

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

University of Dortmund, Germany

Madhu Sudan

Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Peter J. Stuckey (Ed.)

Principles and Practice of Constraint Programming

14th International Conference, CP 2008
Sydney, Australia, September 14-18, 2008
Proceedings

Volume Editor

Peter J. Stuckey
National ICT Australia
Department of Computer Science and Software Engineering
University of Melbourne
Melbourne, Australia
E-mail: pjs@csse.unimelb.edu.au

Library of Congress Control Number: Applied for

CR Subject Classification (1998): D.1.6, D.3.2-3, F.3, I.2.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-540-85957-8 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-85957-4 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media
springer.com

© Springer-Verlag Berlin Heidelberg 2008
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12511280 06/3180 5 4 3 2 1 0

Preface

This volume contains the proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP 2008) held in Sydney, Australia, September 14–18, 2008. The conference was held in conjunction with the International Conference on Automated Planning and Scheduling (ICAPS 2008) and the International Conference on Knowledge Representation and Reasoning (KR 2008). Information about the conference can be found at the website <http://www.unimelb.edu.au/cp2008/>. Held annually, the CP conference series is the premier international conference on constraint programming. The conference focuses on all aspects of computing with constraints. The CP conference series is organized by the Association for Constraint Programming (ACP). Information about the conferences in the series can be found on the Web at <http://www.cs.ualberta.ca/~ai/cp/>. Information about ACP can be found at <http://www.a4cp.org/>.

CP 2008 included two calls for contributions: a call for research papers, describing novel contributions in the field, and a call for application papers, describing applications of constraint technology. For the first time authors could directly submit short papers for consideration by the committee. The research track received 84 long submissions and 21 short submissions and the application track received 15 long submissions. Each paper received at least three reviews, which the authors had the opportunity to see and to react to, before the papers and their reviews were discussed extensively by the members of the Program Committee. Application papers were reviewed by a separate Applications Track Committee. The Research Track Committee and the Application Track Committee then selected 27 research papers and 6 application papers to be published as long papers in the proceedings, and an additional 23 research papers to be published as short papers, including 8 of those submitted as short papers. The full papers were presented at the conference in two parallel tracks and the short papers were presented in a poster session.

A subcommittee—consisting of myself, Laurent Michel and Michael Trick—selected the paper “A New Framework for Sharp and Efficient Resolution of NCSP with Manifolds of Solutions” by Alexandre Goldsztejn and Laurent Granvilliers to receive the ACP best research paper award. The subcommittee also selected the paper “A Branch and Bound Algorithm for Numerical MAX-CSP” by Jean-Marie Normand, Alexandre Goldsztejn, Marc Christia, and Frédéric Benhamou to receive the ACP best student paper award. Finally the Applications Track Program Committee selected the paper “Planning and Scheduling the Operation of a Very Large Oil Pipeline Network” by Arnaldo Moura, Cid de Souza, André Cirè, and Tony Lopes to receive the ACP best application paper award.

The Program Committee invited Alain Colmeraur, one of the fathers of the field, to give a guest lecture. A paper describing the lecture is included in the

proceedings. The Program Committee in collaboration with the ICAPS 2008 Program Committee also invited John Hooker to give a guest lecture to both conferences simultaneously. The Program Committee in collaboration with the KR 2008 Program Committee also invited Adnan Darwiche to give a guest lecture to both conferences simultaneously. An additional invited talk was given by the recipient of the fourth ACP Award for Research Excellence in Constraint Programming. A presentation of the inaugural ACP Doctoral Research Award for the best thesis in constraint programming awarded in 2006–2007 was also made at the conference, and the winner gave a talk on their dissertation. The Tutorial Chair selected two tutorials to be part of the program: “Current Issues in Max-SAT” by Javier Larrosa, and “ILOG CP Optimizer Software Tutorial: An Automatic Search and Modeling Framework for Detailed Scheduling” by Didier Vidal.

The CP Doctoral program, established in 2003, continued at CP 2008. In the doctoral program, PhD students are invited to present their work and discuss it with senior researchers via a mentoring scheme, as well as hear tutorials on career issues. This year, the doctoral program received 27 submissions and selected 20 of them for financial support. The first two days of the conference were set aside for satellite workshops, tutorials, and the doctoral program. This year there were eight workshops tackling active areas of research in constraint programming, one of them was joint with ICAPS. The complete list of workshops is provided below. Each workshop printed its own proceedings.

In conclusion, I would like to thank all the people who helped made this conference a great success. Thank you to Toby Walsh the Conference Chair, who had the huge task of organizing, budgeting, and planning the whole event. Thank you to Maurice Pagnucco the Local Organizer for the unenviable task of coordinating the organization of the three collocated conferences CP 2008, ICAPS 2008, and KR 2008. Thank you to the Program and Conference Chairs of ICAPS 2008 and KR 2008: Jussi Rintanen, Bernhard Nebel, Chris Beck, Eric Hansen, Patrick Doherty, Gerhard Brewka, and Jérôme Lang; for making the sometimes difficult task of coordinating decisions straightforward and collaborative. Thank you to Kostas Stergiou and Roland H.C. Yap, the Doctoral Program Chairs, for organizing a wonderful program for the doctoral students. Thank you to Jimmy H.M. Lee, the Workshop and Tutorial Chair, for his dedication in putting together an excellent workshop and tutorial program. Thank you to Barry O’Sullivan, the Sponsorship Chair, for tracking down sponsors in the current difficult economic environment. Many thanks to Sebastian Brand, the Publicity Chair, for advertising widely and rapidly updating material on the conference website. Thank you to Laurent Michel and Michael Trick for all their hard work in selecting the best research paper and best student paper. Finally, the most important people to thank, as Program Chair, are the members of the Research Track and Applications Track Program Committees without whose tireless work in organizing, reviewing, and discussing the submissions to the conference, the technical program could not exist. The quality of the technical program is the result of their

penetrating reviews and intense discussions. I would like to thank them for all their (unpaid) hard work!

September 2008

Peter J. Stuckey

Organization

Conference Organization

Conference Chair	Toby Walsh, National ICT Australia, Australia
Program Chair	Peter J. Stuckey, National ICT Australia and The University of Melbourne, Australia
Workshop/Tutorial Chair	Jimmy H.M. Lee, Chinese University of Hong Kong, China
Doctoral Program Chairs	Kostas Stergiou, University of the Aegean, Greece Roland H.C. Yap, National University of Singapore, Singapore
Sponsorship Chair	Barry O'Sullivan, 4C, University College Cork, Ireland
Publicity Chair	Sebastian Brand, National ICT Australia, Australia

Research Track Committee

Slim Abdennadher, Egypt	Laurent Michel, USA
Pedro Barahona, Portugal	Robert Nieuwenhuis, Spain
Nicolas Beldiceanu, France	Gilles Pesant, Canada
Frédéric Benhamou, France	Steve Prestwich, Ireland
Christian Bessiere, France	Francesca Rossi, Italy
Lucas Bordeaux, UK	Michel Rueher, France
David Cohen, UK	Thomas Schiex, France
Rina Dechter, USA	Christian Schulte, Sweden
Yves Deville, Belgium	Paul Shaw, France
Alan Frisch, UK	Barbara Smith, UK
Warwick Harvey, UK	Kostas Stergiou, Greece
Hiroshi Hosobe, Japan	Michael Trick, USA
Zeynep Kiziltan, Italy	Peter van Beek, Canada
Arnaud Lallouet, France	Pascal Van Hentenryck, USA
Javier Larrosa, Spain	Roland H.C. Yap, Singapore
Jimmy H.M. Lee, China	Makoto Yokoo, Japan
Michael Maher, Australia	

Applications Track Committee

Andy Chun, China	Barry O'Sullivan, Ireland
Vitaly Lagoon, Australia	Helmut Simonis, Ireland
Michela Milano, Italy	Mark Wallace, Australia

Additional Referees

Magnus Ågren	Katsutoshi Hirayama	Claude-Guy Quimper
A. Anbulagan	Frank Hutter	Jean-Charles Régim
Francisco Azevedo	Christopher Jefferson	Emma Rollon
Thanasis Balafoutis	George Katsirelos	Michel Rueher
Marco Benedetti	Tom Kelsey	Ashish Sabharwal
Simon Boivin	Phil Kilby	Andras Salamon
Magnus Bordewich	Pushmeet Kohli	Horst Samulowitz
Hadrien Cambazard	Frederic Koriche	Marti Sanchez
Mats Carlsson	Mikael Z. Lagerkvist	Frédéric Saubion
Martine Ceberio	Yat Chiu Law	Pierre Schaus
Kenil C.K. Cheng	Yahia Lebbah	Joachim Schimpf
Michael Codish	Christophe Lecoutre	Tom Schrijvers
Remi Coletta	Michel Lemaître	Andrew See
Hélène Collavizza	C. Likitvivatanavong	Charles Siu
Marco Correia	Steve Linton	Harald Søndergaard
Jorge Cruz	Lengning Liu	Guido Tack
Thi Bich Hanh Dao	Matthieu Lopez	Gilles Trombettoni
Romuald Debruyne	Ines Lynce	Charlotte Truchet
Simon de Givry	Radu Marinescu	Willem-Jan van Hoeve
Khalil Djelloul	Toshihiro Matsui	Peter Van Weert
Gregory J. Duck	Amnon Meisels	Jeremie Vautard
Redouane Ezzahir	Pedro Meseguer	K. Brent Venable
Boi Faltings	Claude Michel	Andrew Verden
Hélène Fargier	Ian Miguel	Gérard Verfaillie
Germain Faure	Jean-Noël Monette	Ruben Viegas
Thibaut Feydy	Eric Monfroy	Richard Wallace
Jonathan Gaudreault	Neil Moore	Toby Walsh
Marco Gavanelli	Nina Narodytska	May Woo
Samir Genaim	Bertrand Neveu	Neil Yorke-Smith
Ian Gent	Albert Oliveras	Stéphane Zampelli
Vibhav Gogate	Lars Otten	Alessandro Zanarini
Alexandre Goldsztejn	Thierry Petit	Bruno Zanuttini
Frederic Goualard	Karen Petrie	Yuanlin Zhang
Gopal Gupta	Cedric Piette	Neng Fa Zhou
Hani Hagrais	Maria Silvia Pini	Standa Zivny
Steven Halim	Patrick Prosser	
Patricia M. Hill	Jakob Puchinger	

Workshops

Constraint Modelling and Reformulation

CSP Solver Competition

Constraint Satisfaction Techniques for Planning and Scheduling Problems

Counting Problems in CSP and SAT, and Other Neighboring Problems
Local Search Techniques in Constraint Satisfaction
Quantification in Constraint Programming
Preferences and Soft Constraints
Symmetry and Constraint Satisfaction Problems

Administrative Council of the ACP

President	Barry O'Sullivan, Ireland
Secretary	Jimmy H.M. Lee, China
Treasurer	Thomas Schiex, France
Conference Coordinator	Pedro Meseguer, Spain
Executive Committee	Christian Bessiere, Francesca Rossi, Christian Schulte, Michael Trick

Sponsoring Institutions

Association of Constraint Programming
Cork Constraint Computation Centre
ILOG
Intelligent Information Systems Institute, Cornell University
National ICT Australia
University of New South Wales

Table of Contents

Invited Lecture

Back to the Complexity of Universal Programs	1
<i>Alain Colmerauer</i>	

Applications Track Long Papers

A Constraint Programming Approach for Allocation and Scheduling on the CELL Broadband Engine	21
<i>Luca Benini, Michele Lombardi, Michela Milano, and Martino Ruggiero</i>	
Planning and Scheduling the Operation of a Very Large Oil Pipeline Network	36
<i>Arnaldo V. Moura, Cid C. de Souza, Andre A. Cire, and Tony M.T. Lopes</i>	
Search Strategies for Rectangle Packing	52
<i>Helmut Simonis and Barry O’Sullivan</i>	
Solving a Telecommunications Feature Subscription Configuration Problem	67
<i>David Lesaint, Deepak Mehta, Barry O’Sullivan, Luis Quesada, and Nic Wilson</i>	
Protein Structure Prediction with Large Neighborhood Constraint Programming Search	82
<i>Ivan Dotu, Manuel Cebrián, Pascal Van Hentenryck, and Peter Clote</i>	
An Application of Constraint Programming to Superblock Instruction Scheduling	97
<i>Abid M. Malik, Michael Chase, Tyrel Russell, and Peter van Beek</i>	

Research Track Long Papers

Classes of Submodular Constraints Expressible by Graph Cuts	112
<i>Stanislav Živný and Peter G. Jeavons</i>	
Optimization of Simple Tabular Reduction for Table Constraints	128
<i>Christophe Lecoutre</i>	
Universal Booleanization of Constraint Models	144
<i>Jinbo Huang</i>	

Flow-Based Propagators for the SEQUENCE and Related Global Constraints	159
<i>Michael Maher, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh</i>	
Guiding Search in QCSP ⁺ with Back-Propagation	175
<i>Guillaume Verger and Christian Bessiere</i>	
A New Framework for Sharp and Efficient Resolution of NCSP with Manifolds of Solutions	190
<i>Alexandre Goldsztejn and Laurent Granvilliers</i>	
A Branch and Bound Algorithm for Numerical MAX-CSP	205
<i>Jean-Marie Normand, Alexandre Goldsztejn, Marc Christie, and Frédéric Benhamou</i>	
A Geometric Constraint over k -Dimensional Objects and Shapes Subject to Business Rules	220
<i>Mats Carlsson, Nicolas Beldiceanu, and Julien Martin</i>	
Cost-Based Domain Filtering for Stochastic Constraint Programming ...	235
<i>Roberto Rossi, S. Armagan Tarim, Brahim Hnich, and Steven Prestwich</i>	
Dichotomic Search Protocols for Constrained Optimization	251
<i>Meinolf Sellmann and Serdar Kadioglu</i>	
Length-Lex Bounds Consistency for Knapsack Constraints	266
<i>Yuri Malitsky, Meinolf Sellmann, and Willem-Jan van Hoeve</i>	
A Framework for Hybrid Tractability Results in Boolean Weighted Constraint Satisfaction Problems	282
<i>T.K. Satish Kumar</i>	
From High Girth Graphs to Hard Instances	298
<i>Carlos Ansótegui, Ramón Béjar, César Fernández, and Carles Mateu</i>	
Switching among Non-Weighting, Clause Weighting, and Variable Weighting in Local Search for SAT	313
<i>Wanxia Wei, Chu Min Li, and Harry Zhang</i>	
CPBPV: A Constraint-Programming Framework for Bounded Program Verification	327
<i>Hélène Collavizza, Michel Rueher, and Pascal Van Hentenryck</i>	
Exploiting Common Subexpressions in Numerical CSPs	342
<i>Ignacio Araya, Bertrand Neveu, and Gilles Trombettoni</i>	
A Soft Constraint of Equality: Complexity and Approximability	358
<i>Emmanuel Hebrard, Barry O'Sullivan, and Igor Razgon</i>	

Structural Tractability of Propagated Constraints	372
<i>Martin J. Green and Christopher Jefferson</i>	
Connecting ABT with Arc Consistency	387
<i>Ismel Brito and Pedro Meseguer</i>	
Elicitation Strategies for Fuzzy Constraint Problems with Missing Preferences: Algorithms and Experimental Studies	402
<i>Mirco Gelain, Maria Silvia Pini, Francesca Rossi, K. Brent Venable, and Toby Walsh</i>	
Reformulating Positive Table Constraints Using Functional Dependencies	418
<i>Hadrien Cambazard and Barry O’Sullivan</i>	
Relaxations for Compiled Over-Constrained Problems	433
<i>Alexandre Papadopoulos and Barry O’Sullivan</i>	
Approximate Compilation of Constraints into Multivalued Decision Diagrams	448
<i>Tarik Hadzic, John N. Hooker, Barry O’Sullivan, and Peter Tiedemann</i>	
Quantified Constraint Optimization	463
<i>Marco Benedetti, Arnaud Lallouet, and Jérémie Vautard</i>	
Exploiting Decomposition in Constraint Optimization Problems	478
<i>Matthew Kitching and Fahiem Bacchus</i>	
A Coinduction Rule for Entailment of Recursively Defined Properties . . .	493
<i>Joxan Jaffar, Andrew E. Santosa, and Răzvan Voicu</i>	
Maintaining Generalized Arc Consistency on Ad Hoc r -Ary Constraints	509
<i>Kenil C.K. Cheng and Roland H.C. Yap</i>	

Research Track Short Papers

Perfect Constraints Are Tractable	524
<i>András Z. Salamon and Peter G. Jeavons</i>	
Efficiently Solving Problems Where the Solutions Form a Group	529
<i>Karen E. Petrie and Christopher Jefferson</i>	
Approximate Solution Sampling (and Counting) on AND/OR Spaces . . .	534
<i>Vibhav Gogate and Rina Dechter</i>	
Model Restarts for Structural Symmetry Breaking	539
<i>Daniel Heller, Aurojit Panda, Meinolf Sellmann, and Justin Yip</i>	

An Elimination Algorithm for Functional Constraints	545
<i>Yuanlin Zhang, Roland H.C. Yap, Chendong Li, and Satyanarayana Mariseti</i>	
Crossword Puzzles as a Constraint Problem	550
<i>Anbulagan and Adi Botea</i>	
Recent Hybrid Techniques for the Multi-Knapsack Problem	555
<i>Carlos Diego Rodrigues, Philippe Michelon, and Manoel B. Campêlo</i>	
Edge Matching Puzzles as Hard SAT/CSP Benchmarks	560
<i>Carlos Ansótegui, Ramón Béjar, César Fernández, and Carles Mateu</i>	
Test Strategy Generation Using Quantified CSPs	566
<i>Martin Sachenbacher and Paul Maier</i>	
Perfect Derived Propagators	571
<i>Christian Schulte and Guido Tack</i>	
Refined Bounds for Instance-Based Search Complexity of Counting and Other #P Problems	576
<i>Lars Otten and Rina Dechter</i>	
Transforming Inconsistent Subformulas in MaxSAT Lower Bound Computation	582
<i>Chu Min Li, Felip Manyà, Nouredine Ould Mohamedou, and Jordi Planes</i>	
Semi-automatic Generation of CHR Solvers for Global Constraints	588
<i>Frank Raiser</i>	
Stochastic Local Search for the Optimal Winner Determination Problem in Combinatorial Auctions	593
<i>Dalila Boughaci, Belaid Benhamou, and Habiba Drias</i>	
Revisiting the Upper Bounding Process in a Safe Branch and Bound Algorithm	598
<i>Alexandre Goldsztejn, Yahia Lebbah, Claude Michel, and Michel Rueher</i>	
Computing All Optimal Solutions in Satisfiability Problems with Preferences	603
<i>Emanuele Di Rosa, Enrico Giunchiglia, and Marco Maratea</i>	
On the Efficiency of Impact Based Heuristics	608
<i>Marco Correia and Pedro Barahona</i>	
Probabilistically Estimating Backbones and Variable Bias: Experimental Overview	613
<i>Eric I. Hsu, Christian J. Muise, J. Christopher Beck, and Sheila A. McIlraith</i>	

A New Empirical Study of Weak Backdoors	618
<i>Peter Gregory, Maria Fox, and Derek Long</i>	
Adding Search to Zinc	624
<i>Reza Rafeh, Kim Marriott, Maria Garcia de la Banda, Nicholas Nethercote, and Mark Wallace</i>	
Experimenting with Small Changes in Conflict-Driven Clause Learning Algorithms	630
<i>Gilles Audemard and Laurent Simon</i>	
Search Space Reduction for Constraint Optimization Problems	635
<i>Kenil C.K. Cheng and Roland H.C. Yap</i>	
Engineering Stochastic Local Search for the Low Autocorrelation Binary Sequence Problem	640
<i>Steven Halim, Roland H.C. Yap, and Felix Halim</i>	
Author Index	647

Back to the Complexity of Universal Programs

Alain Colmerauer

Marseilles, France

Abstract. I start with three examples illustrating my contribution to constraint programming: the problem of cutting a rectangle into different squares in Prolog III, a complicated constraint for Prolog IV, the optimal narrowing of the sortedness constraint. Then I switch to something quite different: to machines, in particular to Turing machines. After the declarative aspect, the basic computational aspect!

The paper provides a framework enabling to define and determine the complexity of various universal programs U for various machines. The approach consists of first defining the complexity as the average number of instructions to be executed by U , when simulating the execution of one instruction of a program P with input x .

To obtain a complexity that does not depend on P or x , we introduce the concept of an *introspection coefficient* expressing the average number of instructions executed by U , for simulating the execution of one of its own instructions. We show how to obtain this coefficient by computing a square matrix whose elements are numbers of executed instructions when running selected parts of U on selected data. The coefficient then becomes the greatest eigenvalue of the matrix.

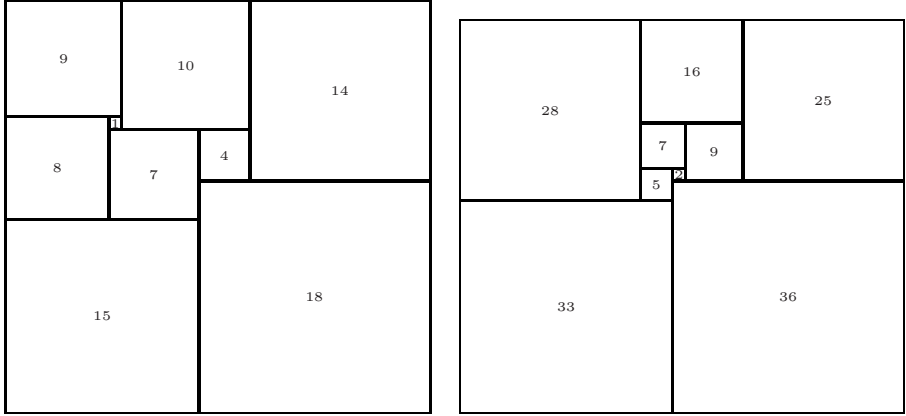
We illustrate the approach using two examples of particularly efficient universal programs: one for a three-symbol Turing Machine (blank symbol not included) with an introspection coefficient of 3 672.98, the other for an indirect addressing arithmetic machine with an introspection coefficient of 26.27.

1 Preface

Let us review my contribution to constraint programming.

1.1 Around 1985

Around 1985 I was interested by constraints, more precisely by numerical linear constraints, by Boolean algebra and by list constraints. That's how Prolog III was born [2]. A good example of a program consists in cutting a rectangle of unknown size into n different squares also of unknown sizes. For $n = 9$ the following result holds:



Here is the program, written in the syntax of Prolog IV. The height of the rectangle to be cut is assumed to be 1, which is not a restriction:

```

rectangle(A,C) :-
    gelin(A,1),
    distinctSizes(C),
    area([-1,A,1],L,C,[]).

distinctSizes([]).
distinctSizes([B|C]) :-
    gtlin(B,0),
    distinctSizes(C),
    out(B,C).

out(B,[]).
out(B,[Bp|C]) :-
    dif(B,Bp),
    out(B,C).

area([V|L],[V|L],C,C) :-
    gelin(V,0).
area([V|L],Lppp,[B|C],Cpp) :-
    lt(V,0),
    square(B,L,Lp),
    area(Lp,Lpp,C,Cp),
    area([V+B,B|Lpp],Lppp,Cp,Cpp).

square(B,[H,0,Hp|L],Lp) :-
    gtlin(B,H),
    square(B,[H+Hp|L],Lp).
square(B,[H,V|L],[-B+V|L]) :-
    B = H.
square(B,[H|L],[-B,H-B|L]) :-
    ltlin(B,H).

```

The predicates *gelin*(x,y), *gtlin*(x,y), *ltlin*(x,y) correspond to the linear constraints $x \geq y$, $x > y$, $x < y$ and *dif*(x,y) to the constraint $x \neq y$. We leave the program uncommented. It is sufficient to ask the query

```
>> size(C)=9, rectangle(A,C).
```

where *size*(x) = y means *size of the list x is y* , to obtain

```

A = 33/32,
C = [15/32,9/16,1/4,7/32,1/8,7/16,1/32,5/16,9/32];
A = 69/61,
C = [33/61,36/61,28/61,5/61,2/61,9/61,25/61,7/61,16/61];
A = 33/32,
C = [9/16,15/32,7/32,1/4,7/16,1/8,5/16,1/32,9/32];
A = 69/61,
C = [36/61,33/61,5/61,28/61,25/61,9/61,2/61,7/61,16/61];

```

$A = 33/32,$
 $C = [9/32, 5/16, 7/16, 1/4, 1/32, 7/32, 1/8, 9/16, 15/32];$
 $A = 69/61,$
 $C = [28/61, 16/61, 25/61, 7/61, 9/61, 5/61, 2/61, 36/61, 33/61];$
 $A = 69/61,$
 $C = [25/61, 16/61, 28/61, 9/61, 7/61, 2/61, 5/61, 36/61, 33/61];$
 $A = 33/32,$
 $C = [7/16, 5/16, 9/32, 1/32, 1/4, 1/8, 7/32, 9/16, 15/32].$

1.2 Around 1990

Prolog IV was finished in 1995 [3]. In addition to the constraints of Prolog III, it includes numerical non-linear constraints which are approximately solved by narrowing of intervals. It also includes the existential quantifier. Here, on the left column, is a constraint in the usual notation and the value of the free variable y . The formula $(x > y)$ denotes the Boolean value *true* ou *false*. On the right column you find the corresponding query and the answer in Prolog IV.

$\exists u \exists v \exists w \exists x \left(\begin{array}{l} y \leq 5 \\ \wedge v_1 = \cos v_4 \\ \wedge \text{size}(u) = 3 \\ \wedge \text{size}(v) = 10 \\ \wedge u \bullet v = v \bullet w \\ \wedge y \geq 2 + (3 \times x) \\ \wedge x = (74 > \lfloor 100 \times v_1 \rfloor) \end{array} \right)$	<pre>>> U ex V ex W ex X ex le(Y,5), V:1 = cos(V:4), size(U) = 3, size(V) = 10, U o V = V o W, ge(Y,2.+(3.*X)), X = bgt(74,floor(100.*V:1)).</pre>
$y = 5$	$Y = 5.$

1.3 Around 2000

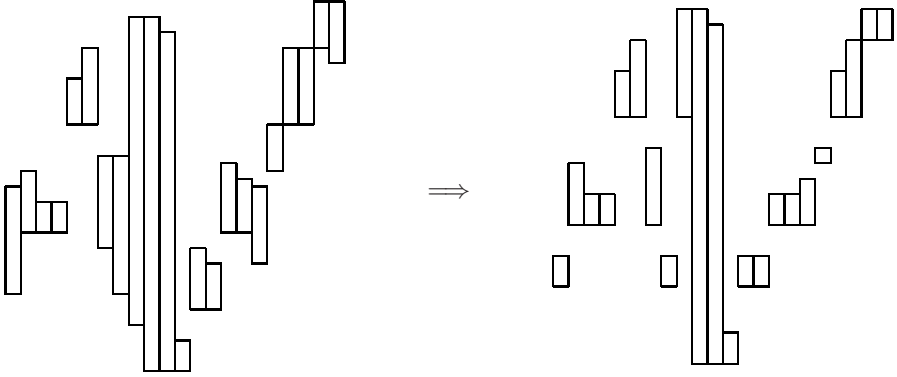
Having been interested by the narrowing of intervals, I focused on a particular instance, the sortedness constraint:

$$\text{sort}(x_1, \dots, x_n, x_{n+1}, \dots, x_{2n}) \equiv \left\{ \begin{array}{l} (x_{n+1}, \dots, x_{2n}) \\ \text{is equal to} \\ (x_1, \dots, x_n) \text{ sorted} \\ \text{in non-decreasing order.} \end{array} \right.$$

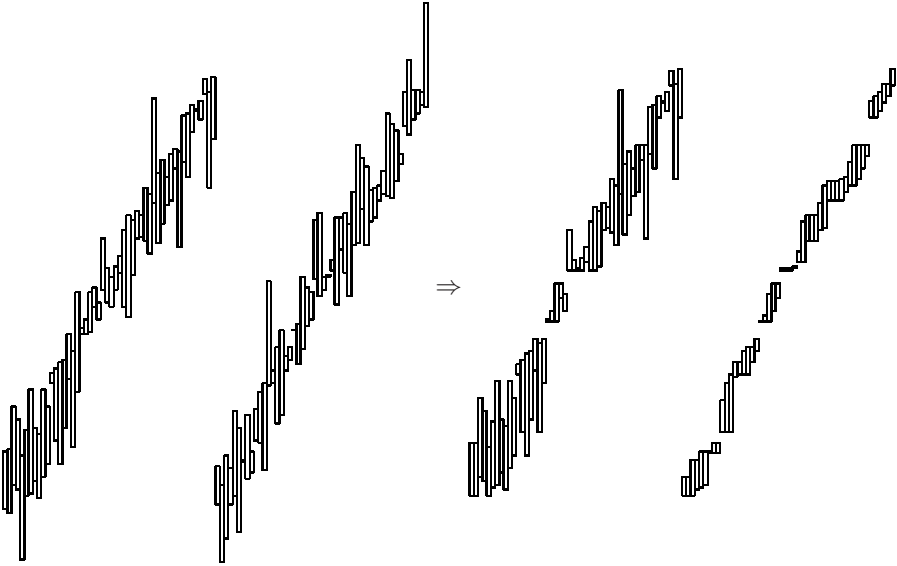
With Noëlle Bleuzen [1], a colleague from the department of Mathematics, we developed an algorithm of complexity $O(n \log n)$ to compute the smallest intervals a'_i from the intervals a_i such that:

$$\begin{aligned} & \text{sort}(x_1, \dots, x_{2n}) \wedge x_1 \in a_1 \wedge \dots \wedge x_{2n} \in a_{2n} \\ & \equiv \\ & \text{sort}(x_1, \dots, x_{2n}) \wedge x_1 \in a'_1 \wedge \dots \wedge x_{2n} \in a'_{2n} \end{aligned}$$

For $2n = 22$ for example, we obtain:



and for $2n = 100$:



2 Introduction

In parallel, around 2000, I was teaching an introductory course designed to initiate undergraduate students to low level programming. My approach was to start teaching them how to program Turing machines. The main exercise in the course consisted of completing and testing a universal program whose architecture I provided. The results were disappointing, the universal program being too slow for executing sizeable programs. Among others it was impossible to run the machine on its own code, in the sense explained in Section 4. In subsequent years, I succeeded in designing considerably more efficient universal programs, even though they became increasingly more complex. These improved programs

were capable to execute their own code in a reasonable time. To simulate the execution of one of its own instructions, the last program executes an average number of 3 672.98 instructions. That is the *introspection coefficient*, a key concept of this paper.

The rest of this paper presents this result in a more general context concerning machines other than Turing machines. Section 2 is this introduction. In Section 3, we formally define the concepts of programmed machine, machine, program, transition and instruction. We illustrate this on a Turing machine, and on an indirect addressing arithmetic machine. In Section 4, we introduce the universal pair, program and coding function. We mention the theorem on how to check the existence of its introspection coefficients and how to compute its value. We omit the proof and refer the reader to [4]. Sections 5 and 6 and also Appendix are devoted to two specially efficient universal programs: the first one, as already mentioned, for a Turing machine, the second one for an indirect addressing arithmetic machine. In Section 7 we conclude about a lack of restriction of our definition of the introspection coefficient.

We are not aware of other work on the design of efficient universal programs. Let us however mention the well known contributions of M. Minsky [5] and Y. Rogozin [7] in the design of universal programs for Turing machines with very small numbers of states. Surprisingly, they seem particularly inefficient in terms of the number of executed instructions.

3 Machines

3.1 Basic Definitions

Definition 1. A machine M is a 5-tuple $(\Sigma, C, \alpha, \omega, I)$, where

- Σ , the alphabet of M , is a finite not empty set;
- C , is a set, generally infinite, of configurations; the ordered pairs (c, c') of elements of C are called transitions;
- α , the input function, maps each element x of Σ^* to a configuration $\alpha(x)$;
- ω , the output function, maps each configuration c to an element $\omega(c)$ of Σ^* ;
- I , is a countable set of instructions, an instruction being a set of compatibles transitions, i.e., whose first components are all distinct.

