}$  denote the index of the right-most non-dummy symbol in  $X$ . The length constraint is  $\forall 1 \leq i \leq \bar{n} : X_i = \# \Leftrightarrow n_X < i$ . String concatenation  $X = Y + Z$  is modelled as  $n_X = n_Y + n_Z \wedge \langle X_1, \dots, X_{n_X} \rangle = \langle Y_1, \dots, Y_{n_Y}, Z_1, \dots, Z_{n_Z} \rangle$  with reification constraints. Regular language membership  $X \in L(R)$ , where  $L(R)$  denotes the language accepted by the regular expression  $R$ , is modelled as  $\text{REGULAR}(X, R\#^*)$ . We use the first-fail search heuristic. Table 3 gives the runtimes of GECODE, SUSHI, and KALUZA for equations 1 to 3 with the *hardest* setting  $n = 37$  and the KALUZA models (for a fair comparison). As KALUZA solves the equations for some  $n \leq \bar{n} < 3n$ , we *pessimistically* set  $\bar{n} = 4n$  for GECODE, and GECODE *still* beats SUSHI and KALUZA, by up to 130 times. GECODE solves our better models than the KALUZA ones of equations 4 and 5 within 0.10 seconds, beating SUSHI and KALUZA by up to 3000 times.

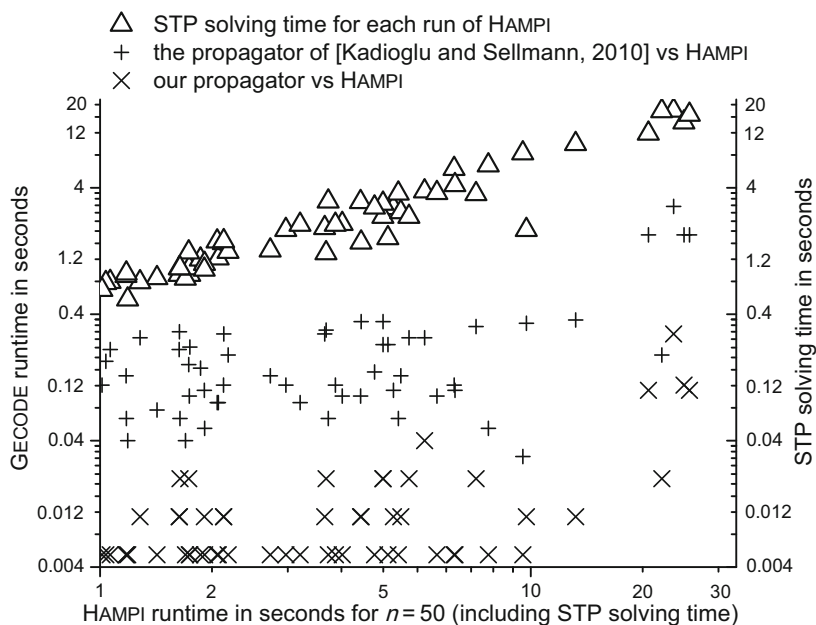


**Fig. 2.** Runtimes for the CFG-intersection problem

**Table 3.** Runtimes (in seconds) for solving string equations

| $n$ | eq1: 3 string variables |       |        | eq2: 2 string variables |       |        | eq3: 4 string variables |       |        |
|-----|-------------------------|-------|--------|-------------------------|-------|--------|-------------------------|-------|--------|
|     | GECODE                  | SUSHI | KALUZA | GECODE                  | SUSHI | KALUZA | GECODE                  | SUSHI | KALUZA |
| 37  | <b>0.15</b>             | 1.34  | 10.40  | <b>0.05</b>             | 1.82  | 3.94   | <b>0.07</b>             | 2.52  | 5.71   |

## 5 Conclusion

We argue that CP solvers are more suitable than existing solvers for verification tools that solve string constraints. Indeed, CP has a rich tradition of constraints for membership in formal languages: their propagators run directly on descriptions, such as automata and grammars, of these languages. Apparently tricky features, such as string equality or multiple string variables (with shared characters), pose no problem to CP. Future work includes designing propagators for string constraints over strings of (un)bounded length.

**Acknowledgements.** The first three authors are supported by grants 2007-6445 and 2011-6133 of the Swedish Research Council (VR), and Jun He is also supported by grant 2008-611010 of China Scholarship Council and the National University of Defence Technology of China. Many thanks to Xiang Fu, Serdar Kadioglu, George Katsirelos, Adam Kiezun, Prateek Saxena, and Guido Tack for useful discussions during the preparation of this work.

## References

1. Axelsson, R., Heljanko, K., Lange, M.: Analyzing context-free grammars using an incremental SAT solver. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfssdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 410–422. Springer, Heidelberg (2008)
2. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 107–122. Springer, Heidelberg (2004)
3. Bessière, C.: Arc-consistency and arc-consistency again. *Artificial Intelligence* 65(1), 179–190 (1994)
4. Demassey, S., Pesant, G., Rousseau, L.-M.: A cost-regular based hybrid column generation approach. *Constraints* 11(4), 315–333 (2006)
5. Fu, X., Powell, M.C., Bantegui, M., Li, C.-C.: Simple linear string constraints. *Formal Aspects of Computing* (2012), Published on-line in January 2012 and available from <http://dx.doi.org/10.1007/s00165-011-0214-3>. SUSHI is available from [http://people.hofstra.edu/Xiang\\_Fu/XiangFu/projects/SAFELI/SUSHI.php](http://people.hofstra.edu/Xiang_Fu/XiangFu/projects/SAFELI/SUSHI.php)
6. 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), STP is available from <https://sites.google.com/site/stpfastprover/>
7. Gange, G., Stuckey, P.J.: Explaining propagators for s-DNNF circuits. In: Beldiceanu, N., Jussien, N., Pinson, É. (eds.) CPAIOR 2012. LNCS, vol. 7298, pp. 195–210. Springer, Heidelberg (2012)
8. Gecode Team. Gecode: A generic constraint development environment (2006), <http://www.gecode.org/>
9. He, J.: Constraints for Membership in Formal Languages under Systematic Search and Stochastic Local Search. PhD thesis, Uppsala University, Sweden (2013), <http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-196347>
10. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley, New York (1979)
11. Kadioglu, S., Sellmann, M.: Grammar constraints. *Constraints* 15(1), 117–144 (2008); An early version is published in the Proceedings of the 23rd AAAI Conference on Artificial Intelligence in 2008
12. Katsirelos, G., Narodytska, N., Walsh, T.: Reformulating global grammar constraints. In: van Hoeve, W.-J., Hooker, J.N. (eds.) CPAIOR 2009. LNCS, vol. 5547, pp. 132–147. Springer, Heidelberg (2009)
13. Katsirelos, G., Narodytska, N., Walsh, T.: The weighted GRAMMAR constraint. *Annals of Operations Research* 184(1), 179–207 (2011), An early version is published in the Proceedings of the 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems in 2008
14. Kiežun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: HAMPI: A solver for string constraints. In: Proceedings of the 18th International Symposium on Software Testing and Analysis, Chicago, USA, July 2009, pp. 105–116. ACM Press (2009), HAMPI is available from <http://people.csail.mit.edu/akiezun/hampi/>
15. Mohr, R., Henderson, T.C.: Arc and path consistency revisited. *Artificial Intelligence* 28(2), 225–233 (1986)
16. 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)

17. Quimper, C.-G., Walsh, T.: Global grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 751–755. Springer, Heidelberg (2006)
18. Quimper, C.-G., Walsh, T.: Decomposing global grammar constraints. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 590–604. Springer, Heidelberg (2007)
19. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for javascript. In: Proceedings of the 31st IEEE Symposium on Security and Privacy, California, USA, pp. 513–528. IEEE Press (May 2010), Kaluza is available from <http://webblaze.cs.berkeley.edu/2010/kaluza/>
20. Sellmann, M.: The theory of grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 530–544. Springer, Heidelberg (2006)

# Blowing Holes in Various Aspects of Computational Problems, with Applications to Constraint Satisfaction

Peter Jonsson, Victor Lagerkvist, and Gustav Nordh

Department of Computer and Information Science, Linköping University, Sweden  
{peter.jonsson,victor.lagerkvist,gustav.nordh}@liu.se

**Abstract.** We consider methods for constructing NP-intermediate problems under the assumption that  $P \neq NP$ . We generalize Ladner's original method for obtaining NP-intermediate problems by using parameters with various characteristics. In particular, this generalization allows us to obtain new insights concerning the complexity of CSP problems. We begin by fully characterizing the problems that admit NP-intermediate subproblems for a broad and natural class of parameterizations, and extend the result further such that structural CSP restrictions based on parameters that are hard to compute (such as tree-width) are covered. Hereby we generalize a result by Grohe on width parameters and NP-intermediate problems. For studying certain classes of problems, including CSPs parameterized by constraint languages, we consider more powerful parameterizations. First, we identify a new method for obtaining constraint languages  $\Gamma$  such that  $CSP(\Gamma)$  are NP-intermediate. The sets  $\Gamma$  can have very different properties compared to previous constructions (by, for instance, Bodirsky & Grohe) and provides insights into the algebraic approach for studying the complexity of infinite-domain CSPs. Second, we prove that the propositional abduction problem parameterized by constraint languages admits NP-intermediate problems. This settles an open question posed by Nordh & Zanuttini.

## 1 Introduction

Ladner [20] explicitly constructed NP-intermediate problems (under the assumption  $P \neq NP$ ) by removing strings of certain lengths from NP-complete languages via a diagonalization technique that is colloquially known as *blowing holes in problems*. The languages constructed via blowing are unfortunately famous for being highly artificial: Arora and Barak [1] write the following.

We do not know of a natural decision problem that, assuming  $NP \neq P$ , is proven to be in  $NP \setminus P$  but not NP-complete, and there are remarkably few candidates for such languages

More natural examples are known under other complexity-theoretic assumptions. For instance, LOGCLIQUE (the problem of deciding whether an  $n$ -vertex



graph contains a clique of size  $\log n$ ) is NP-intermediate under the exponential-time hypothesis (ETH). The lack of natural NP-intermediate computational problems makes it important to investigate new classes of NP-intermediate problems and, hopefully, increase our understanding of the borderline between P and NP.

We begin (in Section 3) by presenting a diagonalization method for obtaining NP-intermediate problems, based on parameterizing decision problems in different ways. In our framework, a parameter, or a *measure function*, is simply a function  $\rho$  from the instances of some decision problem  $X$  to the non-empty subsets of  $\mathbb{N}$ . We say that such a function is *single-valued* if  $\rho(I)$  is a singleton set for every instance of  $X$ , and *multi-valued* otherwise. Depending on the parameter one obtains problems with different characteristics. Simple applications of our method include the connection between the complexity class XP and NP-intermediate problems observed by Chen et al. [9]. Even though our method is still based on diagonalization we claim that the intermediate problems obtained are qualitatively different from the ones obtained by Ladner's original method, and that they can be used for gaining new insights into the complexity of computational problems. We demonstrate this on different CSP problems in the following sections.

In Section 4, we analyze the applicability of the diagonalization method for single-valued measure functions. Under mild additional assumptions, we obtain a full understanding of when NP-intermediate problems arise when the measure function is single-valued and polynomial-time computable. Unfortunately, CSPs under structural restrictions (i.e. when considering instances with bounded width parameters) are not captured by this result since width parameters are typically not polynomial-time computable. To remedy this, we present a fairly general method for obtaining NP-intermediate problems based on structurally restricted CSPs in Section 4.2. This is a generalization of a result by Grohe [15] who has shown that, under the assumption that  $\text{FPT} \neq \text{W}[1]$ , NP-intermediate CSP problems can be obtained by restricting the tree-width of their corresponding primal graphs. Our result imply that this holds also under the weaker assumption that  $\text{P} \neq \text{NP}$  and for many different width parameters. NP-intermediate problems based on structural restrictions have also been identified by Bodirsky & Grohe [4].

Multi-valued measure functions are apparently much harder to study and a full understanding appears difficult to obtain. Despite this, multi-valued measure functions have highly useful properties and we exploit them for studying constraint satisfaction problems parameterized by constraint languages. Our first result is inspired by Bodirsky & Grohe [4] who have proved that there exists an infinite constraint language  $\Gamma$  such that  $\text{CSP}(\Gamma)$  is NP-intermediate. We extend this and prove that whenever an infinite language  $\Gamma$  does not satisfy the so called *local-global* property, i.e. when  $\text{CSP}(\Gamma) \notin \text{P}$  but  $\text{CSP}(\Gamma') \in \text{P}$  for all finite  $\Gamma' \subset \Gamma$ , then there exists a language closely related to  $\Gamma$  such that the resulting CSP problem is NP-intermediate. The only requirement is that  $\Gamma$  can be extended by certain operators  $\langle \cdot \rangle$ . We then provide two very different

extension operators. The first operator  $\langle \cdot \rangle_{pow}$  works for languages over both finite and infinite domains but gives relations of arbitrarily high arity. The second operator  $\langle \cdot \rangle_+$  is limited to idempotent languages over infinite domains but does have the advantage that the arity of any relation is only increased by a small constant factor. Together with the language  $\Gamma^\circ$  from Jonsson & Löw [18] which does not satisfy the local-global property we are thus able to identify a concrete language  $\langle \Gamma^\circ \rangle_+$  such that  $\text{CSP}(\langle \Gamma^\circ \rangle_+)$  is NP-complete,  $\text{CSP}(\Gamma') \in \text{P}$  for any finite  $\Gamma' \subset \langle \Gamma^\circ \rangle_+$ , and there exists a  $\Gamma'' \subset \langle \Gamma^\circ \rangle_+$  such that  $\text{CSP}(\Gamma'')$  is NP-intermediate. The so-called *algebraic approach* [3,6] has been very successful in studying the computational complexity of both finite- and infinite-domain CSPs. However, this approach is, to a large extent, limited to constraint languages that are finite. If one only considers tractable finite subsets of  $\langle \Gamma^\circ \rangle_+$ , we miss that there are both NP-intermediate and NP-complete problems within  $\text{CSP}(\langle \Gamma^\circ \rangle_+)$ . Hence the constraint language  $\langle \Gamma^\circ \rangle_+$  clearly shows the algebraic approach in its present shape is not able to give a full understanding of  $\text{CSP}(\langle \Gamma^\circ \rangle_+)$  and its subclasses.

Our second result (which is presented in Section 5.3) is the propositional *abduction* problem  $\text{ABD}(\Gamma)$ . This problem can be viewed as a non-monotonic extension of propositional logic and it has numerous important applications ranging from automated diagnosis, text interpretation to planning. The complexity of propositional abduction has been intensively studied from a complexity-theoretic point of view (cf. [13,23]) and the computational complexity is known for every finite Boolean constraint language  $\Gamma$  and many infinite languages [23]. In Nordh & Zanuttini [23], the question of whether such a classification is possible to obtain for infinite languages was left open. Since the abduction problem can loosely be described as a combination of the SAT and UNSAT problems, it might be expected that it, like the parameterized  $\text{SAT}(\cdot)$  problem, does not contain any NP-intermediate problems. By exploiting our diagonalization method, we present a constraint language  $\Gamma$  such that  $\text{ABD}(\Gamma)$  is NP-intermediate.

## 2 Preliminaries

Let  $\Gamma$  denote a (possibly infinite) set of finitary relations over some (possibly infinite) set  $D$ . We call  $\Gamma$  a *constraint language*. Given a relation  $R \subseteq D^k$ , we let  $\text{ar}(R) = k$ . The reader should note that we will sometimes express Boolean relations as conjunctions of Boolean clauses. The *constraint satisfaction problem* over  $\Gamma$  (abbreviated as  $\text{CSP}(\Gamma)$ ) is defined as follows.

INSTANCE: A set  $V$  of variables and a set  $C$  of constraint applications  $R(v_1, \dots, v_k)$  where  $R \in \Gamma$ ,  $k = \text{ar}(R)$ , and  $v_1, \dots, v_k \in V$ .

QUESTION: Is there a total function  $f : V \rightarrow D$  such that  $(f(v_1), \dots, f(v_k)) \in R$  for each constraint  $R(v_1, \dots, v_k)$  in  $C$ ?

For an arbitrary decision problem  $X$ , we let  $I(X)$  denote its set of instances, and  $\|I\|$  to denote the number of bits needed for representing  $I \in I(X)$ . By a *polynomial-time reduction* from problem  $X$  to problem  $X'$ , we mean a Turing reduction from  $X$  to  $X'$  that runs in time  $O(p(\|I\|))$  for some polynomial  $p$ .

**Definition 1.** Let  $X$  be a decision problem. A total and computable function  $\rho : I(X) \rightarrow 2^{\mathbb{N}} \setminus \{\emptyset\}$  is said to be a measure function.

If  $\rho(I)$  is a singleton set for every  $I \in I(X)$ , then we say that  $\rho$  is *single-valued*, and otherwise that it is *multi-valued*. We abuse notation in the first case and simply assume that  $\rho : I(X) \rightarrow \mathbb{N}$ . The measure function  $\rho$  combined with a decision problem  $X$  yields a problem  $X_\rho(S)$  parameterized by  $S \subseteq \mathbb{N}$ .

INSTANCE. Instance  $I$  of  $X$  such that  $\rho(I) \subseteq S$ .

QUESTION. Is  $I$  a yes-instance?

For examples of both single- and multi-valued measure functions we refer the reader to Section 3.2. Finally, we prove a simple lemma regarding single-valued measure functions that will be important later on.

**Lemma 2.** Let  $\rho$  be a single-valued and polynomial-time computable measure function. Let  $S \subseteq \mathbb{N}$  and let  $T$  be a non-empty subset of  $S$  such that  $S \setminus T = \{s_1, \dots, s_k\}$ . If  $X_\rho(\{s_i\})$ ,  $1 \leq i \leq k$ , is in  $P$ , then there is a polynomial-time reduction from  $X_\rho(S)$  to  $X_\rho(T)$ .

*Proof.* Let  $I$  be an arbitrary instance of  $X_\rho(S)$ . Compute (in polynomial time)  $\rho(I)$ . If  $\rho(I) \in \{s_1, \dots, s_k\}$ , then we can compute the correct answer in polynomial time. Otherwise,  $I$  is an instance of  $X_\rho(T)$  and the reduction is trivial.  $\square$

### 3 Generation of NP-Intermediate Problems

We will now extend Ladner's method to parameterized problems. Section 3.1 contains the main result and Section 3.2 contains some examples.

#### 3.1 Diagonalization Method

**Theorem 3.** Let  $X_\rho(\cdot)$  be a computational decision problem with a measure function  $\rho$ . Assume that  $X_\rho(\cdot)$  and  $S \subseteq \mathbb{N}$  satisfies the following properties:

P0:  $I(X)$  is recursively enumerable.

P1:  $X_\rho(S)$  is NP-complete.

P2:  $X_\rho(T)$  is in  $P$  whenever  $T$  is a finite subset of  $S$ .

P3:  $X_\rho(S)$  is polynomial-time reducible to  $X_\rho(T)$  whenever  $T \subseteq S$  and  $S \setminus T$  is finite.

Then, if  $P \neq NP$ , there exists a set  $S' \subset S$  such that  $X_\rho(S')$  is in  $NP \setminus P$  and  $X_\rho(S)$  is not polynomial-time reducible to  $X_\rho(S')$ .

Before the proof, we make some observations that will be used without explicit references. If  $\rho$  is single-valued and polynomial-time computable, then P2 implies P3 by Lemma 2. In many examples,  $S = \mathbb{N}$  which means that P1 can be restated as NP-completeness of  $X$ . If P1 holds, then property P3 simply states that  $X_\rho(T)$  is NP-complete for every cofinite  $T \subseteq S$ . Finally, we remind the reader that the polynomial-time bounds may depend on the choice of  $S$  in the definitions of P2 and P3.

The proof is an adaption of Papadimitriou's [24] proof where we use the abstract properties P0 – P3 instead of focusing on the size of instances. Papadimitriou's proof is, in turn, based on Ladner's original proof [20]. It may also be illuminating to compare with Schöning [25] and Bodirsky & Grohe [4].

In the sequel, we let  $X_\rho(\cdot)$  be a computational decision problem that together with  $S \subseteq \mathbb{N}$  satisfies properties P0 – P3. Let  $A_X$  be an algorithm for  $X_\rho(S)$ , let  $M_1, M_2, \dots$  be an enumeration of all polynomial-time bounded deterministic Turing machines, and let  $R_1, R_2, \dots$  be an enumeration of all polynomial-time Turing reductions. Such enumerations are known to exist, cf. Papadimitriou [24].

We define a function  $f : \mathbb{N} \rightarrow \mathbb{N}$  that is computed by a Turing machine  $F$  and the input  $n$  is given to  $F$  in unary representation. We let  $f(0) = f(1) = 0$ . The computation of  $f(n)$  starts with the computation of  $f(0), f(1), f(2), \dots$ , until the total number of steps  $F$  has used in computing this sequence exceeds  $n$ . This is possible since  $F$  has access to its own description by Kleene's fixed point theorem. Let  $i$  be the largest value for which  $F$  was able to completely compute  $f(i)$  (during these  $n$  steps) and let  $k = f(i)$ .

In the final phase of the execution of the machine  $F$  we have two cases depending on whether  $k$  is even or odd. In both cases, if this phase requires  $F$  to run for more than  $n$  computation steps,  $F$  stops and returns  $k$  (i.e.,  $f(n) = k$ ).

The first case is when  $k$  is even: here,  $F$  enumerates all instances  $I$  of  $X_\rho(S)$  — this is possible by property P0. For each instance  $I$ ,  $F$  simulates  $M_{k/2}$  on the encoding of  $I$ , determines whether  $A_X(I)$  is accepted, and finally,  $F$  computes  $f$  for all  $x \in \rho(I)$ . If  $M_{k/2}$  rejects and  $A_X(I)$  was accepted, and  $f(x)$  is even for all  $x \in \rho(I)$ , then  $F$  returns  $k + 1$  (i.e.,  $f(n) = k + 1$ ).  $F$  also returns  $k + 1$  if  $M_{k/2}$  accepts and  $I$  is not accepted by  $A_X$  and  $f(x)$  is even for all  $x \in \rho(I)$ .

The second case is when  $k$  is odd. Again,  $F$  enumerates all instances  $I$  of  $X_\rho(S)$ . Let  $E = \emptyset$ . Now, for each instance  $I$ ,  $F$  begins simulating  $R_{\lfloor k/2 \rfloor}$  on the encoding of  $I$  with an oracle for  $A_X$ . Whenever the simulation notices that  $R_{\lfloor k/2 \rfloor}$  enters an oracle state, we calculate  $\rho(I') = E'$  (where  $I'$  is the  $X_\rho(S)$  instance corresponding to the input of the oracle tape), and add the members of  $E'$  to  $E$ . When the simulation is finished we first calculate  $f(x)$  for every  $x \in E$ . If the result of any  $f(x)$  operation is odd we return  $k + 1$ . We then compare the result of the reduction with  $A_X(I)$ . If the results do not match, i.e. if one is accepted or rejected while the other is not, we return  $k + 1$ . This completes the definition of  $f$ . Note that  $f$  can be computed in polynomial time (regardless of the time complexity of computing  $\rho$  and  $A_X$ ) since the input is given in unary.

We now show that  $f$  is increasing, i.e. for all  $n \geq 0$ ,  $f(n) \leq f(n + 1)$  and  $\{f(n) \mid n \in \mathbb{N}\}$  is an unbounded set, unless  $P = NP$ . To see this, we first prove by induction that  $f(n) \leq f(n + 1)$  for all  $n \geq 0$ . This obviously holds for  $n = 0$  and  $n = 1$ . Assume that this holds for an arbitrary number  $i > 1$ . By definition  $f(i + 1)$  cannot return a smaller number than  $f(i)$  in the first phase of the computation, since the Turing machine  $F$  simulates  $f(i')$  for all  $i' < i$ , and returns the largest  $k$  for which  $f(i')$  was successfully computed within the allotted time. In the second phase, the argument to  $f$  is used to determine the

total amount of computation steps, and since  $f$  will either return the  $k$  from the first phase, or  $k + 1$ , there is no possibility that  $f(i) > f(i + 1)$ .

Let  $S_e = \{x \mid x \in S \text{ and } f(x) \text{ is even}\}$ . We continue by showing that there is no  $n_0$  such that  $f(n) = k_0$  for all  $n > n_0$  unless  $P = NP$ . If there is such a  $n_0$ , then there is also a  $n_1$  such that for all  $n > n_1$  the value  $k$  computed in the phase where  $F$  computes  $f(1), f(2), \dots$  (in  $n$  steps) is  $k_0$ . If  $k_0$  is even, then on all inputs  $n > n_1$  the machine  $M_{k_0/2}$  correctly decides  $X_\rho(S_e)$  and thus  $X_\rho(S_e)$  is in  $P$ . But since  $f(n) = k_0$  for all  $n > n_1$ , we have that  $S \setminus S_e$  is finite, and thus  $X_\rho(S)$  is polynomial-time reducible to  $X_\rho(S_e)$  by Property P3, which is a contradiction since  $X_\rho(S)$  is NP-complete by Property P1. Similarly if  $k_0$  is odd, then on all inputs  $n > n_1$  the function  $R_{\lfloor k_0/2 \rfloor}$  is a valid reduction from  $X_\rho(S)$  to  $X_\rho(S_e)$  and thus  $X_\rho(S_e) \notin P$ . But since  $f(n) = k_0$  for all  $n > n_1$ , we have that  $S_e$  is finite, and we conclude that  $X_\rho(S_e)$  is in  $P$  by Property P2, which is a contradiction since  $X_\rho(S)$  is NP-complete by Property P1.

We conclude the proof by showing that  $X_\rho(S_e)$  is neither in  $P$ , nor is  $X_\rho(S)$  polynomial-time reducible to  $X_\rho(S_e)$ , unless  $P = NP$ . By Property P1,  $X_\rho(S_e)$  is in  $NP$  since  $S_e \subseteq S$ . Assume now that  $X_\rho(S_e)$  is in  $P$ . Then there is an  $i$  such that  $M_i$  solves  $X_\rho(S_e)$ . Thus, by the definition of  $f$ , there is an  $n_1$  such that for all  $n > n_1$  we have  $f(n) = 2i$ ; this contradicts that  $f$  is increasing. Similarly, assume that  $X_\rho(S)$  is polynomial-time reducible to  $X_\rho(S_e)$ . Then, there is an  $i$  such that  $R_i$  is a polynomial-time reduction from  $X_\rho(S)$  to  $X_\rho(S_e)$ . It follows from the definition of  $f$  that there is an  $n_1$  such that  $f(n) = 2i - 1$  for all  $n > n_1$ , and this contradicts that  $f$  is increasing.  $\square$

If the measure function is polynomially bounded (e.g.  $\rho(I) \leq p(\|I\|)$  for some polynomial  $p$ ), then checking whether an integer  $x$  written in binary is in  $S_e$  or not can be decided in polynomial time. This follows from the fact that  $x$  written in binary can be converted to  $x$  written in unary in polynomial time. Another useful observation is the following: it follows from the proof that property P1 (i.e. the NP-hardness of the original problem) can be replaced by hardness for other complexity classes within  $NP$ . By noting that  $X_\rho(S_e)$  is recursively enumerable, this implies that we can construct infinite chains of problems  $X_\rho(T_1), X_\rho(T_2), \dots$  such that  $S_e = T_1 \supset T_2 \supset \dots$ , there does not exist any polynomial-time reductions from  $X_\rho(T_i)$  to  $X_\rho(T_{i+1})$ , and  $X_\rho(T_i)$  is not in  $P$  for any  $i \geq 1$ .

### 3.2 Examples

Ladner's result is now a straightforward consequence of Theorem 3. Let  $X$  be an arbitrary NP-complete problem such that  $I(X)$  is recursively enumerable. For an arbitrary instance  $I \in I(X)$ , we let the single-valued measure function  $\rho$  be defined such that  $\rho(I) = \|I\|$ . We verify that  $X_\rho(\mathbb{N})$  satisfies properties P0 – P3 and conclude that there exists a set  $T \subseteq \mathbb{N}$  such that  $X_\rho(T)$  is NP-intermediate. Properties P0 and P1 hold by assumption and property P2 holds since  $X_\rho(U)$  can be solved in constant time whenever  $U$  is finite. If  $U \subseteq \mathbb{N}$  and  $\mathbb{N} \setminus U = \{x_1, \dots, x_k\}$ , then  $X_\rho(\{x_i\})$ ,  $1 \leq i \leq k$ , is solvable in constant time and we can apply Lemma 2(2). Thus, property P3 holds, too.

Another straightforward application of single-valued measure functions is the following: Chen et al. [9] have discovered a striking connection between NP-intermediate problems and the parameterized complexity class XP (XP denotes the class of decision problems  $X$  that are solvable in time  $O(\|I\|^{f(k)})$  for some polynomial-time computable parameter  $k$  and some computable function  $f$ ).

**Proposition 4.** *Let  $X$  be a decision problem and  $\rho$  a polynomial-time computable single-valued measure function such that  $X_\rho(\cdot)$  satisfies conditions P0 and P1, and  $X_\rho \in \text{XP}$ . Then there exists a  $T \subseteq \mathbb{N}$  such that  $X_\rho(T)$  is NP-intermediate.*

*Proof.* We note that  $X_\rho(S)$  is in P whenever  $S$  is a finite subset of  $\mathbb{N}$ . Hence,  $X_\rho$  satisfies P2 and consequently P3. The result follows from Theorem 3.  $\square$

To illustrate multi-valued measure functions, we turn our attention to the SUBSET-SUM problem [19].

INSTANCE: A finite set  $Y \subseteq \mathbb{N}$  and a number  $k \in \mathbb{N}$ .

QUESTION: Is there a  $Y' \subseteq Y$  such that  $\sum Y' = k$ ?

We define a multi-valued measure function by letting  $\rho((Y, k)) = Y$ . Once again, properties P0 and P1 hold by assumption so it is sufficient to prove that SUBSET-SUM $_\rho(\mathbb{N})$  satisfies P2 and P3. Property P2: instances of SUBSET-SUM can be solved in time  $O(\text{poly}(\|I\|) \cdot c(I))$ , where  $c(I)$  denotes the difference between the largest and smallest number in  $Y$  [14]. This difference is finite whenever we consider instances of SUBSET-SUM $_\rho(S)$  where  $S \subseteq \mathbb{N}$  is finite. Property P3: arbitrarily choose  $S \subseteq \mathbb{N}$  such that that  $\mathbb{N} \setminus S$  is finite. We present a polynomial-time Turing reduction from SUBSET-SUM $_\rho(\mathbb{N})$  to SUBSET-SUM $_\rho(S)$ . Let  $I = (Y, k)$  be an instance of SUBSET-SUM $_\rho(\mathbb{N})$ . Let  $T = Y \setminus S$ , i.e. the elements of the instance which are not members of the smaller set  $S$ . Since  $\mathbb{N} \setminus S$  is finite,  $T$  is a finite set, too. Let  $Z = Y \cap S$ . For every subset  $T'_i = \{x_1, \dots, x_{im}\}$  of  $T$ , we let  $I'_i = (Z, k'_i)$ , where  $k'_i = k - (x_1 + \dots + x_{im})$ . Then, it is easy to see that  $I$  is a yes-instance if and only if at least one  $I'_i$  is a yes-instance. Finally, we note that the reduction runs in time  $O(\text{poly}(\|I\|) \cdot 2^c)$ , where  $c = |\mathbb{N} \setminus S|$ , and this is consequently a polynomial-time reduction for every fixed  $S$ .