Definition 2. A program P for a machine M is a finite subset of the instructions set I of M , such that the transitions of $\bigcup P$ are compatible.¹

3.2 How a Machine Operates

Let $M = (\Sigma, C, \alpha, \omega, I)$ be a machine and P a program for M . The operation of the machine (M, P) is explained by the diagram:



¹ P being a set of sets $\bigcup P$ denotes the set of elements which are member of at least one element of P and thus the set of transitions involved in program P .

and more precisely by the definition of the following functions², where x is a word on Σ :

$$\begin{aligned} orbit_M(P, x) &= \begin{cases} \text{the longest sequence } (c_0, c_1) (c_1, c_2) (c_2, c_3) \dots \text{ with} \\ c_0 = \alpha(x) \text{ and each } (c_i, c_{i+1}) \text{ an element of } \bigcup P. \end{cases} \\ out_M(P, x) &= \begin{cases} \nearrow, & \text{if } orbit(P, x) \text{ is infinite,} \\ \omega(c_n), & \text{if } orbit(P, x) \text{ ends with } (c_{n-1}, c_n) \end{cases} \end{aligned}$$

3.3 Example: Turing Machines

Informally these are classical Turing machines with a bi-infinite tape and instructions written $[q_i, abd, q_j]$, with $d = L$ or $d = R$, meaning : if the machine is in state q_i and the symbol read by the read-write head is a , the machine replaces a by b , then moves its head one symbol to the left or the right, depending whether $d = L$ or $d = R$, and change its state to q_j .

In fact we consider a variant of the Turing machines described above with an internal moving head direction whose initial value is equal to left-right. The instructions are written $[q_i, abs, q_j]$, with $s = +$ or $s = -$, meaning : if the machine is in state q_i and the symbol read by the read-write head is a , the machine replaces a by b , keeps its internal direction or changes it depending whether $s = +$ or $s = -$, moves its read-write head one symbol in the new internal direction, and changes its states to q_j .

Initially the entire tape is filled with blanks except for a finite portion which contains the initial input, the read-write head being positioned on the symbol which precedes this input. When there are no more instructions to be executed the machine output the longest word which contains no blank symbols and which starts just after the position of the read-write head.

Formally one first introduces an infinite countable set $\{q_1, q_2, \dots\}$ of *states* and a special symbol $\mathbf{u}$, *the blank*. For any alphabet word x on an alphabet of the form $\Sigma \cup \{\mathbf{u}\}$, one writes $\cdot x$ for x , with all its beginning blanks erased, and $x \cdot$ for x , with all its ending blanks erased.

Definition 3. A Turing machine is a 5-tuple of the form $(\Sigma, C, \alpha, \omega, I)$ where,

- Σ is a finite set not having $\mathbf{u}$ as an element,
- C is the set of 5-tuples of the form $[d, q_i, \cdot x, a, y \cdot]$, with $d \in \{L, R\}$, q_i being a state, x, y taken from $\Sigma_{\mathbf{u}}^*$ and a taken from $\Sigma_{\mathbf{u}}$, where $\Sigma_{\mathbf{u}} = \Sigma \cup \{\mathbf{u}\}$,
- $\alpha(x) = [R, q_1, \varepsilon, \mathbf{u}, x]$, for all $x \in \Sigma^*$,
- $\omega([d, q_i, \cdot x, a, y \cdot])$ is the longest element of Σ^* beginning y ,
- I is the set of instruction denoted and defined, for all states q_i, q_j , all elements a, b of $\Sigma_{\mathbf{u}}$ and all $s \in \{+, -\}$, by

$$\begin{aligned} [q_i, abs, q_j] &\stackrel{\text{def}}{=} \\ &\{([d, q_i, \cdot xc, a, y \cdot], [L, q_j, \cdot x, c, by \cdot]) \mid (d, s) \in E_1 \text{ and } (x, c, y) \in F\} \cup \\ &\{([d, q_i, \cdot x, a, cy \cdot], [R, q_j, \cdot xb, c, y \cdot]) \mid (d, s) \in E_2 \text{ and } (x, c, y) \in F\}, \\ &\text{with } E_1 = \{(L, +), (R, -)\}, E_2 = \{(R, +), (L, -)\} \text{ and } F = \Sigma_{\mathbf{u}}^* \times \Sigma_{\mathbf{u}} \times \Sigma_{\mathbf{u}}^*. \end{aligned}$$

² Index M is omitted when there is no ambiguity.

3.4 Example: Indirect Addressing Arithmetic Machine

This is a machine with an infinity of registers $r_0, r_1, r_2, \dots$. Each register contains an unbounded natural integer. Each instruction starts with a number and the machine always executes the instruction whose number is contained in r_0 and, except in one case, increases r_0 by 1. There are five types of instructions: assigning a constant to a register, addition and subtraction of a register to/from another, two types of indirect assignment of a register to another and zero-testing of a register content.

More precisely and in accordance with our definition of a machine:

Definition 4. An indirect addressing arithmetic machine is a 5-tuple of the form $(\Sigma, C, \alpha, \omega, I)$, where,

- $\Sigma = \{c_1, \dots, c_m\}$, where the c_i are any symbols,
- C is the set of infinite sequences $r = (r_0, r_1, r_2, \dots)$ of natural integers,
- $\alpha(a_1 \dots a_n) = (0, 25, 1, \dots, 1, r_{24+1}, \dots, r_{24+n}, 0, 0, \dots)$, with r_{24+i} equal to $1, \dots, m$ depending whether a_i equals $c_1, \dots, c_m$,
- $\omega(r_0, r_1, \dots) = a_1 \dots a_n$, with a_i equal to $c_1, \dots, c_m$ depending whether r_{r_1+i} equals $1, \dots, m$, and n being the greatest integer such that $r_{r_1}, \dots, r_{r_1+n}$ are elements of $\{1, \dots, m\}$,
- I is the set of instructions denoted and defined, for all natural integers i, j, k , by:

$$\begin{aligned}
 [i, cst, j, k] &\stackrel{def}{=} \{(r, s) \in C^2 \mid r_0 = i, s := r, s_j := k, s_0 := s_0 + 1\}, \\
 [i, plus, j, k] &\stackrel{def}{=} \{(r, s) \in C^2 \mid r_0 = i, s := r, s_j := s_j + s_k, s_0 := s_0 + 1\}, \\
 [i, sub, j, k] &\stackrel{def}{=} \{(r, t) \in C^2 \mid r_0 = i, s := r, s_j := s_j \div s_k, s_0 := s_0 + 1\}, \\
 [i, from, j, k] &\stackrel{def}{=} \{(r, t) \in C^2 \mid r_0 = i, s := r, s_j := s_{s_k}, s_0 := s_0 + 1\}, \\
 [i, to, j, k] &\stackrel{def}{=} \{(r, t) \in C^2 \mid r_0 = i, s := r, s_{r_j} = r_k, s_0 := s_0 + 1\}, \\
 [i, ifze, j, k] &\stackrel{def}{=} \{(r, t) \in C^2 \mid r_0 = i, s := r, s_0 := \begin{bmatrix} s_k + 1, \text{ if } s_j = 0 \\ s_0 + 1, \text{ if } s_j \neq 0 \end{bmatrix}\}.
 \end{aligned}$$

Here $s_j \div s_k$ stands for $\max\{0, s_j - s_k\}$.

4 Universal Program and Universal Coding

4.1 Universal Pair

Let $M = (\Sigma, C, \alpha, \omega, I)$ be a machine and let us code each program P for M by a word $code(P)$ on Σ .

Definition 5. The pair $(U, code)$, the program U and the coding function $code$, are said to be universal for M , if, for all programs P of M and for all $x \in \Sigma^*$,

$$out(U, code(P) \cdot x) = out(P, x). \quad (1)$$

If in the above formula we replace P by U , and x by $\text{code}(U)^n \cdot x$ we obtain:

$$\text{out}(U, \text{code}(U)^{n+1} \cdot x) = \text{out}(U, \text{code}(U)^n \cdot x)$$

and thus:

Property 1. *If (U, code) is a universal pair, then for all $n \geq 0$ and $x \in \Sigma^*$,*

$$\text{out}(U, \text{code}(U)^n \cdot x) = \text{out}(U, x). \quad (2)$$

4.2 Introspection Coefficient

Let (U, code) be a universal pair for the machine $M = (\Sigma, C, \alpha, \omega, I)$. The *complexity* of this pair is the average number of transitions performed by U for producing the same effect as a transition of the program P occurring in the input of U . More precisely:

Definition 6. *Given a program P for M and a word x on Σ with $\text{orbit}(P, x) \neq \nearrow$, the complexity of (U, code) is the real number defined by*

$$\frac{|\text{orbit}(U, \text{code}(P) \cdot x)|}{|\text{orbit}(P, x)|}.$$

The disadvantage of this definition is that the complexity depends on the input of U . For an intrinsic complexity, independently of the input of U , we introduce the *introspection coefficient* of (U, code) whose definition is justified by Property 2:

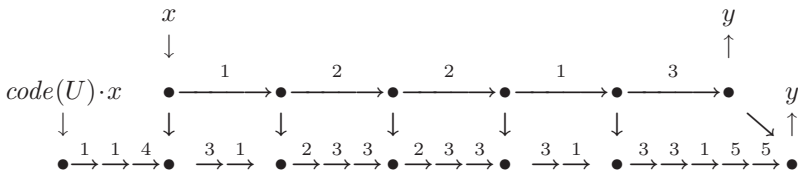
Definition 7. *If for all $x \in \Sigma^*$, with $\text{orbit}(U, x) \neq \nearrow$, the real number*

$$\lim_{n \rightarrow \infty} \frac{|\text{orbit}(U, \text{code}(U)^{n+1} \cdot x)|}{|\text{orbit}(U, \text{code}(U)^n \cdot x)|}$$

exists and does not depend on x , then this real number is the introspection coefficient of the universal pair (U, code) .

4.3 Existence and Value of the Introspection Coefficient

Let (U, code) be a universal pair for a machine $M = (\Sigma, C, \alpha, \omega, I)$. Given a word x on Σ , we assume that the computation of the word y by $y = \text{out}(U, x)$ can be synchronized with the computation of the same word y by $y = \text{out}(U, \text{code}(U) \cdot x)$, according to the following diagram:



More precisely we make the hypothesis:

Hypothesis 1. *There exists $n, nb, \mathcal{A}, \mathcal{B}$ such that, for every pair of traces of the form*

$$(\text{orbit}(U, \text{code}(U) \cdot x, \text{orbit}(U, x)))$$

itself of the form

$$(s_1 \cdots s_l, r_1 \cdots r_k),$$

we have

$$nb(s_1) \cdots nb(s_l) = \mathcal{B} \cdot \mathcal{A}(nb(r_1)) \cdots \mathcal{A}(nb(r_k)),$$

with n positive integer, with $nb(t) \in 1..n$ for each transition t of U , with $\mathcal{A}(i)$ a finite sequence on $1..n$ for each $i \in 1..n$, with $\mathcal{B}$ a finite sequence on $1..n$.

We then introduce the column vector B and the square matrix A :

$$B = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}, \quad b_i = \text{number of occurrences of integer } i \text{ in } \mathcal{B},$$

$$A = \begin{bmatrix} a_{11} & \cdots & a_{nn} \\ \vdots & & \vdots \\ a_{1n} & \cdots & a_{nn} \end{bmatrix}, \quad a_{ij} = \text{number of occurrences of integer } i \text{ in } \mathcal{A}(j).$$
(3)

and we conclude by the theorem, where $\|X\|$ denotes the sum of the components of X :

Theorem 1. *Suppose the matrix A admits a real eigenvalue λ , whose multiplicity is equal to 1, which is strictly greater to 1 and to the greatest modulus λ' of the other eigenvalue.*

If α is a real number with $\lambda' < \alpha < \lambda$, if $X_0 = B$ and $X_{n+1} = \frac{1}{\alpha}AX_n$, then, when $n \rightarrow \infty$, exactly one of the two properties holds:

1. $\|X_n\| \rightarrow 0$,
2. $\|X_n\| \rightarrow \infty$. In this case λ is the inspection coefficient.

Anyone interested in more details may consult [4].

5 Universal Pair for the Turing Machine

5.1 The Universal Pair

We now present a particularly efficient universal pair (U, code) for the Turing machine M with alphabet $\Sigma = \{\text{o}, \text{i}, \text{z}\}$. The program U has 184 instructions and 54 states and $|\text{code}(U)| = 1552$. Its listing and its graph can be seen in the annexes A and B.

5.2 Coding Function of the Universal Pair

Let P be a program for M . We take $\text{code}(P)$ as the word on $\{\text{o}, \text{i}, \text{z}\}$

$$\text{code}(P) = \text{z}I_{4n}\text{z} \dots \text{z}I_{k+1}\text{z}I_k\text{z}I_{k-1}\text{z} \dots \text{z}I_1\text{zoi} \dots \text{izz}.$$

Integer n is the number of states of P and the I_k are the coded instructions. The size of the *shuttle* $\mathbf{oi} \dots \mathbf{iz}$ is equal to the longest size of the I_k minus 5.

In order to assign a position to each coded instruction I_k of $[q_i, abs, q_j]$, we introduce the number:

$$\pi(i, a) = 4(i - 1) + \begin{cases} 1, & \text{if } a = \mathbf{u} \\ 2, & \text{if } a = \mathbf{o} \\ 3, & \text{if } a = \mathbf{i} \\ 4, & \text{if } a = \mathbf{z} \end{cases}$$

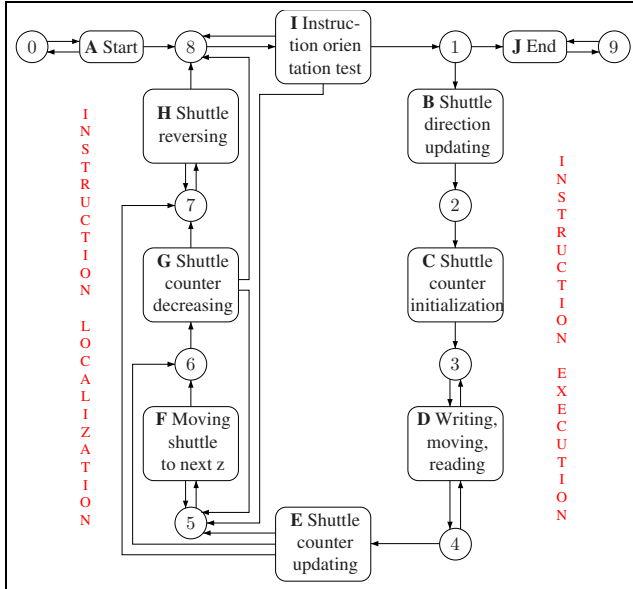
For all $a \in \Sigma_{\mathbf{u}}$ and $i \in 1..n$,

$$I_{\pi(i, a)} = \begin{cases} \overline{[q_i, abs, q_j]}, & \text{if there exists } b, s, j \text{ with } [q_i, abs, q_j] \in P, \\ \mathbf{oi}, & \text{otherwise,} \end{cases}$$

- with $\overline{[q_i, a, b, s, q_j]} = \begin{cases} \mathbf{ia}_m \dots \mathbf{a}_2 \mathbf{o}, & \text{if } \pi(i, a) < \frac{1}{2}(\pi(j, \mathbf{u}) + \pi(j, \mathbf{z})), \\ \mathbf{oa}_2 \dots \mathbf{a}_m \mathbf{i}, & \text{if } \pi(i, a) > \frac{1}{2}(\pi(j, \mathbf{u}) + \pi(j, \mathbf{z})), \end{cases}$
- with $a_2 a_3$ equal to $\mathbf{io}, \mathbf{oi}, \mathbf{ii}$, depending whether b equals $\mathbf{u}, \mathbf{o}, \mathbf{i}, \mathbf{z}$,
- with $a_4 = \mathbf{o}$ or $a_4 = \mathbf{i}$ depending whether $s = +$ or $s = -$ and
- with $\mathbf{ia}_m \dots \mathbf{a}_5$ a binary number ($\mathbf{o}$ for 0 and $\mathbf{i}$ for 1) whose value is equal to $|\pi(j) - \pi(i, a)| + \frac{3}{2}$.

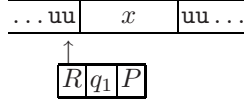
5.3 Operation of the Universal Pair

As already mentioned, the program U has 54 states, $q_1, \dots, q_{54}$, and 184 instructions. These instructions are divided in 10 modules $A, B, C, \dots, J$ organized as follows:

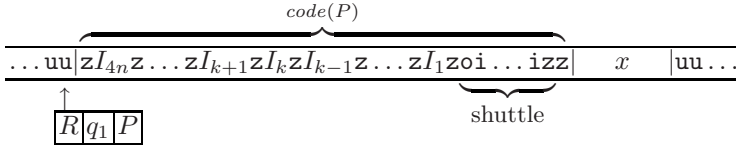


The numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 denote respectively the states $q_1, q_{24}, q_{35}, q_{43}, q_{49}, q_{15}, q_{13}, q_7, q_{10}, q_{23}$. They are called $X_0, X_2, \dots, X_9$ in the program U in the annex A. In the annex B, we also give a graph whose vertices are the states and the edges the instructions of U : each instruction $[q_i, abs, q_j]$ is represented by an arrow, labeled abs , going from q_i to q_j . Note that the vertices a, b, c and 7 have two occurrences which must be merged.

Initial configurations. Initially the machines executing P is in the configuration

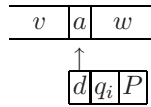


and the machine executing U is in the *corresponding initial configuration*

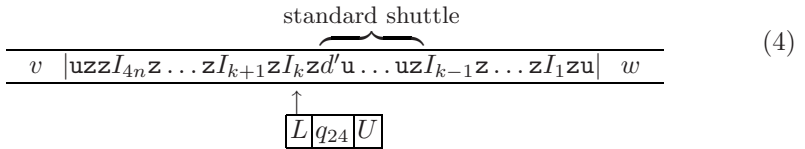


While the machine executing P performs no transitions, the machine executing U performs a sequence of initial transitions, always the same, involving the instructions of module A and some instructions already there in the modules I, H, G, F . Then the machines executing P and U end up respectively in the following current configurations with $k = 1$:

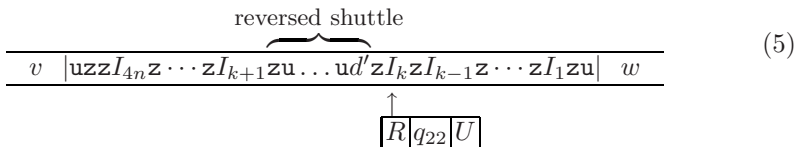
Current configurations. While the machine executing P is in the current configuration



the machine executing U is in the *corresponding current configuration*



or



5.4 Introspection Coefficient of Our Pair for the Turing Machine

First we have chosen a reversing program P such that, for all $n \geq$, one gets $out(P, a_1 a_2 \dots a_n) = a_n \dots a_2 a_1$, with the a_i taken from $\{\mathbf{o}, \mathbf{i}, \mathbf{z}\}$. The program P has 32 instructions and 9 states. We have $|code(P)| = 265$ and $|code(U)| = 1552$. We obtain the following results for the pair $(U, code)$:

x	$ orbit(P, x) $	$ orbit(U, code(P) \cdot x) $	$ orbit(U, code(U) \cdot code(P) \cdot x) $	$\frac{ orbit(U, code(U) \cdot code(P) \cdot x) }{ orbit(U, code(P) \cdot x) }$
ε	2	5 927	22 974 203	3 876.19
$\mathbf{o}$	6	13 335	51 436 123	3 857.23
$\mathbf{oi}$	12	23 095	88 887 191	3 848.76
$\mathbf{oiz}$	20	35 377	136 067 693	3 846.22
$\mathbf{oizo}$	30	49 663	190 667 285	3 839.22

It can be seen that we have succeeded in running the universal program U on its own code and thus to compute a first approximation of the introspection coefficient.

Second, after having computed the column vector B of size $184 \times 4 = 736$ and the matrix A of size 736, using Theorem 1, we have verified that U admits an introspection coefficient and computed its value: for all words x on Σ such that $orbit(P, x) \neq \nearrow$,

$$\lim_{n \rightarrow \infty} \frac{|orbit(U, code_1(U)^{n+1} \cdot x)|}{|orbit(U, code(U)^n \cdot x)|} = 3\,672.98$$

Anyone interested in more details may consult [4]. There, it is also proven that a more classical Turing machine, with 361 instructions and 106 states, has the same introspection coefficient.

6 Universal Pair for the Indirect Addressing Arithmetic Machine

6.1 The Universal Pair

It is interesting to compare the complexities of our universal program for a Turing machine with the complexity of a universal program for the indirect addressing arithmetic machine with same alphabet $\Sigma = \{c_1, c_2, c_3\}$, with $c_1 = \mathbf{o}$, $c_2 = \mathbf{i}$ and $c_3 = \mathbf{z}$. We have written such a universal program U using 103 instructions and with $|code(U)| = 1042$. It can be seen in annex C.

6.2 Operation of the Universal Pair

The universal pair $(U, code)$ for arithmetic machine with indirect addressing operates roughly as following:

Current configuration

r_0	r_1	r_2	
0	3	2	

Execution of one instruction of

$$P = \left\{ \begin{array}{l} [0, plus, 2, 1], \\ [1, cst, 11, 1], \\ [2, from, 5, 2], \\ [3, ifze, 5, 8], \\ [4, sub, 5, 11], \\ [5, to, 2, 5], \\ [6, plus, 2, 11], \\ [7, cst, 0, 1] \end{array} \right\}$$

Next configuration

r_0	r_1	r_2	
1	3	5	

Corresponding configuration

r_0	r_1	r_2	
50			$code(P)$ 0 3 2

Execution of several corresponding instruction of

$$U = \left\{ \begin{array}{l} [0, cst, 8, 0], \\ [1, cst, 10, 2], \\ [2, cst, 11, 11], \\ \dots \\ [99, cst, 0, 49], \\ [100, plus, 9, 1], \\ [101, from, 9, 9], \\ [102, plus, 1, 9] \end{array} \right\}$$

Corresponding configuration

r_0	r_1	r_2	
50			$code(P)$ 1 3 5

6.3 Introspection Coefficient of Our Pair for the Indirect Addressing Machine

On particular examples we obtain the following results:

x	$ orbit(P, x) $	$ orbit(U, code(P) \cdot x) $	$ orbit(U, code(U) \cdot code(P) \cdot x) $	$\frac{ orbit(U, code(U) \cdot code(P) \cdot x) }{ orbit(U, code(P) \cdot x) }$
ε	12	2 372	72 110	30.40
o	16	2 473	74 758	30.23
oi	31	2 860	84 916	29.69
oiz	35	2 961	87 564	29.57
oizo	50	3 348	97 722	29.19

where P is a reversing program of 21 instructions, with $|code(P)| = 216$, such that, for all $n \geq 0$ one obtains $out(P, a_1 a_2 \dots a_n) = a_n \dots a_2 a_1$, with the a_i taken from $\{o, i, z\}$. The introspection coefficient obtained is:

$$\lim_{n \rightarrow \infty} \frac{|orbit(U, code(U)^{n+1} \cdot x)|}{|orbit(U, code(U)^n \cdot x)|} = 26.27$$

Anyone interested in more details may consult [4].

7 Conclusion

Unless one “cheats”, it is difficult to improve the introspection coefficient of our universal Turing machine which took us a considerable effort to develop.

Suppose, which is the case, that we have at our disposal a first universal pair $(U, code)$ for a Turing machine.

A first way of cheating consists of constructing the pair $(U, code')$ from the universal pair $(U, code)$, with

$$code'(P) = \begin{cases} \varepsilon, & \text{if } P = U, \\ code(P), & \text{if } P \neq U. \end{cases}$$

Then we have

$$\frac{|orbit(U, code'(U^{n+1} \cdot x))|}{|orbit(U, code'(U)^n \cdot x)|} = \frac{|orbit(U, x)|}{|orbit(U, x)|} = 1$$

and $(U, code')$ is a universal pair with an introspection coefficient equal to 1.

There is a second more sophisticated way of cheating, without modifying the coding function $code$. Starting from the universal program U we construct a program U' , which, after having erased as many times as possible a given word z occurring as prefix of the input, behaves as U on the remaining input. According to the recursion theorem [6,8], it is possible to take z equal to $code(U')$ and thus to obtain a universal program U' such that, for all $y \in \Sigma^*$ having not $code(U)'$ as prefix,

$$orbit(U', code(U')^n \cdot y) = nk_1 + k_2(y),$$

where k_1 and $k_2(y)$ are positive integers, with k_1 being independent of y . Then we have

$$\frac{|orbit(U', code(U')^{n+1} \cdot y)|}{|orbit(U', code(U')^n \cdot y)|} = \frac{|orbit(U, x)| + (n+1)k_1 + k_2(y)}{|orbit(U, x)| + nk_1 + k_2(y)} = 1 + \frac{k_1}{|orbit(U, x)| + k_2(y) + nk_1}.$$

By letting n tend toward infinity we obtain an introspection coefficient equal to 1 for the pair $(U', code)$.

Unfortunately our introspection coefficient definition, page 8, does not disallow these two kinds of cheating. What one really would like to prevent is that the function $code$ or the program U “behaves differently” on the program P , depending whether P is or is not equal to U . It is an open problem to express this restriction in the definition of the introspection coefficient.

Finally we would like to mention that we tested our universal programs with a package written in MAPLE 8. In each case this package was also used to calculate and manipulate the matrix A and the vectors B . Notably it was used to compute the eigenvalues of A to obtain the introspection coefficient.

References

1. Bleuzen, N., Colmerauer, A.: Optimal Narrowing of a Block of Sortings in Optimal time. Constraints 5(1-2), 85–118 (2000), <http://alain.colmerauer@free.fr>
2. Colmerauer, A.: An Introduction to Prolog III. Communications of the ACM 33(7), 68–90 (1990), <http://alain.colmerauer@free.fr>

3. Colmerauer, A.: Prolog IV (1995), <http://alain.colmerauer@free.fr>
4. Colmerauer, A.: On the complexity of universal programs. In: Machine, Computations and Universality (Saint-Petersburg 2004). LNCS, pp. 18–35 (2005), <http://alain.colmerauer@free.fr>
5. Minsky, M.: shape Computations: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs (1967)
6. Rogers, H.: Theory of Recursive Functions and Effective Computability. McGraw-Hill, New York (1967); fifth printing. MIT Press (2002)
7. Rogozin, Y.: Small universal Turing machines. Theoretical Computer Science 168(2) (November 1996)
8. Sipser, M.: Introduction to the Theory of Computation. PWS Publishing Company (1997)

Appendix

A Annex: Universal Turing Program

```
# A BEGINNING
[X0,uz+,A1],    [X0,oo+,A1],    [X0,ii+,A1],    [X0,zu-,X7],
                [A1,oo+,A1],    [A1,ii+,A1],    [A1,zz+,X0],
                [X7,zz+,X8],

# B INSTRUCTION TAIL COPYING
                [X1,oo+,B1],    [X1,ii+,B1],
                [B1,oo+,B5],    [B1,iu-,B2],
                [B2,oo+,B2],    [B2,ii+,B2],    [B2,zz+,B3],
                [B3,oi-,B4i],    [B3,io-,B4i],
[B4o,uo+,B5],    [B4o,oo+,B4o],    [B4o,ii+,B4o],    [B4o,zz+,B4o],
[B4i,ui+,B5],    [B4i,oo+,B4i],    [B4i,ii+,B4i],    [B4i,zz+,B4i],
[B5,uu-,B6],    [B5,ou-,B4o],    [B5,iu-,B4i],    [B5,zz-,B7],
                [B6,oo+,B4o],    [B6,ii+,B4i],

# Replacement of the remaining u's by o's
[B7,uo+,B9],    [B7,oo+,B7],    [B7,ii+,B7],    [B7,zz+,B7],
[B9,uo+,B9],    [B9,oo+,X2],    [B9,zz-,B10],
                [B10,oo+,B10],    [B10,ii+,B10],    [B10,zz+,B9],

# C INSTRUCTION HEAD COPYING
# Creating the symbol to be written
                [X2,oo+,C2],    [X2,iu-,C1],
[C1,ui+,C2],    [C1,oi+,C1],    [C1,io+,C1],    [C1,zu-,C1],
                [C2,oo-,C3o],    [C2,ii-,C3i],
[C3o,uo+,C4],    [C3o,oo+,C3o],    [C3o,ii+,C3o],    [C3o,zu+,C4],
[C3i,uz+,C4],    [C3i,oo+,C3i],    [C3i,ii+,C3i],    [C3i,zi+,C4],
# Taking in account the direction
                [C4,oi-,C5],    [C4,ii-,X3],
[C5,uu+,C6],    [C5,oo+,C6],    [C5,ii+,C6],    [C5,zz+,C6],
                [C6,oo-,X3],

# D WRITING, MOVING AND READING
# Writing and reading
[X3,uu-,D1u],    [X3,ou-,D1o],    [X3,iu-,D1i],    [X3,zu-,D1z],
[D0,uu-,D1z],    [D0,ou-,D1i],    [D0,iu-,D1o],    [D0,zu-,D1u],
[D1u,uu-,X4],    [D1u,oo+,D1u],    [D1u,ii+,D1u],    [D1u,zz+,D1u],
[D1o,uo-,X4],    [D1o,oo+,D1o],    [D1o,ii+,D1o],    [D1o,zz+,D1o],
[D1i,ui-,X4],    [D1i,oo+,D1i],    [D1i,ii+,D1i],    [D1i,zz+,D1i],
[D1z,uz-,X4],    [D1z,oo+,D1z],    [D1z,ii+,D1z],    [D1z,zz+,D1z],
# Moving
                [X4,zu+,D2z],
                [D2u,ou+,D2o],    [D2u,iu+,D2i],
[D2o,uo+,D2u],    [D2o,oo+,D2o],    [D2o,io+,D2i],    [D2o,zo+,D2z],
[D2i,ui+,D2u],    [D2i,oi+,D2o],    [D2i,ii+,D2i],    [D2i,zi+,D2z],
[D2z,uz+,X3],    [D2z,oz+,D2o],    [D2z,iz+,D2i],    [D2z,zz+,D2zz],
[D2zz,uz+,D0],    [D2zz,oz+,D2o],
```

E SHUTTLE UPDATING

Beginning of the updating

```

      [X4,oo-,E1b], [X4,io-,E1a],
[E1a,uz-,E2a], [E1a,oz-,E2b], [E1a,iz-,X6], [E1a,zz-,X5],
[E1b,uz+,X5], [E1b,oz+,X6], [E1b,iz+,E2b], [E1b,zz+,E2a],

```

End of the updating

```

      [E2a,oo+,E2b], [E2a,iu+,E2b],
      [E2b,oo+,E4], [E2b,ii+,E4],
      [E4,oi+,E4], [E4,io-,E5], [E4,zz-,X7],
[E5,uu+,E5], [E5,oo+,E5], [E5,ii+,E5], [E5,zz-,X6],

```

F SUTTLE MOVING TO NEXT z

```

[X5,uu+,X5], [X5,oo+,X5], [X5,ii+,X5], [X5,zz-,F1],
[F1,uz+,F2u], [F1,oz+,F2o],
[F2u,uu+,F2u], [F2u,ou+,F2o], [F2u,iu+,F2i], [F2u,zu+,F3],
[F2o,uo+,F2u], [F2o,oo+,F2o], [F2o,io+,F2i], [F2o,zo+,F3],
[F2i,ui+,F2u], [F2i,oi+,F2o], [F2i,ii+,F2i], [F2i,zi+,F3],
      [F3,oz-,F4o], [F3,iz-,F4i], [F3,zz-,X6],
[F4o,uu+,F4o], [F4o,oo+,F4o], [F4o,ii+,F4o], [F4o,zo-,F1],
[F4i,uu+,F4i], [F4i,oo+,F4i], [F4i,ii+,F4i], [F4i,zi-,F1],

```

G SHUTTLE DECREASING

```

[X6,uu+,G1], [X6,oo+,G1], [X6,iu+,G1],
[G1,uu-,X7], [G1,oi+,G1], [G1,io+,X5], [G1,zz-,X8],

```

H SHUTTLE REVERSING AFTER BLANK SYMBOLS INTRODUCTION

```

[X7,uu-,E2a], [X7,ou-,E2b], [X7,iu+,X7],
[E2a,uu+,E2a], [E2a,zz-,I1],
[E2b,uu+,E2b], [E2b,zz-,I2],
[I1,uo-,X8],
[I2,ui-,X8],

```

I INSTRUCTION ORIENTATION TEST AFTER BLANK SYMBOLS INTRODUCTION

```

[X8,ui+,X8], [X8,oo+,X8], [X8,iu+,X8], [X8,zz+,I1],
      [I1,oo+,I2], [I1,ii-,I2],
      [I2,oo+,X1], [I2,ii+,X1], [I2,zz+,X5],

```

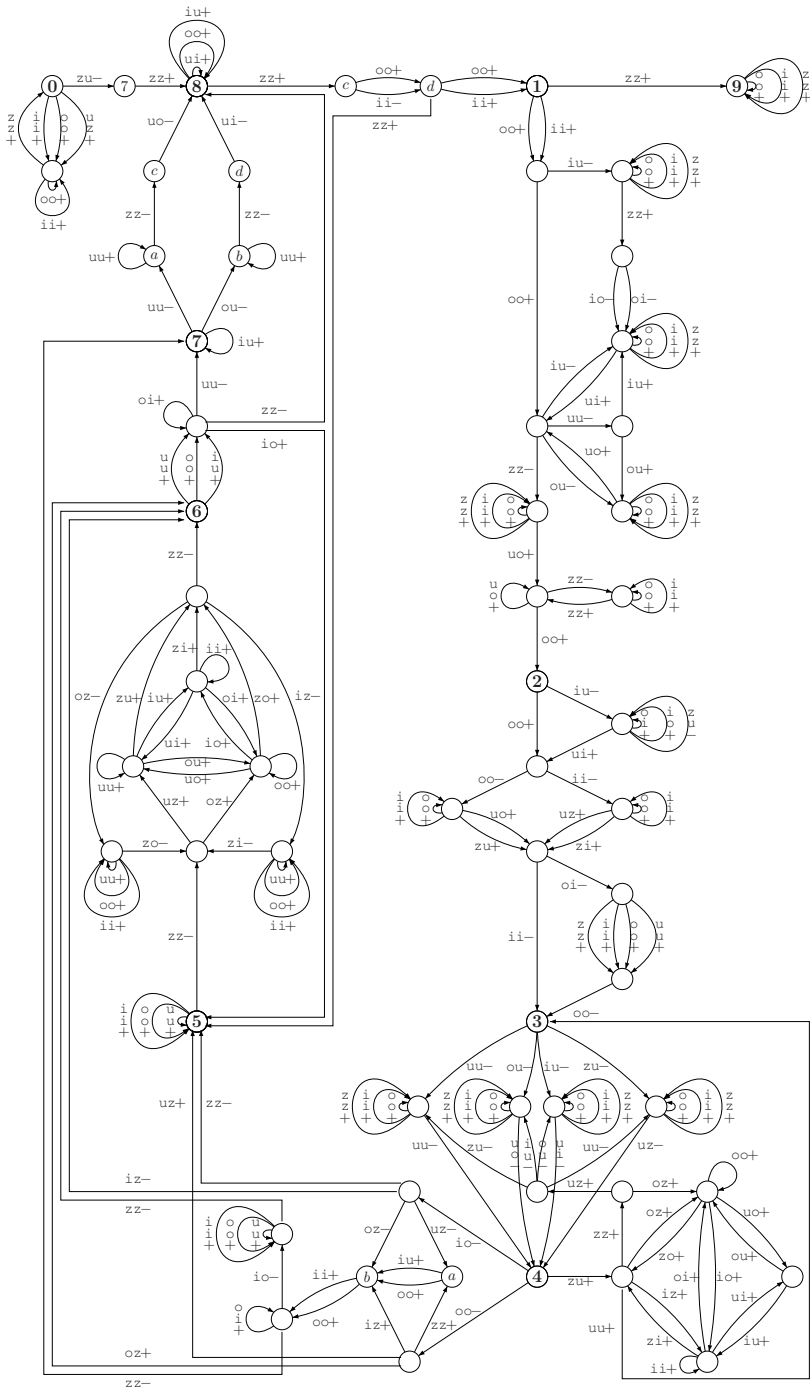
J END OF THE PROGRAM

```

      [X1,zz+,X9],
      [X9,oo+,X9], [X9,ii+,X9], [X9,zz+,X9]];

```

B Annex: Graph of the Universal Turing Program



C Annex: Universal Indirect Addressing Program

```

# INITIALISAT-      [21,ifzero,5,24], # COMPUTING THE      [73,from,5,5],
# ION OF THE        [22,cst,5,0],   # POSITION OF      [74,plus,6,5],
# REGISTERS         [23,cst,4,0],   # INSTRUCTION     [75,to,4,6],
[0,cst,8,0],        [24,plus,4,4],   # NB ZERO         [76,cst,0,46],
[1,cst,10,2],       [25,plus,4,9],   [50,from,7,1],   # MINUS
[2,cst,11,11],      [26,cst,0,15],   [51,cst,6,25],   # INSTRUCTION
[3,cst,12,20],      # CASE a=2     [52,plus,6,7],   [77,from,6,4],
[4,cst,13,26],      [27,ifzero,5,30], [53,plus,6,7],   [78,plus,5,1],
[5,cst,14,33],      [28,cst,5,0],   [54,plus,6,7],   [79,from,5,5],
[6,cst,15,67],      [29,cst,4,0],   # HALTING TEST   [80,sub,6,5],
[7,cst,16,68],      [30,plus,4,4],   [55,cst,7,0],    [81,to,4,6],
[8,cst,17,70],      [31,plus,4,9],   [56,plus,7,1],   [82,cst,0,46],
[9,cst,18,76],      [32,plus,4,9],   [57,sub,7,6],    # FROMINDIRECT
[10,cst,19,82],     [33,cst,0,15],   [58,ifzero,7,100], # INSTRUCTION
[11,cst,20,88],     # CASE a=3     # COMPUTING      [83,plus,5,1],
[12,cst,21,94],     [34,ifzero,5,36], # a:=R[R[6]]     [84,from,5,5],
                   [35,cst,0,40],   [59,from,3,6],   [85,plus,5,1],
# ENCODING OF      [36,plus,2,9],   # COMPUTING      [86,from,5,5],
# THE EMULATED     [37,sub,4,9],   # b:=R[R[6]]+R[1] [87,to,4,5],
# PROGRAM          [38,to,2,4],   [60,plus,6,9],   [88,cst,0,46],
# INITIALISAT-     [39,cst,5,1],   [61,from,4,6],   # TOINDIRECT
# ION OF THE       [40,cst,0,15],   [62,plus,4,1],   # INSTRUCTION
# SOURCE POSIT-    # END          # COMPUTING      [89,from,4,4],
# ION R[1] AND     [41,to,1,8],   # c:=R[R[4]+2]   [90,plus,4,1],
# THE BOOLEAN      [42,cst,4,1],   [63,plus,6,9],   [91,plus,5,1],
# VALUE c          [43,plus,4,1],   [64,from,5,6],   [92,from,5,5],
[13,cst,2,24],     [44,cst,6,25],   # CASE STUDY     [93,to,4,5],
[14,cst,5,1],      [45,to,4,6],   # ACCORDING TO   [94,cst,0,46],
[15,cst,0,16],     # THE VALUE OF a # IFZERO
# INCREASING      # PROGRAM      [65,cst,6,15],   # INSTRUCTION
# THE SOURCE      # EMULATION   [66,plus,6,3],   [95,from,4,4],
# POSITION R[1]    # SKIP INCREM- [67,from,0,6],   [96,ifzero,4,98],
[16,plus,1,9],     # ENTATION OF      # NO             [97,cst,0,46],
# CASE STUDY      # THE INSTRUC- # INSTRUCTION     [98,to,1,5],
# ACCORDING       # TION COUNTER [68,cst,0,99],   [99,cst,0,49],
# TO THE VALUE    [46,cst,0,49],   # CONSTANT       # END
# a OF R[R[1]]    # INCREASING   # INSTRUCTION     [100,plus,9,1],
[17,from,3,1],     # THE INSTRUC-   [69,to,4,5],     [101,from,9,9],
[18,to,1,8],       # TION COUNTER   [70,cst,0,46],   [102,plus,1,9].
[19,plus,3,11],    [47,from,6,1],   # PLUS
[20,from,0,3],     [48,plus,6,9],   # INSTRUCTION
# CASE a=1        [49,to,1,6],   [72,plus,5,1],

```

A Constraint Programming Approach for Allocation and Scheduling on the CELL Broadband Engine

Luca Benini, Michele Lombardi, Michela Milano, and Martino Ruggiero

(1) DEIS, University of Bologna

V.le Risorgimento 2, 40136, Bologna, Italy

{lbenini,mmilano,mlombardi,mruggiero}@deis.unibo.it

Abstract. The Cell BE processor provides both scalable computation power and flexibility, and it is already being adopted for many computational intensive applications like aerospace, defense, medical imaging and gaming. Despite of its merits, it also presents many challenges, as it is now widely known that is very difficult to program the Cell BE in an efficient manner. Hence, the creation of an efficient software development framework is becoming the key challenge for this computational platform.

We have developed a novel software toolkit, called Cellflow, which enables developers to quickly build multi-task applications for Cell-based platform. We support programmers from the initial stage of their work, through a development-time software infrastructure, to the final stage of the application development, proposing a safe and easy-to-use explicit parallel programming model.

A fundamental component of the software toolkit is the off-line allocator and scheduler that manages hardware resources while optimizing performance metrics such as execution time, allocation costs, power. The optimization engine receives as input a task graph representing an application, the hardware resources and produces an optimal allocation and scheduling. We have developed various approaches, either based on decomposition [5] or based on pure Constraint Programming, this latter being the core of this paper. We have identified instance features that guide toward the choice of the best solver for the instance at hand.

Experimental result show that Constraint Programming (possibly combined with Integer Programming) is a proper tool for dealing with this kind of applications achieving very good performance.

1 Introduction

Single-chip multicore platforms are becoming widespread in high-end embedded computing applications (networking, communication, graphics, signal processing). The Cell Broadband Engine is probably one of the highest-volume multicore platforms in use today, targeting interactive graphics and advanced signal processing¹. It is a heterogeneous multi-core architecture composed by a standard general purpose microprocessor (called PPE), with eight coprocessing units (called SPEs) integrated on the same chip. The SPE is a processor designed for streaming workloads, featuring a local memory, and a globally-coherent DMA (Direct Memory Access) engine [15], [28].

¹ Sony's Playstation 3, powered by Cell BE, had sold more than 10M pieces at the end of 2007.

The heterogeneity of its processing elements and, above all, the limited explicitly-managed on-chip memory and the multiple options for exploiting hardware parallelism, make efficient application design and implementation on the Cell BE a major challenge. Efficient programming requires one to explicitly manage the resources available to each SPE, as well the allocation and scheduling of activities on them, the storage resources, the movement of data and synchronization. As a result, even with the help of APIs and advanced programming environments, programming Cell in an efficient fashion is a daunting task. Therefore, significant effort is being focused on the development of software optimization tools and methods to automate the mapping of complex parallel applications onto the Cell BE platform.

The final goal of this work is to enable developers to quickly build multi-task applications using a high-level explicitly parallel programming model. Low-level compilers and hardware-optimized core functions are provided by the the SDK from IBM [12]. However, the basic SDK does not offer any facility for optimizing the resource utilization in terms of both allocation and scheduling, memory transfers and utilization. We want to set programmers free from the issue of managing allocation and scheduling tasks, so they can focus on developing the core algorithms of the application.

The allocation and scheduling problems that are at the core of the mapping task are quite large and extremely challenging, and they are usually tackled using incomplete approaches. Even though incomplete approaches can be computationally efficient, they generally produce sub-optimal solutions. This is a significant shortcoming especially for demanding applications with tight execution time constraints, as incomplete optimizers may fail to find a feasible solution even when it does exist. Hence, efficient complete approaches are of great practical interest: not only they help programmers in taking hard design decisions, but also they can significantly extend the size and complexity of applications that can be run on the target hardware platform while meeting performance constraint.

For the problem at hand we have developed two approaches. One is based on Logic Based Benders Decomposition [8], and in particular on a recursive application of the technique. This approach has been proposed in [5] and will be recalled here for making the paper self contained. The second approach, which is the core of the present paper, is a pure CP model targeting both allocation and scheduling. We have experimentally compared the two approaches and identified instance features that guide toward the choice of the best solving strategy.

2 The Problem

The current design methodology for multicore systems on chip is hampered by a lack of appropriate design tools, leading to low efficiency and productivity. Software optimization is a key requirement for building cost- and power-efficient electronic systems, while meeting tight real-time constraints and ensuring predictability and reliability, and is one of the most critical challenges in today's high-end computing.

Embedded devices are not general purpose, but run a set of predefined applications during the entire system lifetime. Therefore software compilation can be optimized once for all at design time thus improving the performance of the overall system. Thus,

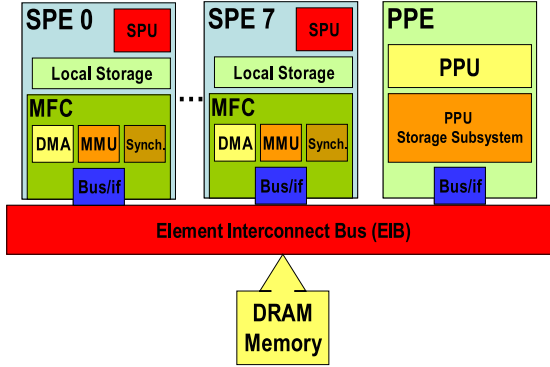


Fig. 1. Cell Broadband Engine Hardware Architecture

optimization is a critical component in the design of next-generation, highly program mable, intelligent embedded devices.

We focus on a well-known multicore platform, namely the IBM Cell BE processor (described in section 2.1), and we address the problem of allocating and scheduling its processors, communication channels and memories. The application that runs on top of the target platform is abstracted as a task graph (described in section 2.2). Each task is labelled with its execution time, memory and communication requirements. Arcs in the task graph represent data dependencies and communications between pairs of tasks. The optimization metric we take into account is the application execution time that should be minimized.

2.1 Cell BE Hardware Architecture

In this section we give a brief overview of the Cell hardware architecture, focusing on the features that are most relevant for our optimization tools. Cell is a non-homogeneous multi-core processor [32] which includes a 64-bit PowerPC processor element (PPE) and eight synergistic processor elements (SPEs), connected by an internal high bandwidth Element Interconnect Bus (EIB) [29]. Figure 1 shows a pictorial overview of the Cell Broadband Engine Hardware Architecture. The PPE is dedicated to the operating system and acts as the master of the system, while the eight synergistic processors are optimized for computation-intensive applications. The PPE is a multithreaded core and has two levels of on-chip cache. However, the main computing power of the Cell processor is provided by the eight SPEs. The SPE is a computation-intensive coprocessor designed to accelerate media and streaming workloads [27]. Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC). The MFC includes a DMA controller, a memory management unit (MMU), a bus interface unit, and an atomic unit for synchronization with other SPUs and the PPE.

Efficient SPE software should heavily optimize memory usage, since the SPEs operate on a limited on-chip memory (only 256 KB local store) that stores both instructions and data required by the program. The local memory of the SPEs is not coherent with

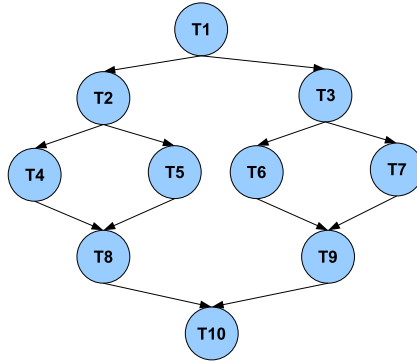


Fig. 2. Example of task graph

the PPE main memory, and data transfers to and from the SPE local memories must be explicitly managed by using asynchronous coherent DMA commands.

2.2 The Target Application

The target application to be executed on top of the hardware platform is input to our methodology, and for this purpose it must be represented as a task graph. This latter consists of a graph pointing out the parallel structure of the program. The application workload is therefore partitioned into computation sub-units denoted as tasks, which are the nodes of the graph. Graph edges connecting any two nodes indicate task dependencies due to communication and/or synchronization. Tasks communicate through queues and each task can handle several input/output queues. For example task T_9 in Figure 2 reads two input queues from tasks T_6 and T_7 and writes an output queue for task T_{10} .

Task execution is modeled and structured in three phases (see Figure 3): all input communication queues are read (Input Reading), task computation activity is performed (Task Execution) and finally all output queues are written (Output Writing). Each phase consists of an atomic activity. Each task also has two kinds of associated memory requirements:

1. Program Data: storage locations are required for computation data and for processor instructions;
2. Communication queues: each task needs queues to transmit and receive messages to/from other tasks, eventually mapped on different SPEs.

Both these memory requirements can be either allocated on the local storage of each SPE or in the shared memory (DRAM in Figure 1).



Fig. 3. Three phases behavior of Tasks

Durations are linked to the allocation choices: the duration of an execution phase in case of remote allocation of program data ($dmax^{ex}$) is greater than in case of a local allocation $dmin^{ex}$. Writing (and reading) operations have their minimum possible value ($dmin^{wr}, dmin^{rd}$) if the communication queue is on the local memory of the producer (resp. consumer) tasks, a higher value ($dmed^{wr}, dmed^{rd}$) if it is allocated on the local memory of the consumer (resp. producer) task, an even higher value ($dmax^{wr}, dmax^{rd}$) in case of remote allocation (on the on-chip DRAM memory).

3 Why CP

The main goal of this paper is to apply software optimization for maximizing the exploit of the hardware resources of the CELL BE architecture.

Scientific literature related to our problem explores many directions: we here recall the main research trends:

- exploitation of heterogeneous parallelism provided by the CELL architecture possibly performing automated scheduling and allocation;
- software optimization for other (yet similar) multicore platforms.

The Cell architecture supports a wide range of heterogeneous parallelism levels. To our knowledge, prior work is mainly focused on trying to exploit fine grained parallelism of Cell, such as at instruction and functional level, while our work is one of the few approaches at task level. In [14] authors present a framework for the automatic exploitation of the functional parallelism of a sequential program through the different SPEs. Their work is based on annotation of the source code of target application. A runtime library deals with generating threads, scheduling them on the SPEs, and transferring data to/from them. The authors in [30] present a realtime software platform for the Cell processor. It is based on the virtualization of the processing resources and a real-time resource scheduler which runs on the PPE. The compiler described in [20] implements techniques for optimizing the execution of scalar code in SIMD units, subword optimization and other techniques. Authors in [19] describe several compiler techniques that aim at automatically generating high-quality code over a wide range of heterogeneous parallelism available on the CELL processor.

At task level, the authors in [33] propose a programming model based on micro-tasks communicating through message passing interface. The micro-task represents a unit of computation that causes communication at its beginning and end. They tackle the mapping and scheduling problem by a suboptimal heuristic solver. The work in [34] describes a multicore streaming layer whose main goal is to abstract away the architecture-specific details that complicate the scheduling of computation and communication activities in a stream program. They propose both dynamic and static scheduling facilities, but without any optimality guarantee.

The literature on optimization of other multicore architectures uses heuristic approaches for mapping and scheduling task graphs onto the target platforms. In [16] a re-timing heuristic is used to implement pipelined scheduling, that optimizes the initiation interval, the number of pipeline stages and memory requirements of a particular design alternative. Pipelined execution of a set of periodic activities is also addressed

in [17], for the case where tasks have deadlines larger than their periods. Palazzari et al. [31] focus on scheduling to sustain the throughput of a given periodic task set and to serve aperiodic requests associated with hard real-time constraints. Mapping of tasks to processors, pipelining of system specification and scheduling of each pipeline stage have been addressed in [18], aiming at satisfying throughput constraints at minimal hardware cost. A comparative study of well-known heuristic search techniques (genetic algorithms, simulated annealing and tabu search) is reported in [21]. Eles et al. [22] compare the use of simulated annealing and tabu search for partitioning a graph into hardware and software parts while trying to reduce communication and synchronization between parts. More scalable versions of these algorithms for large real-time systems are introduced in [23]. Many heuristic scheduling algorithms are variants and extensions of list scheduling [24], a scheduling algorithm coming from the real time literature.

Heuristic approaches provide no guarantees about the quality of the final solution. On the other hand, complete approaches which compute the optimum solution (possibly, with a high computational cost), can be attractive for statically scheduled systems, where the solution is computed once and applied throughout the entire lifetime of the system.

Our previous work [3], [4] was aimed at optimally solving task graphs allocation and scheduling on a different multicore platform (called MPARM and based on ARM processors) using a Logic Based Benders Decomposition approach. The allocation part is solved through Integer Programming and the scheduling problem via Constraint Programming. We have applied and extended this approach for the CELL BE platform in [5]. We will summarize this paper in section 4. In this paper we propose a pure Constraint Programming approach for this problem.

CP has been previously used to solve similar, yet simplified, problems. The work in [25] is based on Constraint Logic Programming to represent system synthesis problem, and leverages a set of finite domain variables and constraints imposed on these variables. Optimal solutions can be obtained for small problems, while large problems require the use of heuristic algorithms. The proposed framework is able to create pipelined implementations in order to increase the design throughput. In [26] the embedded system is represented by a set of finite domain constraints defining different requirements on process timing, system resources and interprocess communication. The assignment of processes to processors and interprocess communications to buses as well as their scheduling are then defined as an optimization problem tackled by means of constraint solving techniques.

4 How CP

For the problem of allocating and scheduling task graphs onto the CELL BE platform we have implemented two approaches. One is based on a recursive application of Logic Based Benders Decomposition [8] and is extensively described in [5]. We recall here the main structure of the solution technique, while we refer to [5] for modeling details and extensive comparison with a traditional (two-stage) decomposition approach.

The second model we propose is the core of this paper and is a pure CP model where both allocation and scheduling are solved using a single monolithic model.

We describe in detail this second approach and propose an experimental evaluation in section 5 along with a comparison with the decomposition approach.

4.1 Decomposition Based Approach

The problem at hand can be solved using a Logic Based Benders decomposition approach similarly to [3], [4], [7], [6], [9], [10], and [11], where the allocation is modelled and solved in the master problem (usually using Integer Programming) while the scheduling problem is tackled as a subproblem (possibly via Constraint Programming). This approach does not scale well and in [5] we have shown that the reason is the poor balancing between the allocation and the scheduling components, as the first is much more complicated.

Therefore, we have experimented a multi-stage decomposition, which is actually a recursive application of standard Logic based Benders' Decomposition (LBD), that aims at obtaining balanced and lighter components. The allocation part should be decomposed again in two subproblems, each part being easily solvable.

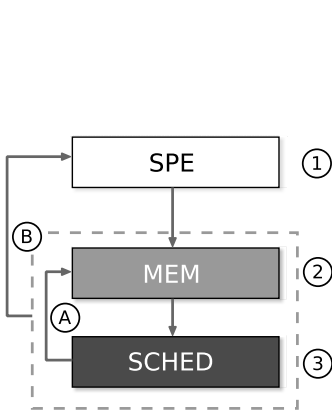


Fig. 4. Solver architecture: two level Logic based Benders' Decomposition

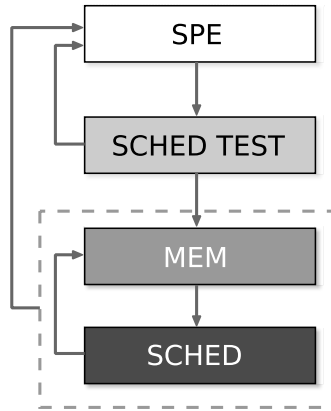


Fig. 5. Solver architecture with schedulability test

In Figure 4 at level one the SPE assignment problem (SPE stage) that computes task to processor assignment acts as the master problem, while memory device assignment and scheduling as a whole are the subproblem. At level two (the dashed box in Figure 4) the memory assignment (MEM stage) is the master and the scheduling (SCHED stage) is the correspondent subproblem. The first step of the solution process is the computation of a task-to-SPE assignment; then, based on that assignment, allocation choices for all memory requirements are taken. Finally, a scheduling problem with fixed resource assignments and fixed durations is solved. When the SCHED problem is solved (no matter if a solution has been found), one or more cuts (labeled A) are generated to forbid (at least) the current memory device allocation and the process is restarted from the MEM stage; in addition, if the scheduling problem is feasible, an upper bound on the

value of the next solution is also posted. When the MEM & SCHED subproblem ends (either successfully or not), more cuts (labeled B) are generated to forbid the current task-to-SPE assignment. When the SPE stage becomes infeasible the process is over, and converges to the optimal solution for the problem overall.

We found that quite often SPE allocation choices are by themselves very relevant: in particular, a bad SPE assignment is sometimes sufficient to make the scheduling problem unfeasible. Thus, after the task to processor allocation, we can first check whether the SPE allocation is schedulable, as depicted in Figure 5 (SCHED TEST). In practice, if the given allocation with minimal task durations is already infeasible for the scheduling component, then it is useless to complete it with the memory assignment that cannot lead to any feasible solution overall.

4.2 Pure CP Model

In alternative to the decomposition approach, we have implemented a pure CP model that is solved using the commercial tool ILOG Scheduler/Solver 6.3.

Let n be the number of tasks, m the number of arcs and p the number of processing elements.

The possible allocation choices are modeled by means of the following variables:

$$\begin{aligned} TPE_i &\in \{0, \dots, p-1\} & \forall i = 0, \dots, n-1 \\ M_i &\in \{0, 1\} & \forall i = 0, \dots, n-1 \\ APE_r &\in \{-1, \dots, p-1\} & \forall r = 0, \dots, m-1 \end{aligned}$$

TPE_i is the processing element assigned to task t_i . Similarly, if $APE_r = j$ then the communication buffer related to arc a_r is on the local memory of the processing element j , while if $APE_r = -1$ the communication buffer is allocated on the remote memory. Finally, M_i is 1 if program data of task t_i are allocated locally to the same processor of task t_i .

Due to architectural restrictions, a communication buffer can be allocated either on the local memory of the source task, or that of the target task, or on the remote memory; therefore for the arc r connecting nodes representing tasks t_h and t_k :

$$APE_r = TPE_h \vee APE_r = TPE_k \vee APE_r = -1$$

From a scheduling standpoint, each task is modeled as a set of non preemptive activities a , each with a start variable $start(a)$ and an end variable $end(a)$. In particular, a task t_i is split into an activity modeling its execution phase ex_i , and a set of activities modeling each one the reading and writing of a communication buffer, i.e., wr_r for each outgoing arc r and rd_r for each incoming arc r :

$$\begin{aligned} ex_i(ED_i) & \quad \forall t_i \\ wr_r(WD_r) & \quad \forall a_r = (t_i, t_k) \\ rd_r(RD_r) & \quad \forall a_r = (t_h, t_i) \end{aligned}$$

The duration of each activity is defined by the proper variable and is reported between round brackets after its name. It depends on the related memory allocation choices; hence we define a variable for each execution and communication task:

$$\begin{aligned} ED_i &\in \{0, \dots, \dots eoh\} & \forall i = 0, \dots, n-1 \\ WD_r &\in \{0, \dots, \dots eoh\} & \forall r = 0, \dots, m-1 \\ RD_r &\in \{0, \dots, \dots eoh\} & \forall r = 0, \dots, m-1 \end{aligned}$$

ED_i is the duration of the communication phase of task t_i , WD_r and RD_r respectively are the time needed to write and read buffer r . Their range is the whole temporal horizon (eoh is the end of horizon).

As stated in section 2.2, durations are linked to the allocation choices; the duration of an execution phase in case of remote allocation of program data ($dmax^{ex}$) is greater than in case of local allocation. Writing (and reading) operations have their minimum possible value ($dmin^{wr}$, $dmin^{rd}$) if the communication queue is on the local memory of the producer task (resp. consumer), a higher value ($dmed^{wr}$, $dmed^{rd}$) if it is allocated on the local memory of the consumer (resp. producer) task, an even higher value ($dmax^{wr}$, $dmax^{rd}$) in case of remote allocation of communication queue in DRAM. All those properties are enforced by means of the following constraints:

$$\begin{aligned} \forall i = 0, \dots, n-1 \quad & ED_i = dmin_i^{ex} + \\ & (dmax_i^{ex} - dmin_i^{ex})(1 - M_i) \\ \forall r = 0, \dots, m-1, a_r = (t_h, t_k) \quad & WD_i = dmin_r^{wr} + \\ & (dmax_r^{wr} - dmin_r^{wr})(APE_r = -1) + \\ & (dmed_r^{wr} - dmin_r^{wr})(APE_r = TPE_k) \\ \forall r = 0, \dots, m-1, a_r = (t_h, t_k) \quad & RD_i = dmin_r^{rd} + \\ & (dmax_r^{rd} - dmin_r^{rd})(APE_r = -1) + \\ & (dmed_r^{rd} - dmin_r^{rd})(APE_r = TPE_h) \end{aligned}$$

All reading operations are performed immediately before the execution, and all writing operations start immediately after. Let $r_0, \dots, r_{h-1}$ be the indices of the ingoing arcs of task t_i and $r_h, \dots, r_{k-1}$ those of the outgoing arcs; then:

$$\begin{aligned} end(rd_{r_j}) &= start(rd_{r_{j+1}}) & \forall j = 0, h-2 \\ end(rd_{r_{h-1}}) &= start(ex_i) \\ end(ex_i) &= start(wr_{r_h}) \\ end(rd_{r_j}) &= start(rd_{r_{j+1}}) & \forall j = h, k-2 \end{aligned}$$

All resource constraints are triggered when the TPE allocation variables are assigned; in particular if $TPE_i = j$, all reading, writing and execution activities related to task t_i require processing element j . The resource capacity constraint is enforced by a timetable constraint and a precedence graph constraint available in ILOG Scheduler 6.3 [13].

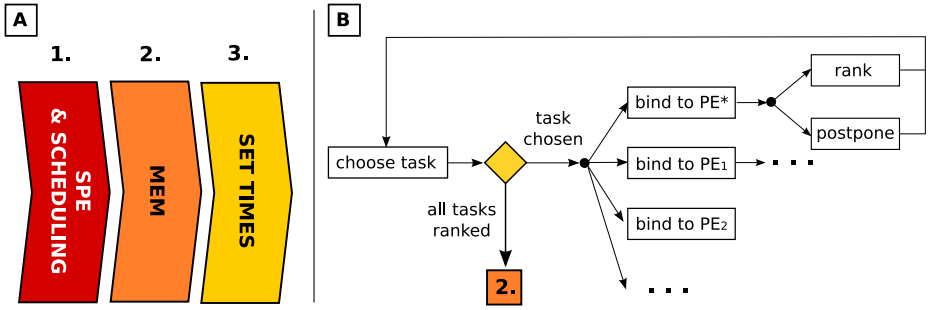


Fig. 6. A: Structure of the dynamic search strategy; B: Operation schema for phase 1

Search Strategy

The model is solved by means of a dynamic search strategy where resource allocation and scheduling decisions are interleaved.

We chose this approach since most resource constraints are not able to effectively prune start and end variables as long as the time windows are large and no task (or just a few of them) has an obligatory region: in particular it is difficult, before scheduling decisions are taken, to effectively exploit the presence of precedence relations and makespan bounds. In our approach, tasks are scheduled immediately after they are assigned to a processing element: this results in immediate updates of the time windows for all tasks linked by precedence relations.

The main drawback with this method is that an early bad choice is likely to lead to thrashing, due to the size of the search space resulting from the mixture of allocation and scheduling decisions; a pure two phases allocation and scheduling approach, like the decomposition based one presented in the previous section, would be able to recover faster from such a situation.

Intuitively, the presence of many precedence constraints strongly shrinks the set of good allocation choices and is likely to guide the allocation toward promising choices, whereas if the graph mostly contains independent or loosely related tasks a two stages approach is probably to be preferred.

A considerable difficulty in our specific case is set by the need to assign each task and arc both to a processing element and to a storage device: this makes the number of possible allocations too big to completely define the allocation of each task right before it is scheduled. Therefore we chose to postpone the memory allocation stage after the main scheduling decisions are taken, as depicted in Figure 6A.

Since task durations directly depend on memory assignment, scheduling decisions taken in phase 1 of Figure 6 had to be relaxed to enable the construction of a *fluid* schedule with variable durations. In practice we adopted a Precedence Constraint Postponing approach [1,2], by just adding precedence relations to fix the order of tasks at the time they are assigned to SPEs: they will be given a start time only once the memory devices are assigned. Note this time setting step is done in polynomial time. Figure 7A shows an example of fluid schedule where tasks have variable durations and precedence relations have been added to fix the order of the tasks on each SPE;

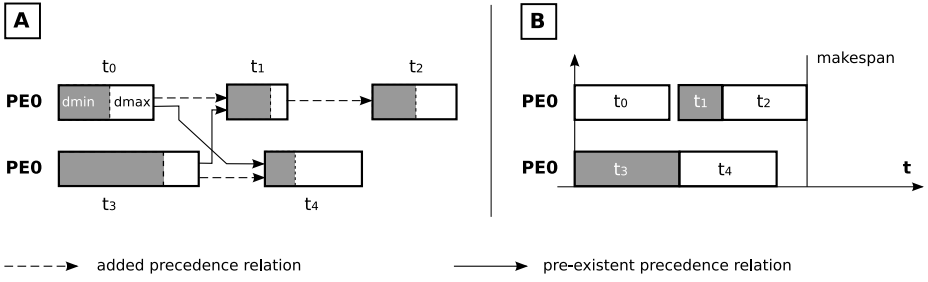


Fig. 7. A: A fluid schedule; B: A possible fixed schedule

Figure 7B show a corresponding schedule where all durations are decided (a grey box means the minimum duration is used, a white box means the opposite).