## 4 Single-Valued Measure Functions

This section is divided into two parts: Section 4.1 is concerned with polynomial-time computable single-valued measure functions and Section 4.2 is concerned with structurally restricted CSPs.

### 4.1 Polynomial-Time Computable Measure Functions

By Theorem 3, we know that properties P0 – P3 are sufficient to assure the existence of NP-intermediate problems. A related question is to what degree the properties are also necessary. Here, we investigate the scenario when P2 and P3 do not necessarily hold.

**Theorem 5.** *Assume  $X$  is a decision problem and  $\rho$  is a single-valued measure function such that  $X_\rho(\mathbb{N})$  satisfies P0 and P1. Let  $S_P = \{s \in \mathbb{N} \mid X_\rho(\{s\}) \in P\}$  and assume membership in  $S_P$  is a decidable problem. Then, at least one of the following holds: (1) there exists a set  $T \subseteq S_P$  such that  $X_\rho(T)$  is NP-intermediate, (2) there exists a  $t \in \mathbb{N}$  such that  $X_\rho(\{t\})$  is NP-intermediate, or (3)  $X_\rho$  admits no NP-intermediate subproblems.*

*Proof.* If  $X_\rho(\{s\})$  is NP-complete for every  $s \in \mathbb{N}$ , then we are in case (3) so we assume this is not the case. If there exists  $s \in \mathbb{N}$  such that  $X_\rho(\{s\})$  is NP-intermediate, then we are in case (2) so we assume this does not hold either. Thus, we may henceforth assume that there exists  $s \in \mathbb{N}$  such that  $X_\rho(\{s\}) \in P$  and that  $X_\rho(\{u\})$  is NP-complete whenever  $u \in \mathbb{N} \setminus S_P$ . This implies that  $S_P$  is non-empty. Once again, we single out two straightforward cases: if  $X_\rho(S_P)$  is NP-intermediate, then we are in case (1), and if  $X_\rho(S_P)$  is in P, then we are in case (3) (since  $X_\rho(\{u\})$  is NP-complete whenever  $u \notin S_P$ ). Hence, we may assume that  $X_\rho(S_P)$  is NP-complete (note that  $X_\rho(S_P) \in \text{NP}$  since  $X_\rho(\mathbb{N}) \in \text{NP}$  by P1), i.e.  $X_\rho(S_P)$  satisfies P1. Furthermore,  $X_\rho(S_P)$  satisfies P0 since  $S_P$  is a decidable set and the instances of  $X$  are recursively enumerable. To generate the instances of  $X_\rho(S_P)$ , we generate the instances of  $X$  one after another and output instance  $I$  if and only if  $\rho(I)$  is in  $S_P$ .

We finally show that  $X_\rho(S_P)$  satisfies P2 and P3. It is sufficient to prove that  $X_\rho(S_P)$  satisfies P2 since  $\rho$  is single-valued. Assume there exists a finite set  $K \subseteq S_P$  such that  $X_\rho(K) \notin P$ . Let  $\emptyset \subset K' \subseteq K$  be a subset such that  $X_\rho(K')$  is a member of P; such a set exists since  $K \subseteq S_P$ . For every  $k' \in K'$ , we know that  $X_\rho(\{k'\}) \in P$ . Hence, we can apply Lemma 2 and deduce that there exists a polynomial-time reduction from  $X_\rho(K)$  to  $X_\rho(K')$ . This contradicts the fact that  $X_\rho(K)$  is not a polynomial-time solvable problem. We can now apply Theorem 3 and conclude that there exists a set  $T \subseteq S_P$  such that  $X_\rho(T)$  is NP-intermediate, i.e. we are in case (1).  $\square$

Problems parameterized by multi-valued measure functions are apparently very different from those parameterized by single-valued functions. For instance, Lemma 2 breaks down which indicates that the proof strategy used in Theorem 5 is far from sufficient to attack the multi-valued case.

## 4.2 Structurally Restricted CSPs

When identifying tractable (i.e. polynomial-time solvable) fragments of constraint satisfaction problems and similar problems, two main types of results have been considered in the literature. The first one is to identify constraint languages  $\Gamma$  such that  $\text{CSP}(\Gamma) \in P$ , and the second one is to restrict the structure induced by the constraints on the variables. The second case is often concerned with associating some structure with each instance and then identifying sets of structures that yield tractable problems. The classical example of this approach is to study the *primal graph* or *hypergraph* of CSP instances. Given a CSP instance  $I$  with variable set  $V$ , we define its primal graph  $G = (V, E)$  such

that  $(v_i, v_j) \in E$  if and only if variables  $v_i, v_j$  occur simultaneously in some constraint, and we define the hypergraph  $\mathcal{H} = (V, \mathcal{E})$  such that the hyperedge  $\{v_{i_1}, \dots, v_{i_k}\} \in \mathcal{E}$  if and only if there is a constraint  $R(v_{i_1}, \dots, v_{i_k})$  in  $I$ .

When it comes to defining structurally restricted problems that are tractable, one is typically interested in certain parameters of these (hyper)graphs such as *tree-width*, *fractional hypertree width* [16], or *submodular width* [22]. It is, for instance, known that any finite-domain CSP instance  $I$  with primal graph  $G = (V, E)$  can be solved in  $\|I\|^{O(\text{tw}(G))}$  time [11] where  $\text{tw}(G)$  denotes the tree-width of  $G$ , and it can be solved in  $\|I\|^{O(\text{flw}(\mathcal{H}))}$  time [16] where  $\text{flw}(\mathcal{H})$  denotes the fractional hypertree width of  $\mathcal{H}$ . Since these results rely on the domains being finite, we restrict ourselves to finite-domain CSPs throughout this section. Now note that if given a finite constraint language  $\Gamma$ , then the instances of  $\text{CSP}(\Gamma)$  are recursively enumerable and  $\text{CSP}(\Gamma)$  is in NP. If  $\Gamma$  is infinite, then this is not so evident and it may, in fact, depend on the representation of relations. We adopt a simplistic approach and represent a relation by listing its tuples. Under this assumption, the instances of  $\text{CSP}(\Gamma)$  are recursively enumerable and  $\text{CSP}(\Gamma)$  is in NP.

By restricting the CSP problem to instances with tree-width or fractional hypertree width  $\leq k$  (for some constant  $k$ ), it is known that the resulting problem is solvable in polynomial time. This immediately implies that problems like  $\text{CSP}_{\text{tw}}$  and  $\text{CSP}_{\text{flw}}^1$  have property P2. If the width parameter under consideration is polynomial-time computable, then we have property P3 (via Lemma 2), too, and conclude that NP-intermediate fragments exist. Unfortunately, this is typically not the case. It is for instance NP-complete to determine whether a given graph  $G$  has treewidth at most  $k$  or not [2] if  $k$  is part of the input. This is a common feature that holds for, or is suspected to hold for, many different width parameters. Hence, width parameters are a natural source of single-valued measure functions that are not polynomial-time computable. Such measure functions are problematic since we cannot prove the existence of NP-intermediate subproblems by using simplifying results like Proposition 4 or Theorem 5. By a few additional assumptions we can however still prove the applicability of Theorem 3. Note that if  $k$  is fixed, and thus not part of the input, then the graphs with tree-width  $\leq k$  can be recognized in linear time [5]. This is not uncommon when studying width parameters — determining the width exactly is computationally hard but it can be computed or estimated in polynomial time under additional assumptions. We arrive at the following result.

**Proposition 6.** *Assume that  $X$  is a decision problem and  $\rho$  is a single-valued measure function such that  $X_\rho(\cdot)$  satisfies conditions P0 and P1. Furthermore suppose that for each set  $\{0, \dots, k\}$  there exists a promise algorithm  $A_k$  for  $X_\rho(\{0, \dots, k\})$  with the following properties:*

- if  $\rho(I) \leq k$ , then  $A_k$  returns the correct answer in  $p_k(\|I\|)$  steps where  $p_k$  is a polynomial only depending on  $k$ , and

---

<sup>1</sup> We slightly abuse notation since  $\text{tw}$  and  $\text{flw}$  are not directly defined on problem instances.



- if  $\rho(I) > k$ , then  $A_k$  either return a correct answer or do not answer at all.

Then there exists a set  $S \subset \mathbb{N}$  such that  $X_\rho(S)$  is NP-intermediate.

*Proof.* Let  $X^k$  denote the computational problem  $X$  restricted to instances  $I \in I(X)$  such that  $\rho(I) \geq k$ . Assume there exists a  $k$  such that  $X^k \in P$  and let  $B$  be an algorithm for this problem running in time  $q(|I|)$  for some polynomial  $q$ . For  $X_\rho(\{0, \dots, k-1\})$ , we have algorithm  $A_{k-1}$  described above. Given an arbitrary instance  $I$  of  $X$ , we may not be able to compute  $\rho(I)$  and choose which algorithm to run. Do as follows: run algorithm  $A_{k-1}$  for  $p_{k-1}(|I|)$  steps on input  $I$ . If  $A_{k-1}$  produces an answer, then this is correct. If  $A_{k-1}$  does not produce an answer, then we know that  $\rho(I) > k-1$  and we can apply algorithm  $B$ . All in all, this takes  $O(p_{k-1}(|I|) + q(|I|))$  time so  $X \in P$  which leads to a contradiction.

If  $X^k$  is in NPI for some  $k$ , then we simply let  $S = \{k, k+1, \dots\}$ . We can henceforth assume that  $X^k$  is NP-complete for all  $k$ . Obviously,  $X_\rho(\mathbb{N})$  satisfies property P2 since algorithm  $A_k$ ,  $k \geq 0$ , runs in polynomial time. We show that it satisfies property P3, too. Let  $T \subseteq \mathbb{N}$  be a finite set and let  $m = \max T$ . We know that  $X^{m+1}$  is NP-complete. Hence, there exists a polynomial-time reduction from the NP-complete problem  $X_\rho(\mathbb{N})$  to  $X^{m+1}$  which, in turn, admits a trivial polynomial-time reduction to  $X_\rho(\mathbb{N} \setminus T)$  since  $\{m+1, m+2, \dots\} \subseteq \mathbb{N} \setminus T$ . We can now apply Theorem 3 and obtain the set  $S$ .  $\square$

We apply this result to  $\text{CSP}_{\text{tw}}$  and  $\text{CSP}_{\text{fhw}}$ , respectively. Clearly, both these problems satisfy properties P0 and P1 due to the assumptions that we have made. For  $\text{CSP}_{\text{tw}}$ , we let  $A_k$  work as follows: given a CSP instance  $I$ , check whether  $I$  has treewidth  $\leq k$  using Bodlaender's [5] algorithm. If the algorithm answers “no”, then go into an infinite loop. Otherwise, decide whether  $I$  has a solution or not in  $|I|^{O(k)}$  time. Proposition 6 implies that there exists a set  $T \subseteq \mathbb{N}$  such that  $\text{CSP}_{\text{tw}}(T)$  is NP-intermediate. We observe that Grohe [15] has shown a similar result under the assumption that  $\text{FPT} \neq \text{W}[1]$  instead of  $P \neq \text{NP}$ . Many other width parameters can also be used for obtaining NP-intermediate problems. One example is  $\text{CSP}_{\text{fhw}}$  for which the proof is very similar but is instead based on Theorem 4.1 in Marx [21].

## 5 Multi-valued Measure Functions

In this section we turn our attention to multi-valued measure functions and apply them to constraint problems. Throughout this section we assume that  $P \neq \text{NP}$ . Here, we want to associate the complexity of CSPs with constraint languages and multi-valued measure functions are convenient for this purpose. Given a constraint satisfaction problem parameterized with a constraint language  $\Gamma$ , let  $\rho$  denote the single-valued measure function defined to return the highest arity of any constraint in a given instance:  $\rho((V, C)) = \max\{k \mid R(v_1, \dots, v_k) \in C\}$ . Let  $\text{CSP}_\rho^*(X)$  denote the  $\text{CSP}(\Gamma)$  problem restricted to instances  $I$  such that  $\rho(I) \in X$ , and assume there exists a set  $X \subset \mathbb{N}$  such that  $\text{CSP}_\rho^*(X)$  is NP-intermediate. Can we from this conclude that there exists a constraint language

$\Gamma' \subset \Gamma$  such that  $\text{CSP}(\Gamma')$  is NP-intermediate? In general, the answer is no since the set of valid instances of  $\text{CSP}_\rho^*(X)$  are not in a one-to-one correspondence with any constraint language restriction. Note that  $\text{CSP}_\rho^*(X)$  is not the same problem as  $\text{CSP}(\{R \in \Gamma \mid \text{ar}(R) \in X\})$ . If we on the other hand define the multi-valued measure function  $\sigma((V, C)) = \{k \mid R(v_1, \dots, v_k) \in C\}$ , then for every  $X \subset \mathbb{N}$  the problem  $\text{CSP}_\sigma^*(X)$  is equivalent to  $\text{CSP}(\{R \in \Gamma \mid \text{ar}(R) \in X\})$ .

### 5.1 Constraint Satisfaction Problems and the Local-Global Conjecture

A constraint language  $\Gamma$  is said to have the *local-global* property [4] if  $\text{CSP}(\Gamma') \in \text{P}$  for every finite set  $\Gamma' \subset \Gamma$  implies  $\text{CSP}(\Gamma) \in \text{P}$ . The non-existence of languages not having the local-global property is known as the *local-global conjecture*. In Bodirsky & Grohe [4] it is proven that if  $\Gamma$  is a constraint language over a finite domain  $D$  that does not exhibit the local-global property, then there exists a constraint language  $\Gamma'$  over  $D$  such that  $\text{CSP}(\Gamma')$  is NP-intermediate. In this section we prove a more general result not restricted to finite domains based on the notion of *extension operators*. If  $R$  is a  $k$ -ary relation and  $\Gamma$  a constraint language over a domain  $D$  we say that  $R$  has a *primitive positive* (p.p.) definition in  $\Gamma$  if  $R(x_1, \dots, x_k) \equiv \exists y_1, \dots, y_l. R_1(\mathbf{x}_1) \wedge \dots \wedge R_l(\mathbf{x}_l)$ , where each  $R_j \in \Gamma \cup \{=\}$  and each  $\mathbf{x}_i$  is a vector over  $x_1, \dots, x_k, y_1, \dots, y_l$ .

**Definition 7.** Let  $\Gamma$  be a recursively enumerable constraint language (with a suitable representation of relations in  $\Gamma$ ). We say that  $\langle \cdot \rangle$  is an extension operator if (1)  $\langle \Gamma \rangle$  is a recursively enumerable set of p.p. definable relations over  $\Gamma$  and (2) whenever  $\Delta \subset \langle \Gamma \rangle$  and  $\langle \Gamma \rangle \setminus \Delta$  is finite, then every  $R \in \langle \Gamma \rangle \setminus \Delta$  is p.p. definable in  $\Delta$ .

Another way of viewing this is that the expressive power of  $\langle \Gamma \rangle$  does not change when removing finitely many relations. Since  $\Gamma$  and  $\langle \Gamma \rangle$  are recursively enumerable we can enumerate relations in  $\Gamma$  or  $\langle \Gamma \rangle$  as  $R_1, R_2, \dots$ , and it is not hard to see that this implies that instances of  $\text{CSP}(\Gamma)$  and  $\text{CSP}(\langle \Gamma \rangle)$  are also recursively enumerable. Given an instance  $I$  of  $\text{CSP}(\Gamma)$  containing the relations  $R_{i_1}, \dots, R_{i_k}$ , we let  $\rho(I) = \{i_1, \dots, i_k\}$ . Let  $\text{CSP}_\rho^*(S)$  denote the  $\text{CSP}(\Gamma)$  problem over instances  $I$  such that  $\rho(I) \subseteq S$ . Define the measure function  $\rho'$  analogous to  $\rho$  but for instances over  $\text{CSP}(\langle \Gamma \rangle)$ , and let  $\text{CSP}_{\rho'}^*(S)$  be the  $\text{CSP}(\langle \Gamma \rangle)$  problem restricted to instances  $I$  such that  $\rho'(I) \subseteq S$ .