In deeper detail, the SPE allocation and scheduling phase operates according to the schema of Figure 6B: first, the task with minimum start time is selected – ties are broken looking at the (least) maximum end time and then at the task index. Second, the SPE where the task can be allocated at its minimum start time is identified (let it be SPE^*), then a choice point is open, with a branch for each SPE. Along each branch the task is bound to the corresponding resource and a *rank or postpone* decision is taken: we try to rank the task immediately after the last activity on the selected resource, otherwise the task is postponed and not considered ready until its minimum start time changes due to propagation (this is analogous to the standard schedule or postpone strategy in ILOG). The process is reiterated as long as there are unranked tasks.

In phase 2, memory requirements are allocated to storage devices, selecting at each step the variable with the smallest domain; in phase 3 a start time is assigned to each task. Finally, since the processing elements are symmetric resources the procedure embeds quite standard symmetry breaking techniques to prevent the generation of useless branches in the search tree.

5 Computational Efficiency

The decomposition based approach has been implemented using the state of the art solvers ILOG Cplex 10.1 and Scheduler/Solver 6.3, while the pure CP model has been implemented on Scheduler/Solver 6.3.

Since the main goal of the paper is to study and compare the performance of the two approaches it would be not realistic to assume the availability of such a large benchmark set that would allow us to sample a large variety of problem instances. Therefore we resorted to synthetic benchmarks as follows.

A first group of 90 instances is coming from the actual execution of multi tasking programs on a CELL BE architecture. These benchmarks have been created by synthesising code (matrix multiplication) tuning the computation vs. communication effort which is related to matrix size. For the instances in the first group the duration variability is very small or even null depending on memory allocation (i.e., $dmin^{ex}$ and $dmax^{ex}$ are very close or equal, and analogously durations of reading and writing activities are similar).

Table 1. Results on the set of instances where task durations are not strongly influenced by allocation decisions

		CP			TD		
Number of tasks	Number of arcs	time (sec.)	SbB	> TL	time (sec.)	SbB	> TL
15	9-13	0.01	0.01	0	0.31	0.31	0
15	14-26	0.02	0.02	0	0.62	0.62	0
25	30-55	0.10	0.11	0	369.66	369.66	2
25	56-65	0.05	0.05	0	530.96	530.96	2
30	47-71	1.25	0.82	2	620.13	620.13	11
30	73-82	0.12	0.09	0	834.45	834.45	8

A second group of instances has been generated by using the same task graph structure of the first group and by changing randomly the durations of communication activities depending on the allocation choices; we chose to generate 200 instances instead of 90 to increase the reliability of the evaluation. Compared to the previous ones, instances of this second group have a higher variability of minimal and maximal task durations.

The first set of instances is representative of high computational intensive applications in general, like many signal processing kernels. In this scenario the overall task duration is dominated by the data computation section, while the variability induced by different memory allocations is negligible. On the other hand, the second set is representative of more communication intensive applications. In this case, the overall task duration can be drastically affected by different memory allocations. Several video and image processing algorithms are good examples of applications which fit in this category. The Cell configuration we used for the tests has 6 available SPEs.

Results on the first set of instances, where task duration is not much influenced by memory allocation, are reported in table 5. Every row reports results on 15 instances. Each instance is characterized by the number of tasks and a variable number of arcs in the interval reported in the table. We recall that arcs in the task graph represent communications and should be modelled with two communication activities (writing and reading). For each solver the computation time is reported in seconds and is the average execution time on instances solved to optimality (in which case the two approaches yield the same solution quality). In the column SbB the time computation is restricted to instances solved by both methods; finally column > TL reports the number of timed out instances (out of 15). The time limit has been set to 1800 seconds.

As we can see the CP approach achieves significant speed ups with respect to the decomposition approach and the number of timed out instances is significantly smaller in this case. The produced schedules were validated on the same platform used for characterization of the instances.

On the other hand, results on the second set of instances where tasks have high duration variability due to allocation choices are reported in table 2. Every row reports results on 20 instances. Each instance is characterized by the number of tasks (variable in the range reported in table) and the number of arcs. The time is reported in seconds and is the average execution time on instances solved within the time limit; as in the previous table in the column SbB the time computation is restricted to instances solved

Table 2. Result on the set of instances where task durations are strongly influenced by allocation decisions

Number of tasks	Number of arcs	CP			TD		
		time (sec.)	SbB	> TL	time (sec.)	SbB	> TL
10-11	4-11	16.70	16.70	0	3.67	3.67	0
12-13	8-14	116.92	116.92	2	11.19	4.59	0
14-15	8-15	81.50	81.50	8	10.25	7.67	0
16-17	11-17	34.66	34.66	11	29.53	18.17	0
18-19	13-19	66.47	66.47	15	72.56	33.92	1
20-21	16-22	400.41	400.41	16	248.00	82.50	2
22-23	19-26	30.78	30.78	18	355.15	395.00	3
24-25	20-29	—	—	20	200.00	—	9
26-27	23-29	—	—	20	425.00	—	6
28-29	25-35	—	—	20	742.73	—	9

by both approaches. In the column > TL we report the number of timed out instances (out of 20). Also in this case the time limit has been set to 1800 seconds.

As we can see, the performances of the pure CP approach now start decreasing. For the difficult instances (last three rows), all 20 instances have achieved the time limit while the decomposition approach is still able to produce optimal results for half of the instances.

It appears that the CP solver, during the initial PE assignment and scheduling phase, has difficulties in computing good makespan bounds taking into account the impact of memory allocation choices. On the other hand those choices are anticipated, and thus better managed, by the decomposition based solver, at the price of a weakness in exploiting resource constraints to compute makespan bounds. Benders' cuts seem to be a quite robust device to partially overcome the limitations of the decomposition approach: perhaps they could be introduced as well in the CP solver to give to it the ability to handle memory allocation.

These results give a clear indication about the type of solver we have to use depending on the instance structure. If the allocation part is predominant since it greatly influences task durations, the decomposition approach should be used. On the contrary, if choosing resource assignments should respect resource capacity constraints but it does not influence significantly task durations, the pure CP approach greatly outperforms the (more complex) decomposition approach.

6 Conclusions

The work presented in this paper is part of a wider project aimed at developing a software development infrastructure, called Cellflow to help programmers in software implementation on the Cell Broadband Engine processor. Although an off-line development framework and an on-line runtime support are needed in Cellflow, the optimization engine is a fundamental component. We are designing an algorithm portfolio and a selection algorithm based on the instance structure.

Acknowledgement

The work described in this publication was supported by the PREDATOR Project funded by the European Community's 7th Framework Programme, Contract FP7-ICT-216008.

References

1. Policella, N., Cesta, A., Oddi, A., Smith, S.F.: From precedence constraint posting to partial order schedules A CSP approach to Robust Scheduling. *AI Communications* 20(3), 163–180 (2007)
2. Laborie, P.: Complete MCS-Based Search: Application to Resource Constrained Project Scheduling. In: *Proc. of IJCAI 2005*, pp. 181–186 (2005)
3. Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation and scheduling for MPSOCs via decomposition and no-good generation. In: van Beek, P. (ed.) *CP 2005*. LNCS, vol. 3709, pp. 107–121. Springer, Heidelberg (2005)
4. Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation, Scheduling and Voltage Scaling on Energy Aware MPSOCs. In: Beck, J.C., Smith, B.M. (eds.) *CPAIOR 2006*. LNCS, vol. 3990, pp. 44–58. Springer, Heidelberg (2006)
5. Benini, L., Lombardi, M., Mantovani, M., Milano, M., Ruggiero, M.: Multi-stage Benders Decomposition for Optimizing Multicore Architectures. In: Perron, L., Trick, M.A. (eds.) *CPAIOR 2008*. LNCS, vol. 5015, pp. 36–50. Springer, Heidelberg (2008)
6. Bockmayr, A., Piskunov, N.: Detecting infeasibility and generating cuts for MIP using CP. In: *Int. Workshop Integration AI OR Techniques Constraint Programming Combin. Optim. Problems CP-AI-OR 2003*, Montreal, Canada (2003)
7. Grossmann, I.E., Jain, V.: Algorithms for hybrid milp/cp models for a class of optimization problems. *INFORMS Journal on Computing* 13, 258–276 (2001)
8. Hooker, J.N., Ottosson, G.: Logic-based benders decomposition. *Mathematical Programming* 96, 33–60 (2003)
9. Hooker, J.N.: A hybrid method for planning and scheduling. In: Wallace, M. (ed.) *CP 2004*. LNCS, vol. 3258, pp. 305–316. Springer, Heidelberg (2004)
10. Hooker, J.N.: Planning and scheduling to minimize tardiness. In: van Beek, P. (ed.) *CP 2005*. LNCS, vol. 3709, pp. 314–327. Springer, Heidelberg (2005)
11. Sadykov, R., Wolsey, L.A.: Integer Programming and Constraint Programming in Solving a Multimachine Assignment Scheduling Problem with Deadlines and Release Dates. *INFORMS Journal on Computing* 18(2), 209–217 (2006)
12. IBM CELL Broadband Engine software development kit, <http://www.alphaworks.ibm.com/tech/cellsw/download>
13. Laborie, P.: Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Journal of Artificial Intelligence* 143, 151–188 (2003)
14. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: Cellss: a programming model for the cell be architecture. In: *SC 2006: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, p. 86. ACM Press, New York (2006)
15. Chen, T., Raghavan, R., Dale, J., Iwata, E.: Cell broadband engine architecture and its first implementation. In: *IBM White paper* (2005)
16. Chatha, K.S., Vemuri, R.: Hardware-software partitioning and pipelined scheduling of transformative applications, vol. 10, pp. 193–208 (2002)
17. Fohler, G., Ramamritham, K.: Static scheduling of pipelined periodic tasks in distributed real-time systems. In: *Procs. of the 9th EUROMICRO Workshop on Real-Time Systems - EUROMICRO-RTS 1997*, Toledo, Spain, pp. 128–135. IEEE, Los Alamitos (1997)

18. Bakshi, S., Gajski, D.D.: A scheduling and pipelining algorithm for hardware/software systems. In: Proceedings of the 10th international symposium on System synthesis - ISSS 1997, Washington, DC, USA, pp. 113–118. IEEE Computer Society, Los Alamitos (1997)
19. Eichenberger, A., et al.: Optimizing compiler for the cell processor. In: PACT 2005: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, Washington, DC, USA, pp. 161–172. IEEE Computer Society, Los Alamitos (2005)
20. Eichenberger, A.E., et al.: Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. *IBM Syst. J.* 45(1), 59–84 (2006)
21. Axelsson, J.: Architecture synthesis and partitioning of real-time synthesis: a comparison of 3 heuristic search strategies. In: *Procs. of the 5th Intern. Workshop on Hardware/Software Codesign (CODES/CASHE 1997)*, Braunschweig, Germany, pp. 161–166. IEEE, Los Alamitos (1997)
22. Eles, P., Peng, Z., Kuchcinski, K., Doboli, A.: System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems* 2, 5–32 (1997)
23. Kodase, S., Wang, S., Gu, Z., Shin, K.: Improving scalability of task allocation and scheduling in large distributed real-time systems using shared buffers. In: *Procs. of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003)*, Toronto, Canada, pp. 181–188. IEEE, Los Alamitos (2003)
24. Eles, P., Peng, Z., Kuchcinski, K., Doboli, A., Pop, P.: Scheduling of conditional process graphs for the synthesis of embedded systems, Paris, France, pp. 132–139 (1998)
25. Kuchcinski, K., Szymanek, R.: A constructive algorithm for memory-aware task assignment and scheduling. In: *Procs of the Ninth International Symposium on Hardware/Software Codesign - CODES 2001*, Copenhagen, Denmark, pp. 147–152. ACM Press, New York (2001)
26. Kuchcinski, K.: Embedded system synthesis by timing constraint solving. *IEEE Transactions on CAD* 13, 537–551 (1994)
27. Flachs, B., et al.: A streaming processing unit for a cell processor. In: *IEEE International Solid-State Circuits Conference, 2005 (ISSCC 2005)*. Digest of Technical Papers, pp. 134–135 (2005)
28. Hofstee, H.: Cell broadband engine architecture from 20,000 feet. In: *IBM White paper* (2005)
29. Kistler, M., Perrone, M., Petrini, F.: Cell multiprocessor communication network: Built for speed. *IEEE Micro*. 26(3), 10–23 (2006)
30. Maeda, S., Asano, S., Shimada, T., Awazu, K., Tago, H.: A real-time software platform for the cell processor. *IEEE Micro*. 25(5), 20–29 (2005)
31. Palazzari, P., Baldini, L., Coli, M.: Synthesis of pipelined systems for the contemporaneous execution of periodic and aperiodic tasks with hard real-time constraints. In: *18th International Parallel and Distributed Processing Symposium - IPDPS 2004*, pp. 121–128 (2004)
32. Pham, D., et al.: The design and implementation of a first-generation cell processor. In: *IEEE International Solid-State Circuits Conference ISSCC 2005*, vol. 1, pp. 184–592 (2005)
33. Ohara, M., Inoue, H., Sohda, Y., Komatsu, H., Nakatani, T.: MPI microtask for programming the Cell Broadband Engine processor. *IBM System Journal* 45(1) (2006)
34. Zhang, D., Li, Q.J., Rabbah, R., Amarasinghe, S.: A Lightweight Streaming Layer for Multicore Execution. In: *Proceedings of Workshop on Design, Architecture and Simulation of Chip Multi-Processors, dasCMP 2007* (2007)

Planning and Scheduling the Operation of a Very Large Oil Pipeline Network

Arnaldo V. Moura*, Cid C. de Souza, Andre A. Cire, and Tony M.T. Lopes

Institute of Computing - University of Campinas, 13084-971 - Campinas, Brazil
{arnaldo,cid}@ic.unicamp.br, {andre.cire,tony.lopes}@gmail.com

Abstract. Brazilian PETROBRAS is one of the world largest oil companies. Recurrently, it faces a very difficult over-constrained planning challenge: how to operate a large pipeline network in order to adequately transport oil derivatives and biofuels from refineries to local markets. In spite of being more economical and environmentally safer, the use of a complex pipeline network poses serious operational difficulties. The network has a complex topology, with around 30 interconnecting pipelines, over 30 different products in circulation, and about 14 distribution depots which harbor more than 200 tanks, with a combined capacity for storing up to 65 million barrels. The problem is how to schedule individual pumping operations, given the daily production and demand of each product, at each location in the network, over a given time horizon. We describe a solution based on a two-phase problem decomposition strategy. A novel Constraint Programming (CP) model plays a key role in modeling operational constraints that are usually overlooked in literature, but that are essential in order to guarantee viable solutions. The use of CP was crucial, since it allowed the modeling of complex constraints, including nonlinearities. The full strategy was implemented and produced very adequate results when tested over large real instances. In contrast, other approaches known from the literature failed, even when applied to much less complex networks.

1 Introduction

PETROBRAS is ranked as the 14th largest oil company in the world (see www.energyintel.com). One of the major sources of costs faced by PETROBRAS is related to transportation, specially regarding petroleum derivatives, such as gasoline, and biofuel, like ethanol. In this context, pipeline networks are considered the main inland transportation mode in contrast to rail and road, since they are much more economical and environmentally safer.

However, these advantages ensue a very high operational complexity. For instance, the Brazilian pipeline network owned and operated by PETROBRAS has an extension of 7,000 kilometers, comprising 29 individual interconnecting pipelines in which more than 30 different types of products are in circulation. There are 14 distribution depots that can store up to 65 millions barrels of these products,

* Corresponding author.

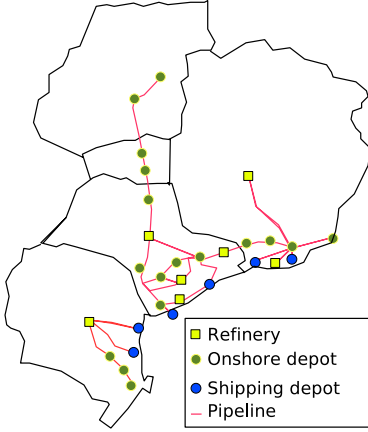


Fig. 1. PETROBRAS pipeline networks

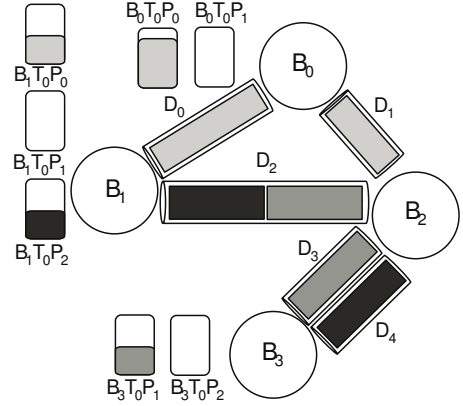


Fig. 2. A pipeline network example

stocked in more than 200 tanks located at such depots. A partial illustration of the Brazilian southeastern network is shown in Figure 1. Pipelines must always be completely filled with products, meaning that a volume must be *pushed* into a duct in order to pump out the same volume at the other extremity. Moreover, due to chemical properties, certain products can not make contact with each other - they are called *incompatibles*. Also, each product has its own flow rate interval, and that depends on the flow direction and on the particular pipeline being used. At depots, not all departing and arriving operations can be simultaneous, due to restrictions imposed both by the internal valve and ducts layout, as well as by the number of local pumps. Tanks can store just one type of product, and extraction or injection of volumes can not be simultaneous.

The problem is how to schedule all individual *pumping operations* in order to fulfill market demands and store all the planned production. Each pumping operation is defined by origin and destination tanks, a pipeline *route*, start and end times, a specific product and its respective volume. The operations must obey all constraints over the given time horizon. The management of all these resources gives rise to a complex planning and scheduling problem.

Currently, the problem is solved manually by executing a trial-and-error process with the aid of a proprietary simulator that checks whether some simple physical constraints are being satisfied. This process is very time consuming and, not rarely, the final results still violate some of the more complex restrictions. Clearly, this manual process is far from optimal and limits the efficiency of the network operation. In fact, it is common for the company to use trucks for transporting pending volumes, thus increasing the overall transportation costs, a situation that could be avoided by a more intelligent use of the pipeline network.

Due to its size and complexity, as well as to its financial impact, the efficient operation of this large oil pipeline network is one of the most strategic problems faced by logistics at PETROBRAS today. As will be discussed later, CP was at

the core of a computational model devised and used to find good operational solutions for real problem instances, in an adequate amount of computer time.

Problem Description. As an illustration, Figure 2 shows a sample network with 4 depots, B_0, B_1, B_2 , and B_3 , interconnected by 5 pipelines. Between depots B_2 and B_3 , there are 2 pipelines, which is common to occur in practice. Each depot also has its own tank farm. For instance, depot B_1 has storage tanks for products P_0, P_1 and P_2 . Each tank contains an initial volume. Ducts must always be completely filled. All of these quantities are measured in standardized units.

The following constraints must be satisfied:

- (1) During the whole planning horizon, a tank can store only a pre-defined product and its capacity must always be respected. But a depot not necessarily contains tanks to store all types of products. All injection and extraction operations in a tank must be disjunctive in time.
- (2) Pipelines operate in an intermittent fashion and must always be completely filled. No interface losses between products are considered. Furthermore, volumes pumped out can either enter a tank or move directly into another pipe in an assigned route. The initial sequence of products inside each pipe is given.
- Flows in pipelines can change direction dynamically, an event called *pipeline flow reversal*. An example of reversal is illustrated in Figure 3 for a single pipeline topology. From instants $t = 0$ to $t = 2$, a product extracted from tank $B_0T_0P_2$ in depot B_0 is being used to push another product into the tank $B_1T_0P_0$ in depot B_1 . As soon as the first product is completely injected into its destination tank at $t = 3$, the second volume must return to the first depot, since there is no tank for it in depot B_1 . This is done by using the product from tank $B_1T_0P_1$ to push it back to the origin tank, changing the pipeline direction at $t = 4$ and $t = 5$.
- (3) Depending on the internal arrangement of a depot, certain operations can not be active simultaneously. Such sets of operations are called *forbidden alignment configurations*. Also, each depot has an upper limit on the number of outgoing pump operations, which depends on the number of available pumps.
- (4) A *route* is an alternating sequence of depots and non-repeating connecting ducts. For example, the sequence $(B_0, D_1, B_2, D_3, B_3)$ represents a valid route in figure 2. Each product in circulation must have a route assigned to it, and a volume can only leave its route at the final destination tank. Although there is no restriction barring the creation of new routes, the most common choices obtained from human experience should be preferred.
- (5) Least maximum flow rates among all products in any route must be enforced.
- (6) To separate two incompatible products, it is possible to use a third product, called a *plug*, compatible with both products it separates.
- (7) Production and demand volumes are defined per depot and per product, each with its own duration interval.

A solution is defined by a set of *pumping operations*. Each such operation is taken as a continuous and atomic pumping stream. An operation is defined by specifying information about the product, volume, route, origin and destination tanks, as well as start and end pumping times. Once a pumping operation starts,

the volume must follow its designated route until it reaches the destination tank. However, a pumping operation can be stopped at any time, as long as no pumped volume (*i.e.* a volume that composes a whole operation) is interleaved with other products at any intermediate depot along its assigned route. The main goal is to find a solution that respects all operational and physical constraints of the network, as well as that uses stocks and productions to satisfy all local demands, while storing away any remaining production.

2 Why CP and Related Work

Previous studies from the literature frequently have focused on more restricted or much smaller network topologies. Usually, they consist of a single pipe connecting one origin (a refinery) to multiple depots. Different problem decompositions together with several MILP formulations [1,2,3,4] were proposed for these cases. Some studies also deal with variable pumping flow rates and other non-linear constraints [5,6]. Other approaches handle multiple origins and destinations within a more realistic network, albeit neglecting most of the hard constraints in order to make the problem tractable [7]. In [8], a MILP based on a network flow model was created to solve a relaxed version of the problem, but it took more than 50 hours of computer time only to find the LP initial basis.

As our research indicated, taking advantage of the problem structure using single MILP models is not practical for two reasons. First, most of the problem restrictions are computationally costly, or even impossible, to model as linear constraints, specially those related with variable flow rates and transmission between pipelines. Besides, MILP models would have to deal with multiple pipelines and depots, and investigation showed that the number of integer variables and constraints would increase at an unacceptable rate. On the other hand, heuristic and meta-heuristic strategies *per se* are greatly impaired when too many operational constraints are considered. This is particularly disturbing when slight modifications in a solution give rise to serious collateral perturbations over the problem structure as a whole. For example, since products can flow directly from one pipeline to another, changing a single pumping start time may delay the arrival of a number of other products that pass through connected pipelines. This can easily render a candidate solution into an infeasible one.

In face of all these issues, the use of CP was seen to offer great advantages for modeling and solving this problem. Firstly, its powerful modeling language allowed for the implementation of operationally crucial constraints, besides providing enough flexibility to extend the model if new restrictions were risen by pipeline operators. Secondly, and most importantly, it was possible to exploit specific problems patterns explicitly. This is done, for instance, by modeling multiple subproblem representations in order to use specialized and adequate constraint propagation mechanisms to solve each of the subproblems. In fact, such multiple perspectives played an important role in the final model, greatly improving domain reductions. Furthermore, a preliminary study [9] already indicated that CP would be flexible and powerful enough to treat the real problem faced

by PETROBRAS. Finally, the use of CP was further fostered by its well-known good performance when treating scheduling problems [10]. In addition, CP is more suitable for our case since any feasible solution is enough.

3 How CP ?

The complete problem was solved using a hybrid approach that combined a randomized constructive heuristic and a novel CP model. The hybridization main cycle is schematically presented in Figure 4. The *planning phase*, implemented as a constructive heuristic, is responsible for creating a set of *delivery orders*. Each such order is defined by a volume, origin and destination tanks, product type, route and a delivery deadline. The planning phase must guarantee that, if all delivery orders are completed within their respective deadlines, local market demands will be fulfilled and the excess production will be correctly stored away. The *scheduling phase* takes the set of delivery orders generated by the planning phase. It must both sequence the pumping operations at the initial pipeline in each route present in a delivery order, as well as determine the start times of each of the pumping operations, while ensuring that no network operational constraint is violated at any time. The scheduling phase represents the problem's central decision process and it was implemented as a CP model. In the sequel, each phase will be discussed, with the CP model described in more detail.

Planning and Routing. To generate delivery orders, we created a randomized constructive heuristic that makes use of the accumulated experience at PETROBRAS. The purpose of the randomization is to generate diversified sets of orders in case the main cycle restarts, increasing the chance of finding solutions. Also, it takes into consideration other criteria that are difficult to handle manually, such as estimating the time for product volumes to arrive at depots.

Delivery orders are created incrementally as follows: **(1)** randomly select a local product demand in any depot, giving higher priority to demands that must be fulfilled earlier in time; **(2)** randomly choose depots that could supply volumes of the required products, as well as the routes that these volumes should traverse. In order to do so, consider factors such as pipeline occupation rate,

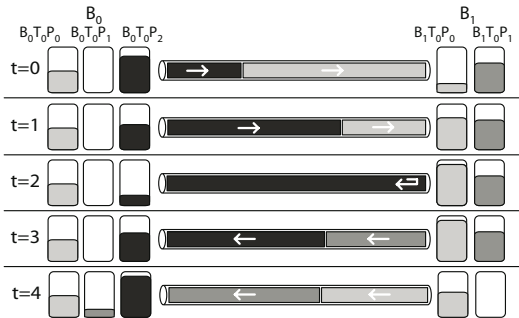


Fig. 3. Example of a flow reversal

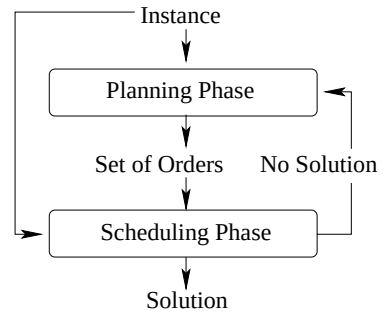


Fig. 4. Solver Framework

production schedules, present product stocks and estimated time of arrivals; **(3)** select origin and destination tanks, setting order volumes accordingly. Also, set order deadlines so as to guarantee demand fulfillment.

As soon as there is no more demands to choose from, the planning phase ends. At this point network operators can interfere adding, modifying, or removing orders according to their particular needs. This flexibility is interesting since sudden needs might unexpectedly arise. For example, there might be exceptional cases where the operators want to empty certain tanks for emergency maintenance purposes. This could be achieved by issuing new orders that remove products from those particular tanks.

All demands are guaranteed to be satisfied if the resulting orders can be scheduled to arrive by their respective deadlines. Of course, at this stage, it is not possible to know if the whole pipeline network can be operated in a way that meets all delivery order deadlines, while satisfying all problem constraints.

Orders are *indivisible*, i.e., once a volume starts to be pumped in, no other pump operation, at that same origin, can be started before the first one completes. However, orders are *preemptive* in the sense that they can be interrupted and be resumed at a later time. For instance, it is possible that a segment of the route that is being used in this pumping must also carry other products, with more pressing deadlines, along another route that has that segment in common. In such cases, it may be necessary to interrupt the present pumping operation, allowing for the more urgent products to circulate in the common pipeline segment, later resuming the first pumping.

Sequencing and Scheduling Orders. The scheduling phase must determine the pumping parameters in order to meet all delivery order deadlines, also taking into account the network operational constraints. Or it must prove that the present set of delivery orders can not be sequenced nor have their start times set in a way that observe all their assigned deadlines. At this point, orders already have their routes, volumes and origin/destination tanks assigned by the planning and routing phase, besides their deadlines.

In a typical scenario comprising 14400 minutes (*i.e.* 10 days), the model is expected to deal with around 900 delivery orders, involving dozens of products, leaving and reaching several tanks, circulating through many interconnected pipelines, and subject to thousands of constraints. In order to cope with this problem complexity, the CP model was further divided into two steps. A first model treats the sequencing of delivery orders, generating *time intervals* for the start of the respective pumping operations. After one such sequencing is completed, the most difficult constraints are guaranteed to be satisfied. Then a second, simpler, CP model takes over and determines the number of pumping operations for each delivery order (*i.e.* taking into account possible preemptions), as well as the start time of each operation.

All time variables represent minutes, the unit currently adopted by network operators. Therefore, all variables have integer domains. Time value roundings, *e.g.* due to some particular combination of flow rate and pipeline extension, can

be safely neglected given the large volumes that are involved. Variable domains are easy to infer from the input data instances and are not further detailed here.

The Sequencing Model. This model must take into account product pair incompatibilities, tank capacities, pipeline flow direction restrictions and other essential operational constraints, such as no two products being pumped into a pipeline simultaneously. Furthermore, it must consider order deadlines and flow rates, in order to determine valid time bounds for the pumping operations.

The model interrelates two different *viewpoints* [11]. Firstly, the *order viewpoint* provides a *global* view of the problem, dealing mainly with routes and volume transmission between pipelines. In contrast, the *operations viewpoint* captures a *local* view of the problem, representing the pumping operation constraints in each pipeline. Both viewpoints are connected by *channeling* constraints.

The Order Viewpoint. The *order viewpoint* handles the problem globally, focusing on the relationship between orders and the pipelines that occur in their assigned routes. It also enforces constraints related to flow rates, deliver deadlines, disjunctions of pipeline operations, product incompatibilities, and tanks.

Let $\mathbf{P}$ be the pipeline set, $\mathbf{T}$ the tank set and $\mathbf{O} = \{o_1, \dots, o_n\}$ the set of delivery orders received from the planning phase. For each $o_i \in \mathbf{O}$, let $\text{route}(o_i) = (p_l, \dots, p_m)$ be the sequence of pipelines that order o_i must traverse. For each $p \in \text{route}(o_i)$, the volume specified by o_i can have one of four possible pipeline flow attributes when traversing pipeline p : N , if it follows the normative, or preferred, pipeline flow direction; R , if it follows the reverse direction; NR , if it starts in the normative direction, but later changes to the reverse direction, thus leaving the pipeline through the same extremity it was pumped into; and RN , similar to NR but starting in the reverse direction. Let variable $\text{direct}_{i,p}$ specify one among such possibilities. Finally, let $\text{origin}(o_i)$, $\text{destin}(o_i) \in \mathbf{T}$ be the origin and destination tanks, respectively, for order o_i .