**Theorem 8.** Assume  $\Gamma$  is a constraint language such that  $\text{CSP}_\rho^*(\mathbb{N})$  satisfies property P0 – P2. Let  $\langle \cdot \rangle$  be an extension operator such that  $\text{CSP}_{\rho'}^*(\langle \Gamma \rangle)$  satisfies property P0 – P1. Then there exists a  $\Gamma' \subset \langle \Gamma \rangle$  such that  $\text{CSP}(\Gamma')$  is NP-intermediate.

*Proof.* We prove that  $\text{CSP}_{\rho'}^*(\mathbb{N})$  satisfies property P0 – P3. The first two properties are trivial by assumption. For property P2 let  $T = \{i_1, \dots, i_k\}$  be an arbitrary finite subset of  $\mathbb{N}$  and let  $\Theta = \{R_{i_1}, \dots, R_{i_k}\}$ . Note that  $\Theta$  might contain relations which are not included in  $\Gamma$ . For every such relation  $R \in \Theta$  we can

however replace it by its p.p. definition in  $\Gamma$ . Let the resulting set of relations be  $\Theta'$  and let  $S = \{i \mid R_i \in \Theta'\}$ . Then  $\text{CSP}_{\rho'}^\times(T)$  and  $\text{CSP}_{\rho'}^*(S)$  are polynomial-time equivalent since  $T$  is a finite set. Since  $\text{CSP}_{\rho'}^*(S)$  is solvable in polynomial time by assumption,  $\text{CSP}_{\rho'}^\times(T)$  is polynomial-time solvable too.

For property P3 let  $T \subset \mathbb{N}$  such that  $\mathbb{N} \setminus T = \{t_1, \dots, t_k\}$ . To see that there exists a polynomial-time reduction from  $\text{CSP}_{\rho'}^\times(\mathbb{N})$  to  $\text{CSP}_{\rho'}^\times(T)$ , we let  $I$  be an arbitrary instance of  $\text{CSP}_{\rho'}^\times(\mathbb{N})$ . Assume  $I$  contains the constraint  $R_i(x_1, \dots, x_m)$ ,  $i \in \mathbb{N} \setminus T$ . Since  $\langle \cdot \rangle$  is an extension operator the relation  $R_i$  is p.p. definable in  $\langle \Gamma \rangle \setminus \Delta$  where  $\Delta = \{R_i \mid i \in \mathbb{N} \setminus T\}$ . Thus, we can replace  $R_i(x_1, \dots, x_m)$  with its p.p. definition in  $\langle \Gamma \rangle \setminus \Delta$ , and by doing this for all constraints that are not allowed by  $T$ , we end up with an instance  $I'$  of  $\text{CSP}_{\rho'}^\times(T)$  that is satisfiable if and only if  $I$  is satisfiable. This is a polynomial-time reduction since  $\mathbb{N} \setminus T$  is a finite set.

By applying Theorem 3, we can now identify a set  $S \subset \mathbb{N}$  such that  $\text{CSP}_{\rho'}^\times(S)$  is NP-intermediate. This implies that  $\text{CSP}(\Gamma')$  is NP-intermediate when  $\Gamma' = \{R_i \in \langle \Gamma \rangle \mid i \in S\}$ .  $\square$

Our first extension operator is based on the idea of extending a relation into a relation with higher arity. For any relation  $R \subseteq D^n$ , we define the  $k$ th power of  $R$  to be the relation  $R^k(x_0, \dots, x_{k \cdot n - 1}) \equiv R(x_0, \dots, x_{n-1}) \wedge R(x_n, \dots, x_{2n-1}) \wedge R(x_{2n}, \dots, x_{3n-1}) \wedge \dots \wedge R(x_{(k-1)n}, \dots, x_{kn-1})$ . Given a constraint language  $\Gamma$ , let  $\langle \Gamma \rangle_{\text{pow}} = \{R^k \mid R \in \Gamma \text{ and } k \in \mathbb{N}\}$ . We represent each relation in  $\langle \Gamma \rangle_{\text{pow}}$  as a pair  $(R, k)$ . It is easy to see that  $\text{CSP}(\langle \Gamma \rangle_{\text{pow}}) \in \text{NP}$  if  $\text{CSP}(\Gamma) \in \text{NP}$  from which it follows that  $\text{CSP}(\langle \Gamma \rangle_{\text{pow}})$  is NP-complete. Now assume that  $\Delta \subset \langle \Gamma \rangle_{\text{pow}}$  and that  $\langle \Gamma \rangle_{\text{pow}} \setminus \Delta$  is finite. First, for every  $R^k \in \langle \Gamma \rangle_{\text{pow}} \setminus \Delta$  we can p.p. define  $R^k$  in  $\Delta$  as  $R(x_1, \dots, x_n) \equiv \exists x_{n+1}, \dots, x_{k' \cdot n - 1}. R^{k'+1}(x_1, \dots, x_n, x_{n+1}, \dots, x_{k' \cdot n - 1})$ , where  $k' > k$ . Such a  $k'$  must exist since we have only removed finitely many relations from  $\langle \Gamma \rangle_{\text{pow}}$ . Hence  $\langle \cdot \rangle_{\text{pow}}$  is an extension operator. Extension operators are not uncommon in the literature. Well studied examples (provided relations can be suitably represented) include closure under p.p. definitions (known as *co-clones*) and closure under p.p. definitions without existential quantification (known as *partial co-clones*). These are indeed extension operators since  $\langle \Gamma \rangle_{\text{pow}}$  is always a subset of the partial co-clone of  $\Gamma$  and hence also of the co-clone of  $\Gamma$ . For a general introduction to the field of clone theory we refer the reader to Lau [26].

Let  $R_{a,b,c,U} = \{(x, y) \in \mathbb{Z}^2 \mid ax - by \leq c, 0 \leq x, y \leq U\}$  for arbitrary  $a, b, U \in \mathbb{N}$  and  $c \in \mathbb{Z}$ . Furthermore let  $\Gamma'_U = \{R_{a,b,c,U} \mid a, b \in \mathbb{N}, c \in \mathbb{Z}\}$  for any  $U \in \mathbb{N}$  and the language  $\Gamma^\circ$  be defined as  $\Gamma^\circ = \bigcup_{i=0}^\infty \Gamma'_i$ . Note that we can represent each relation in  $\Gamma^\circ$  compactly by four integers written in binary. Due to Jonsson & Löw [18] it is known that  $\Gamma^\circ$  does not satisfy the local-global property. By combining the language  $\Gamma^\circ$  and the extension operator  $\langle \cdot \rangle_{\text{pow}}$  with Theorem 8 we thus obtain the following result.

**Theorem 9.** *There exists a  $\Gamma' \subset \langle \Gamma^\circ \rangle_{\text{pow}}$  such that  $\text{CSP}(\Gamma')$  is NP-intermediate.*

Due to the work of Bodirsky & Grohe [4] we already know that the CSP problem over infinite domains is non-dichotomizable. Their result is however

based on reducing an already known NP-intermediate problem to a CSP problem while our language  $\Gamma' \subset \langle \Gamma^\circ \rangle_{pow}$  is an explicit example of a locally tractable language obtained via blowing holes.

## 5.2 Locally Tractable Languages with Bounded Arity

The downside of the  $\langle \cdot \rangle_{pow}$  operator is that the construction creates relations of arbitrary high arity even if the language only contain relations of bounded arity. In this section we show that simpler extensions are sometimes applicable for constraint languages over infinite domains. For any  $k$ -ary relation  $R$  we define the  $(k+1)$ -ary relation  $R_a$  as  $R_a(x_1, \dots, x_n, y) \equiv R(x_1, \dots, x_n) \wedge (y = a)$ , where  $a \in D$  and  $(y = a)$  is the constraint application of the relation  $\{(a)\}$ . Let  $\langle \Gamma \rangle_+ = \{R_a \mid R \in \Gamma, a \in D\}$ . If we represent each relation in  $\langle \Gamma \rangle_+$  as a tuple  $(R, a)$  then obviously  $\langle \Gamma \rangle_+$  is recursively enumerable if  $\Gamma$  is recursively enumerable. Now assume that  $\Gamma$  is an infinite constraint language and that  $\langle \Gamma \rangle_+ \setminus \Delta$  is finite. For any relation  $R_a \in \langle \Gamma \rangle_+ \setminus \Delta$  we first determine a  $b$  such that  $R_b \in \Delta$ . By construction there exists such a  $b$  since  $\langle \Gamma \rangle_+ \setminus \Delta$  is finite. Then, since  $\Gamma$  is infinite, there exists an  $m$ -ary relation  $R' \in \Gamma$  such that  $R'_a \in \Delta$ . Hence we can implement  $R_a$  as  $R_a(x_1, \dots, x_n, y) \equiv \exists y', x'_1, \dots, x'_m. R_b(x_1, \dots, x_n, y') \wedge R'_a(x'_1, \dots, x'_m, y)$ , by which it follows that  $\langle \cdot \rangle_+$  is an extension operator.

Say that a language  $\Gamma$  is *idempotent* if for all  $a \in D$  it holds that  $\{(a)\}$  is p.p. definable in  $\Gamma$ . We assume that we can find the p.p. definition of  $\{(a)\}$  in  $\Gamma$  in polynomial time.

**Theorem 10.** *Let  $\Gamma$  be an idempotent language over an infinite domain such that  $\Gamma$  does not satisfy the local-global property. Then there exists a constraint language  $\Gamma'$  such that (1)  $\text{CSP}(\Gamma')$  is NP-intermediate and (2)  $\Gamma'$  contains only relations of arity  $k+1$ , where  $k$  is the highest arity of a relation in  $\Gamma$ .*

*Proof.* Let  $R_1, R_2, \dots$  be an enumeration of  $\Gamma$  and define the measure function  $\rho$  over an instance  $I$  containing the relations  $R_{i_1}, \dots, R_{i_k}$  as  $\rho(I) = \{i_1, \dots, i_k\}$ . We note that  $\Gamma$  must be infinite since it does not satisfy the local-global property. Let  $\text{CSP}^*_\rho(S)$  denote the  $\text{CSP}(\Gamma)$  problem over instances  $I$  such that  $\rho(I) \subseteq S$ . Then  $\text{CSP}^*_\rho(\mathbb{N})$  obviously satisfies property P0–P2, and since  $\langle \cdot \rangle_+$  is an extension operator, we only need to prove that  $\text{CSP}(\langle \Gamma \rangle_+)$  is NP-complete. NP-hardness is easy since  $\text{CSP}(\Gamma)$  is trivially polynomial-time reducible to  $\text{CSP}(\langle \Gamma \rangle_+)$ . For membership in NP we give a polynomial-time reduction from  $\text{CSP}(\langle \Gamma \rangle_+)$  to  $\text{CSP}(\Gamma)$ . Let  $I$  be an arbitrary instance of  $\text{CSP}(\langle \Gamma \rangle_+)$ . For any constraint  $R_a(x_1, \dots, x_n, y)$  we replace it by  $R(x_1, \dots, x_n) \wedge \phi(x'_1, \dots, x'_m, y)$ , where  $\exists x'_1, \dots, x'_m. \phi$  is the p.p. definition of  $y = a$ , which is computable in polynomial time by assumption. If we repeat the procedure for all  $R_a$  in  $I$  we get an instance  $I'$  of  $\text{CSP}(\Gamma)$  which is satisfiable if and only if  $I$  is satisfiable. Hence there exists a  $\Gamma' \subset \langle \Gamma \rangle_+$  such that  $\text{CSP}(\Gamma')$  is NP-intermediate by Theorem 8. Let  $k$  denote the highest arity of a relation in  $\Gamma$ . By definition every relation in  $\langle \Gamma \rangle_+$  then has its arity bounded by  $k+1$ , which trivially also holds for  $\Gamma'$ .  $\square$

It is not hard to see that for the constraint language  $\Gamma^\circ$  defined in the previous section any constant relation is p.p. definable in polynomial time. For any  $a \in \mathbb{N}$  we simply let  $(y = a) \equiv \exists x. R_{0,1,a}(x, y)$ , i.e. the relation  $0 \cdot x - 1 \cdot y \leq a \wedge 0 \leq x, y \leq a$ . By Theorem 10 and the fact that  $\Gamma^\circ$  only contains relations of arity 2 we therefore obtain the following.

**Theorem 11.** *There exists a  $\Gamma' \subset \langle \Gamma^\circ \rangle_+$  such that (1)  $CSP(\Gamma')$  is NP-intermediate and (2)  $\Gamma'$  contains only relations of arity 3.*

### 5.3 Propositional Abduction

Abduction is a fundamental form of nonmonotonic reasoning whose computational complexity has been thoroughly investigated [10,13,23]. It is known that the abduction problem parameterized with a finite constraint language is always in P, NP-complete, coNP-complete or  $\Sigma_2^P$ -complete. For infinite languages the situation differs and the question of whether it is possible to obtain a similar classification was left open in [23]. We will show that there exists an infinite constraint language such that the resulting abduction problem is NP-intermediate.

Let  $\Gamma$  denote a constraint language and define the propositional abduction problem  $ABD(\Gamma)$  as follows.

**INSTANCE.** An instance  $I$  of  $ABD(\Gamma)$  consists of a tuple  $(V, H, M, KB)$ , where  $V$  is a set of Boolean variables,  $H$  is a set of literals over  $V$  (known as the *set of hypotheses*),  $M$  is a literal over  $V$  (known as the *manifestation*), and  $KB$  is a set of constraint applications  $C_1(\mathbf{x}_1) \wedge \dots \wedge C_k(\mathbf{x}_k)$  where  $C_i$  denotes an application of some relation in  $\Gamma$  and  $\mathbf{x}_i$ ,  $1 \leq i \leq k$ , is a vector of variables in  $V$  ( $KB$  is known as the *knowledge base*).

**QUESTION.** Does there exist an *explanation* for  $I$ , i.e., a set  $E \subseteq H$  such that  $KB \wedge \bigwedge E$  is satisfiable and  $KB \wedge \bigwedge E \models M$ , i.e.  $KB \wedge \bigwedge E \wedge \neg M$  is not satisfiable.

Let  $\Gamma_{IHSB-}$  be the infinite constraint language consisting of the relations expressed by the clauses  $(x), (\neg x \vee y)$  and all negative clauses, i.e.,  $\{(\neg x_1 \vee \dots \vee \neg x_n) \mid n \geq 1\}$ . We may represent each relation in  $\Gamma_{IHSB-}$  with a natural number in the obvious way. Let the finite constraint language  $\Gamma_{IHSB-}/k$  be the subset of  $\Gamma_{IHSB-}$  that contains all clauses  $C$  such that  $ar(C) = k$ . In light of this we define the multi-valued measure function  $\rho(I) = \{ar(C) \mid C \text{ is a negative clause of } KB \text{ in } I\}$ . With the chosen representation of relations,  $\rho$  is obviously polynomial-time computable. We define the corresponding parameterized abduction problem  $ABD_\rho^*(\Gamma)$  such that  $I(ABD^*)$  is the set of abduction instances over  $\Gamma_{IHSB-}$ . We now verify that  $ABD_\rho^*(\mathbb{N})$  fulfills property P0 – P3.

Property P0 holds trivially while property P1 follows from [23]. For property P2, we note that if  $T$  is an arbitrary finite subset of  $\mathbb{N}$ , then there exists a  $k \in T$  such that the clauses of every  $ABD_\rho^*(T)$  instance is bounded by  $k$ . By [23], we know that  $ABD(\Gamma_{IHSB-}/k)$  is in P for every  $k$ , and hence that  $ABD_\rho^*(T)$  is in P for every finite subset of  $S$ . To show property P3, we present a polynomial-time reduction from  $ABD_\rho^*(\mathbb{N})$  to  $ABD_\rho^*(T)$  when  $\mathbb{N} \setminus T$  is finite. Let  $k = \max(\mathbb{N} \setminus T)$ . Arbitrarily choose an instance  $I = (V, H, M, KB)$  of  $ABD_\rho^*(\mathbb{N})$ . Then, for every

clause  $C = (\neg x_1 \vee \dots \vee \neg x_l) \in KB$  such that  $l \in S \setminus T$ , replace  $C$  by the logically equivalent clause  $C' = (\neg x_1 \vee \dots \vee \neg x_{l-1} \vee \neg x_l \vee \underbrace{\neg x_l \dots \vee \neg x_l}_{k+1-l \text{ } \neg x_l\text{'s}})$  of length  $k+1$ .

If we let the resulting knowledge base be  $KB'$  then  $I' = (V, H, M, KB')$  is an instance of  $ABD_\rho^*(T)$  which has a solution if and only if  $I$  has a solution.

From this and Theorem 3 it follows that there exists a  $S' \subset \mathbb{N}$  such that  $ABD_\rho^*(S')$  is NP-intermediate. Hence we conclude the following.

**Theorem 12.** *There exists a constraint language  $\Gamma'_{IHSB-} \subset \Gamma_{IHSB-}$  such that  $ABD(\Gamma'_{IHSB-})$  is NP-intermediate.*

## 6 Future Work

One way of obtaining genuinely new NP-intermediate problems is to consider other complexity-theoretic assumptions than  $P \neq NP$ . We have pointed out that the LOGCLIQUE problem is NP-intermediate under the ETH, and that the main difficulty is to provide a lower bound, i.e. proving that LOGCLIQUE  $\notin P$ . One may suspect that providing lower bounds is the main difficulty also when considering other problems. We have seen that CSP problems constitute a rich source of NP-intermediate problems via different kinds of parameterization. Hence, it appears feasible that methods for studying the complexity of parameterized problems will become highly relevant. In particular, *linear fpt-reductions* [7,8] have been used for proving particularly strong lower bounds which may be used for linking together NP-intermediate problems, parameterized problems, and lower bound assumptions. Another way is to adapt and use recent methods for studying the time complexity of Boolean CSP problems [17]. These methods aim at obtaining reductions that provide a fine-grained picture of time complexity and this may be useful when studying NP-intermediate problems. Additionally, recent results by Dell and van Melkebeek [12] can be used for proving the non-existence of such reductions.

We have shown that the propositional abduction problem has NP-intermediate fragments. One may view abduction as a problem that is closely related to Boolean CSPs. However, there is an important difference: the CSP( $\Gamma$ ) problem is either a member of P or NP-complete for *all* choices of Boolean  $\Gamma$ . Hence, it would be interesting to determine which finite-domain CSP-related problems can be used for obtaining NP-intermediate problems and which of them have the local-global property. Inspired by our result on the abduction problem, we view other forms of non-monotonic reasoning such as circumscription and default logic as potential candidates. Unfortunately, many problems of this type are polynomial-time solvable only in very restricted cases, which makes it hard to find a candidate language resulting in a problem not having the local-global property. Thus, more powerful methods than blowing may be needed for identifying NP-intermediate problems in this and similar cases.

## References

1. Arora, S., Barak, B.: *Computational Complexity: A Modern Approach*, 1st edn. Cambridge University Press, New York (2009)
2. Arnborg, S., Corneil, D., Proskurowski, A.: Complexity of finding embeddings in a  $k$ -tree. *SIAM Journal on Matrix Analysis and Applications* 8(2), 277–284 (1987)
3. Bodirsky, M.: *Complexity Classification in Infinite-Domain Constraint Satisfaction*. Habilitation thesis. Univ. Paris 7 (2012)
4. Bodirsky, M., Grohe, M.: Non-dichotomies in constraint satisfaction complexity. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfssdóttir, A., Walukiewicz, I. (eds.) *ICALP 2008, Part II*. LNCS, vol. 5126, pp. 184–196. Springer, Heidelberg (2008)
5. Bodlaender, H.: A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM Journal on Computing* 25(6), 1305–1317 (1996)
6. Bulatov, A., Jeavons, P., Krokhin, A.: Classifying the computational complexity of constraints using finite algebras. *SIAM Journal on Computing* 34(3), 720–742 (2005)
7. Chen, J., Chor, B., Fellows, M., Huang, X., Juedes, D., Kanj, I., Xia, G.: Tight lower bounds for certain parameterized np-hard problems. In: *Proc. IEEE Conference on Computational Complexity (CCC 2004)*, pp. 150–160 (2004)
8. Chen, J., Huang, X., Kanj, I., Xia, G.: Linear fpt reductions and computational lower bounds. In: *Proc. 36th ACM Symposium on Theory of Computing (STOC-2004)*, pp. 212–221 (2004)
9. Chen, Y., Thurley, M., Weyer, M.: Understanding the complexity of induced subgraph isomorphisms. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfssdóttir, A., Walukiewicz, I. (eds.) *ICALP 2008, Part I*. LNCS, vol. 5125, pp. 587–596. Springer, Heidelberg (2008)
10. Creignou, N., Schmidt, J., Thomas, M.: Complexity of propositional abduction for restricted sets of boolean functions. In: *Proc. 12th International Conference on the Principles of Knowledge Representation and Reasoning, KR 2010* (2010)
11. Dechter, R.: *Constraint Processing*. Elsevier Morgan Kaufmann (2003)
12. Dell, H., van Melkebeek, D.: Satisfiability allows no nontrivial sparsification unless the polynomial-time hierarchy collapses. In: *Proc. 42nd ACM Symposium on Theory of Computing (STOC 2010)*, pp. 251–260 (2010)
13. Eiter, T., Gottlob, G.: The complexity of logic-based abduction. *Journal of the ACM* 42(1), 3–42 (1995)
14. Garey, M., Johnson, D.: “Strong” NP-completeness results: motivation, examples and implications. *Journal of the ACM* 25(3), 499–508 (1978)
15. Grohe, M.: The complexity of homomorphism and constraint satisfaction problems seen from the other side. *Journal of the ACM* 54(1), article 1 (2007)
16. Grohe, M., Marx, D.: Constraint solving via fractional edge covers. In: *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-2006)*, pp. 289–298 (2006)
17. Jonsson, P., Lagerkvist, V., Nordh, G., Zanuttini, B.: Complexity of SAT problems, clone theory and the exponential time hypothesis. In: *Proc. the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013* (2013)
18. Jonsson, P., Löw, T.: Computational complexity of linear constraints over the integers. *Artificial Intelligence* 195, 44–62 (2013)
19. Karp, R.M.: Reducibility Among Combinatorial Problems. In: Miller, R.E., Thatcher, J.W. (eds.) *Complexity of Computer Computations*, pp. 85–103. Plenum Press (1972)

20. Ladner, R.: On the structure of polynomial time reducibility. *Journal of the ACM* 22, 155–171 (1975)
21. Marx, D.: Approximating fractional hypertree width. *ACM Transactions on Algorithms* 6(2) (2010)
22. Marx, D.: Tractable hypergraph properties for constraint satisfaction and conjunctive queries. In: *Proc. 42nd ACM Symposium on Theory of Computing (STOC 2010)*, pp. 735–744 (2010)
23. Nordh, G., Zanuttini, B.: What makes propositional abduction tractable. *Artificial Intelligence* 172, 1245–1284 (2008)
24. Papadimitriou, C.H.: *Computational Complexity*. Addison-Wesley (1994)
25. Schöning, U.: A uniform approach to obtain diagonal sets in complexity classes. *Theoretical Computer Science* 18, 95–103 (1982)
26. Lau, D.: *Function Algebras on Finite Sets: Basic Course on Many-Valued Logic and Clone Theory*. Springer Monographs in Mathematics. Springer-Verlag New York, Inc., Secaucus (2006)



# Solving QBF with Free Variables

William Klieber<sup>1</sup>, Mikoláš Janota<sup>2</sup>, Joao Marques-Silva<sup>2,3</sup>, and Edmund Clarke<sup>1</sup>

<sup>1</sup> Carnegie Mellon University, Pittsburgh, PA, USA

<sup>2</sup> IST/INESC-ID, Lisbon, Portugal

<sup>3</sup> University College Dublin, Ireland

**Abstract.** An *open* quantified boolean formula (QBF) is a QBF that contains free (unquantified) variables. A solution to such a QBF is a quantifier-free formula that is logically equivalent to the given QBF. Although most recent QBF research has focused on closed QBF, there are a number of interesting applications that require one to consider formulas with free variables. This article shows how clause/cube learning for DPLL-based closed-QBF solvers can be extended to solve QBFs with free variables. We do this by introducing *sequents* that generalize clauses and cubes and allow learning facts of the form “under a certain class of assignments, the input formula is logically equivalent to a certain quantifier-free formula”.

## 1 Introduction

In recent years, significant effort has been invested in developing efficient solvers for Quantified Boolean Formulas (QBFs). So far this effort has been almost exclusively directed at solving closed formulas — formulas where each variable is either existentially or universally quantified. However, in a number of interesting applications (such as symbolic model checking and automatic synthesis of a boolean reactive system from a formal specification), one needs to consider *open* formulas, i.e., formulas with free (unquantified) variables. A solution to such a QBF is a formula equivalent to the given one but containing no quantifiers and using only those variables that appear free in the given formula. For example, a solution to the open QBF formula  $\exists x. (x \wedge y) \vee z$  is the formula  $y \vee z$ .

This article shows how DPLL-based closed-QBF solvers can be extended to solve QBFs with free variables. In [14], it was shown how clause/cube learning for DPLL-based QBF solvers can be reformulated in terms of *sequents* and extended to non-CNF, non-prenex formulas. This technique uses *ghost variables* to handle non-CNF formulas in a manner that is symmetric between the existential and universal quantifiers. We show that this sequent-based technique can be naturally extended to handle QBFs with free variables.

A naïve way to recursively solve an open QBF  $\Phi$  is shown in Figure 1. Roughly, we Shannon-expand on the free variables until we’re left with only closed-QBF problems, which are then handed to a closed-QBF solver. As an example, consider the formula  $(\exists x. x \wedge y)$ , with one free variable,  $y$ . Substituting  $y$  with  $\text{true}$  in  $\Phi$  yields  $(\exists x. x)$ ; this formula is given to a closed-QBF solver, which yields

---

```

function solve( $\Phi$ ) {
  if ( $\Phi$  has no free variables) {return closed_qbf_solve( $\Phi$ );}
   $x :=$  (a free variable in  $\Phi$ );
  return ite( $x$ , solve( $\Phi$  with  $x$  substituted with True),
            solve( $\Phi$  with  $x$  substituted with False));
}

```

---

**Fig. 1.** Naïve algorithm. The notation “ $\text{ite}(x, \phi_1, \phi_2)$ ” denotes a formula with an *if-then-else* construct that is logically equivalent to  $(x \wedge \phi_1) \vee (\neg x \wedge \phi_2)$ .

**true.** Substituting  $y$  with **false** in  $\Phi$  immediately yields **false**. So, our final answer is the formula  $(y ? \text{true} : \text{false})$ , which simplifies to  $y$ . In general, if the free variables are always branched on in the same order, then the algorithm effectively builds an ordered binary decision diagram (OBDD) [7], assuming that the **ite** function is memoized and performs appropriate simplification.

The above-described naïve algorithm suffers from many inefficiencies. In terms of branching behavior, it is similar to the DPLL algorithm, but it lacks non-chronological backtracking and an equivalent of clause learning. The main contribution of this paper is to show how an existing closed-QBF algorithm can be modified to directly handle formulas with free variables by extending the existing techniques for non-chronological backtracking and clause/cube/sequent learning.

## 2 Preliminaries

**Grammar.** We consider prenex formulas of the form  $Q_1X_1...Q_nX_n.\phi$ , where  $Q_i \in \{\exists, \forall\}$  and  $\phi$  is quantifier-free and represented as a DAG. The logical connectives allowed in  $\phi$  are conjunction, disjunction, and negation. We say that  $Q_1X_1...Q_nX_n$  is the **quantifier prefix** and that  $\phi$  is the **matrix**.

**Assignments.** Let  $\pi$  be a partial assignment of boolean values to variables. For convenience, we identify  $\pi$  with the set of literals made true by  $\pi$ . For example, we identify the assignment  $\{(e_1, \text{true}), (u_2, \text{false})\}$  with the set  $\{e_1, \neg u_2\}$ . We write “ $\text{vars}(\pi)$ ” to denote the set of variables assigned by  $\pi$ .

**Quantifier Order.** In a formula such as  $\forall x.\exists y.\phi$ , where the quantifier of  $y$  occurs inside the scope of the quantifier of  $x$ , and the quantifier type of  $x$  is different from the quantifier type of  $y$ , we say that  $y$  is **downstream** of  $x$ . Likewise, we say that  $x$  is **upstream** of  $y$ . All quantified variables in a formula are considered downstream of all free variables in the formula. In the context of an assignment  $\pi$ , we say that a variable is an **outermost** unassigned variable iff it is not downstream of any variables unassigned by  $\pi$ .

**QBF as a Game.** A closed QBF formula  $\Phi$  can be viewed as a game between an existential player (Player  $\exists$ ) and a universal player (Player  $\forall$ ):

- Existentially quantified variables are *owned* by Player  $\exists$ .
- Universally quantified variables are *owned* by Player  $\forall$ .
- Players assign variables in quantification order (starting with outermost).
- The *goal* of Player  $\exists$  is to make  $\Phi$  be true.
- The *goal* of Player  $\forall$  is to make  $\Phi$  be false.
- A player *owns* a literal  $\ell$  if the player owns  $\text{var}(\ell)$ .

If both players make the best moves possible, then the existential player will win iff the formula is true, and the universal player will win if the formula is false.

**Substitution.** Given a partial assignment  $\pi$ , we define “ $\Phi|_\pi$ ” to be the result of the following: For every assigned variable  $x$ , we replace all occurrences of  $x$  in  $\Phi$  with the assigned value of  $x$  (and delete the quantifier of  $x$ , if any).

**Gate Variables.** We label each conjunction and disjunction with a *gate variable*. If a formula  $\phi$  is labelled by a gate variable  $g$ , then  $\neg\phi$  is labelled by  $\neg g$ . The variables originally in the formulas are called “input variables”, in distinction to gate variables.

## 2.1 Tseitin Transformation’s Undesired Effects in QBF

The Tseitin transformation [20] is the usual way of converting a formula into CNF. In the Tseitin transformation, all the gate variables (i.e., Tseitin variables) are existentially quantified in the innermost quantification block and clauses are added to equate each gate variable with the subformula that it represents. For example, consider the following formula:

$$\Phi_{\text{in}} := \exists e. \forall u. \underbrace{(e \wedge u)}_{g_1} \vee \underbrace{(\neg e \wedge \neg u)}_{g_2}$$

This formula is converted to:

$$\Phi'_{\text{in}} = \exists e. \forall u. \exists g. (g_1 \vee g_2) \wedge (g_1 \Leftrightarrow (e \wedge u)) \wedge (g_2 \Leftrightarrow (\neg e \wedge \neg u)) \quad (1)$$

The biconditionals defining the gate variables are converted to clauses as follows:

$$(g_1 \Leftrightarrow (e \wedge u)) = (\neg e \vee \neg u \vee g_1) \wedge (\neg g_1 \vee e) \wedge (\neg g_1 \vee u)$$

Note that the Tseitin transformation is asymmetric between the existential and universal players: In the resulting CNF formula, the gate variables are existentially quantified, so the existential player (but not the universal player) loses if a gate variable is assigned inconsistently with the subformula that it represents. For example, in Equation 1, if  $e|_\pi = \text{false}$  and  $g_1|_\pi = \text{true}$ , then the existential player loses  $\Phi'_{\text{in}}|_\pi$ . This asymmetry can be harmful to QBF solvers. For example, consider the QBF

$$\forall \mathbf{x}. \exists y. y \vee \underbrace{\psi(\mathbf{x})}_{g_1} \quad (2)$$

This formula is trivially true. A winning move for the existential player is to make  $y$  be true, which immediately makes the matrix of the formula true, regardless of  $\psi$ . Under the Tseitin transformation, Equation 2 becomes:

$$\forall \mathbf{x}. \exists y. \exists g. (y \vee g_1) \wedge (\text{clauses equating gate variables})$$

Setting  $y$  to be true no longer immediately makes the matrix true. Instead, for each assignment of universal variables  $\mathbf{x}$ , the QBF solver must actually find a satisfying assignment to the gate variables. This makes it much harder to detect when the existential player has won. Experimental results [1,22] indicate that purely CNF-based QBF solvers would, in the worst case, require time exponential in the size of  $\psi$  to solve the CNF formula, even though the original problem (before translation to CNF) is trivial.