For each $o_i \in \mathbf{O}$ and $p \in \text{route}(o_i)$, we define two *activities* [12], $\text{snd}_{i,p}$ and $\text{rcv}_{i,p}$, each composed by start and end time variables and an inferred non-negative duration variable. The first activity represents the time interval during which order o_i is being pumped into p , while the second represents the time interval during which the order is being pumped out of p . Using these activities, we can give bounds on flow rates and state delivery deadline constraints for each $o_i \in \mathbf{O}$ and each $p \in \text{route}(o_i)$:

$$\text{EndTime}(\text{rcv}_{i,p}) \leq \text{deadline}(o_i), \quad (1)$$

$$\text{Duration}(\text{snd}_{i,p}).\text{max\_flow\_rate}_{p,\text{direct}_{i,p}} \geq \text{volume}(o_i), \quad (2)$$

$$\text{Duration}(\text{rcv}_{i,p}).\text{max\_flow\_rate}_{p,\text{direct}_{i,p}} \geq \text{volume}(o_i). \quad (3)$$

Before an order exits a pipeline, it must first traverse all the pipeline extension. Thus, for each $o_i \in \mathbf{O}$ and $p \in \text{route}(o_i)$, we require:

$$\text{StartTime}(\text{rcv}_{i,p}) \geq \text{StartTime}(\text{snd}_{i,p}) + \left\lfloor \frac{\text{volume}(p)}{\text{max\_flow\_rate}_{p,\text{direct}_{i,p}}} \right\rfloor. \quad (4)$$

When an order is being pumped out of a pipeline, it is immediately pumped into the next pipeline in its route, without volume loss. This can be done by

unifying [13] send and receive activity variables in the following way. For each $o_i \in \mathbf{O}$ and for each pair (p_l, p_m) of consecutive pipeline pairs in $\text{route}(o_i)$, let:

$$\text{StartTime}(\text{rcv}_{i,p_l}) = \text{StartTime}(\text{snd}_{i,p_m}), \quad (5)$$

$$\text{EndTime}(\text{rcv}_{i,p_l}) = \text{EndTime}(\text{snd}_{i,p_m}). \quad (6)$$

Order activities in a pipeline must all be *disjunctive* with respect to time; a send (or receive) activity from a certain order must not overlap with the send (or receive) activity of other orders in the pipe. In order to guarantee this, for each $p \in \mathbf{P}$, we define two *unary resources*¹: SndResource_p and RcvResource_p , and we associate the send and receive activities to these resources, respectively. Since an unary resource defines a mutually exclusive relationship between activities that are linked to it, each resource constraint is ranked during the solving process, *i.e.*, it is ordered along the time line. This ranking is explicitly represented in our model using positional variables $\text{sndPos}_{i,p}$ and $\text{rcvPos}_{i,p}$, accounting for, respectively, the send and receive activities positions of order o_i in pipeline $p \in \text{route}(o_i)$. The positional variables are connected directly to the resource's *precedence graph* [10,12], expressed by the constraints:

$$\text{snd}_{i,p} \text{ startsBefore } \text{snd}_{j,p} \iff \text{sndPos}_{i,p} < \text{sndPos}_{j,p}, \quad (7)$$

$$\text{rcv}_{i,p} \text{ startsBefore } \text{rcv}_{j,p} \iff \text{rcvPos}_{i,p} < \text{rcvPos}_{j,p}, \quad (8)$$

$$\forall o_i, o_j \in \mathbf{O}, \forall p \in \text{route}(o_i) \cap \text{route}(o_j).$$

We also add redundant *all different* global constraints [13]. For each $p \in \mathbf{P}$,

$$\text{all\_diff}(\text{sndPos}_{i,p}) \wedge \text{all\_diff}(\text{rcvPos}_{i,p}), \forall o_i \in \mathbf{O} \text{ s.t. } p \in \text{route}(o_i). \quad (9)$$

In case two orders $o_i, o_j \in \mathbf{O}$ share at least one common consecutive pipeline pair $(p_l, p_m) \in \text{route}(o_i) \cap \text{route}(o_j)$, the activities precedence relations must be preserved in both pipelines. Here, we present the restrictions for flow directions N and R , the other cases being similar.

$$\begin{aligned} \text{sndPos}_{i,p_l} > \text{sndPos}_{j,p_l} &\iff \text{sndPos}_{i,p_m} > \text{sndPos}_{j,p_m} \\ \wedge \text{rcvPos}_{i,p_l} > \text{rcvPos}_{j,p_l} &\iff \text{rcvPos}_{i,p_m} > \text{rcvPos}_{j,p_m}, \\ &\forall o_i, o_j \in \mathbf{O}, \forall (p_l, p_m) \in \text{route}(o_i) \cap \text{route}(o_j). \end{aligned} \quad (10)$$

Positional variables also help discarding sequences that violate product incompatibilities. Given two orders $o_i, o_j \in \mathbf{O}$, if $\text{product}(o_i)$ and $\text{product}(o_j)$ are incompatible, then they can not make contact in a pipeline. A necessary condition for contact is that both orders enter consecutively at the same pipeline extremity, and this can only happen if they have the same entering (or leaving) pipeline flow direction. This scenario is represented by the following constraints, for each $o_i, o_j \in \mathbf{O}$, $p \in \text{route}(o_i) \cap \text{route}(o_j)$ and $\text{product}(o_i)$ **incompatible** $\text{product}(o_j)$.

$$|\text{sndPos}_{i,p} - \text{sndPos}_{j,p}| > 1 \text{ if } (\text{direct}_{i,p} \neq N \vee \text{direct}_{j,p} \neq RN) \quad (11)$$

$$\wedge (\text{direct}_{i,p} \neq R \vee \text{direct}_{j,p} \neq NR),$$

$$|\text{rcvPos}_{i,p} - \text{rcvPos}_{j,p}| > 1 \text{ if } (\text{direct}_{i,p} \neq N \vee \text{direct}_{j,p} \neq NR) \quad (12)$$

$$\wedge (\text{direct}_{i,p} \neq R \vee \text{direct}_{j,p} \neq RN).$$

¹ An *unary resource* is a resource that allows for only one activity at a time [12].

Next, we define two new activities: $ext_{i, \text{origin}(o_i)}$ and $inj_{i, \text{destin}(o_i)}$, representing volume extraction and injection, respectively, from the assigned tanks associated with order o_i . The relationship between send (receive) variables and tanks activities is the same as those for pipeline volume transmissions. For each $o_i \in \mathbf{O}$, letting p_0 and p_m be the first and last pipeline in $\text{route}(o_i)$, we state:

$$\text{StartTime}(\text{snd}_{i, p_0}) = \text{StartTime}(ext_{i, \text{origin}(o_i)}), \quad (13)$$

$$\text{EndTime}(\text{snd}_{i, p_0}) = \text{EndTime}(ext_{i, \text{origin}(o_i)}), \quad (14)$$

$$\text{StartTime}(\text{rcv}_{i, p_m}) = \text{StartTime}(inj_{i, \text{destin}(o_i)}), \quad (15)$$

$$\text{EndTime}(\text{rcv}_{i, p_m}) = \text{EndTime}(inj_{i, \text{destin}(o_i)}). \quad (16)$$

Injecting and extracting volumes from tanks must not overlap in time as well. Hence, activities $ext_{i, t}$ and $inj_{i, t}$ are associated with a new unary resource $TkDisj_t$, created for each tank $t \in \mathbf{T}$. However, capacities must also be taken into account in this case, requiring the combined use of a different type of resource $TkRes_t$, $t \in \mathbf{T}$, called a *reservoir* [12]. Such activities can both increase capacity (volume injection) or deplete capacity (volume extraction) from reservoirs.

Finally, we must also consider production and demand volumes in order to appropriately represent the behavior of tank capacities. Let $\mathbf{Dem}$ and $\mathbf{Pr}$ be, respectively, the sets of demands and productions given as input. For each $d \in \mathbf{Dem}$, an activity dem_d is created, with its associated constraints, considering demand time bounds and volume extraction from tanks. Similarly, an activity $prod_p$ is created for each $p \in \mathbf{Pr}$, but now considering volume injection instead of extraction. These activities are associated with the unary resources $TkDisj_t$ and reservoirs $TkRes_t$. Additional constraints are stated as follows.

$$\text{StartTime}(dem_d) \geq \text{DemandMinStartTime}(d), \quad (17)$$

$$\text{EndTime}(dem_d) \leq \text{DemandMaxEndTime}(d), \quad \forall d \in \mathbf{Dem}, \quad (18)$$

$$\text{StartTime}(prod_p) \geq \text{ProductMinStartTime}(p), \quad (19)$$

$$\text{EndTime}(prod_p) \leq \text{ProductMaxEndTime}(p), \quad \forall p \in \mathbf{Pr}. \quad (20)$$

The Operations Viewpoint. The main intuition for this viewpoint is to consider each pipeline individually (a *local* vision), since time variables domains will be automatically propagated by force of constraints defined in the order viewpoint. Although time bounds and disjunctions were already established, it is still necessary to model the fact that, in order for a certain volume to leave a pipeline, the exact amount of volume must be pumped in from the other extremity. Besides that, restrictions such as variable flow rates which depend on the products inside a pipeline, must also be considered. We will also use some ideas from previous studies [2,5,6], that treated the case of a single pipeline.

Given a pipeline $p \in P$, two time-ordered sets of *operation activities* are defined, SndPipe_p and RcvPipe_p , where $|\text{SndPipe}_p| = |\text{RcvPipe}_p| = |\{o_i : o_i \in \mathbf{O}, p \in \text{route}(o_i)\}|$. As in the order viewpoint, they represent send and receive activities in p , respectively, but now with new precedence relations of the form

$$i < j \iff \text{sndOp}_{p,i} \text{ startsBefore } \text{sndOp}_{p,j}, \quad \forall \text{sndOp}_{p,i}, \text{sndOp}_{p,j} \in \text{SndPipe}_p,$$

$$i < j \iff \text{rcvOp}_{p,i} \text{ startsBefore } \text{rcvOp}_{p,j}, \quad \forall \text{rcvOp}_{p,i}, \text{rcvOp}_{p,j} \in \text{RcvPipe}_p.$$

A *volume* and a *product* variables are additionally associated with each activity belonging to $SndPipe_p$ and $RcvPipe_p$. We thus say that both sequences represent a valid ranking of *undetermined* delivery orders; they will only be *determined* when orders are ranked in their unary resources. However, since they are already time-ordered, we are able to create a more intuitive and compact model to represent pipeline flow behavior, in which constraints will also enforce propagation in the order viewpoint variable domains.

Let $rcvOp_{p,j} \in RcvPipe_p$ be an activity. A certain volume associated with it can only be received when an activity $sndOp_{p,i} \in SndPipe_p$ is being pumped at the other extremity of the pipe. In order to define which send activity i pushes a receive activity j , it is necessary to consider three factors: the pipeline volume, the volumes of the activities and the volumes between activities $sndOp_{p,i}$ and $rcvOp_{p,j}$, *i.e.*, the volume still in the pipeline before sending $sndOp_{p,i}$ and after receiving $rcvOp_{p,j}$. For the latter, a new variable $acc_{p,i,j}$ is created for each $sndOp_{p,i} \in SndPipe_p$, $rcvOp_{p,j} \in RcvPipe_p$, for $i \leq j$, as follows:

$$acc_{p,i,j} = \sum_{k < i} volume(sndOp_{p,k}) - \sum_{k \leq j} volume(rcvOp_{p,k}). \quad (21)$$

It can be shown that $sndOp_{p,i}$ never pushes $rcvOp_{p,j}$ off the pipeline, for $i > j$, due to the time-ordering of the relation. If $acc_{p,i,j} \geq volume(p)$, then it is not possible for $sndOp_{p,i}$ to push $rcvOp_{p,j}$, since a quantity greater or equal than the pipeline volume was already injected between activities i and j . On the other hand, if $acc_{p,i,j} + volume(sndOp_{p,i}) + volume(rcvOp_{p,j}) \leq volume(p)$, then the volume in $sndOp_{p,i}$ is not enough to push $rcvOp_{p,j}$ out of the pipeline. Thus, a necessary and sufficient condition for activity i to push activity j out of the pipeline (a **push** _{i,j} event), can be stated as:

$$\begin{aligned} \text{push}_{i,j} &\iff acc_{p,i,j} < volume(p) \\ &\wedge acc_{p,i,j} + volume(sndOp_{p,i}) + volume(rcvOp_{p,j}) > volume(p). \end{aligned} \quad (22)$$

Similar ideas can be used to determine the exact amount of volume involved in the pumping. Let $flow_{i,j}$ be the volume used in $sndOp_{p,i}$ to push the same volume $flow_{i,j}$ from $rcvOp_{p,j}$ out of the pipeline. We have:

$$\begin{aligned} \text{push}_{i,j} \implies flow_{i,j} = &\min[volume(rcvOp_{p,j}), volume(rcvOp_{p,j}) \\ &+ volume(sndOp_{p,i}) + acc_{p,i,j} - volume(p)] \\ &- \max[0, volume(rcvOp_{p,j}) + acc_{p,i,j} - volume(p)], \end{aligned} \quad (23)$$

$$\neg \text{push}_{i,j} \implies flow_{i,j} = 0. \quad (24)$$

These flow variables can be seen as flow edges in a capacitated network. Assuming that each operation activity represents a node, the amount of volume (flow) pushed into a pipe must be equal to the volume pushed out it, and both are equal to the operation's total volume. To model this restriction, we can use a **flow** global constraint [10] for send and receive variable sequences.

In order to ensure flow rate bounds consistency, it is necessary to limit the flow rate of send and receive activities according to the products that are inside the

pipeline when the pumping activity occurs. This can now be easily done using the earlier condition $acc_{p,i,j} + volume(sndOp_{p,i}) + volume(rcvOp_{p,j}) \leq volume(p)$, which is true if $sndOp_{p,i}$ and $rcvOp_{p,j}$ are both inside the pipeline at the moment of pumping. Let $MaxFl_i$ be a variable representing the maximum flow rate for activity $sndOp_{p,i}$, related to the variable $product(sndOp_{p,i})$, and let $MaxFl_j$ stand similarly for activity $rcvOp_{p,j}$. We state:

$$\begin{aligned} & acc_{p,i,j} + volume(sndOp_{p,i}) + volume(rcvOp_{p,j}) \leq volume(p) \quad (25) \\ \Rightarrow & \left[\frac{volume(sndOp_{p,i})}{(EndTime(sndOp_{p,i}) - StartTime(sndOp_{p,i}))} \right] \leq MaxFl_j \\ \wedge & \left[\frac{volume(rcvOp_{p,j})}{(EndTime(rcvOp_{p,j}) - StartTime(rcvOp_{p,j}))} \right] \leq MaxFl_i. \end{aligned}$$

Finally, flow directions in the pipeline must be consistent as well. For instance, if an activity has its direction attribute set to N , the next activity along the pipe must necessarily have its direction attributes set to N or NR . Direction attributes such as R and RN are only consistent after a sequence of NR activities whose volume sum is equal to the pipeline volume. Attribute RN is treated similarly. These valid pairs are enforced using a *Table Constraint* [12].

For the pipeline reversal, a special constraint **reversal** was created, encapsulating the rules for the reversal of flow direction. This global constraints also controls the relation between the sequences $sndPos_{i,p}$ and $rcvPos_{i,p}$, since orders do not enter and live a pipe in the same order when there is a flow reversal, as showed in figure 3. If there is no flow reversal in a pipe p , then a constraint $sndPos_{i,p} = rcvPos_{i,p}$ is added to the model.

The Channeling Constraints. The order and operation viewpoints can be easily connected using the *element* constraint [13] and positional variables. Notice that a similar set of constraints is applied to the receive sequence.

$$StartTime(snd_{i,p}) = StartTime(sndOp_{sndPos_{i,p}, p}), \quad (26)$$

$$EndTime(snd_{i,p}) = EndTime(sndOp_{sndPos_{i,p}, p}), \quad (27)$$

$$volume(o_i) = Volume(sndOp_{sndPos_{i,p}, p}), \quad (28)$$

$$direct(o_i) = Direct(sndOp_{sndPos_{i,p}, p}), \quad (29)$$

$$product(o_i) = Product(sndOp_{sndPos_{i,p}, p}), \quad (30)$$

$$\forall o_i \in \mathbf{O}, p \in route(o_i).$$

The Scheduling Model. The second part of the complete CP model is a simpler model which is responsible for assigning the exact times to pumping operations, respecting forbidden alignment configurations and avoiding simultaneous pipe usage. The pumping operations are created by checking the $flow_{p,i,j}$ variables values for each activity i and pipeline p . If $flow_{p,i,j} > 0$, for a certain j , then there is a pumping operation of volume $flow_{p,i,j}$ with flow rate and time bounds already established by the sequencing step. In that case, a new activity, $pumpOp$, is created and its time constraints are included in the model. Note that the precedence among activities can be inferred from the orders' sequence.

Let **Dep** be the set of depots, and let $PumpOps_d$ give the pumping operations that will start at depot d , for each $d \in \mathbf{Dep}$. The simultaneous sending constraint can be implemented using a *discrete resource*, $DiscSending_d$, a resource which limits the number of consecutive operations by a certain capacity [12]. Thus, we associate each operation in $PumpOps_d$ in its respective resource $DiscSending_d$, limited by the input $DepotMaxSimultaneousOperations_d$. Similarly, the forbidden alignment configurations are enforced with discrete resources $AlignDisc_{a,d}$, created for each alignment restriction a . The operations associated with each resource are easily identifiable by checking their product type and flow direction.

Free Delivery Orders. In certain scenarios, the orders created by the planning phase are not enough to guarantee a valid pumping solution. For instance, suppose that only two orders need to be scheduled, and they have incompatible products. Consequently, one can not push the other in a pipeline. A third product must be used between them. For that, *free delivery orders* are arbitrarily created before entering the scheduling phase. In contrast to regular orders, their volumes, products, and origin/destination tanks are treated as variables instead of constants, and they do not have a deadline. Note also that free orders may have a null volume associated with them. Furthermore, their routes are previously determined by choosing among the ones typically used by pipeline operators. Operators can change such routes by editing a configuration file.

Free orders are also used to represent products that remain in the pipeline at the end of the process, for the purpose of ensuring that all pipes are always completely filled. These orders do not have a destination tank, and constraints are used to indicate they are the last ones to be pumped into the pipelines.

Only minor changes to the previous model are necessary in order to accommodate free orders. Among them, in the order viewpoint, constraints where volume and product were constants should be changed to variables, and an *Alternative Resource Set* [12] can be used to indicate that an origin and destination tank must be assigned to free orders. The operation viewpoint remains unchanged.

Search Strategy. Different types of search strategies were tested for solving both the sequencing and the scheduling models. The currently implemented version is shown as Algorithm 1. It combines a *backtracking* mechanism [13] with a special variable ordering, being divided into three consecutive parts: *disjunctive components determination*, *adaptive backtracking* and *time assignment*. In the *Disjunctive Components Determination*, a disjunctive component is defined as a subset of the network which can be scheduled separately, without affecting other regions. Two pipelines belong to the same component if they are both contained in at least one order's route. The same reasoning applies to tanks.

For the *Adaptive Backtracking*, we implemented backtracking using positional variables for each pipeline. The term *adaptive* comes from the fact that it is based on a *restart* strategy [14]. As such, the pipeline sequencing order is changed dynamically according to the number of fails that occurred during the search. The values of positional variables are randomly chosen, giving higher probabilities to orders with the earliest deadlines. For free orders, volumes, products and tanks are set after their respective positional variables are labeled.

Algorithm 1. Procedure for search strategy

```

1 begin
2   Identify network disjunctive components  $C$ 
3   for each  $c \in C$  do
4     Build pipe graph  $G(c)$  and sort it topologically, obtaining order  $N$ 
5      $N' := N$ ;  $k := \text{initial\_}k$ 
6     while  $N' \neq \emptyset$  do
7        $p :=$  first element from sequence  $N'$ ;  $N' := N' / \{p\}$ 
8       Label positional variables, and volumes/tanks in case of free orders
9       if fails in labeling  $\geq k$  and not cyclic condition then
10         $k := k + \text{incremental factor}$ ;  $N' := N$ 
11        Move  $p$  to the beginning of sequence  $N'$ 
12   while no scheduling solution found do
13     Create scheduling model and assign times as earliest as possible
14     if no solution then request next sequencing solution
15 end

```

The initial sequencing is constructed as follows. Firstly a *pipe graph* is created, in which pipelines are nodes and there is a direct arc from node p to q if there is a consecutive pair (p, q) in some order's route. In case there are two arcs (p, q) and (q, p) , only the one associated with the order having the earliest deadline is maintained. Secondly, the graph is topologically sorted, the result being the desired initial sequencing. Clearly, this strategy considers first those pipelines with the least number of orders that come directly from other pipelines.

After the occurrence of k fails involving a pipeline, the backtrack tree is reinitialized with that pipeline as the first element in the topological ordering, and k is incremented by a constant. This implementation was motivated by the fact that, during test runs, it was observed that a fair number of fails were caused by earlier decisions taken when instantiating variables in related pipelines in the given sequencing. We empirically determined $k = 150$ and 100 as the increment.

Finally, in *Time Assignment*, executed after the sequencing is completed, the CP scheduling model is created and the time variables are instantiated with the least possible value in their domains. This forces pumping to start as soon as possible. In case a failure ensues, a new sequencing solution is requested, most certainly a different one due to the randomization present in the model.

4 Results

Solutions were obtained on a *Intel Pentium D* 3.40 Ghz CPU platform, with 4GB of memory. The planning and scheduling phases were coded in C++ and compiled using *GCC-4.0*. The CP model was solved using *ILOG Solver 6.2* and *ILOG Scheduler 6.2*, with medium to high propagation enforcement. Part of a typical solution is presented in Table 1. As described earlier, a solution is a sequence of pump operations and each line in the table describes one such

Table 1. Solution Example

T_i	T_f	Vol.	Pd	Tk _{Or}	Tk _{Dt}	Route
2075	2362	858	<i>G</i>	T004	T005	SUG03
4857	4868	30	<i>N</i>	T160	T087	GUG03
4870	5111	722	<i>D</i>	T008	T005	BUG03
...	...	...	...	...	...	...

Table 2. Solver Results

<i>Instance</i>	1	2	3	4
Horizon	10 days	7 days	7 days	7 days
Orders	924	645	724	693
Planning Phase Time	4 min	5 min	4 min	6 min
Planning Phase Peak Memory	78MB	61MB	67MB	63MB
Sequencing Model Variables	37,326	21,381	25,938	24,315
Sequencing Model Constraints	382,565	148,075	160,302	155,409
Sequencing Choice Points	3,355	2,462	3,417	2,518
Sequencing Fails	2,301	1,291	987	1,902
Sequencing Time	2 min	1 min	1 min	1 min
Sequencing Peak Memory	450 MB	240 MB	310 MB	270 MB
Scheduling Model Variables	12,350	7,530	8,931	8,032
Scheduling Model Constraints	27,088	16,768	19,231	18,292
Scheduling Choice Points	1,516	1,164	801	1,810
Scheduling Fails	301	429	210	120
Scheduling Time	2 min	1 min	1 min	1 min
Scheduling Peak Memory	450 MB	250 MB	290 MB	280 MB
Total Time	8 min	7 min	6 min	8 min

operation. Column headings are as follows: T_i and T_f are the start and end times (in minutes), respectively; Vol is the pumped volume (in m^3); Pd is the product code (*G* is gasoline, *N* is naphtha and *D* is diesel); Tk_{Or} and Tk_{Dt} are origin and destination tank codes, respectively; and $Route$ is the route code. A full solution table would contain several hundred such lines.

We used four real field instances to test the models. The first two rows in Table 2 indicate the time horizon and the number of deliver orders generated by the planning phase, respectively, for each of the test instances. The remaining lines give details of typical runs. All instances share the same network topology of 14 depots, 29 pipelines, 32 different product types and 242 tanks distributed among the depots. Pipelines volumes range from 30 to 8,000 m^3 , and most of the tank capacities are between 4,000 and 30,000 m^3 .

In all cases the solver found a solution in a reasonable amount of computer time, *e.g.*, within 10 minutes. Most variables were instantiated as a result of constraint propagation. The search heuristic, which proved crucial in the planning phase, was also instrumental to improve other important aspects of the solution quality, as noticed by logistic engineers. For instance, usually, a typical solution showed only a very small number of pipeline flow reversions, the kind of operation that engineers prefer to keep to a minimum. Also, new and interesting routes were identified. Some of them came as a surprise to logistic engineers, who

were biased towards the same traditional routes they were using when manually planning the network operation.

5 Added Value and Conclusions

We proposed a novel procedure for generating feasible solutions for real instances stemming from planning and scheduling the operation of a very-large pipeline network used to move petroleum derivatives. The operation of such a network is subject to a complex set of physical and operational constraints, and it makes possible the delivery of oil and biofuel to local markets, as well as the storing of the excess production from refineries. Using the CP paradigm, these constraints were adequately modeled. Problems of this size and complexity, as known by the authors, would not be solved by other approaches reported in the literature to date, in which much of the difficult constraints and topologies are overlooked.

The procedure is already integrated with a proprietary flow simulation tool and the company is currently considering it for routine use on a daily basis. The tool has already proved its value, showing that it can save many valuable work hours of skilled engineers. Also, using the tool, many different planning and scheduling scenarios can be easily setup and quickly tested, by varying local demand needs and production schedules at refineries.

The present modeling and implementation stage was reached after 2 years of problem specification, data gathering, model development, and testing. As work progresses, it is expected that new constraints will be introduced. Such could include inventory management restrictions, limitations on energy use at specific time intervals and at specific depots, and shutdown periods or partial operation intervals for tanks and pipelines. When modeling such new constraints, we feel that the flexibility of the CP paradigm will again prove to be crucial.

As for future directions, one could implement more sophisticated search heuristics for both the planning and scheduling phases, making the overall approach capable of dealing with more specific instance classes. Finally, one could consider objective functions that would help guide the heuristics. This would provide a yardstick that could be used to gauge solution quality.

Acknowledgments. This research was supported by grants 05/57343-0 and 05/57344-7 from FAPESP and grants 301732/07-8, 478470/06-1, 472504/07-0, and 473726/07-6, 305781/2005-7 from CNPq. The authors are also grateful to Fernando Marcellino and the team of engineers from PETROBRAS-TI/SP.

References

1. Rejowski, R., Pinto, J.M.: Efficient MILP formulations and valid cuts for multi-product pipeline scheduling. *Comput. Chem. Eng.* 28(8), 1511–1528 (2004)
2. Cafaro, D.C., Cerdá, J.: Optimal scheduling of multiproduct pipeline systems using a non-discrete MILP formulation. *Comput. Chem. Eng.* 28(10), 2053–2058 (2004)

3. Relvas, S., Barbosa-Póvoa, A.P.F.D., Matos, H.A., Fialho, J., Pinheiro, A.S.: Pipeline scheduling and distribution centre management - a real-world scenario at clc. In: 16th Eur. Symp. Comput. Aided Process Eng. and 9th Int. Symp. Process Syst. Eng., pp. 2135–2140. Garmisch-Partenkirchen, Germany (2006)
4. Relvas, S., Matos, H.A., Barbosa-Póvoa, A.P.F.D., Fialho, J., Pinheiro, A.S.: Pipeline scheduling and inventory management of a multiproduct distribution oil system. *Ind. Eng. Chem. Res.* 45(23), 7841–7855 (2006)
5. Rejowski, R., Pinto, J.M.: A novel continuous time representation for the scheduling of pipeline systems with pumping yield rate constraints. *Comput. Chem. Eng.* 32, 1042–1066 (2008)
6. Relvas, S., Matos, H.A., Barbosa-Póvoa, A.P.F.D., Fialho, J.: Reactive scheduling framework for a multiproduct pipeline with inventory management. *Ind. Eng. Chem. Res.* 46(17), 5659–5672 (2007)
7. Crane, D.S., Wainwright, R.L., Schoenfeld, D.A.: Scheduling of multi-product fungible liquid pipelines using genetic algorithms. In: *Proc. of the 1999 ACM Symposium on Applied Computing*, San Antonio, USA, pp. 280–285 (1999)
8. Camponogara, E.: A-Teams para um problema de transporte de derivados de petróleo. Master's thesis, Instituto de Matemática, Estatística e Ciência da Computação, Universidade Estadual de Campinas, Campinas, Brazil (1995)
9. Moura, A.V., de Souza, C.C., Cire, A.A., Lopes, T.M.T.: Heuristics and constraint programming hybridizations for a real pipeline planning and scheduling problem. In: *Proc. IEEE 11th Int. Conf. Comput. Sci. Eng.*, São Paulo, Brazil (to appear, 2008)
10. Hooker, J.N.: *Integrated Methods for Optimization* (International Series in Operations Research & Management Science). Springer, Secaucus (2006)
11. Cheng, B.M.W., Choi, K.M.F., Lee, J.H.M., Wu, J.C.K.: Increasing constraint propagation by redundant modeling: an experience report. *Constraints* 4(2), 167–192 (1999)
12. ILOG: *ILOG Scheduler 6.2: User's Manual* (2006)
13. Marriot, K., Stuckey, P.: *Programming with Constraints: An Introduction*, 1st edn. MIT Press, Cambridge (1998)
14. Kautz, H., Horvitz, E., Ruan, Y., Gomes, C., Selman, B.: Dynamic restarts policies. In: *Proc. of the 18th Nat. Conf. on Artif. Intell.*, Edmonton, Alberta (July 2002)

Search Strategies for Rectangle Packing

Helmut Simonis and Barry O'Sullivan

Cork Constraint Computation Centre

Department of Computer Science, University College Cork, Ireland

{h.simonis, b.osullivan}@4c.ucc.ie

Abstract. Rectangle (square) packing problems involve packing all squares with sizes 1×1 to $n \times n$ into the minimum area enclosing rectangle (respectively, square). Rectangle packing is a variant of an important problem in a variety of real-world settings. For example, in electronic design automation, the packing of blocks into a circuit layout is essentially a rectangle packing problem. Rectangle packing problems are also motivated by applications in scheduling. In this paper we demonstrate that an “off-the-shelf” constraint programming system, SICStus Prolog, outperforms recently developed ad-hoc approaches by over three orders of magnitude. We adopt the standard CP model for these problems, and study a variety of search strategies and improvements to solve large rectangle packing problems. As well as being over three orders of magnitude faster than the current state-of-the-art, we close eight open problems: two rectangle packing problems and six square packing problems. Our approach has other advantages over the state-of-the-art, such as being trivially modifiable to exploit multi-core computing platforms to parallelise search, although we use only a single-core in our experiments. We argue that rectangle packing is a domain where constraint programming significantly outperforms hand-crafted ad-hoc systems developed for this problem. This provides the CP community with a convincing success story.

1 Introduction

Rectangle (square) packing problems involve packing all squares with sizes 1×1 to $n \times n$ into an enclosing rectangle (square) of minimum area. Rectangle packing is an important problem in a variety of real-world settings. For example, in electronic design automation, the packing of blocks into a circuit layout is essentially a rectangle packing problem [12, 14]. Rectangle packing problems are also motivated by applications in scheduling [10, 11, 13]. Rectangle packing is an important application domain for constraint programming, with significant research into improved constraint propagation methods reported in the literature [1, 2, 3, 4, 5, 6, 7, 15].

The *objective of this paper* was to demonstrate that a current “off-the-shelf” constraint programming system, in our case SICStus Prolog [8], is competitive with the hand-crafted ad-hoc solutions to rectangle packing that have been reported in the literature. Our *methodology* was to consider the standard formulation of the rectangle packing problem, and study the performance of SICStus Prolog using several appropriate search strategies that we explore in this paper. We have developed no new constraint programming technology, such as ad-hoc global constraints, restricting ourselves

entirely to the facilities provided by the standard solver. The surprising, and extremely encouraging, result is that rather than being simply competitive on this problem class, SICStus Prolog outperforms recently developed ad-hoc approaches [10, 11, 13] by over three orders of magnitude. In addition, we close eight open problems in this area: two rectangle packing problems and six square packing problems. Therefore, we claim that rectangle (square) packing provides the CP community with a convincing success story.

We consider rectangle packing to be a more attractive benchmark for general placement problems than the perfect square placement problems considered in [1, 2, 3, 4, 5, 6, 7, 15] for several reasons. Firstly, the perfect square placement problem contains no wasted space (slack), a situation rarely found in practical problems. It is tempting to improve the reasoning for this special case [4], while most practical problems obtain little benefit from such reasoning. Secondly, by providing a single parameter n , it is easy to create increasingly more complex problems. Note though, that problem complexity does not necessarily increase directly with problem size, as the amount of unused space varies with n . Thirdly, for the specific case of rectangle packing, we may choose to solve the problem by testing different combinations of the width and height of candidate rectangles, each with different slack values. This nicely tests the generality of a search method. Finally, for some candidate rectangle sizes, there is no solution that packs all n rectangles into the candidate solution, although the simple lower bounds on required area are satisfied. This means that the proof of optimality for these cases is non-trivial, and may require significant enumeration.

Our future work is to develop a fully constraint-based solution to circuit placement and routing where we pack the blocks of a circuit into a bounding rectangle such that the linear sum of the rectangle area and the total length of wiring is minimised. This is an extremely important problem in electronic design automation [14].

2 Constraint Programming Model

We use the established constraint model [2, 6] for the rectangle packing problem. Each item to be placed is defined by domain variables X and Y for the origin in the x and y

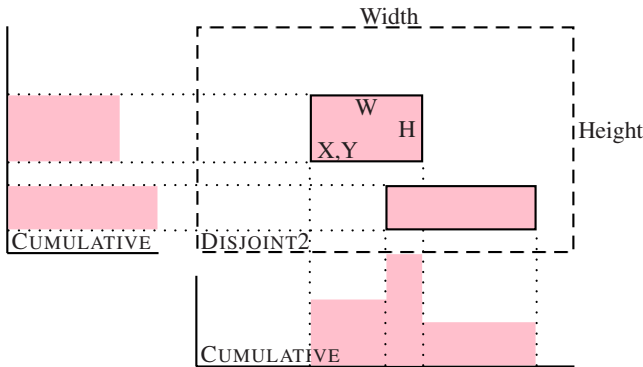


Fig. 1. The basic constraint programming model

dimension respectively, and two integer constants W and H for the width and the height of the rectangle, respectively. In the particular case of packing squares, W and H are identical. The constraints are expressed by a non-overlapping constraint in two dimensions and two (redundant) CUMULATIVE constraints that work on the projection of the packing problem in x or y direction. This is illustrated by Figure 1. We use SICStus Prolog 4.0.2, which provides both CUMULATIVE [1] and DISJOINT2 [3] constraints.

2.1 Problem Decomposition

To find the enclosing rectangle with smallest area, we need a decomposition strategy that generates sub-problems with fixed enclosing rectangle sizes. We enumerate on demand all pairs $Width, Height$ in order of increasing area $Width \times Height$ that satisfy

$$\begin{aligned}
 [Width, Height] &:: n.. \infty, Width \geq Height \\
 \sum_{i=1}^n i^2 &\leq Width * Height \\
 k = \left\lfloor \frac{Height + 1}{2} \right\rfloor, Width &\geq \sum_{j=k}^n j
 \end{aligned} \tag{1}$$

Equation 1 provides a simple bound on the required area, considering all large squares that cannot be stacked on top of each other, which, thus, must fit horizontally. For solutions with the same area, we try them by increasing $Height$, i.e. for two solutions with the same surface we try the “less square-like” solution first. We then solve the rectangle packing problem for each such rectangle in turn, until we find the first feasible solution. By construction, this is an optimal solution. The number of candidates seems to grow linearly with the amount of slack allowed.

Figure 2 shows possible candidate rectangles for $n = 26$. The diagram plots surface area on the x-axis, and height of the rectangle on the y-axis. The lower bound (LB) is marked by a line on the left, the optimal solution is marked by the label *Optimal*. We also show an arrow between two rectangles R_1 and R_2 if one subsumes the other, i.e. $W_1 \leq W_2, H_1 \leq H_2$. Unfortunately, none of the candidates to the left of the optimal solution subsumes another, we therefore have to check each candidate individually.

This decomposition approach differs from both [11] and [13]. Moffit and Pollack do not impose *a-priori* limits on the rectangle to be filled, while Korf builds solutions starting from an initial wide rectangle. Both methods are *anytime* algorithms, while our method is not. Whether this distinction is important will depend on the intended application. Korf will have to show infeasibility of the same or larger, more difficult rectangles to prove optimality, while the search space for Moffit and Pollack looks very different. An advantage of our method is it can be trivially extended to multiple processor cores by exploring candidates in parallel. Korf’s method is inherently sequential. A more fine grain parallelization can be applied to both Moffit and Pollack’s, and our approaches by unfolding the top choices in the search tree to run as different processes.

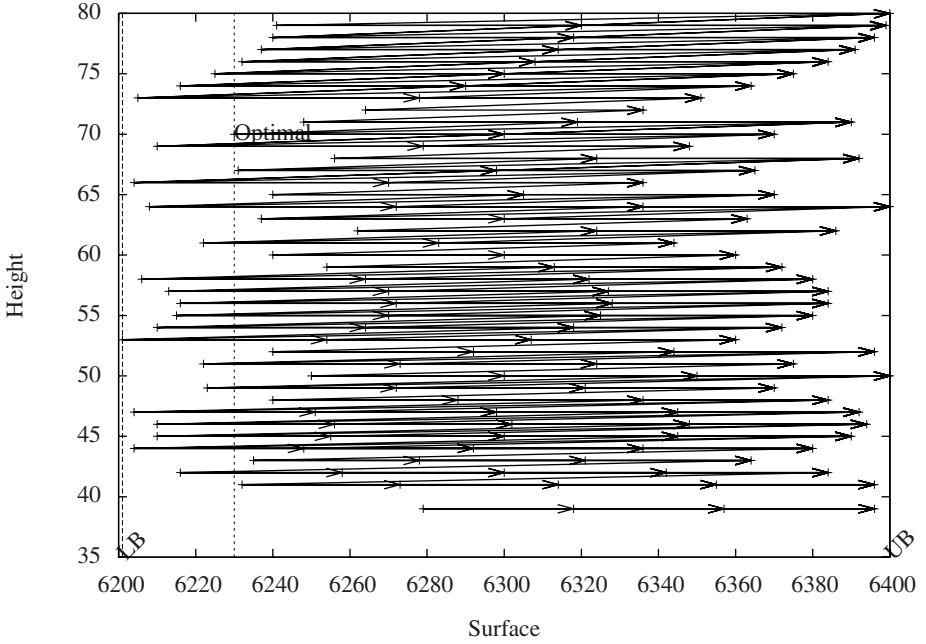


Fig. 2. Candidate plot for $n = 26$

2.2 Symmetry Removal

The model so far contains a number of symmetries, which we need to remove as we may have to explore the complete search space. We restrict the domain of the largest square of size $n \times n$ to be placed in an enclosing rectangle of size $Width \times Height$ to

$$X :: 1..1 + \left\lfloor \frac{Width - n}{2} \right\rfloor, Y :: 1..1 + \left\lfloor \frac{Height - n}{2} \right\rfloor.$$

For the square packing problem we can apply a slightly stronger restriction, due to the increased number of symmetries. For an enclosing square of size $Size \times Size$ we use the following restriction for the largest square to be placed

$$X :: 1..1 + \left\lfloor \frac{Size - n}{2} \right\rfloor, Y \leq X.$$

3 Search Strategies

For finite domain constraint problems, the choice of a search strategy usually follows naturally from the model. We first need to decide which variables to enumerate (*model*), we then have to consider the order in which they are assigned (*variable selection*), and the order in which possible values are tried (*value selection*). In case the default, complete, depth-first search is not sufficient, we also may have to decide on a incomplete search strategy. For the problem considered here, the choice is much simpler. The

squares should be assigned by decreasing size, so that the largest squares are assigned early on; there is no need for a dynamic variable ordering. Note that this is not necessarily true for the general rectangle placement problem, where items may be incomparable. As we may have to explore the complete search space for many subproblems, the choice of a good value ordering is not so critical, since it will only have an effect on feasible sub-problems and, as we need to explore the search space completely for the infeasible subproblems, there is little that incomplete search strategies can contribute. Given these restrictions, it is surprising how many different search methods can be applied to this problem type. The following paragraphs describe the nine alternatives we considered.

3.1 Naive

The most basic routine places the squares one after the other, in order of decreasing size, by choosing a value for the x and y variables. On backtracking, the next alternative position is tested. The fundamental problem with this method is the large number of alternative values to be tested.

3.2 X Then Y

An alternative method would assign all x variables first, before assigning any of the y variables. The advantage is that after fixing the x values, there are few if any choices left for the y values, reducing the effective depth of the search tree to n . Unfortunately, if this does not work, this method will lead to deep backtracking (*thrashing*), making finding a solution all but impossible.

3.3 Disjunctive

An alternative way of placing the rectangles is deciding on the relative position of each pair. A rectangle can be placed to the left, to the right, above or below another rectangle,

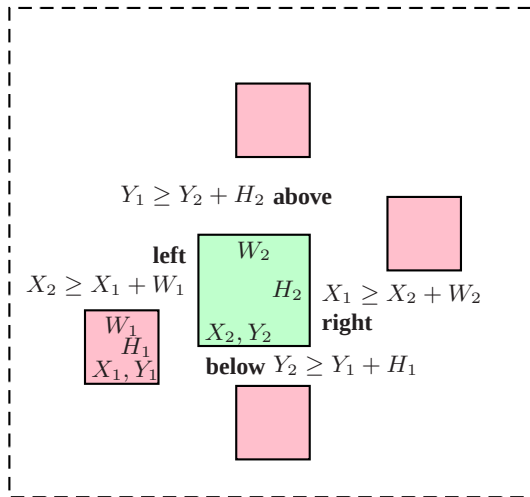


Fig. 3. Relative positioning of pairs of squares

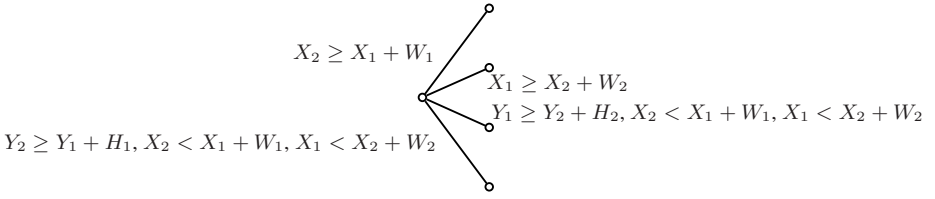


Fig. 4. Semantic disjunctive: showing four branches. Note some constraints appear twice.

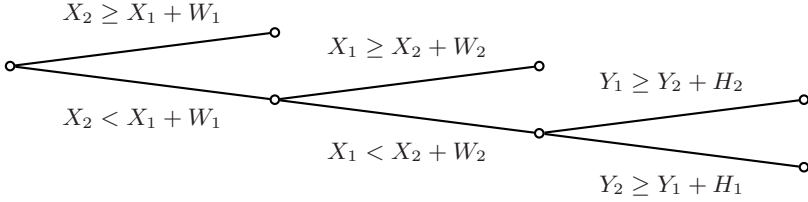


Fig. 5. Semantic disjunctive with binary choices

as shown in Figure 3. Each choice is enforced by imposing a constraint on the x or y variables of the two rectangles.

3.4 Semantic Disjunctive

A problem with the above disjunctive strategy is that the alternative cases are not exclusive: a rectangle can be, for example, both to the left and above another one. This means that we will consider some alternatives twice in the search, that is not a good idea given the overall size of the search space. One possible way of dealing with this overlap is to exclude left and right choices for the placement above and below. This leads to the four alternatives shown in Figure 4. This method is called *semantic4* in the experiments. Instead of trying these four alternatives for one choice, we can also split the decision into three binary choices. This maximizes the information that is available at each point and can help to reduce the number of choices to be explored. These binary decisions are shown in Figure 5. This method is called *semantic* in the experiments. The name *semantic disjunctive* is taken from [13], although it is not clear which variant is used in that paper.

3.5 Dual

This strategy is an example of a non-deterministic variable selection, followed by a deterministic value selection. This version, called *dual*, first assigns all the x variables, and then the y variables. It is the strategy used for the perfect square placement problems in [2, 6]. It works by choosing increasing values for the variables, and then deciding for each variable whether it should take that value or not. Once all x variables have been fixed, finding values for the y variables should be straightforward. There is a risk that due to a lack of propagation no valid assignment for the y variables exists, which will cause deep backtracking.

3.6 Forcing Obligatory Parts

The following strategies try to avoid the large branching factor caused by choosing individual values for the variables by splitting the domain into intervals first. The key idea is to make the size of the interval dependent on the size of the rectangle, it should be chosen large enough so that *obligatory parts* are generated for the CUMULATIVE and possibly the DISJOINT constraint. Figure 6 shows the effect of changing the interval size. Beldiceanu et al. [4] suggest the interval size $\lfloor \frac{S}{2} \rfloor + 1$ for a square of size S , which creates obligatory parts of at least half the size of the item. We show below that for the problem and constraints considered here this value is too aggressive, and smaller interval sizes lead to better performance. We tested three variants of this approach:

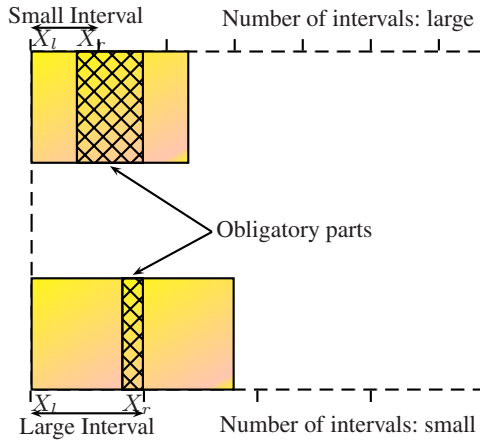


Fig. 6. Forcing obligatory parts

Interval. First split the x variables into intervals, then fix values for them, followed by splitting the y variables into intervals, and finally fixing the values of the y variables. This method is proposed in [4] for the perfect square packing problem. It is quite a risky strategy. But by ignoring the y variables when assigning the x variables, we can again possibly reduce the height of the search tree by a factor of 2, dramatically reducing the overall search space. Unfortunately, there is no guarantee that this will work in general, in particular if there is significant slack and/or the constraint propagation is weak.

Split. First split the x variables into intervals, then the y variables, before fixing the x variables, and then the y variables. This is even more risky than the previous strategy.

XY Intervals. For each rectangle, split the x and y variables into intervals, creating an obligatory part for both CUMULATIVE and the DISJOINT constraint. Once this is done for all rectangles, fix the values for x and y variables for each rectangle. This method is less aggressive, but, by interleaving x and y variables, may create larger search trees.

4 Model Improvements

We consider some runtime performance enhancing improvements.

4.1 Empty Strip Dominance

In [10] one of the pruning methods is a dominance criterion that eliminates certain partial placements from consideration, since an equivalent placement has already been investigated. This is a special case of symmetry breaking, a very active field of research for constraint programming [9]. Such reasoning cannot be directly put inside a DISJOINT or CUMULATIVE constraint, as it removes feasible, but dominated assignments; it has to be added either as a modification of the search routine, or a specific constraint.

We do not use the same problem representation as [10], and therefore have to adapt the approach to the possibilities of our model. We introduce two variants, one dealing with the border of the problem space, the other dealing with interaction of two squares. We do not consider the case where multiple squares form a “wall”.

Initial Domain Reduction. Following the reasoning in [10], we can remove some values from the domain of the X and Y variables for a square with edge size k . Suppose the square is placed d units from the border. Then the gap can only be used by squares up to length d . If all squares $1 \times 1, 2 \times 2, \dots, d \times d$ fit into the space $k \times d$, then it would be possible to shift the larger square to the border, moving all these smaller squares into the now vacant space. As we will consider the placement of the big square on the border, we do not have to consider the placement d units from the border, this value can be removed from the domain a priori. For each size k , we can easily compute all values that can be removed, by considering the placement problem of d squares of increasing size in a $k \times d$ area. Note that this can be easily solved by hand, checking which squares cannot be placed on top of each other. This leads to the *generic* domain reductions shown in Table 1 for squares from size 2 up to size 45. These reductions (called *domain*) can be applied when setting up the problem, and are independent of the size of the enclosing space. For the problem of packing squares considered in this paper we can strengthen the bounds slightly, as shown in the *specific* row in Table 1. This uses the fact that each square occurs only once, so for the square of size 3 we can remove gap 3 as well, as only squares 1×1 and 2×2 can fill the gap.

Table 1. Forbidden gaps due to dominance

size	2	3	4	5-8	9-11	12-17	18-21	22-29	30-34	34-44	45
generic	1	2	2	3	4	5	6	7	8	9	10
specific	2	3									

Interaction of two squares. A similar pruning (called *gap*) can be used to eliminate the placement of squares that *face* a larger square at a certain distance. As the search routines do not just place one square after the other, this check has to be data-driven,

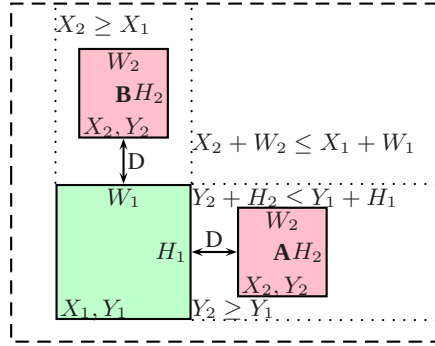


Fig. 7. Dominance condition between squares

it will be tested as soon as both squares are placed. The situation is shown in Figure 7 (case **A**). Square 2 is to the right of the larger square 1, and facing it, i.e.

$$Y_2 \geq Y_1, Y_2 + H_2 \leq Y_1 + H_1.$$

The distance $D = X_2 - (X_1 + W_1)$ between the squares cannot be any of the forbidden gap values for H_2 . The same argument can be made if square 2 is above square 1 and facing it (Figure 7, case **B**): $X_2 \geq X_1, X_2 + W_2 \leq X_1 + W_1, D = Y_2 - (Y_1 + H_1)$.

4.2 Ignoring Size 1 Squares

The square of size 1 can be placed in any available location. We therefore do not need to include it in the constraint model (we call this method *notone*), saving the cost of constantly updating its domains and checking its interaction with the other squares. Contrary to [4] we observe a significant improvement in performance when the smallest square is removed. For their problem of perfect square placement, the opposite occurs: execution times increase dramatically by a factor of 7. The probable cause is that the step from no slack in the perfect placement problem to a single unit of slack in the problem without the 1×1 square reduces the effectiveness of some propagation mechanism. In our case, most problems already contain a significant amount of slack, so the reduction in propagation overhead becomes more visible. Note that we still have to consider the 1×1 square when calculating the required area to fill, so that there is room for it even if it is not represented in the constraints.

4.3 Ignoring Size 2 Squares

We can also try to ignore the 2×2 square when setting up the constraints. If the resulting problem is infeasible, then the original problem is also infeasible. If it is feasible, then we might get lucky, and the solution leaves place for both 2×2 and 1×1 squares. If this is not the case, we have to check the candidate again, with the 2×2 square included. For the candidates studied, only one instance (size 21, 37×90) is feasible when ignoring the 2×2 square, and infeasible for the original problem. We do not use this simplification in our experiments.

5 Results

We now report some experimental results for our programs using SICStus Prolog 4.0.2 on a 3GHz Intel Xeon E5450 with 3.25Gb of memory running WindowsXP SP2. We use a single processor core for the experiments.

We first compare the different strategies in Table 2, showing the execution times (in seconds) required for problem sizes 15 to 27. Missing entries indicate that times were significantly exceeding competing methods. The *disj* strategy is performing worst, even slower than the *naive* enumeration. This is not surprising, considering that the choices are not exclusive. We also include the combination of *naive* strategy with the *gap* improvement. This is the only case where this redundant constraint improves results significantly. Enumerating all x and then the y variables (*xtheny* strategy) achieves a much better result than the *naive* enumeration which interleaves their enumeration. The *dual* strategy performs badly when solving all candidate problems, but is competitive for some instances with no or very little slack, even for large problem sizes. The *semantic* branching works quite well up to problem size 21, with the binary choices leading to a slightly better result. The clear winners are the branching methods based on intervals, where the more conservative *xy* strategy is out-performed by the *interval* and *split* strategies, which split the x variables before the y variables. For each method we use the interval size (indicated as a fraction of the square length) which produces the most stable results over all problem instances.

Even when we consider individual candidates, we find that the *interval* strategy performs best for nearly all cases. There are some exceptions for problems with no slack, where the *dual* method sometimes wins, and for some feasible problems, for which the *split* strategy seems to work well.

Figure 8 presents the result in graphical form, and adds the times for previous approaches (Korf and BlueBlocker results from [13]) for comparison. Note the logarithmic scale for the execution time. With the exception of the naive strategies, all other methods outperform the previously known approaches.

Figure 9 shows the impact of the interval length for the *interval* strategy. The interval length (as a fraction of the length of the square to be placed) is plotted on the x-axis, the

Table 2. Strategy comparison

n	naive	naive +gap	xtheny	disj	semantic4	semantic	dual	interval 0.3	split 0.2	xy 0.75
15	2.92	2.16	0.09	12.12	0.55	0.45	2.63	-	0.05	-
16	10.44	7.02	0.11	98.25	1.31	1.03	0.89	-	0.05	-
17	20.75	13.81	0.27	23.57	1.48	1.13	0.33	0.05	0.05	0.81
18	667.33	325.56	18.37	-	30.53	23.05	118.58	1.83	1.13	13.94
19	4140.09	1823.15	13.73	-	83.42	63.25	80.66	1.11	1.88	36.78
20	-	-	13.08	-	216.07	167.61	149.79	2.14	1.47	108.28
21	-	-	143.72	-	1138.98	865.13	-	8.09	10.59	619.45
22	-	-	1708.89	-	-	-	-	52.21	32.36	1668.59
23	-	-	-	-	-	-	-	245.07	265.54	9521.73
24	-	-	-	-	-	-	-	452.73	545.82	37506.20
25	-	-	-	-	-	-	-	2533.64	4127.41	-
26	-	-	-	-	-	-	-	14158.15	-	-
27	-	-	-	-	-	-	-	43529.87	-	-

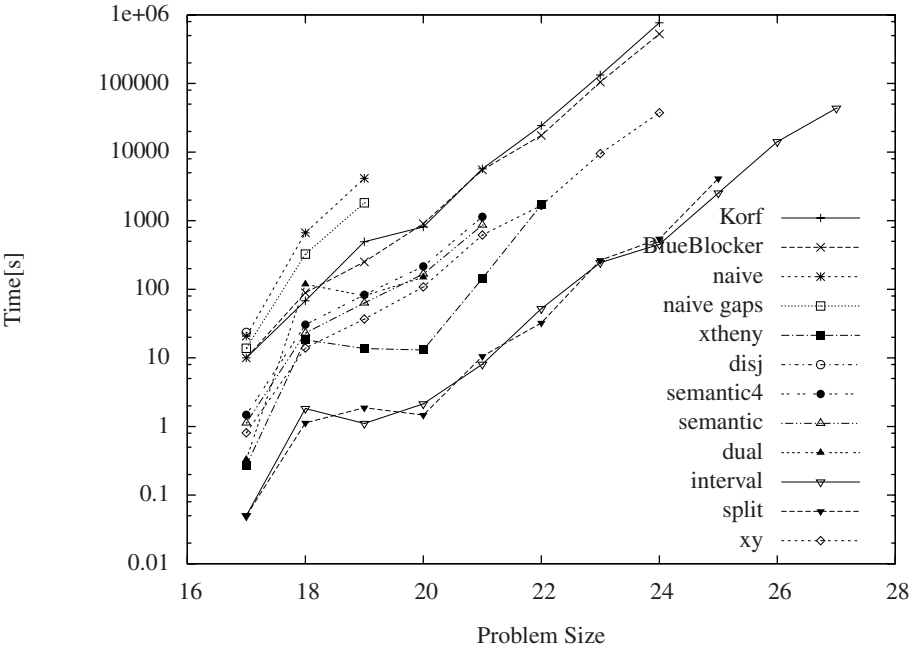


Fig. 8. Strategy comparison plot, including methods from the literature

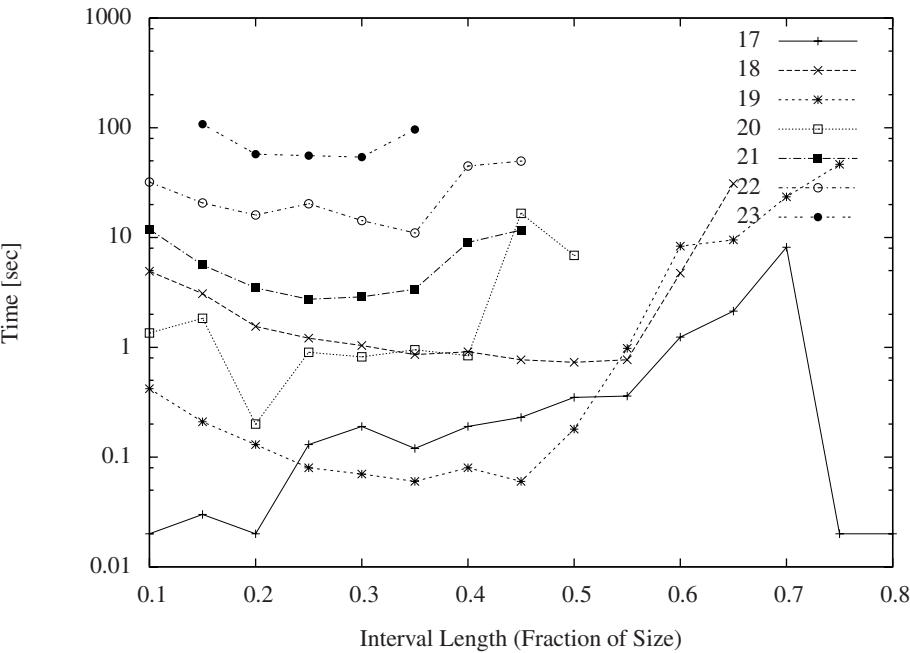


Fig. 9. Interval strategy: impact of interval length

Table 3. Rectangle placement overview

n	Surface	K	Width	Height	Area	Loss	Back	Time	Clautiaux	Korf	BlueBlocker
18	2109	14	31	69	2139	1.42	25781	00:01	31:33	1:08	1:29
19	2470	12	47	53	2491	0.85	18747	00:01	72:53:18	8:15	4:11
20	2870	14	34	85	2890	0.70	28841	00:02	-	13:32	15:03
21	3311	19	38	88	3344	1.00	128766	00:07	-	1:35:08	1:32:01
22	3795	15	39	98	3822	0.71	566864	00:51	-	6:46:15	4:51:23
23	4324	19	64	68	4352	0.65	2802479	03:58	-	36:54:50	29:03:49
24	4900	18	56	88	4928	0.57	4541284	05:56	-	213:33:00	146:38:48
25	5525	17	43	129	5547	0.40	28704074	40:38	-	see text	-
26	6201	21	70	89	6230	0.47	143544214	03:41:43	-	-	-
27	6930	21	47	148	6956	0.38	420761107	11:30:02	-	-	-

execution time on the y-axis. Time points missing indicate that no solution was found within a timeout of 120 seconds. The impact of the interval size is more pronounced for the larger problem sizes, where values 0.2-0.3 seem to provide the best results. Values 0.4 and higher lead to thrashing in some instances, and can therefore not be recommended.

Table 3 shows the best results with the *interval* strategy for the rectangle packing problem of sizes 18 to 27, problem sizes 26 and 27 were previously open. Diagrams of the solutions can be found on our website (<http://www.4c.ucc.ie/~hsimonis>). The columns have the following meaning:

- n is the problem size;
- *Surface* is the total surface area of all squares to be packed;
- K is the number of subproblems that had to be checked;
- *Width* and *Height* are the size of the optimal rectangle;
- *Area* is its surface area;
- *Loss* is the spare space in the optimal rectangle as a percentage;
- B is the number of backtrack steps as reported by SICStus Prolog;
- *Time* is the time (in HH:MM:SS) required.

For ease of comparison, we also include in Table 3 the results reported in [13]. The times for Clautiaux, Korf and BlueBlocker were obtained on a Linux Opteron 2.2GHz machine with 8Gb of RAM. Our results use SICStus 4.0.2 on a 3GHz Intel Xeon 5450 with 3.25Gb of memory, we estimate that our hardware is about twice faster. The previous best time for size 25 in [11] was over 42 days, although on a significantly slower machine.

Table 4 shows the impact of the different improvements to our model, giving the required runtime as a percentage of the pure model. The best combination ignores the square of size 1×1 (option *notone*), and uses the initial domain reduction from the dominance criterion (*domain*), but does not use the additional constraint about the gap between squares (*gap*). The massive improvement when ignoring the 1×1 square cannot be completely explained by reduced propagation. It is most likely caused by reducing bad choices at the end of the x interval splitting. We noted that it pays off not to include the small squares in this part of the search.

In our decomposition approach, we have to show infeasibility of multiple subproblems before reaching the optimal solution. The times required for the subproblems vary

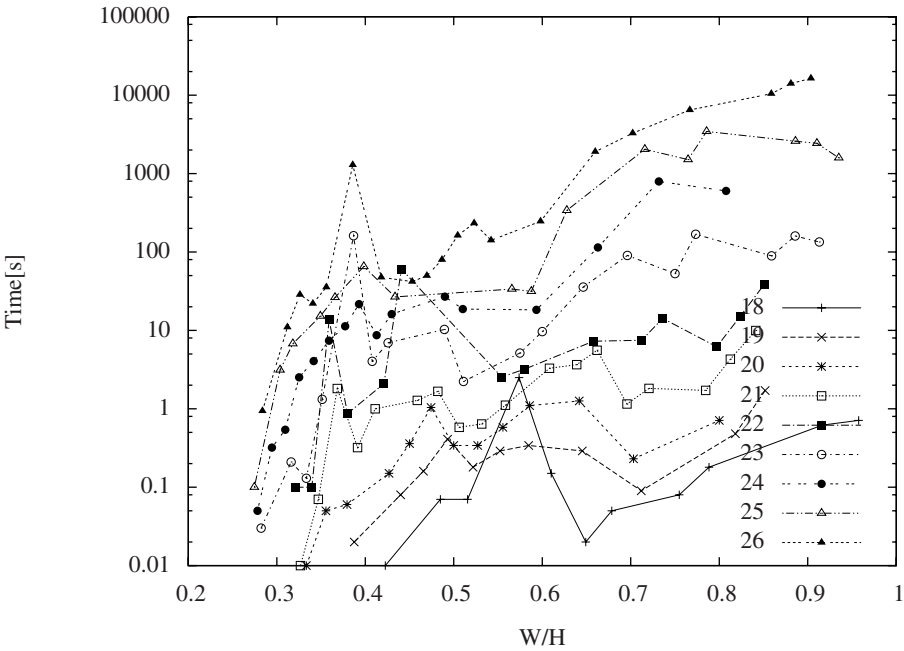


Fig. 10. Interval strategy: impact of squariness

Table 4. Method comparison

n	pure	gap	domain	notone	all	best
18	100.00	99.37	78.96	12.93	9.77	9.78
19	100.00	101.61	87.14	48.55	38.26	37.31
20	100.00	105.26	92.24	18.93	16.20	15.39
21	100.00	100.94	81.90	63.57	50.82	49.58
22	100.00	100.24	90.56	23.66	19.46	19.00
23	100.00	99.81	78.92	30.33	23.18	22.80
24	100.00	101.77	77.69	36.43	29.16	28.58

widely. For the *interval* strategy, this does not seem to be caused by the amount of slack in the problem, the shape of the enclosing rectangle has a much more direct impact. Although not uniform, Figure 10 shows a clear connection between the “*squariness*” of the rectangle and the runtime. It is much harder to show infeasibility for near-square rectangles. For the *dual strategy*, the opposite happens. Runtimes explode when the slack increases, but there is little impact from the “*squariness*”.

6 Incomplete Heuristics

We also considered incomplete heuristics to find good solutions for the problem and evaluated these on the square packing problem. They are based on the well-known observation that good packing solutions place the large items in the corner and on the edges of the enclosing field without any lost space. The smaller items and the slack

Table 5. New optimal solutions for square packing

Problem Size	26	27	29	30	31	35
Optimal Solution	80	84	93	98	103	123
T_{opt}	12:26	00:04	11:06	2:07	00:18	1:10:07
T_{proof}	1:25:22	-	-	-	-	-

space are used inside the packing area. We only consider one side, say the bottom one, of the board for our heuristic, and assume that the biggest square is placed in the bottom left corner. We then try to find combinations of $K - 1$ other squares that fill the bottom edge completely, not considering very small squares.

We precompute all possible solutions with a small finite domain constraint program. Once all solutions are found, we order them by decreasing area of the selected squares, and use them as initial branches in our packing model, setting the y coordinate of the selected squares to 1, as well as fixing the biggest square in position $(1, 1)$. Note that we do not fix the relative placement in the x direction, this is determined by the remainder of the search routine. If no solution for the given *Size* is found, we backtrack and recompute the heuristic for the next larger value.

Optimal solutions for the square packing problem up to size 25 are already known from [11]. We find six new optimal values shown in Table 5, T_{opt} is the time required to find the optimal solution, T_{proof} the time for the proof of optimality with the full model. A dash indicates that a lower bound is reached and no further proof is required.

7 Conclusion

In this paper we have demonstrated that in the domains of optimal rectangle and square packing an “off-the-shelf” constraint programming system, SICStus Prolog, outperforms recently developed ad-hoc approaches by over three orders of magnitude. We have also closed eight open problems: two rectangle packing problems and six square packing problems. We argue that rectangle packing is a domain in which current constraint programming technology significantly outperforms hand-crafted ad-hoc systems developed for this problem. This provides the CP community with a convincing success story.

Acknowledgment

This work was supported by Science Foundation Ireland (Grant Number 05/IN/I886). The authors wish to thank Mats Carlsson, who provided the SICStus Prolog 4.0.2 used for the experiments.

References

1. Aggoun, A., Beldiceanu, N.: Extending CHIP in order to solve complex scheduling problems. *Journal of Mathematical and Computer Modelling* 17(7), 57–73 (1993)
2. Beldiceanu, N., Bourreau, E., Simonis, H.: A note on perfect square placement. In: *Prob009 in CSPLIB* (1999)

3. Beldiceanu, N., Carlsson, M.: Sweep as a generic pruning technique applied to the non-overlapping rectangles constraint. In: Walsh [16], pp. 377–391
4. Beldiceanu, N., Carlsson, M., Poder, E.: New filtering for the cumulative constraint in the context of non-overlapping. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 21–35. Springer, Heidelberg (2008)
5. Beldiceanu, N., Carlsson, M., Poder, E., Sadek, R., Truchet, C.: A generic geometrical constraint kernel in space and time for handling polymorphic n -dimensional objects. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 180–194. Springer, Heidelberg (2007)
6. Beldiceanu, N., Contejean, E.: Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling* 20(12), 97–123 (1994)
7. Beldiceanu, N., Guo, Q., Thiel, S.: Non-overlapping constraints between convex polytopes. In: Walsh [16], pp. 392–407.
8. Carlsson, M., et al.: SICStus Prolog User's Manual, 4th edn. Swedish Institute of Computer Science (2007) ISBN 91-630-3648-7
9. Gent, I., Petrie, K., Puget, J.F.: Symmetry in constraint programming. In: Rossi, F., van Beek, P., Walsh, T. (eds.) *Handbook of Constraint Programming*, ch. 10. Elsevier, Amsterdam (2006)
10. Korf, R.E.: Optimal rectangle packing: Initial results. In: Giunchiglia, E., Muscettola, N., Nau, D.S. (eds.) ICAPS, pp. 287–295. AAAI, Menlo Park (2003)
11. Korf, R.E.: Optimal rectangle packing: New results. In: Zilberstein, S., Koehler, J., Koenig, S. (eds.) ICAPS, pp. 142–149. AAAI, Menlo Park (2004)
12. Moffitt, M.D., Ng, A.N., Markov, I.L., Pollack, M.E.: Constraint-driven floorplan repair. In: Sentovich, E. (ed.) DAC, pp. 1103–1108. ACM, New York (2006)
13. Moffitt, M.D., Pollack, M.E.: Optimal rectangle packing: A meta-CSP approach. In: Long, D., Smith, S.F., Borrajo, D., McCluskey, L. (eds.) ICAPS, pp. 93–102. AAAI, Menlo Park (2006)
14. Roy, J.A., Markov, I.L.: Eco-system: Embracing the change in placement. In: ASP-DAC, pp. 147–152. IEEE, Los Alamitos (2007)
15. Van Hentenryck, P.: Scheduling and packing in the constraint language cc(FD). In: Zweben, M., Fox, M. (eds.) *Intelligent Scheduling*. Morgan Kaufmann Publishers, San Francisco (1994)
16. Walsh, T. (ed.): *Principles and Practice of Constraint Programming - CP 2001*. LNCS, vol. 2239. Springer, Heidelberg (2001)

Solving a Telecommunications Feature Subscription Configuration Problem

David Lesaint¹, Deepak Mehta², Barry O’Sullivan², Luis Quesada², and Nic Wilson²

¹ Intelligent Systems Research Centre, British Telecom, UK
david.lesaint@bt.com

² Cork Constraint Computation Centre, University College Cork, Ireland
{d.mehta,b.osullivan,l.quesada,n.wilson}@4c.ucc.ie

Abstract. Call control features (e.g., call-divert, voice-mail) are primitive options to which users can subscribe off-line to personalise their service. The configuration of a feature subscription involves choosing and sequencing features from a catalogue and is subject to constraints that prevent undesirable feature interactions at run-time. When the subscription requested by a user is inconsistent, one problem is to find an optimal relaxation. In this paper, we show that this problem is NP-hard and we present a constraint programming formulation using the variable weighted constraint satisfaction problem framework. We also present simple formulations using partial weighted maximum satisfiability and integer linear programming. We experimentally compare our formulations of the different approaches; the results suggest that our constraint programming approach is the best of the three overall.