### 3 Ghost Variables and Sequents

We employ *ghost variables* to provide a modification of the Tseitin transformation that is symmetric between the two players. The idea of using a symmetric transformation was first explored in [22], which performed the Tseitin transformation twice: once on the input formula, and once on its negation. Similar ideas have been used to handle non-prenex formulas in [14] and to handle “don’t care” propagation in [12].

For each gate variable  $g$ , we introduce two *ghost variables*: an existentially quantified variable  $g^\exists$  and a universally quantified variable  $g^\forall$ . We say that  $g^\exists$  and  $g^\forall$  represent the formula labeled by  $g$ . Ghost variables are considered to be downstream of all input variables.

We now introduce a semantics with ghost variables for the game formulation of QBF. As in the Tseitin transformation, the existential player should lose if an existential ghost variable  $g^\exists$  is assigned a different value than the subformula that it represents. Additionally, the universal player should lose if an universal ghost variable  $g^\forall$  is assigned a different value than the subformula that it represents.

In this paper, we never consider formulas (other than single literals) in which ghost variables occur as actual variables. In particular, if  $\Phi$  is the input formula to the QBF solver, then in a substitution  $\Phi|_\pi$ , ghost variables in  $\pi$  have no effect.

**Definition 1 (Consistent assignment to ghost literal).** Given a quantifier type  $Q \in \{\exists, \forall\}$  and an assignment  $\pi$ , we say that a ghost literal  $g^Q$  is assigned **consistently** under  $\pi$  iff  $g^Q|_\pi = (\text{the formula represented by } g^Q)|_\pi$ .

**Definition 2 (Winning under a total assignment).** Given a formula  $\Phi$ , a quantifier type  $Q \in \{\exists, \forall\}$ , and an assignment  $\pi$  to all the input variables and a subset of the ghost variables, we say “Player  $Q$  **wins**  $\Phi$  under  $\pi$ ” iff:

- $\Phi|_\pi = \text{true}$  if  $Q$  is  $\exists$ , and
  - $\Phi|_\pi = \text{false}$  if  $Q$  is  $\forall$ , and
  - Every ghost variable owned by  $Q$  in  $\text{vars}(\pi)$  is assigned consistently.
- (Intuitively, a winning player’s ghost variables must “respect the encoding”).

For example, if  $\Phi = \exists e. \forall u. (e \wedge u)$  and  $g$  labels  $(e \wedge u)$  then neither player wins  $\Phi$  under  $\{\neg e, u, g^\forall, \neg g^\exists\}$ . The existential player loses because  $\Phi|_\pi = \text{false}$ , and the universal player loses because  $g^\forall|_\pi \neq$  (the formula represented by  $g^\forall|_\pi$ ).

**Definition 3 (Losing under a total assignment).** Given a formula  $\Phi$  and an assignment  $\pi$  that assigns all the input variables, we say “Player  $Q$  loses  $\Phi$  under  $\pi$ ” iff Player  $Q$  does not win  $\Phi$  under  $\pi$ .

**Definition 4 (Losing under a partial assignment).** Given a formula  $\Phi$ , an assignment  $\pi$ , and an outermost unassigned input variable  $x$ , we say “Player  $Q$  loses  $\Phi$  under  $\pi$ ” iff either:

- Player  $Q$  loses  $\Phi$  under both  $\pi \cup \{(x, \text{true})\}$  and  $\pi \cup \{(x, \text{false})\}$ , or
- $Q$ ’s opponent owns  $x$  and Player  $Q$  loses  $\Phi$  under either  $\pi \cup \{(x, \text{true})\}$  or  $\pi \cup \{(x, \text{false})\}$ .

For example, consider a formula  $\Phi = \exists e. x \wedge e$ , where  $x$  is a free variable. The existential player loses  $\Phi$  under  $\{\neg x\}$  and under  $\{\neg e\}$ . Neither player can be said to lose  $\Phi$  under the empty assignment, because the value of  $\Phi$  depends on the free variable  $x$ . Now let us make a few general observations about when a player loses under an arbitrary partial assignment.

**Observation 1.** If  $\Phi|_\pi = \text{true}$ , then Player  $\forall$  loses  $\Phi$  under  $\pi$ .

**Observation 2.** If  $\Phi|_\pi = \text{false}$ , then Player  $\exists$  loses  $\Phi$  under  $\pi$ .

**Observation 3.** If a ghost variable owned by  $Q$  in  $\text{vars}(\pi)$  is assigned inconsistently under  $\pi$ , then Player  $Q$  loses  $\Phi$  under  $\pi$ .

**Observation 4.** If the opponent of  $Q$  owns a literal  $\ell$  that is unassigned under  $\pi$ , and  $Q$  loses  $\Phi$  under  $\pi \cup \{\ell\}$ , then  $Q$  loses  $\Phi$  under  $\pi$ .

**Definition 5 (Game-State Specifier, Match).** A **game-state specifier** is a pair  $\langle L^{\text{now}}, L^{\text{fut}} \rangle$  consisting of two sets of literals,  $L^{\text{now}}$  and  $L^{\text{fut}}$ . We say that  $\langle L^{\text{now}}, L^{\text{fut}} \rangle$  **matches** an assignment  $\pi$  iff:

1. for every literal  $\ell$  in  $L^{\text{now}}$ ,  $\ell|_\pi = \text{true}$ , and
2. for every literal  $\ell$  in  $L^{\text{fut}}$ , either  $\ell|_\pi = \text{true}$  or  $\ell \notin \text{vars}(\pi)$ .

For example,  $\langle \{u\}, \{e\} \rangle$  matches the assignments  $\{u\}$  and  $\{u, e\}$ , but does not match  $\{\}$  or  $\{u, \neg e\}$ . Note that, for any literal  $\ell$ , if  $\{\ell, \neg \ell\} \subseteq L^{\text{fut}}$ , then  $\langle L^{\text{now}}, L^{\text{fut}} \rangle$  matches an assignment  $\pi$  only if  $\pi$  doesn’t assign  $\ell$ . The intuition behind the names “ $L^{\text{now}}$ ” and “ $L^{\text{fut}}$ ” is as follows: Under the game formulation of QBF, the assignment  $\pi$  can be thought of as a state of the game, and  $\pi$  matches  $\langle L^{\text{now}}, L^{\text{fut}} \rangle$  iff every literal in  $L^{\text{now}}$  is already true in the game and, for every literal  $\ell$  in  $L^{\text{fut}}$ , it is possible that  $\ell$  can be true in a future state of the game.

**Definition 6 (Game Sequent).** The sequent “ $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (Q \text{ loses } \Phi)$ ” means “Player  $Q$  loses  $\Phi$  under all assignments that match  $\langle L^{\text{now}}, L^{\text{fut}} \rangle$ .”

As an example, let  $\Phi$  be the following formula:

$$\forall u. \exists e. (e \vee \neg u) \wedge (u \vee \neg e) \wedge \overbrace{(x_1 \vee e)}^{g_3}$$

Note that sequent  $\langle \{u\}, \{e\} \rangle \models (\forall \text{ loses } \Phi)$  holds true: in any assignment  $\pi$  that matches it,  $\Phi|_{\pi} = \text{true}$ . However,  $\langle \{u\}, \emptyset \rangle \models (\forall \text{ loses } \Phi)$  does not hold true: it matches the assignment  $\{u, \neg e\}$ , under which Player  $\forall$  does not lose  $\Phi$ . Finally,  $\langle \{g_3^{\forall}\}, \{e, \neg e\} \rangle \models (\forall \text{ loses } \Phi)$  holds true. Let us consider why Player  $\forall$  loses  $\Phi$  under the assignment  $\{g_3^{\forall}\}$ . The free variable  $x_1$  is the outermost unassigned variable, so under Definition 4, Player  $\forall$  loses under  $\{g_3^{\forall}\}$  iff Player  $\forall$  loses under both  $\{g_3^{\forall}, x_1\}$  and  $\{g_3^{\forall}, \neg x_1\}$ . Under  $\{g_3^{\forall}, x_1\}$ , Player  $\forall$  loses because  $\Phi|_{\{g_3^{\forall}, x_1\}}$  evaluates to **true**. Under  $\{g_3^{\forall}, \neg x_1\}$ , Player  $\forall$  loses because  $e$  is owned by the opponent of Player  $\forall$  and  $g_3^{\forall}$  is assigned inconsistently under  $\{g_3^{\forall}, \neg x_1, \neg e\}$ .

Note that a clause  $(\ell_1 \vee \dots \vee \ell_n)$  in a CNF formula  $\Phi_{\text{in}}$  is equivalent to the sequent  $\langle \{\neg \ell_1, \dots, \neg \ell_n\}, \emptyset \rangle \models (\exists \text{ loses } \Phi_{\text{in}})$ . (Sequents in this form can also be considered similar to *nogoods* [19].) Likewise, a cube  $(\ell_1 \wedge \dots \wedge \ell_n)$  in a DNF formula  $\Phi_{\text{in}}$  is equivalent to the sequent  $\langle \{\ell_1, \dots, \ell_n\}, \emptyset \rangle \models (\forall \text{ loses } \Phi_{\text{in}})$ .

### 3.1 Sequents with Free Variables

Above, we introduced sequents that indicate if a player loses a formula  $\Phi$ . Now, we will generalize sequents so that they can indicate that  $\Phi$  evaluates to a quantifier-free formula involving the free variables. To do this, we first introduce a logical semantics for QBF with ghost variables. Given a formula  $\Phi$  and an assignment  $\pi$  that assigns all the input variables, we want the semantic evaluation  $\llbracket \Phi \rrbracket_{\pi}$  to have the following properties:

1.  $\llbracket \Phi \rrbracket_{\pi} = \text{true}$  iff the existential player wins  $\Phi$  under  $\pi$ .
2.  $\llbracket \Phi \rrbracket_{\pi} = \text{false}$  iff the universal player wins  $\Phi$  under  $\pi$ .

Note that the above properties cannot be satisfied in a two-valued logic if both players lose  $\Phi$  under  $\pi$ . So, we use a three-valued logic with a third value **dontcare**. We call it “don’t care” because we are interested in the outcome of the game when both players make the best possible moves, but if both players fail to win, then clearly at least one of the players failed to make the best possible moves. In our three-valued logic, a conjunction of boolean values evaluates to **false** if any conjunct is **false**, and otherwise it evaluates to **dontcare** if any conjunct is **dontcare**. Disjunction is defined analogously. The negation of **dontcare** is **dontcare**. In a truth table:

| $x$   | $y$      | $x \wedge y$ | $x \vee y$ |
|-------|----------|--------------|------------|
| true  | dontcare | dontcare     | true       |
| false | dontcare | false        | dontcare   |

**Definition 7.** Given an assignment  $\pi$  to all the input variables and a subset of the ghost variables, we define  $\llbracket \Phi \rrbracket_\pi$  as follows:

$$\llbracket \Phi \rrbracket_\pi := \begin{cases} \text{true} & \text{if Player } \exists \text{ wins } \Phi \text{ under } \pi \\ \text{false} & \text{if Player } \forall \text{ wins } \Phi \text{ under } \pi \\ \text{dontcare} & \text{if both players lose } \Phi \text{ under } \pi \end{cases}$$

For convenience in defining  $\llbracket \Phi \rrbracket_\pi$  for a partial assignment  $\pi$ , we assume that the formula is prepended with a dummy “quantifier” block for free variables. For example,  $(\exists e. e \wedge z)$  becomes  $(\mathcal{F}z. \exists e. e \wedge z)$ , where  $\mathcal{F}$  denotes the dummy block for free variables. If  $\Phi$  contains free variables unassigned by  $\pi$  then  $\llbracket \Phi \rrbracket_\pi$  is a formula in terms of these free variables. We define  $\llbracket \Phi \rrbracket_\pi$  as follows for a partial assignment  $\pi$  that assigns only a proper subset of the input variables:

$$\begin{aligned} \llbracket Qx. \Phi \rrbracket_\pi &= \llbracket \Phi \rrbracket_\pi \quad \text{if } x \in \text{vars}(\pi) \\ \llbracket \exists x. \Phi \rrbracket_\pi &= \llbracket \Phi \rrbracket_{(\pi \cup \{x\})} \vee \llbracket \Phi \rrbracket_{(\pi \cup \{\neg x\})} \quad \text{if } x \notin \text{vars}(\pi) \\ \llbracket \forall x. \Phi \rrbracket_\pi &= \llbracket \Phi \rrbracket_{(\pi \cup \{x\})} \wedge \llbracket \Phi \rrbracket_{(\pi \cup \{\neg x\})} \quad \text{if } x \notin \text{vars}(\pi) \\ \llbracket \mathcal{F}x. \Phi \rrbracket_\pi &= x ? \llbracket \Phi \rrbracket_{(\pi \cup \{x\})} : \llbracket \Phi \rrbracket_{(\pi \cup \{\neg x\})} \quad \text{if } x \notin \text{vars}(\pi) \end{aligned}$$

The notation “ $x ? \phi_1 : \phi_2$ ” denotes a formula with an *if-then-else* construct that is logically equivalent to  $(x \wedge \phi_1) \vee (\neg x \wedge \phi_2)$ . Note that the branching on the free variables here is similar to the Shannon expansion [17].

**Remark.** Do we really need to add the dummy blocks for free variables and have the rule for  $\llbracket \mathcal{F}x. \Phi \rrbracket_\pi$  in Definition 7? Yes, because if  $\pi$  contains a ghost literal  $g^Q$  that represents a formula containing variables free in  $\Phi$ , then it doesn’t make sense to ask if  $g^Q$  is assigned consistently under  $\pi$  unless all the variables in the formula represented by  $g^Q$  are assigned by  $\pi$ .

**Definition 8 (Sometimes-Dontcare).** A formula  $\phi$  is said to be **sometimes-dontcare** iff there is an assignment  $\pi$  under which  $\phi$  evaluates to **dontcare**. For example,  $(x \vee \text{dontcare})$  is sometimes-dontcare, while  $(x \vee (x \wedge \text{dontcare}))$  is not sometimes-dontcare (because it evaluates to **true** if  $x$  is **true** and evaluates to **false** if  $x$  is **false**).

**Definition 9 (Free Sequent).** The sequent “ $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models \Phi \Leftrightarrow \psi$ ” means “for all assignments  $\pi$  that match  $\langle L^{\text{now}}, L^{\text{fut}} \rangle$ , if  $\llbracket \Phi \rrbracket_\pi$  is not sometimes-dontcare, then  $\llbracket \Phi \rrbracket_\pi$  is logically equivalent to  $\psi|_\pi$ ”.