1 Introduction

Information and communication services, from news feeds to internet telephony, are playing an increasing, and potentially disruptive, role in our daily lives. As a result, providers seek to develop personalisation solutions allowing customers to control and enrich their service. In telephony, for instance, personalisation relies on the provisioning of call control features. A feature is an increment of functionality which, if activated, modifies the basic service behaviour in systematic or non-systematic ways, e.g., do-not-disturb, multi-media ring-back tones, call-divert-on-busy, credit-card-calling, find-me.

Modern service delivery platforms provide the ability to implement features as modular applications and compose them on demand when setting up live sessions, that is, consistently with the feature subscriptions preconfigured by participants. In this context, a personalisation approach consists of exposing feature catalogues to subscribers and letting them select and sequence the features of their choice.

Not all sequences of features are acceptable though due to the possible occurrence of feature interactions. A feature interaction is “some way in which a feature modifies or influences the behaviour of another feature in generating the system’s overall behaviour” [1]. For instance, a do-not-disturb feature will block any incoming call and cancel the effect of any subsequent feature subscribed by the callee. This is an undesirable interaction: as shown in Figure 1, the call originating from X will never reach

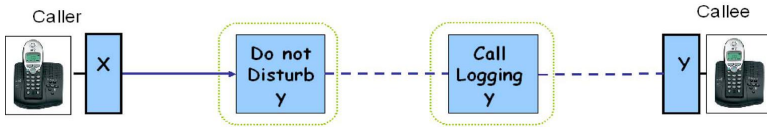


Fig. 1. An example of an undesirable feature interaction

call-logging. However, if call-logging is placed before do-not-disturb then both features will play their role.

Distributed Feature Composition (DFC) provides a method and a formal architecture model to address feature interactions [1,2,3]. The method consists of constraining the selection and sequencing of features by prescribing constraints that prevent undesirable interactions. These feature interaction resolution constraints are represented in a feature catalogue as precedence or exclusion constraints. A precedence constraint, $f_i \prec f_j$, means that if the features f_i and f_j are part of the same sequence then f_i must precede f_j in the sequence. An exclusion constraint between f_i and f_j means that they cannot be together in any sequence. Undesirable interactions are then avoided by rejecting any sequence that does not satisfy the catalogue constraints.

A *feature subscription* is defined by a set of features, a set of user specified precedence constraints and a set of feature interaction constraints from the catalogue. The main task is to find a sequence of features that is consistent with the constraints in the catalogue. It may not always be possible to construct a sequence of features that consists of all the user selected features and respect all user specified precedence constraints. In such cases, *the task is to find a relaxation of the feature subscription that is closest to the initial requirements of the user.*

In this paper, we shall show that checking the consistency of a feature subscription is polynomial in time, but finding an optimal relaxation of a feature subscription, when inconsistent, is NP-hard. We shall then present the formulation of finding an optimal relaxation using *constraint programming*. In particular, we shall use the variable weighted constraint satisfaction problem framework. In this framework, a branch and bound algorithm that maintains some level of consistency is usually used for finding an optimal solution. We shall investigate the impact of maintaining three different levels of consistency. The first one is Generalised Arc Consistency (GAC) [4], which is commonly used. The others are mixed consistencies. Here, mixed consistency means maintaining different levels of consistency on different sets of variables of a given problem. The first (second) mixed consistency enforces (a restricted version of) singleton GAC on some variables and GAC on the remaining variables of the problem.

We shall also consider *partial weighted maximum satisfiability*, an artificial intelligence technique, and *integer linear programming*, an operations research approach. We shall present the formulations using these approaches and shall discuss their differences with respect to the constraint programming formulation.

We have conducted experiments to compare the different approaches. The experiments are performed on a variety of random catalogues and random feature subscriptions. We shall present empirical results that demonstrate the superiority of maintaining mixed consistency on the generalised arc consistency. For hard problems, we see a

difference of up to three orders of magnitude in terms of search nodes and one order of magnitude in terms of time. Our results suggest that, when singleton generalised arc consistency is used, the constraint programming approach considerably outperforms our integer linear programming and partial weighted maximum satisfiability formulations. We highlight the factors that deteriorate the scalability of the latter approaches.

The rest of the paper is organised as follows. Section 2 provides an overview of the DFC architecture, its composition style and subscription configuration method. Section 3 presents the relevant definitions and theorems. Section 4 describes the constraint programming formulation for finding an optimal relaxation and discusses branch and bound algorithms that maintain different levels of consistency. The integer linear programming and partial weighted maximum satisfiability formulations of the problem are described in Section 5. The empirical evaluation of these approaches is shown in Section 6. Finally our conclusions are presented in Section 7.

2 Configuring Feature Subscriptions in DFC

In DFC each feature is implemented by one or more modules called *feature box types* (FBT) and each FBT has many run-time instances called *feature boxes*. We assume in this paper that each feature is implemented by a single FBT and we associate features with FBTs. As shown in Figure 2, a call session between two end-points is set up by chaining feature boxes. The routing method decomposes the connection path into a source and a target region and each region into *zones*. A source (target) zone is a sequence of feature boxes that execute for the same source (target) address.

The first source zone is associated with the source address encapsulated in the initial setup request, e.g., zone of *X* in Figure 2. A change of source address in the source region, caused for instance by an identification feature, triggers the creation of a new source zone [5]. If no such change occurs in a source zone and the zone cannot be expanded further, routers switch to the target region. Likewise, a change of target address in the target region, as performed by Time-Dependent-Routing (TDR) in Figure 2, triggers the creation of a new target zone. If no such change occurs in a target zone and the zone cannot be expanded further (as for *Z* in Figure 2), the request is sent to the final box identified by the encapsulated target address.

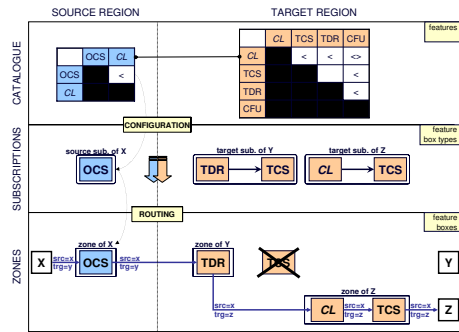


Fig. 2. DFC: Catalogues, subscriptions and sessions

DFC routers are only concerned with locating feature boxes and assembling zones into regions. They do not make decisions as to the type of feature boxes (the FBTs) appearing in zones or their ordering. They simply fetch this information from the *feature subscriptions* that are preconfigured for each address in each region based on the *catalogue* published by the service provider.

A catalogue is a set of features subject to precedence and exclusion constraints. Features fall into three classes: *source*, *target* and *reversible*, i.e., a subset of features that are both source and target. Constraints are formulated by designers on pairs of source features and pairs of target features to prevent undesirable feature interactions in each zone [6]. Specifically, a precedence constraint imposes a routing order between two features, as for the case of Terminating-Call-Screening (TCS) and Call-Logging (CL) in Figure 2. An exclusion constraint makes two features mutually exclusive, as for the case of CL and Call-Forwarding-Unconditional (CFU) in Figure 2.

A subscription is a subset of catalogue features and a set of user precedence constraints between features in each region. For instance, the subscription of Y in the target region includes the user precedence $\text{TDR} \prec \text{TCS}$. Configuring a subscription involves selecting, parameterising and sequencing features in each region consistently with the catalogue constraints and other integrity rules [3]. In particular, the source and target regions of a subscription must include the same reversible features in inverse order, i.e. source and target regions are not configured independently.

3 Formal Definitions

Let f_i and f_j be features, we write a precedence constraint of f_i before f_j as $\langle f_i, f_j \rangle$, or as $f_i \prec f_j$. An exclusion constraint between f_i and f_j expresses that these features cannot appear together in a sequence of features. We encode this as the pair of precedence constraints $\langle f_i, f_j \rangle$ and $\langle f_j, f_i \rangle$.

Definition 1 (Feature Catalogue). A catalogue is a tuple $\langle F, P \rangle$, where F is a set of features that are available to users and P is a set of precedence constraints on F .

The *transpose* of a catalogue $\langle F, P \rangle$ is the catalogue $\langle F, P^T \rangle$ such that $\forall \langle f_i, f_j \rangle \in P^2 : \langle f_i, f_j \rangle \in P \Leftrightarrow \langle f_j, f_i \rangle \in P^T$. In DFC the precedence constraints between the features in the source (target) catalogue are specified with respect to the direction of the call. For the purpose of configuration, we combine the source catalogue $\langle F_s, P_s \rangle$ and the target catalogue $\langle F_t, P_t \rangle$ into a single catalogue $\langle F_c, P_c \rangle \equiv \langle F_s \cup F_t, P_s \cup P_t^T \rangle$.

Definition 2 (Feature Subscription). A feature subscription S of catalogue $\langle F_c, P_c \rangle$ is a tuple $\langle F, C, U, W_F, W_U \rangle$, where $F \subseteq F_c$, C is the projection of P_c on F , i.e., $P_c \downarrow_F = \{f_i \prec f_j \in P_c : \{f_i, f_j\} \subseteq F\}$, U is a set of (user defined) precedence constraints on F , $W_F : F \rightarrow \mathbb{N}$ is a function that assigns weights to features and $W_U : U \rightarrow \mathbb{N}$ is a function that assigns weights to user precedence constraints. The value of S is defined by $\text{Value}(S) = \sum_{f \in F} W_F(f) + \sum_{p \in U} W_U(p)$.

Note that a weight associated with a feature signifies its importance for the user. These weights could be elicited directly, or using data mining or analysis of user interactions.

Definition 3 (Consistency). A feature subscription $\langle F, C, U, W_F, W_U \rangle$ of some catalogue is defined to be consistent if and only if the directed graph $\langle F, C \cup U \rangle$ is acyclic.

Due to the composition of the source and target catalogues into a single catalogue, a feature subscription S is consistent if and only if both source and target regions are consistent in the DFC sense.

Theorem 1 (Complexity of Consistency Checking). *Determining whether a feature subscription $\langle F, C, U, W_F, W_U \rangle$ is consistent or not can be checked in $\mathcal{O}(|F| + |C| + |U|)$.*

Proof. We use *Topological Sort* [7]. In *Topological Sort* we are interested in ordering the nodes of a directed graph such that if the edge $\langle i, j \rangle$ is in the set of edges of the graph then node i is less than node j in the order. In order to use *Topological Sort* for detecting whether a feature subscription is consistent, we associate nodes with features and edges with precedence constraints. Then, the subscription is consistent iff for all edges $\langle i, j \rangle$ in the graph associated with the subscription we have that $i \prec j$ in the order computed by *Topological Sort*. As the complexity of *Topological Sort* is linear with respect to the size of the graph, detecting whether a feature subscription is consistent is $\mathcal{O}(|F| + |C| + |U|)$. $\square$

If an input feature subscription is not consistent then the task is to relax the given feature subscription by dropping one or more features or user precedence constraints to generate a consistent feature subscription with maximum value.

Definition 4 (Relaxation). *A relaxation of a feature subscription $\langle F, C, U, W_F, W_U \rangle$ is a subscription $\langle F', C', U', W'_F, W'_U \rangle$ such that $F' \subseteq F$, $C' = P_c \downarrow_{F'}$, $U' \subseteq U \downarrow_{F'}$, $W_{F'}$ is W_F restricted to F' , and $W_{U'}$ is W_U restricted to U' .*

Definition 5 (Optimal Relaxation). *Let R_S be the set of all consistent relaxations of a feature subscription S . We say that $S_i \in R_S$ is an optimal relaxation of S if it has maximum value among all relaxations, i.e., if and only if there does not exist $S_j \in R_S$ such that $\text{Value}(S_j) > \text{Value}(S_i)$.*

Theorem 2 (Complexity of Finding an Optimal Relaxation). *Finding an optimal relaxation of a feature subscription is NP-hard.*

Proof. Given a directed graph $\langle V, E \rangle$, the *Feedback Vertex Set Problem* is to find a smallest $V' \subseteq V$ whose deletion makes the graph acyclic. This problem is known to be NP-hard [8]. We prove that finding an optimal relaxation is NP-hard by reducing the feedback vertex set problem to the latter. Given a feature subscription $S = \langle F, C, U, W_F, W_U \rangle$, the feedback vertex set problem can be reduced to our problem by associating the nodes of the directed graph V with features F , the edges E with catalogue precedence constraints C , the empty set $\emptyset$ with U , and the constant function that maps every element of its domain to 1 ($\lambda x.1$) with both W_F and W_U . Notice that, as $U = \emptyset$, the only way of finding an optimal relaxation of S is by removing a set of features from F . Assuming that an optimal relaxation is $S' = \langle F', C', U', W'_F, W'_U \rangle$, the set of features $F - F'$ corresponds to the smallest set of nodes V' whose deletion makes the directed graph acyclic. Thus, we can conclude that finding an optimal relaxation S' is NP-hard. $\square$

4 A Constraint Programming Approach

Constraint programming has been successfully used in many applications such as planning, scheduling, resource allocation, routing, and bio-informatics [9]. Here problems are primarily stated as a Constraint Satisfaction Problems (CSP), that is a finite set of variables, together with a finite set of constraints. A solution to a CSP is an assignment of a value to each variable such that all constraints are satisfied simultaneously. The basic approach to solving a CSP instance is to use a backtracking search algorithm that interleaves two processes: *constraint propagation* and *labeling*. Constraint propagation helps in pruning values that do not lead to a solution of the problem. Labeling involves assigning values to variables that may lead to a solution.

Various generalisations of the CSP have been developed to find a solution that is optimal with respect to certain criteria such as costs, preferences or priorities. One of the most significant is the Constraint Optimisation Problem (COP). Here the goal is to find an optimal solution that maximises (minimises) the objective function. The simplest COP formulation retains the CSP limitation of allowing only hard Boolean-valued constraints but adds an objective function over the variables.

4.1 Formulation

In this section we model the problem of finding an optimal relaxation of a feature subscription $\langle F, C, U, W_F, W_U \rangle$ as a COP.

Variables and Domains. We associate each feature $f_i \in F$ with two variables: a *Boolean variable* bf_i and an *integer variable* pf_i . A Boolean variable bf_i is instantiated to 1 or 0 depending on whether f_i is included in the subscription or not, respectively. The domain of each integer variable pf_i is $\{1, \dots, |F|\}$. Assuming that the computed subscription is consistent, an integer variable pf_i corresponds to the position of the feature f_i in a sequence. We associate each user precedence constraint $p_{ij} \equiv (f_i \prec f_j) \in U$ with a *Boolean variable* bp_{ij} . A Boolean variable bp_{ij} is instantiated to 1 or 0 depending on whether p_{ij} is respected in the computed subscription or not respectively.

Constraints. A catalogue precedence constraint $p_{ij} \in C$ that feature f_i should be before feature f_j can be expressed as follows:

$$bf_i \wedge bf_j \Rightarrow (pf_i < pf_j).$$

Note that the constraint is activated only if the selection variables bf_i and bf_j are instantiated to 1. A user precedence constraint $p_{ij} \in U$ that f_i should be placed before f_j can be expressed as follows:

$$bp_{ij} \Rightarrow (bf_i \wedge bf_j \wedge (pf_i < pf_j)).$$

Note that if a user precedence constraint holds then the features f_i and f_j are included in the subscription and also the feature f_i is placed before f_j , that is, the selection variables bf_i and bf_j are instantiated to 1 and $pf_i < pf_j$ is true.

Objective Function. The objective of finding an optimal relaxation of a feature subscription can be expressed as follows:

$$\text{Maximise } \sum_{f_i \in F} bf_i \times W_F(f_i) + \sum_{p_{ij} \in U} bp_{ij} \times W_U(p_{ij}).$$

4.2 Solution Technique

A depth-first branch and bound algorithm (BB) is generally used to find an optimal solution. In case of maximisation, BB keeps the current optimal value of the solution while traversing the search tree. This value is a lower bound (lb) of the objective function. At each node of the search tree BB computes an overestimation of the global value. This value is an upper bound (ub) of the best solution that can be found as long as the current search path is maintained. If $ub \leq lb$, then a solution of a greater value than the current optimal value cannot be found below the current node, so the current branch is pruned and the algorithm backtracks.

Enforcing local consistency enables the computation of $ub_{(i,a)}$, which is a specialisation of ub for a value a of an unassigned variable i . If $ub_{(i,a)} \leq lb$, then value a can be removed because it will not be present in any solution better than the current one. Removed values are restored when BB backtracks above the node where they were eliminated. The quality of the upper bound can be improved by increasing the level of local consistency that is maintained at each node of the search tree. The different levels of local consistencies that we have considered are *generalised Arc Consistency* (GAC) [4] and *mixed consistency* [10].

A problem is said to be *generalised arc consistent* if it has non-empty domains and for any assignment of a variable each constraint in which that variable is involved can be satisfied. A problem is said to be *singleton generalised arc consistent* [11] if it has non-empty domains and for any assignment of a variable, the resulting subproblem can be made GAC. Enforcing Singleton generalised Arc Consistency (SGAC) in a SAC-1 manner [12] works by having an outer loop consisting of variable-value pairs of the form (x, a) . For each a in the domain of x if there is a domain wipeout while enforcing arc consistency then a is removed from the domain of x and arc consistency is enforced. The main problem with SAC-1 is that deleting a single value triggers the outer loop again. The Restricted SAC (RSAC) algorithm avoids this triggering by considering each variable-value pair only once [13].

Mixed consistency means maintaining different levels of consistency on different variables of a problem. In [14] it has been shown that maintaining mixed consistency, in particular maintaining SAC on some variables and maintaining arc consistency on some variables, can reduce the solution time for some CSPs. In this paper we shall study the effect of maintaining different levels of consistency on different sets of variables within a branch and bound search. We shall investigate the effect of Maintaining generalised Singleton Arc Consistency (MGSAC) on the Boolean variables and Maintaining generalised Arc Consistency (MGAC) on the remaining variables of the problem. We shall also investigate the effect of Maintaining Restricted Singleton generalised Arc Consistency (MRSGAC) on the Boolean variables and MGAC on the remaining variables. The

former shall be denoted by MSGAC_b and the latter by MRGSAC_b . Results presented in Section 6 suggest that maintaining singleton generalised arc consistency on the Boolean variables of the random instances of the feature subscription configuration problem reduces the search space and time of the branch and bound algorithm significantly.

5 Other Approaches

We present a partial weighted maximum Boolean satisfiability and an integer linear programming formulation for finding an optimal relaxation of a feature subscription.

5.1 Partial Weighted Maximum Boolean Satisfiability

The Boolean Satisfiability Problem (SAT) is a decision problem whose instance is an expression in propositional logic written using only $\wedge$, $\vee$, $\neg$, variables and parentheses. The problem is to decide whether there is an assignment of *true* and *false* values to the variables that will make the expression *true*. The expression is normally written in conjunctive normal form. The Partial Weighted Maximum Boolean Satisfiability Problem (PWMSAT) is an extension of SAT that includes the notions of hard and soft clauses. Any solution should respect the hard clauses. Soft clauses are associated with weights. The goal is to find an assignment that maximises the sum of the weights of the satisfied clauses. The PWMSAT formulation of finding an optimal relaxation of a feature subscription $\langle F, C, U, W_F, W_U \rangle$ is outlined below.

Variables. Let PrecDom be the set of possible precedence constraints that can be defined on F , i.e., $\{f_i \prec f_j : \{f_i, f_j\} \subseteq F \wedge f_i \neq f_j\}$. For each feature $f_i \in F$ there is a Boolean variable bf_i , which is true or false depending on whether feature f_i is included or not in the computed subscription. For each precedence constraint p_{ij} there is a Boolean variable bp_{ij} , which is true or false depending on whether the precedence constraint $f_i \prec f_j$ holds or not in the computed subscription.

Clauses. In our model, clauses are represented with a tuple $\langle w, c \rangle$, where w is the weight of clause and c is the clause itself. Note that the hard clauses are associated with weight $\top$, which represents an infinite penalty for not satisfying the clause. Each precedence constraint $p_{ij} \in C$ must be satisfied if the features f_i and f_j are included in the computed subscription. We model this by adding the following clause

$$\langle \top, (\neg bf_i \vee \neg bf_j \vee bp_{ij}) \rangle.$$

The precedence relation should be transitive and asymmetric in order to ensure that the subscription graph is acyclic. In order to ensure this, for every $\{p_{ij}, p_{jk}\} \subseteq \text{PrecDom}$, we add the following clause:

$$\langle \top, (\neg bp_{ij} \vee \neg bp_{jk} \vee bp_{ik}) \rangle. \quad (1)$$

Note that Rule (1) need only be applied to $\langle i, j, k \rangle$ such that $i \neq k$ because of Rule (2) below. In our model, both bp_{ij} and bp_{ji} can be false. However, if one of them is true

the other one is false. As this should be the case for any precedence relation, we add the following clause for every $p_{ij} \in \text{PrecDom}$:

$$\langle \top, (\neg bp_{ij} \vee \neg bp_{ji}) \rangle. \quad (2)$$

We make sure that each precedence constraint $p_{ij} \in \text{PrecDom}$ is only satisfied when its features are included by considering the following clauses:

$$\langle \top, (bf_i \vee \neg bp_{ij}) \rangle \quad \langle \top, (bf_j \vee \neg bp_{ij}) \rangle.$$

We need to penalise any solution that does not include a feature $f_i \in F$ or a user precedence constraint $p_{ij} \in U$. This is done by adding the following clauses:

$$\langle wf_i, (bf_i) \rangle \quad \langle wp_{ij}, (bp_{ij}) \rangle,$$

where $wf_i = W_F(f_i)$ and $wp_{ij} = W_U(\langle f_i, f_j \rangle)$. The cost of violating these clauses is the weight of the feature f_i and the weight of the precedence constraint p_{ij} respectively.

The number of Boolean variables in the PWMSAT model (approximately $|F|^2$) is greater than the number of Boolean variables in the CP model ($|F| + |U|$). These extra variables are used by Rule (1) and (2) to avoid cycles in the final subscription graph. We remark that the subscription contains a cycle if and only if the transitive closure of $C \cup U$ contains a cycle. Therefore, it is sufficient to associate Boolean variables only with the precedence constraints in the transitive closure of $C \cup U$. Reducing these variables will also reduce the transitive clauses, especially when the input subscription graph is not dense. Otherwise, Rule (1) will generate $|F| \times (|F| - 1) \times (|F| - 2)$ transitivity clauses. For example, for the subscription $\langle F, C, U, W_F, W_U \rangle$ with $F = \{f_1, f_2, f_3, f_4, f_5, f_6\}$, $C = \{p_{12}, p_{21}, p_{34}, p_{43}, p_{56}, p_{65}\}$, and $U = \emptyset$, Rule (1) will generate 120 transitive clauses. Since any relaxation of the given subscription respecting the clauses generated by Rule (2) is acyclic, the 120 transitive clauses are useless. Thus, if PrecDom is instead set to be the transitive closure of $C \cup U$, then Rule (1) would not generate any clause for the mentioned example. Another way to reduce the number of transitive clauses is by not considering the ones where $\{p_{ji}, p_{kj}, p_{ik}\} \cap C \neq \emptyset$, especially when the input subscription graph is not sparse. The reason is that these transitive clauses are always entailed due to the enforcement of the catalogue precedence constraints.

Note that the two techniques described before for reducing the number of transitive clauses complement each other. This reduction in the number of clauses might have an impact on the runtime of the PWMSAT approach, since less memory might be needed. Even though it is sufficient to associate a Boolean variable with each precedence constraint in the transitive closure of $C \cup U$, it is still greater than $|F| + |U|$. Another way of reducing the number of variables is to associate a feature with a finite domain variable representing its position (as done in the CP model), log-encode the finite domain variables, and express the precedence constraints using a lexicographical comparator [15]. This approach indeed uses fewer variables than the implemented approach since only $|F| \times \log |F|$ variables are needed for encoding the position variables. However, it is not so straightforward to automatically translate the resulting Boolean formula into its corresponding conjunctive normal form.

5.2 Integer Linear Programming

In Linear Programming the goal is to optimise an objective function subject to linear equality and inequality constraints. When all the variables are forced to be integer-valued, the problem is an Integer Linear Programming (ILP) problem. The standard way of expressing these problems is by presenting the function to be optimised, the linear constraints to be respected and the domain of the variables involved. Both the CP and the PWMSAT formulations for finding an optimal relaxation of a feature subscription $\langle F, C, U, W_F, W_U \rangle$ can be modeled in ILP. The translation of the PWMSAT formulation into ILP formulation is straightforward. For this particular model, we observed that CPLEX was not able to solve even simple problems within a time limit of 4 hours. Due to the lack of space we shall describe neither the formulation nor its corresponding results. The ILP formulation that is equivalent to the CP formulation is outlined below.

Variables. For each $f_i \in F$, we use a binary variable bf_i and an integer variable pf_i . A binary variable bf_i is equal to 1 or 0 depending on whether feature f_i is included or not. An integer variable pf_i is the position of feature f_i in the final subscription. For each user precedence constraint $p_{ij} \in U$, we use a binary variable bp_{ij} . It is instantiated to 1 or 0 depending on whether the precedence constraint $f_i \prec f_j$ holds or not.

Linear Inequalities. If the features f_i and f_j are included in the computed subscription and if $p_{ij} \in C$ then the position of feature f_i must be less than the position of feature f_j . To this effect, we need to translate the underlying implication $(bf_i \wedge bf_j \Rightarrow (pf_i < pf_j))$ into the following linear inequality:

$$pf_i - pf_j + n * bf_i + n * bf_j \leq 2n - 1. \quad (3)$$

Here, n is a constant that is used to refer to the number of features $|F|$ selected by the user. When both bf_i and bf_j are 1, Inequality (3) will force $(pf_i < pf_j)$. Note that this is not required for any user precedence constraint $p_{ij} \in U$, since it can be violated.

A user precedence $p_{ij} \in U$ is equivalent to the implication $bp_{ij} \Rightarrow pf_i < pf_j \wedge bf_i \wedge bf_j$, which in turn is equivalent to the conjunction of the three implications $(bp_{ij} \Rightarrow (pf_i < pf_j))$, $(bp_{ij} \Rightarrow bf_i)$ and $(bp_{ij} \Rightarrow bf_j)$. These implications can be translated into the following inequalities:

$$pf_i - pf_j + n * bp_{ij} \leq n - 1 \quad (4)$$

$$bp_{ij} - bf_i \leq 0 \quad (5)$$

$$bp_{ij} - bf_j \leq 0. \quad (6)$$

Inequality (4) means that $bp_{ij} = 1$ forces $pf_i < pf_j$ to be true. Also, if $bp_{ij} = 1$ then both bf_i and bf_j are equal to 1 from Inequalities (5) and (6) respectively.

Objective Function. The objective is to find an optimal relaxation of a feature subscription configuration problem $\langle F, C, U, W_F, W_U \rangle$ that maximises the sum of the weights of the features and the user precedence constraints that are selected:

$$\text{Maximise } \sum_{f_i \in F} wf_i bf_i + \sum_{p_{ij} \in U} wp_{ij} bp_{ij}.$$

6 Experimental Results

In this section, we shall describe the empirical evaluation of finding an optimal relaxation of randomly generated feature subscriptions using constraint programming, partial weighted maximum Boolean satisfiability and integer linear programming.

6.1 Problem Generation and Solvers

We generated and experimented with a variety of *random catalogues* and many classes of *random feature subscriptions*. All the random selections below are performed with uniform distributions. A random catalogue is defined by a tuple $\langle f_c, B_c, T_c \rangle$. Here, f_c is the number of features, B_c is the number of binary constraints and $T_c \subseteq \{<, >, <>\}$ is a set of types of constraints. Note that $f_i <> f_j$ means that in any given subscription both f_i and f_j cannot exist together. A random catalogue is generated by selecting B_c pairs of features randomly from $f_c(f_c - 1)/2$ pairs of features. Each selected pair of features is then associated with a type of constraint that is selected randomly from T_c . A random feature subscription is defined by a tuple $\langle f_u, p_u, w \rangle$. Here, f_u is the number of features that are selected randomly from f_c features, p_u is the number of user precedence constraints between the pairs of features that are selected randomly from $f_u(f_u - 1)/2$ pairs of features, and w is an integer greater than 0. Each feature and each user precedence constraint is associated with an integer weight that is selected randomly between 1 and w inclusive.

We generated catalogues of the following forms: $\langle 50, 250, \{<, >\} \rangle$, $\langle 50, 500, \{<, >, <>\} \rangle$ and $\langle 50, 750, \{<, >\} \rangle$. For each random catalogue, we generated classes of feature subscriptions of the following forms: $\langle 10, 5, 4 \rangle$, $\langle 15, 20, 4 \rangle$, $\langle 20, 10, 4 \rangle$, $\langle 25, 40, 4 \rangle$, $\langle 30, 20, 4 \rangle$, $\langle 35, 35, 4 \rangle$, $\langle 40, 40, 4 \rangle$, $\langle 45, 90, 4 \rangle$ and $\langle 50, 5, 4 \rangle$. Note that $\langle 50, 250, \{<, >\} \rangle$ is the default catalogue by and the value of w is 4 by default, unless stated otherwise. For the catalogue $\langle 50, 250, \{<, >\} \rangle$ we also generated $\langle 5, 0, 1 \rangle$, $\langle 10, 0, 1 \rangle$, $\dots$, $\langle 50, 0, 1 \rangle$ and $\langle 5, 5, 1 \rangle$, $\langle 10, 10, 1 \rangle$, $\dots$, $\langle 50, 50, 1 \rangle$ classes of random feature subscriptions. For each class 10 instances were generated and their mean results are reported in this paper.

The CP model was implemented and solved using CHOCO [16], a Java library for constraint programming systems. The PWMSAT model of the problem was implemented and solved using SAT4J [17], an efficient library of SAT solvers in Java. The ILP model of the problem was solved using ILOG CPLEX [18]. All the experiments were performed on a PC Pentium 4 (CPU 1.8 GHZ and 768MB of RAM) processor. The performances of all the approaches are measured in terms of search nodes (#nodes) and runtime in milliseconds (time). We used the time limit of 4 hours to cut the search.

6.2 Maintaining Different Levels of Consistency in CP

For the CP model, we first investigated the effect of Maintaining generalised Arc Consistency (MGAC) during branch and bound search. We then studied the effect of maintaining different levels of consistency on different sets of variables within a problem. In particular we investigated, (1) maintaining generalised singleton arc consistency on the Boolean variables and MGAC on the remaining variables, and (2) maintaining restricted

singleton generalised arc consistency on the Boolean variables and MGAC on the remaining variables; the former is denoted by MSGAC_b and the latter by MRSGAC_b . The results are presented in Table 1 for these three branch and bound search algorithms.

Table 1 clearly shows that maintaining (R)SGAC on the Boolean variables and GAC on the integer variables dominates maintaining GAC on all the variables. To the best of our knowledge this is the first time that such a significant improvement has been observed by maintaining a partial form of singleton arc consistency. We also see that there is no difference in the number of nodes visited by MRSGAC_b and MSGAC_b for the first two classes of feature subscriptions. However, as the problem size increases the difference in terms of the number of nodes also increases significantly. Note that in the remainder of the paper the results that correspond to the CP approach are obtained by using MSGAC_b algorithm.