**Remark.** The sequent definitions in Definitions 9 and 6 are related as follows:

- “ $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\exists \text{ loses } \Phi)$ ” means the same as “ $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \text{false})$ ”.
- “ $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\forall \text{ loses } \Phi)$ ” means the same as “ $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \text{true})$ ”.

We treat a game sequent as interchangeable with the corresponding free sequent.

Sequents of the form  $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models \Phi \Leftrightarrow \psi$  extend clause/cube learning by allowing  $\psi$  to be a formula (in terms of the variables free in  $\Phi$ ) in addition to the constants **true** and **false**. This enables handling of formulas with free variables.

## 4 Algorithm

The top-level algorithm, shown in Figure 2, is based on the well-known DPLL algorithm, except that sequents are used instead of clauses. Similar to how SAT solvers maintain a *clause database* (i.e., a set of clauses whose conjunction is equisatisfiable with the original input formula  $\Phi_{\text{in}}$ ), our solver maintains a *sequent database*. A SAT solver's clause database is initialized to contain exactly the set of clauses produced by the Tseitin transformation of the input formula  $\Phi_{\text{in}}$  into CNF. Likewise, our sequent database is initialized (§ 4.1) to contain a set of sequents analogous to the clauses produced by the Tseitin transformation.

In the loop on lines 4–7, the solver chooses an outermost unassigned literal, adds it to  $\pi_{\text{cur}}$ , and performs boolean constraint propagation (BCP). BCP may add further literals to  $\pi_{\text{cur}}$ , as described in detail in § 4.4; such literals are referred to as *forced literals*, in distinction to the literals added by **DecideLit**, which are referred to as *decision literals*. The stopping condition for the loop is when the current assignment matches a sequent already in the database. (The analogous stopping condition for a SAT solver would be when a clause is falsified.) When this stopping condition is met, the solver performs an analysis similar to that of *clause learning* [18] to learn a new sequent (line 8). If the  $L^{\text{now}}$  component of the learned sequent is empty, then the solver has reached the final answer, which it returns (line 9). Otherwise, the solver backtracks to the earliest decision level at which the newly learned sequent will trigger a forced literal in BCP. (The learning algorithm guarantees that this is possible.) The solver then performs BCP (line 11) and returns to the inner loop at line 4.

The intuition behind BCP for quantified variables is fairly straightforward; a literal owned by  $Q$  is forced by a sequent if the sequent indicates that  $Q$  need to make  $\ell$  true to avoid losing. For free variables, the intuition is slightly different. Free variables are forced to prevent the solver from re-exploring parts of the

---

```

1.  initialize_sequent_database();
2.   $\pi_{\text{cur}} := \emptyset$ ; Propagate();
3.  while (true) {
4.    while ( $\pi_{\text{cur}}$  doesn't match any database sequent) {
5.      DecideLit();
6.      Propagate();
7.    }
8.    Learn();
9.    if (learned seq has form  $\langle \emptyset, L^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$ ) return  $\psi$ ;
10.   Backtrack();
11.   Propagate();
12. }
```

---

**Fig. 2.** Top-Level Algorithm. Details have been omitted for sake of clarity.



search space that it has already seen, so that the solver is continuously making progress in exploring the search space, thereby guaranteeing it would eventually terminate (given enough time and memory). (Actually, this intuition also applies to quantified variables.)

The solver maintains a list of assigned literals in the order in which they were assigned; this list is referred to as the *trail* [9]. Given a decision literal  $\ell_d$ , we say that all literals that appear in the trail after  $\ell_d$  but before any other decision literal belong to the same *decision level* as  $\ell_d$ .

For prenex formulas without free variables, the algorithm described here is operationally very similar to standard DPLL QBF solvers, except that  $L^{\text{now}}$  and  $L^{\text{fut}}$  do not need to be explicitly separated, since  $L^{\text{now}}$  always consists exactly of all the loser's literals. However, for formulas with free variables, it is necessary to explicitly record which literals belong in  $L^{\text{now}}$  and which in  $L^{\text{fut}}$ .

#### 4.1 Initial Sequents

We initialize the sequent database to contain a set of *initial sequents*, which correspond to the clauses produced by the Tseitin transformation of the input formula  $\Phi_{\text{in}}$ . The set of initial sequents must be sufficient to ensure the loop on line 4–6 of Figure 2 (which adds unassigned literals to the current assignment until it matches a sequent in the database) operates properly. That is, for every possible total assignment  $\pi$ , there must be at least one sequent that matches  $\pi$ .

First, let us consider a total assignment  $\pi$  in which both players assign all their ghost variables consistently (Definition 1). In order to handle this case, we generate the following two initial sequents, where  $g_{\text{in}}$  is the label of the input formula  $\Phi_{\text{in}}$ :  $\langle \{-g_{\text{in}}^{\exists}\}, \emptyset \rangle \models (\exists \text{ loses } \Phi_{\text{in}})$  and  $\langle \{g_{\text{in}}^{\forall}\}, \emptyset \rangle \models (\forall \text{ loses } \Phi_{\text{in}})$ .

Since all ghost variables are assigned consistently in  $\pi$ , it follows that, for each gate  $g$ ,  $g^{\exists}|\pi$  must equal  $g^{\forall}|\pi$ , since both  $g^{\exists}$  and  $g^{\forall}$  must each be assigned the same value as the formula that  $g$  labels. In particular,  $g_{\text{in}}^{\exists}|\pi$  must be equal to  $g_{\text{in}}^{\forall}|\pi$ , so  $\pi$  must match exactly one of the two above initial sequents.

Now let us consider a total assignment  $\pi$  in which at least one player assigns a ghost variable inconsistently. In order to handle this case, we generate a set of initial sequents for every conjunction and disjunction in  $\Phi_{\text{in}}$ . Let  $g_*$  be the label of an arbitrary conjunction in  $\Phi_{\text{in}}$  of the form

$$(x_1 \wedge \dots \wedge x_n \wedge \underbrace{\phi_1}_{g_1} \wedge \dots \wedge \underbrace{\phi_m}_{g_m})$$

where  $x_1$  through  $x_n$  are input literals. The following initial sequents are produced from this conjunction for each  $Q \in \{\exists, \forall\}$ :

1.  $\langle \{g_*^Q, \neg x_i\}, \emptyset \rangle \models (Q \text{ loses } \Phi_{\text{in}})$  for  $i \in \{1, \dots, n\}$
2.  $\langle \{g_*^Q, \neg g_i^Q\}, \emptyset \rangle \models (Q \text{ loses } \Phi_{\text{in}})$  for  $i \in \{1, \dots, m\}$
3.  $\langle \{\neg g_*^Q, x_1, \dots, x_n, g_1^Q, \dots, g_m^Q\}, \emptyset \rangle \models (Q \text{ loses } \Phi_{\text{in}})$

Now let  $g_*^Q$  denote a ghost literal such that (1)  $g_*^Q$  is inconsistently assigned under  $\pi$  and (2) no proper subformula of the formula represented by  $g_*^Q$  is labelled by a inconsistently-assigned ghost variable. Then  $\pi$  must match one of the above-listed initials sequents.

## 4.2 Normalization of Initial Sequents

Note that all the initial sequents have the form  $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (Q \text{ loses } \Phi)$  where  $L^{\text{fut}} = \emptyset$ . We normalize these sequents by moving all literals owned by  $Q$ 's opponent from  $L^{\text{now}}$  to  $L^{\text{fut}}$ , in accordance with the following inference rule:

$$\frac{\begin{array}{l} \text{The opponent of } Q \text{ owns } \ell, \text{ and } \neg\ell \notin L^{\text{fut}} \\ \langle L^{\text{now}} \cup \{\ell\}, L^{\text{fut}} \rangle \models (Q \text{ loses } \Phi) \end{array}}{\langle L^{\text{now}}, L^{\text{fut}} \cup \{\ell\} \rangle \models (Q \text{ loses } \Phi)}$$

To prove the above inference rule, we consider an arbitrary assignment  $\pi$  that matches  $\langle L^{\text{now}}, L^{\text{fut}} \cup \{\ell\} \rangle$ , assume that the premises of inference rule hold true, and prove that Player  $Q$  loses under  $\pi$ :

1.  $\pi$  matches  $\langle L^{\text{now}}, L^{\text{fut}} \cup \{\ell\} \rangle$  (by assumption).
2.  $\pi \cup \{\ell\}$  matches  $\langle L^{\text{now}} \cup \{\ell\}, L^{\text{fut}} \rangle$  (using the premise that  $\neg\ell \notin L^{\text{fut}}$ ).
3.  $Q$  loses  $\Phi$  under  $\pi \cup \{\ell\}$  (by the premise  $\langle L^{\text{now}} \cup \{\ell\}, L^{\text{fut}} \rangle \models (Q \text{ loses } \Phi)$ ).
4.  $Q$  loses  $\Phi$  under  $\pi$  (by Observation 4 on page 419).

## 4.3 Properties of Sequents in Database

After the initial sequents have been normalized (as described in § 4.2), the solver maintains the following invariants for all sequents in the sequent database, including sequents added to the database as a result of learning (§ 4.5):

1. In a sequent of the form  $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (Q \text{ loses } \Phi_{\text{in}})$ :
  - (a) Every literal in  $L^{\text{now}}$  either is owned by  $Q$  or is free in  $\Phi_{\text{in}}$ .
  - (b) Every literal in  $L^{\text{fut}}$  is owned by the opponent of  $Q$ .
2. In a sequent of the form  $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$ , every variable in  $\psi$  appears both positively and negatively in  $L^{\text{fut}}$  (i.e., if  $r$  occurs in  $\psi$ , then  $\{r, \neg r\} \subseteq L^{\text{fut}}$ ). This is guaranteed by the learning algorithm in § 4.5.

## 4.4 Propagation

The **Propagate** procedure is similar to that of closed-QBF solvers. Consider a sequent  $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$  in the sequent database. If, under  $\pi_{\text{cur}}$ ,

1. there is exactly one unassigned literal  $\ell$  in  $L^{\text{now}}$ , and
2. no literals in  $L^{\text{now}} \cup L^{\text{fut}}$  are assigned **false**, and
3.  $\ell$  is not downstream of any unassigned literals in  $L^{\text{fut}}$ ,

then  $\neg\ell$  is *forced* — it is added to the current assignment  $\pi_{\text{cur}}$ . In regard to the 3rd condition, if an unassigned literal  $r$  in  $L^{\text{fut}}$  is upstream of  $\ell$ , then  $r$  should get assigned before  $\ell$ , and if  $r$  gets assigned **false**, then  $\ell$  shouldn't get forced at all by the sequent. Propagation ensures that the solver never re-explores areas of the search space for which it already knows the answer, ensuring continuous progress and eventual termination. It is instructive to consider how the propagation rule applies in light of the properties of sequents discussed in § 4.3:

1. A sequent of the form  $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (Q \text{ loses } \Phi_{\text{in}})$  can force a literal that is either owned by  $Q$  or free in  $\Phi_{\text{in}}$ ; it cannot force a literal owned by  $Q$ 's opponent. If  $\ell$  is owned by  $Q$ , then the reason for forcing  $\neg\ell$  is intuitive: the only way for  $Q$  to avoid losing is to add  $\neg\ell$  to the current assignment. If  $\ell$  is free in  $\Phi_{\text{in}}$ , then  $\neg\ell$  is forced because the value of  $\llbracket \Phi_{\text{in}} \rrbracket_{\pi_{\text{cur}} \cup \{\ell\}}$  is already known and the solver shouldn't re-explore that same area of the search space.
2. A sequent of the form  $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$ , where  $\psi$  contains free variables, can only force a literal that is free in  $\Phi_{\text{in}}$ . Although  $L^{\text{now}}$  can contain literals owned by Player  $\exists$  and Player  $\forall$ , such literals cannot be forced by the sequent. To prove this, we consider two cases: either there exists a variable  $v$  that occurs in  $\psi$  and is assigned by  $\pi_{\text{cur}}$ , or all variables that occur  $\psi$  are left unassigned by  $\pi_{\text{cur}}$ . If there is variable  $v$  in  $\psi$  that is assigned by  $\pi_{\text{cur}}$ , then  $\pi_{\text{cur}}$  cannot match  $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$ , since  $\{v, \neg v\} \subseteq L^{\text{fut}}$ . If there is a variable  $v$  in  $\psi$  that is left unassigned by  $\pi_{\text{cur}}$ , then  $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$  cannot force any quantified variable, since  $v$  occurs in  $L^{\text{fut}}$  and all quantified variables are downstream of free variable  $v$ .

We employ a variant of the watched-literals rule designed for SAT solvers [16] and adapted for QBF solvers [10]. For each sequent  $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (\Phi \Leftrightarrow \psi)$ , we watch two literals in  $L^{\text{now}}$  and one literal in  $L^{\text{fut}}$ .

## 4.5 Learning

In the top-level algorithm in Figure 2, the solver performs *learning* (line 8) after the current assignment  $\pi_{\text{cur}}$  matches a sequent in the database. The learning procedure is based on the clause learning introduced for SAT in [18] and adapted for QBF in [24]. We use inference rules shown in Figure 4 to add new sequents to the sequent database. These rules, in their  $L^{\text{now}}$  components, resemble the *resolution* rule used in SAT (i.e., from  $(A \vee r) \wedge (\neg r \vee B)$  infer  $A \vee B$ ). The learning algorithm ensures that the solver remembers the parts of the search space for which it has already found an answer. This, together with propagation, ensures that solver eventually covers all the necessary search space and terminates.

The learning procedure, shown in Figure 3, works as follows. Let *seq* be the database sequent that matches the current assignment  $\pi_{\text{cur}}$ . Let  $r$  be the literal in the  $L^{\text{now}}$  component of *seq* that was most recently added to  $\pi_{\text{cur}}$  (i.e., the latest one in the *trail*). Note that  $r$  must be a forced literal (as opposed to a decision literal), because only an outermost unassigned literal can be picked as a decision literal, but if  $r$  was outermost immediately before it added to  $\pi_{\text{cur}}$ ,

---

```

func Learn() {
  seq := (the database sequent that matches  $\pi_{\text{cur}}$ );
  do {
     $r :=$  (the most recently assigned literal in  $\text{seq.L}^{\text{now}}$ )
    seq := Resolve(seq, antecedent[ $r$ ]);
  } until ( $\text{seq.L}^{\text{now}} = \emptyset$  or has_good_UIP(seq));
  return seq;
}

```

---

**Fig. 3.** Procedure for learning new sequents

---

**Resolving on a literal  $r$  owned by Player  $Q$  (case 1):**

The quantifier type of  $r$  in  $\Phi$  is  $Q$

$$\langle L_1^{\text{now}} \cup \{r\}, L_1^{\text{fut}} \rangle \models (Q \text{ loses } \Phi_{\text{in}})$$

$$\langle L_2^{\text{now}} \cup \{\neg r\}, L_2^{\text{fut}} \rangle \models (Q \text{ loses } \Phi_{\text{in}})$$

$r$  is not downstream of any  $\ell$  such that  $\ell \in L_1^{\text{fut}}$  and  $\neg \ell \in (L_1^{\text{fut}} \cup L_2^{\text{fut}})$

---


$$\langle L_1^{\text{now}} \cup L_2^{\text{now}}, L_1^{\text{fut}} \cup L_2^{\text{fut}} \rangle \models (Q \text{ loses } \Phi_{\text{in}})$$

**Resolving on a literal  $r$  owned by Player  $Q$  (case 2):**

The quantifier type of  $r$  in  $\Phi$  is  $Q$

$$\langle L_1^{\text{now}} \cup \{r\}, L_1^{\text{fut}} \rangle \models (Q \text{ loses } \Phi_{\text{in}})$$

$$\langle L_2^{\text{now}} \cup \{\neg r\}, L_2^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$$

$r$  is not downstream of any  $\ell$  such that  $\ell \in L_1^{\text{fut}}$  and  $\neg \ell \in (L_1^{\text{fut}} \cup L_2^{\text{fut}})$

---


$$\langle L_1^{\text{now}} \cup L_2^{\text{now}}, L_1^{\text{fut}} \cup L_2^{\text{fut}} \cup \{\neg r\} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi)$$

**Resolving on a variable  $r$  that is free in  $\Phi_{\text{in}}$ :**

Literal  $r$  is free

$$\langle L_1^{\text{now}} \cup \{r\}, L_1^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi_1)$$

$$\langle L_2^{\text{now}} \cup \{\neg r\}, L_2^{\text{fut}} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \psi_2)$$

---


$$\langle L_1^{\text{now}} \cup L_2^{\text{now}}, L_1^{\text{fut}} \cup L_2^{\text{fut}} \cup \{r, \neg r\} \rangle \models (\Phi_{\text{in}} \Leftrightarrow (r ? \psi_1 : \psi_2))$$


---

**Fig. 4.** Resolution-like inference rules

then no unassigned literal in the  $L^{\text{fut}}$  component of  $seq$  was upstream of  $r$ , so  $seq$  would have forced  $\neg r$  in accordance with §4.4. We use the inference rules in Figure 4 to infer a new sequent from  $seq$  and the *antecedent* of  $r$  (i.e., the sequent that forced  $r$ ). This is referred to as *resolving* due to the similarity of the inference rules to the clause resolution rule. We stop and return the newly inferred sequent if it has a “good” unique implication point (UIP) [24], i.e., if there is a literal  $\ell$  in the  $L^{\text{now}}$  component such that

1. Every literal in  $(L^{\text{now}} \setminus \{\ell\})$  belongs to an earlier decision level than  $\ell$ ,
2. Every literal in  $L^{\text{fut}}$  upstream of  $\ell$  belongs to a decision level earlier than  $\ell$ .
3. If  $seq$  has the form  $\langle L^{\text{now}}, L^{\text{fut}} \rangle \models (Q \text{ loses } \Phi_{\text{in}})$ , then the decision variable of the decision level of  $\ell$  is not owned by the opponent of  $Q$ .

Otherwise, we resolve the sequent with the antecedent of the most recently assigned literal in its  $L^{\text{now}}$  component, and continue this process until the stopping conditions above are met or  $L^{\text{now}}$  is empty. Note that if the most recently assigned literal in  $L^{\text{now}}$  is a decision literal, then it is a good UIP.

Note that in the resolution rule for resolving on a free variable  $r$ , we add both  $r$  and  $\neg r$  to  $L^{\text{fut}}$ . This is not necessary for soundness of the resolution itself. Rather, it is to ensure that the properties in §4.3 hold true. Without these properties, a quantified variable could be forced by a sequent that is not equivalent to a clause or a cube.

**Example.** Below, we give several applications of the resolution rules. For brevity, we omit free variables from the  $L^{\text{fut}}$  component.

$$\exists e_3. \underbrace{(i_1 \wedge e_3)}_{g_5} \vee \underbrace{(i_2 \wedge \neg e_3)}_{g_4}$$

1. Start:  $\langle \{\neg i_1, \neg i_2\}, \{\} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \text{false})$
2. Resolve  $\neg i_1$  via  $\langle \{i_1, \neg g_5^\forall\}, \{e_3\} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \text{true})$   
Result:  $\langle \{\neg i_2, \neg g_5^\forall\}, \{e_3\} \rangle \models (\Phi_{\text{in}} \Leftrightarrow i_1)$
3. Resolve  $\neg i_2$  via  $\langle \{i_2, \neg g_4^\forall\}, \{\neg e_3\} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \text{true})$   
Result:  $\langle \{\neg g_5^\forall, \neg g_4^\forall\}, \{e_3, \neg e_3\} \rangle \models (\Phi_{\text{in}} \Leftrightarrow (i_1 \vee i_2))$
4. Resolve  $\neg g_4^\forall$  via  $\langle \{g_4^\forall\}, \{\} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \text{true})$   
Result:  $\langle \{\neg g_5^\forall\}, \{e_3, \neg e_3, \neg g_4^\forall\} \rangle \models (\Phi_{\text{in}} \Leftrightarrow (i_1 \vee i_2))$
5. Resolve  $\neg g_5^\forall$  via  $\langle \{g_5^\forall\}, \{\} \rangle \models (\Phi_{\text{in}} \Leftrightarrow \text{true})$   
Result:  $\langle \{\}, \{e_3, \neg e_3, \neg g_4^\forall, \neg g_5^\forall\} \rangle \models (\Phi_{\text{in}} \Leftrightarrow (i_1 \vee i_2))$

#### 4.6 Justification of Inference Rules

The first inference rule in Figure 4 is analogous to long-distance resolution [23] and can be proved by similar methods (e.g., [2]). Intuitively, if the current

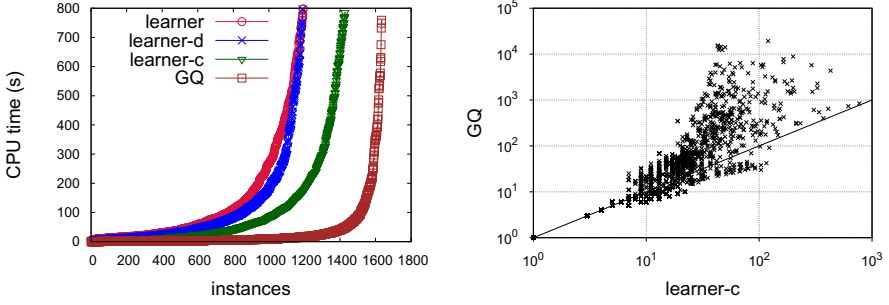


**Fig. 5.** Time and size comparisons, instances solved by all solvers in less than 10 s are not included in the time comparison

assignment matches  $\langle L_1^{\text{now}} \cup L_2^{\text{now}}, L_1^{\text{fut}} \cup L_2^{\text{fut}} \rangle$ , then the opponent of  $Q$  can make  $Q$  lose  $\Phi_{\text{in}}$  by assigning **true** to all the literals in  $L_1^{\text{fut}}$  that are upstream of  $r$ . This forces  $Q$  to assign  $r = \text{false}$  to avoid matching the first sequent in the premise of the inference rule, but assigning  $r = \text{false}$  makes the current assignment match the second sequent in the premise.

If the current assignment  $\pi_{\text{cur}}$  matches the sequent in the conclusion of the second inference rule, there are two possibilities. For simplicity, assume that  $\pi_{\text{cur}}$  assigns all free variables and that neither  $L_1^{\text{fut}}$  nor  $L_2^{\text{fut}}$  contains any free literals (since, as mentioned earlier, free literals can be removed from  $L^{\text{fut}}$  without affecting soundness of the sequent). If  $Q$  loses  $\psi$  under  $\pi_{\text{cur}}$ , then the situation is similar to first inference rule. If the opponent of  $Q$  loses  $\psi$  under  $\pi_{\text{cur}}$ , then  $Q$  can make his opponent lose  $\Phi_{\text{in}}$  by assigning  $r = \text{false}$ , thereby making the current assignment match the second sequent of the premise.

For the third inference rule, we don't need a condition about  $r$  not being downstream of other literals, since no free variable is downstream of any variable.

## 5 Experimental Results

We extended the existing closed-QBF solver GhostQ [14] to implement the techniques described in this paper. For comparison, we used the solvers and load-balancer benchmarks from [3].<sup>1</sup> The benchmarks contain multiple alternations of quantifiers and are derived from problems involving the automatic synthesis of a reactive system from a formal specification. The experimental results were obtained on Intel Xeon 5160 3-GHz machines with 4 GB of memory. The time limit was 800 seconds and the memory limit to 2 GB.

<sup>1</sup> The results do not exactly match the results reported in [3] because we did not preprocess the QDIMACS input files. We found that sometimes the output of the preprocessor was not logically equivalent to its input. With the unpreprocessed inputs, the output formulas produced by the learner family of solvers were always logically equivalent to the output formulas of GhostQ.

There are three solvers from [3], each with a different form of the output: CDNF (a conjunction of DNFs), CNF, and DNF. We will refer to these solvers as “Learner” (CDNF), “Learner-C” (CNF), and “Learner-D” (DNF). Figure 5 compares these three solvers with GhostQ on the “hard” benchmarks (those that not all four solvers could solve within 10 seconds). As can be seen on the figure, GhostQ solved about 1600 of these benchmarks, Learner-C solved about 1400, and Learner-D and Learner each solved about 1200. GhostQ solved 223 instances that Learner-C couldn’t solve, while Learner-C solved 16 instances that GhostQ couldn’t solve. GhostQ solved 375 instances that neither Learner-DNF nor Learner could solve, while there were only 2 instances that either Learner-DNF or Learner could solve but GhostQ couldn’t solve.

Figure 5 shows a comparison of the size of the output formulas for GhostQ and Learner-C, indicating that the GhostQ formulas are often significantly larger. The size is computed as 1 plus the number of edges in the DAG representation of the formula, not counting negations, and after certain simplifications. E.g., the size of  $x$  is 1, the size of  $\neg x$  is also 1, and the size of  $x \wedge y$  is 3.

## 6 Related Work

Ken McMillan [15] proposed a method to use SAT solvers to perform quantifier elimination on formulas of the form  $\exists \mathbf{x}. \phi$ , generating CNF output. This problem (i.e, given a formula  $\exists \mathbf{x}. \phi$ , return a logically equivalent quantifier-free CNF formula) has received attention recently. Brauer, King, and Kriener [6] designed an algorithm that combines model enumeration with prime implicant generation. Goldberg and Manolios [11] developed a method based on *dependency sequents*; experimental results show that it works very well on forward and backward reachability on the Hardware Model Checking Competition benchmarks. For QBFs with arbitrary quantifier prefixes, the only other work of which we are aware is that of Becker, Ehlers, Lewis, and Marin [3], which uses computational learning to generate CNF, DNF, or CDNF formulas, and that of Benedetti and Mangassarian [5], which adapts sKizzo [4] for open QBF. The use of SAT solvers to build unordered BDDs [21] and OBDDs [13] has also been investigated.

## 7 Conclusion

This paper has shown how a DPLL-based closed-QBF solver can be extended to handle free variables. The main novelty of this work consists of generalizing clauses/cubes (and the methods involving them), yielding sequents that can include a formula in terms of the free variables. Our extended solver GhostQ produces unordered BDDs, which have several favorable properties [8]. However, in practice, the formulas tended to fairly large in comparison to equivalent CNF representations. Unordered BDDs can often be larger than equivalent OBDDs, since logically equivalent subformulas can have multiple distinct representations in an unordered BDD, unlike in an OBDD. Although our BDDs are necessarily unordered due to unit propagation, in future work it may be desirable to investigate techniques aimed at reducing the size of the output formula.

## References

1. Ansótegui, C., Gomes, C.P., Selman, B.: The Achilles' Heel of QBF. In: AAAI 2005 (2005)
2. Balabanov, V., Jiang, J.-H.R.: Unified QBF certification and its applications. *Formal Methods in System Design* 41(1), 45–65 (2012)
3. Becker, B., Ehlers, R., Lewis, M., Marin, P.: ALLQBF Solving by Computational Learning. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 370–384. Springer, Heidelberg (2012)
4. Benedetti, M.: sKizzo: A Suite to Evaluate and Certify QBFs. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 369–376. Springer, Heidelberg (2005)
5. Benedetti, M., Mangassarian, H.: QBF-Based Formal Verification: Experience and Perspectives. In: JSAT (2008)
6. Brauer, J., King, A., Kriener, J.: Existential Quantification as Incremental SAT. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 191–207. Springer, Heidelberg (2011)
7. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 100(8), 677–691 (1986)
8. Darwiche, A., Marquis, P.: A Knowledge Compilation Map. *J. Artif. Intell. Res (JAIR)* 17, 229–264 (2002)
9. 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)
10. Gent, I.P., Giunchiglia, E., Narizzano, M., Rowley, A.G.D., Tacchella, A.: Watched Data Structures for QBF Solvers. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 25–36. Springer, Heidelberg (2004)
11. Goldberg, E., Manolios, P.: Quantifier elimination by Dependency Sequents. In: FMCAD, pp. 34–43. IEEE (2012)
12. Goultiaeva, A., Bacchus, F.: Exploiting QBF Duality on a Circuit Representation. In: AAAI (2010)
13. Huang, J., Darwiche, A.: Using DPLL for Efficient OBDD Construction. In: H. Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, pp. 157–172. Springer, Heidelberg (2005)
14. Klieber, W., Sapa, 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)
15. McMillan, K.L.: Applying SAT Methods in Unbounded Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 250–264. Springer, Heidelberg (2002)
16. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: DAC 2001 (2001)
17. Shannon, C.E.: The Synthesis of Two Terminal Switching Circuits. *Bell System Technical Journal* 28, 59–98 (1949)
18. Silva, J.P.M., Sakallah, K.A.: GRASP - a new search algorithm for satisfiability. In: ICCAD, pp. 220–227 (1996)
19. Stallman, R.M., Sussman, G.J.: Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artif. Intell.* 9(2), 135–196 (1977)
20. Tseitin, G.S.: On the complexity of derivation in propositional calculus. *Studies in Constructive Mathematics and Mathematical Logic* 2(115-125), 10–13 (1968)



21. Wille, R., Fey, G., Drechsler, R.: Building free binary decision diagrams using SAT solvers. *Facta Universitatis-series: Electronics and Energetics* (2007)
22. Zhang, L.: Solving QBF by Combining Conjunctive and Disjunctive Normal Forms. In: *AAAI 2006* (2006)
23. Zhang, L., Malik, S.: Conflict Driven Learning in a Quantified Boolean Satisfiability Solver. In: *ICCAD 2002* (2002)
24. 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)

# Globalizing Constraint Models<sup>\*</sup>

Kevin Leo<sup>1</sup>, Christopher Mears<sup>1</sup>, Guido Tack<sup>1,2</sup>, and Maria Garcia de la Banda<sup>1,2</sup>

<sup>1</sup> Faculty of IT, Monash University, Australia

<sup>2</sup> National ICT Australia (NICTA), Victoria Laboratory

{kevin.leo,chris.mears,guido.tack,maria.garciadelabanda}@monash.edu

**Abstract.** We present a method that, given a constraint model, suggests global constraints to replace parts of it. This helps non-expert users to write higher-level models that are easier to reason about and may result in better solving performance. Our method exploits the *structure* of the model by considering combinations of the constraints, collections of variables, parameters and loops already present in the model, as well as parameter data from several data files. We assign a score to a candidate global constraint by comparing a sample of its solution space with that of the part of the model it is intended to replace. The top-scoring global constraints are presented to the user through an interactive display, which shows how they could be incorporated into the model. The *MiniZinc Globalizer*, our implementation of the method for the MiniZinc modelling language, is available on the web.

## 1 Introduction

Constraint problems can usually be modelled in many different ways, and the choice of model can have a significant impact on the effectiveness of the resulting constraint program. Developing good models is often a very challenging iterative process that requires considerable levels of expertise and consumes significant amounts of resources. This paper introduces a method that supports users through this iterative process: given a constraint problem model and a few input data files, the method suggests global constraints as possible replacements for certain sets of constraints in the model.

Replacing simpler constraints by global constraints — “*globalizing*” the model — has three significant advantages. First, many solvers implement specialised algorithms for global constraints. Therefore, having the global constraint in the model can improve the efficiency of the solving process considerably. Second, more information is made available regarding the underlying structure of the model. The additional information can help, for example, to detect symmetries, which can then be broken either by adding symmetry breaking constraints or by modifying the search. As another example, even if the chosen solver does not yet support the inferred global constraint, its presence in the model can be used to select better decompositions than the ones originally used by the modeller. And third, the higher-level model obtained by the globalization may improve the modeller’s understanding of the problem and even make it more readable.

---

<sup>\*</sup> NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council. This research was partly sponsored by the Australian Research Council grant DP110102258.