Table 1. Average results of MGAC, MRSGAC_b and MSGAC_b

$\langle f, p \rangle$	MGAC		MRSGAC_b		MSGAC_b	
	time	#nodes	time	#nodes	time	#nodes
$\langle 10, 5 \rangle$	17	21	23	16	26	16
$\langle 15, 20 \rangle$	92	726	34	41	42	41
$\langle 20, 10 \rangle$	203	1,694	39	47	50	46
$\langle 25, 40 \rangle$	14,985	88,407	595	187	678	169
$\langle 30, 20 \rangle$	6,073	29,211	653	184	768	161
$\langle 35, 35 \rangle$	124,220	481,364	7,431	1,279	8,379	1,093
$\langle 40, 40 \rangle$	1,644,624	5,311,838	67,798	9,838	76,667	8,475

6.3 Comparison between the Alternative Approaches

The performances of using constraint programming (CP), partial weighted maximum satisfiability (PWMSAT) and integer linear programming (CPLEX) approaches are presented in Tables 2 and 3. If any approach failed to find and prove an optimal relaxation within a time limit of 4 hours then that time limit is used as the runtime of the algorithm and the number of nodes visited in that time limit is used as the number of nodes of the algorithm in order to compute the average runtime and average search nodes of a given problem class. In the tables, the column labelled as #us is used to denote the number of instances for which the time limit was exceeded. If this column is not present for any approach then it means that all the instances of all the problem classes were solved within the time limit. In general finding an optimal relaxation is NP-hard. Therefore, we need to investigate which approach can do it in reasonable time.

Tables 2 and 3 suggest that our CP approach performs better than our ILP and PWM-SAT approaches. Although in very few cases the CP approach is outperformed by the other two approaches, it performs significantly better in all other cases. Nevertheless,

Table 2. Catalogue $\langle 50, 250, \{<, >\} \rangle$

$\langle f, p \rangle$	optimal value	PWMSAT			CPLEX			CP	
		#nodes	time	#us	#nodes	time	#us	#nodes	time
$\langle 10, 5 \rangle$	36	167	345	0	0	11	0	16	23
$\langle 15, 20 \rangle$	69	721	1,039	0	51	61	0	41	34
$\langle 20, 10 \rangle$	62	1,295	1,619	0	50	47	0	47	39
$\langle 25, 40 \rangle$	115	5,039	4,391	0	3,482	1,945	0	187	595
$\langle 30, 20 \rangle$	93	5,415	6,397	0	1,901	1,025	0	184	653
$\langle 35, 35 \rangle$	118	30,135	23,955	0	35,247	22,763	0	1,279	7,431
$\langle 40, 40 \rangle$	123	186,913	282,760	0	299,829	247,140	0	9,838	67,798
$\langle 45, 90 \rangle$	173	6,291,957	12,638,251	8	5,280,594	7,690,899	2	104,729	1,115,515
$\langle 50, 4 \rangle$	96	165,928	195,717	0	1,164,755	1,010,383	0	60,292	413,611

it is also true that a remarkable improvement in our CP approach is due to maintaining (restricted) singleton arc consistency on the Boolean variables. For example, for feature subscription $\langle 40, 40 \rangle$ and catalogue $\langle 50, 250, \{<, >\} \rangle$ constraint programming (with MSGAC_b), on average, requires approximately only 1 minute whereas MGAC requires approximately half an hour.

The CP approach solved all the instances within the time limit. CPLEX could not solve 2 instances. More precisely, it could not prove their optimality within the time limit. SAT4J exceeded the time limit for 9 instances. This could be a consequence of $\mathcal{O}(n^3)$ transitive clauses, where $n = |F|$. Figure 3 depicts a plot between the number of clauses and the runtime of SAT4J. This plot clearly suggests that the runtime of SAT4J increases as the number of clauses increases. The high number of clauses restricts the scalability of the PWMSAT approach. For large instances SAT4J also runs out of the default memory (64MB). For instance, for catalogue $\langle 50, 250, \{<, >\} \rangle$ and feature subscription $\langle 45, 90 \rangle$, SAT4J runs out of memory when solving one of the instances. Note that the results for SAT4J presented in this section correspond to the instances that are generated after reducing the variables and the clauses by applying the techniques described in Section 5.1. The application of these techniques reduces the runtime up to 65%. However, this only enabled one of the previously unsolvable instances to be solved.

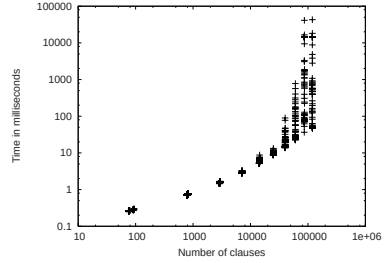


Fig. 3. Clauses vs Time

Figure 4 presents the comparison of the different approaches in terms of their runtimes for the subscriptions, when $U = \emptyset$ and the weight of each feature is 1. The runtimes of the approaches for the instances when $|F| = |U|$ are presented in Figure 5. Overall, the CP approach performs best. Although, the SAT4J solver performs best when $|F| > 35$ and $U = \emptyset$, it would be interesting to find out whether its performance will deteriorate when $|F| > 50$. In Figure 5, when $|F| = 50$, neither the ILP approach nor the PWMSAT approach managed to solve all the instances. This is the reason that their average runtimes, for the case of 50 features, are close to the timeout. If the timeout was higher, the gap between the CP approach and the other approaches, for the case of 50 features in Figure 5 would be even more significant.

Table 3. Results for more dense catalogues

$\langle f, p \rangle$	Catalogue $\langle 50, 500, \{<, >, <>\} \rangle$						Catalogue $\langle 50, 750, \{<, >\} \rangle$					
	PWMSAT		CPLEX		CP		PWMSAT		CPLEX		CP	
	#nodes	time	#nodes	time	#nodes	time	#nodes	time #us	#nodes	time	#nodes	time
$\langle 10, 5 \rangle$	326	528	0	10	13	3	246	500 0	28	33	16	7
$\langle 15, 20 \rangle$	1,066	1,173	4	53	31	28	1,111	985 0	306	261	40	45
$\langle 20, 10 \rangle$	2,583	1,981	18	85	49	59	2,484	1,542 0	798	540	82	145
$\langle 25, 40 \rangle$	5,753	2,961	76	554	110	250	6,904	3,158 0	7,043	5,741	236	910
$\langle 30, 20 \rangle$	9,738	4,092	90	447	158	417	11,841	5,025 0	22,253	18,461	591	2,381
$\langle 35, 35 \rangle$	12,584	6,841	300	1,824	461	1,643	31,214	18,278 0	109,472	126,354	2,288	12,879
$\langle 40, 40 \rangle$	22,486	11,310	711	3,018	892	3,914	68,112	92,105 0	354,454	514,275	6,363	42,268
$\langle 45, 90 \rangle$	60,504	59,267	2,130	17,452	2,286	14,803	602,192	2,443,228 1	1,969,716	3,780,539	19,909	188,826
$\langle 50, 4 \rangle$	43,765	21,472	1,500	3,771	4,208	16,921	184,584	319,531 0	1,646,752	3,162,084	51,063	342,492

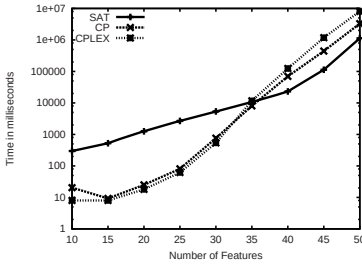


Fig. 4. Results for $\langle f_u, 0, 1 \rangle$, where f_u varies from 5 to 50 in steps of 5

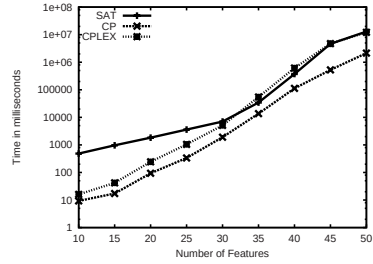


Fig. 5. Results for $\langle f_u, p_u, 1 \rangle$, where $f_u = p_u$ and f_u varies from 5 to 50 in steps of 5

7 Conclusions

We presented, and evaluated, three optimisation-based approaches to finding optimal reconfigurations of call-control features when the user's preferences violate the technical constraints defined by a set of DFC rules. We proved that finding an optimal relaxation of a feature subscription is NP-hard. For the constraint programming approach, we studied the effect of maintaining generalised arc consistency and two mixed consistencies during branch and bound search. Our experimental results suggest that maintaining (restricted) generalised singleton arc consistency on the Boolean variables and generalised arc consistency on the integer variables outperforms MGAC significantly. Our results also suggest that the CP approach when applied with stronger consistency, is able to scale well compared to the other approaches. Finding an optimal relaxation for a reasonable size catalogue (e.g., [19] refers to a catalogue with up to 25 features) is feasible using constraint programming.

Acknowledgements. This material is based upon works supported by the Science Foundation Ireland under Grant No. 05/IN/I886, and Embark Post Doctoral Fellowships No. CT1080049908 and No. CT1080049909. The authors would also like to thank Hadrien Cambazard, Daniel Le Berre and Alan Holland for their support in using CHOCO, SAT4J and CPLEX respectively.

References

1. Bond, G.W., Cheung, E., Purdy, H., Zave, P., Ramming, C.: An Open Architecture for Next-Generation Telecommunication Services. *ACM Transactions on Internet Technology* 4(1), 83–123 (2004)
2. Jackson, M., Zave, P.: Distributed Feature Composition: a Virtual Architecture for Telecommunications Services. *IEEE TSE* 24(10), 831–847 (1998)
3. Jackson, M., Zave, P.: The DFC Manual. AT&T (November 2003)
4. Bessiere, C.: Constraint propagation. Technical Report 06020, LIRMM, Montpellier, France (March 2006)
5. Zave, P., Goguen, H., Smith, T.M.: Component Coordination: a Telecommunication Case Study. *Computer Networks* 45(5), 645–664 (2004)

6. Zave, P.: An Experiment in Feature Engineering. In: McIver, A., Morgan, C. (eds.) *Programming Methodology*, pp. 353–377. Springer, Heidelberg (2003)
7. Cormen, T., Leiserson, C., Rivest, R.: *Introduction to Algorithms*. The MIT Press, Cambridge (1990)
8. Garey, M., Johnson, D.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York (1979)
9. Wallace, M.: Practical applications of constraint programming. *Constraints Journal* 1(1), 139–168 (1996)
10. Dooms, G., Deville, Y., Dupont, P.: CP(Graph): Introducing a graph computation domain in constraint programming. In: van Beek, P. (ed.) *CP 2005*. LNCS, vol. 3709, pp. 211–225. Springer, Heidelberg (2005)
11. Bessière, C., Stergiou, K., Walsh, T.: Domain filtering consistencies for non-binary constraints. *Artificial Intelligence* (2007)
12. Debruyne, R., Bessière, C.: Some practical filtering techniques for the constraint satisfaction problem. In: *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, Nagoya, Japan, pp. 412–417 (1997)
13. Prosser, P., Stergiou, K., Walsh, T.: Singleton Consistencies. In: *CP 2000*, pp. 353–368 (September 2000)
14. Lecoutre, C., Patrick, P.: Maintaining singleton arc consistency. In: *Proceedings of the 3rd International Workshop on Constraint Propagation And Implementation (CPAI 2006) held with CP 2006*, Nantes, France, pp. 47–61 (September 2006)
15. Hawkins, P., Lagoon, V., Stuckey, P.J.: Solving set constraint satisfaction problems using ROBDDs. *Journal of Artificial Intelligence Research* 24, 109–156 (2005)
16. Laburthe, F., Jussien, N.: JChoco: A java library for constraint programming
17. Le Berre, D.: SAT4J: An efficient library of SAT solvers in java
18. ILOG: CPLEX solver 10.1, <http://www.ilog.com/products/cplex/>
19. Bond, G.W., Cheung, E., Goguen, H., Hanson, K.J., Henderson, D., Karam, G.M., Purdy, K.H., Smith, T.M., Zave, P.: Experience with Component-Based Development of a Telecommunication Service. In: Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyper-ski, C.A., Wallnau, K. (eds.) *CBSE 2005*. LNCS, vol. 3489, pp. 298–305. Springer, Heidelberg (2005)

Protein Structure Prediction with Large Neighborhood Constraint Programming Search

Ivan Dotu¹, Manuel Cebrián¹, Pascal Van Hentenryck¹, and Peter Clote²

¹ Department of Computer Science, Brown University, Box 1910, Providence, RI 02912

² Biology Department, Boston College, Chestnut Hill, MA 02467

Abstract. Protein structure predictions is regarded as a highly challenging problem both for the biology and for the computational communities. Many approaches have been developed in the recent years, moving to increasingly complex lattice models or even off-lattice models. This paper presents a Large Neighborhood Search (LNS) to find the native state for the Hydrophobic-Polar (HP) model on the Face Centered Cubic (FCC) lattice or, in other words, a self-avoiding walk on the FCC lattice having a maximum number of H-H contacts. The algorithm starts with a tabu-search algorithm, whose solution is then improved by a combination of constraint programming and LNS. This hybrid algorithm improves earlier approaches in the literature over several well-known instances and demonstrates the potential of constraint-programming approaches for *ab initio* methods.

1 Introduction

In 1973, Nobel laureate C.B. Anfinsen [2] denatured the 124 residue protein, bovine ribonuclease A, by the addition of urea. Upon removal of the denaturant, the ribonuclease, an enzyme, was determined to be fully functional, thus attesting the successful reformation of functional 3-dimensional structure. Since no chaperone molecules were present, Anfinsen's experiment was interpreted to mean that the native state of a protein is its minimum free energy conformation, and hence that protein structure determination is a computational problem which can in principle be solved by applying a combinatorial search strategy to an appropriate energy model.

Protein structure prediction is historically one of the oldest, most important, yet stubbornly recalcitrant problems of bioinformatics. Solution of this problem would have an enormous impact on medicine and the pharmaceutical industry, since successful tertiary structure prediction, given only the amino acid sequence information, would allow the computational screening of potential drug targets, in that a drug (small chemical ligand) must dock to a complementary portion of the protein surface (such as a G-coupled protein receptor, the most common drug target)¹ of successful drug. Indeed, it has been stated that: "Prediction of protein structure *in silico* has thus been the 'holy grail' of computational biologists for many years" [39]. Despite the quantity of work on this problem over the past 30 years, and despite the variety of methods developed for structure prediction, no truly accurate *ab initio* methods exist to predict the

¹ The design of HIV protease inhibitors, first described in [29], exploited the target structure.

3-dimensional structure from amino acid sequence. Indeed, Helles (2008) [24] benchmarked the accuracy of 18 *ab initio* methods, whose average normalized root mean square deviation ranged from 11.17 Å to 3.48 Å, while Dalton and Jackson (2007) [19] similarly benchmarked five well-known homology modeling programs and three common sequence-structure alignment programs. In contrast, computational drug screening requires atomic scale accuracy, since the size of a single water molecule is about 1.4 Å.

In this paper, we describe a combination of constraint programming and Large Neighborhood Search (LNS) to determine close-to-optimal conformations for the Lau-Dill HP-model on the face-centered cubic lattice. Before describing our contribution, we first present an overview of computational methods for protein structure prediction. In general, methods are classified as *homology (comparative) modeling*, *threading*, *lattice model*, and *ab initio*. Protein structure prediction is an immense field that cannot be adequately surveyed in this introduction. Numerous books (e.g., [50]) and excellent reviews, (e.g., [21]) are available. Nevertheless, to situate the contribution of our work within the broader scope of protein structure prediction, we briefly describe each of the methods – homology, threading, *ab initio* –, before focusing on lattice models.

In homology modeling, the amino acid sequence of a novel protein P is aligned against sequences of proteins Q , whose tertiary structure is available in the Protein Data Bank (PDB) [10]. Regions of P aligned to regions of Q are assumed to have the same fold, while non-aligned regions are modeled by interconnecting loops. Examples of comparative modeling software are SWISS-MODEL, developed by M. Peitsch, T. Schwede et al., and recently described in [3], as well as MODELER developed by the Šali Lab [26]. Comparative modeling relies on the assumption that evolutionarily related (homologous) proteins retain high sequence identity and adopt the same fold.

Threading [31,40], though known to be NP-complete [30], is a promising *de novo* protein structure approach, which relies on *threading* portions $a_i, \dots, a_j$ of the amino acid sequence $a_1, \dots, a_n$ onto a *fragment library*, which latter consists of frequently adopted partial folds. Pseudo-energy (aka knowledge-based potential) is computed from the frequency of occurrence of certain folds with certain types of amino acid sequence. Impressive results have been obtained with the Skolnick Lab program I-TASSER [47] with web server [51], which yielded the best-ranked structure predictions in the blind test CASP-7 (Critical Assessment of Techniques for Protein Structure Prediction) in 2006. Success of threading hinges on two things: *energetics*, i.e., that the PDB is relatively saturated and contains occurrences of almost all protein folds, and *search strategy*, i.e., usually Monte-Carlo or some type of branch-and-bound algorithm. According to a study of Zhang and Skolnick [52], the PDB is currently sufficiently saturated to permit adequate threading approaches, albeit with insufficient accuracy for the requirements of computational drug design.²

Despite advances in comparative modeling and threading, there is an interest in *ab initio* protein structure prediction, since this is the only method that attempts to understand protein folding from basic principles, i.e., by applying a search strategy with (generally) a physics-based energy function. Moreover, only *ab initio* methods can be applied for proteins having no homology with proteins of known structure. In

² According to [52], using the TASSER algorithm, “in 408 cases the best of the top five full-length models has a RMSD < 6.5 Ångstroms.”

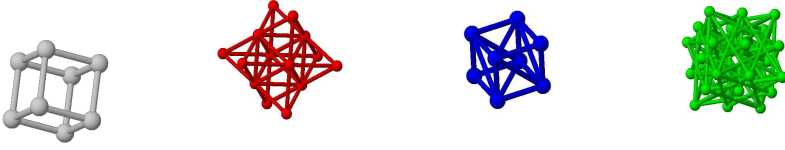


Fig. 1. Lattices used in protein structure modeling. (a) Points (x, y, z) in cubic lattice, satisfying $0 \leq x, y, z \leq 1$. (b) Points (x, y, z) in FCC lattice, satisfying $0 \leq x, y, z \leq 2$. (c) Points (x, y, z) in tetrahedral lattice, satisfying $0 \leq x, y, z \leq 1$. (d) Points (x, y, z) in 210 (knight's move) lattice, satisfying $0 \leq x, y, z \leq 2$.

molecular dynamics (MD), protein structure is predicted by iteratively solving Newton's equations for all pairs of atoms (possibly including solvent) using mean force potentials, that generally include pairwise (non-contact) terms for Lennard-Jones, electrostatic, hydrogen bonding, etc. Well-known MD software CHARMM [14] and Amber [20], as well as variant Molsoft ICM [1], the latter employing internal coordinates (dihedral angle space) and local optimization, are used to simulate protein docking, protein-ligand interactions, etc. since molecular dynamics generally is too slow to allow *ab initio* folding of any but the smallest proteins. Other *ab initio* methods include the Baker Lab program Rosetta [12], benchmarked in [24] with comparable accuracy as the Skolnick Lab program I-TASSER [47]. Search strategies of *ab initio* methods include molecular dynamics simulation, Metropolis Monte-Carlo (Rosetta [12]), Monte-Carlo with replica exchange (I-TASSER [47]), branch-and-bound (ASTROFOLD [27]), integer linear programming (ASTROFOLD [27]), Monte-Carlo with simulated annealing, evolutionary algorithms, and genetic algorithms.

2 Problem Formalization

A *lattice* is a discrete integer approximation to a vector space, formally defined to be the set of *integral* linear combinations of a finite set of vectors in $\mathbb{Z}^n$; i.e.,

$$L = \left\{ \sum_{i=1}^k a_i \mathbf{v}_i : a_i \in \mathbb{Z} \right\} \quad (1)$$

where $\mathbf{v}_1, \dots, \mathbf{v}_k \in \mathbb{Z}^n$. If k is the minimum value for which (1) holds, then $\mathbf{v}_1, \dots, \mathbf{v}_k$ form a *basis*, and k is said to be the *dimension* (also called *coordination* or *contact* number) of L . Two lattice points $p, q \in L$ are said to be *in contact* if $q = p + \mathbf{v}_i$ for some vector $\mathbf{v}_i$ in the basis of L . Historically, many different lattices have been considered, some of which are depicted in Figure 1. For more details on properties of these and other lattices, see the book by Conway and Sloane [16]. In this paper, we consider the face-centered cubic (FCC) lattice which is generated by the following 12 basis vectors (identified with compass directions [46]):

$$\begin{array}{lll} N : (1, 1, 0) & S : (-1, -1, 0) & W : (-1, 1, 0) \\ E : (1, -1, 0) & NW_+ : (0, 1, 1) & NW_- : (0, 1, -1) \\ NE_+ : (1, 0, 1) & NE_- : (1, 0, -1) & SE_+ : (0, -1, 1) \\ SW_+ : (-1, 0, 1) & SE_- : (0, -1, -1) & SW_- : (-1, 0, -1). \end{array}$$

	H	P
H	-1	0
P	0	0

	H	P	N	X
H	-4	0	0	0
P	0	+1	-1	0
N	0	-1	+1	0
X	0	0	0	0

Fig. 2. Energy for HP- and HPNX-model

It follows that the FCC lattice consists of all integer points (x, y, z) , such that $(x + y + z) \bmod 2 = 0$, and that lattice points $p = (x, y, z)$ and $q = (x', y', z')$ are in *contact*, denoted by $co(p, q)$, if $(x - x') + (y - y') + (z - z') \bmod 2 \equiv 0$, $|x - x'| \leq 1$, $|y - y'| \leq 1$, and $|z - z'| \leq 1$. We will sometimes state that lattice points p, q are at *unit distance*, when we formally mean that they are in contact. Since the distance between two successive alpha-carbon atoms is on average $3.8\text{\AA}$ with a standard deviation of $0.04\text{\AA}$, a reasonable coarse-grain approach is to model an n -residue protein by a self-avoiding walk $p_1, \dots, p_n$ on a lattice.

In 1972, Lau and Dill [32] proposed the *hydrophobic-hydrophilic* (HP) model, which provides a coarse approximation to the most important force responsible for the hydrophobic collapse which has been experimentally seen in protein folding. Amino acids are classified into either hydrophobic (e.g. Ala, Gly, Ile, Leu, Met, Phe, Pro, Trp, Val) or hydrophilic (e.g. Arg, Asn, Asp, Cys, Glu, Gln, His, Lys, Ser, Thr, Tyr) residues. In the HP-model, there is an energy of -1 contributed by any two non-consecutive hydrophobic residues that are in *contact* on the lattice. For this reason, the HP-model is said to have a contact potential, depicted in the left panel of Figure 2, where ‘H’ designates hydrophobic, while ‘P’ designates polar (i.e., hydrophilic). To account for electrostatic forces involving negatively charged residues (Asp, Glu) and positively charged residues (Arg, His, Lys), the HP-model has been extended to the HPNX-model, with hydrophobic (H), positively charged (P), negatively charged (N) and neutral hydrophilic (X) terms. The right panel of Figure 2 depicts the HPNX-potential used in [11].

Though Lau and Dill [32] originally considered only the 2-dimensional square lattice, their model allowed the formulation of the following simply stated combinatorial problem. For a given lattice and an arbitrary HP-sequence, determine a self-avoiding walk on the lattice having minimum energy, i.e., a minimum energy lattice conformation. This problem was shown to be NP-complete for the 2-dimensional square lattice by [17] and for the 3-dimensional cubic lattice by Berger and Leighton [9].

3 Related Work

Approaches to the HP Model. We first survey some search strategies for the HP-model. In [48], Yue and Dill applied “constraint-based exhaustive search”³ to determine the minimum energy conformation(s) of several small proteins including crambin, when represented as HP-sequences on the cubic lattice. Necessarily, any exhaustive search is limited to very small proteins, since the number of conformations for an n -mer on the 3-dimensional cubic lattice is estimated to be approximately 4.5^n [33]. In [43],

³ Despite the name, the method of Yue and Dill did not involve constraint programming.

Unger and Moulton described a genetic algorithm for the HP-model on the 2-dimensional square lattice, where pointwise mutation corresponds to a conformation pivot move. This approach was extended in Backofen, Will, and Clote [6] to a genetic algorithm on the FCC lattice, in order to quantify hydrophobicity in protein folding.

In [4,5,7], Backofen applied constraint programming to the HP-model and to the HPNX-model, thus providing an exact solution for small HP- and HPNX-sequences beyond the reach of exhaustive methods. In [8,45], Will and Backofen precomputed *hydrophobic cores*, maximally compact face-centered cubic self-avoiding walks of (only) hydrophobic residues. By threading an HP-sequence onto hydrophobic cores, the optimum conformation could be found for certain examples; however, if threading is not possible (which is often the case), no solution is returned.

Dal Palu et al. [18] use secondary structure and disulfide bonds used as constraints using constraint logic programming over finite domains to compute a predicted structure on the face-centered cubic lattice. They describe tests ranging from the 12 residue fragment (PDB code 1LE0) with RMSD of 2.8 Å achieved in 1.3 seconds, to the 63 residue protein (PDB code 1YPA) with RMSD of 17.1 Å in 10 hours. Further optimization was performed after the alpha-carbon trace was replaced by an all-atom model (presumably using well-known Holm-Sander method [25]), thus achieving an all-atom prediction of the 63 residue protein (PDB code 1YPA) with RMSD of 9.2 Å within 116.9 hours computation time. This study suggests that protein structure prediction might best proceed in a hierarchical fashion, first taking into account secondary structure on a coarse-grain lattice model and subsequently performing all-atom refinement.

Beyond the HP Model. The HP-model can be viewed as a coarse approximation of more complex *contact potentials*. In [35], Miyazawa and Jernigan introduced two kinds of contact potential matrices, i.e., 20×20 matrices that determine a residue-dependent energy potential to be applied in the case that two residues are in contact (either on the lattice, or within a fixed threshold such as 7 Å from each other). Recently, Pokarowski et al. [37] analyzed 29 contact matrices and showed that in essence all known contact potentials are one of the two types they introduced in [35]. Their first contact potential is given by the formula $e(i, j) = h(i) + h(j)$, where $1 \leq i \leq 20$ ranges over the 20 amino acids and h is a residue-type dependent factor that is highly correlated with frequency of occurrence of a given amino acid type in a non-redundant collection of proteins. Their second contact potential is given by the formula $e(i, j) = c_0 - h(i)h(j) + q(i)q(j)$, where c_0 is a constant, h is highly correlated with the Kyte-Doolittle hydrophobicity scale [28], and a residue-type dependent factor q is highly correlated isoelectric points pI. The “knight’s move” 210 lattice was used by Skolnick and Kolinski [41] to fold the 99-residue beta protein, apoplastocyanin, to within 2 Å of its crystal structure with PDB accession code 2PCY.

4 Why Constraint Programming?

Our earlier work [13] applied a tabu-search algorithm to obtain approximate solutions for protein folding for the HP-model on FCC lattice. The goal of this paper is to evaluate

a similar model using a large neighborhood search based on constraint programming, both to improve earlier results and to assess their quality. The improvements obtained by the CP-based LNS indicate that this approach provides significant benefits over a pure local search algorithm. More generally, as explained in the introduction, protein structure prediction can be viewed as the application of a search engine (Monte-Carlo, Monte-Carlo with replica exchange, genetic algorithm, integer programming, ...) to a physics-based or knowledge-based energy function. This paper evaluates CP-Based large neighborhood search on the Harvard instances, a standard benchmark for assessing accuracy of structure prediction for the HP-model. Our successful application of LNS to the face-centered cubic lattice suggests the potential of using this constraint-programming strategy in a hierarchical manner with successive refinements to perform all-atom structure prediction – a task for future research.

5 The Implementation

5.1 The CP Model

The CP model receives as input a sequence of binary values H_i ($0 \leq i < n$) denoting whether aminoacid i is hydrophobic ($H_i = 1$). Its output associates each aminoacid i with a point (x_i, y_i, z_i) in the FCC lattice. Recall that the FCC lattice is the closure of 12 vectors $V = \{v_0, \dots, v_{11}\}$ defined as follows:

$$\begin{aligned} v_0 &= \{1, 1, 0\} & v_1 &= \{-1, -1, 0\} & v_2 &= \{-1, 1, 0\} & v_3 &= \{1, -1, 0\} \\ v_4 &= \{1, 0, 1\} & v_5 &= \{-1, 0, -1\} & v_6 &= \{-1, 0, 1\} & v_7 &= \{1, 0, -1\} \\ v_8 &= \{0, 1, 1\} & v_9 &= \{0, -1, -1\} & v_{10} &= \{0, -1, 1\} & v_{11} &= \{0, 1, -1\}. \end{aligned}$$

Decision Variables. Although the output maps each aminoacid i into a FCC lattice point, the model uses move vectors as decision variables. These vectors (m_i^x, m_i^y, m_i^z) specify how to move from point $i - 1$ to point i in the self-avoiding walk. The use of *move variables* greatly simplifies the modeling: Self-avoidance is maintained through the lattice points, but move vectors along with a lexicographical variable ordering allow us to implicitly check chain connection and drastically reduces the search space.

The Domain Constraints. Each move variable (m_i^x, m_i^y, m_i^z) has a finite domain consisting of the FCC lattice vectors $\{v_0, \dots, v_{11}\}$, that is

$$(m_i^x, m_i^y, m_i^z) \in \{v_0, \dots, v_{11}\}.$$

Each coordinate x_i, y_i , and z_i in the 3D point (x_i, y_i, z_i) associated with aminoacid i has a finite domain $0..2n$.

The Lattice Constraints. The lattice constraints link the *move variables* and the points in the FCC lattice. They are specified as follows:

$$\forall 0 < i < n : x_i = x_{i-1} + m_i^x \ \& \ y_i = y_{i-1} + m_i^y \ \& \ z_i = z_{i-1} + m_i^z.$$

The model also uses the redundant constraints $(x_i + y_i + z_i) \bmod 2 = 0$ which are implied by the FCC lattice. In addition, the initial point is fixed.

The Self-Avoiding Walk Constraints. To express that all aminoacids are assigned different points in the FCC lattice, the model uses a constraint

$$abs(\sum_{k \in i..j} m_k^x) + abs(\sum_{k \in i..j} m_k^y) + abs(\sum_{k \in i..j} m_k^z) \neq 0$$

for each pair (i, j) of aminoacids, ensuring the moves from the position of aminoacid i do not place j at the same position as i . Indeed, the two points (x_i, y_i, z_i) and (x_j, y_j, z_j) are at the same position if each of the sums in the above expression is zero.

The Objective Function. The objective function maximizes the number of contacts between hydrophobic aminoacids $\sum_{i,j|i+1 < j} (d_{ij} = 2) \times H_i \times H_j$ where d_{ij} denotes the square of Euclidean distance between aminoacids i and j . Since the minimal distance in the FCC lattice is $\sqrt{2}$, the condition $d_{ij} = 2$ holds when there exists a contact between aminoacids i and j .

5.2 The Search Procedure

The search procedure assigns positions to the aminoacids in sequence by selecting moves in their domains. The only heuristic choice thus concerns which moves to select, which uniquely determines the position of the next aminoacid. In the course of this research, a number of move selection heuristics were evaluated. Besides the traditional lexicographic and random value selections, the heuristics included

1. **Minimizing the distance to the origin:** Choosing the move minimizing the distance of the corresponding aminoacid to the origin.
2. **Minimizing the distance to the centroid:** Choosing the move minimizing the distance of the corresponding aminoacid to the centroid.
3. **Maximizing density:** Choosing the move maximizing the density of the structure.
4. **Maximizing hydrophobic density:** Choosing the move that maximizing the density of the structure consisting only of the hydrophobic aminoacids.

The centroid of the conformation is defined as $(\frac{1}{n} \sum_{i=0}^{n-1} x_i, \frac{1}{n} \sum_{i=0}^{n-1} y_i, \frac{1}{n} \sum_{i=0}^{n-1} z_i)$. Most of the dedicated heuristics bring significant improvements in performance, although those minimizing the distance to the origin and the centroid seem to be most effective. Our implementation randomly selects one of the two heuristics.

5.3 Strengthening the Model during Search

We now describe a number of tightenings of the model which are applied during search. Their main benefit is to strengthen the bound on the objective function.

Linking FCC Moves and Distance Constraints. In the model described so far, the distance between two aminoacids ignores the fact that the points are placed on the FCC lattice. The model may be improved by deriving the fact that two aminoacids are necessarily placed at a distance greater than $\sqrt{2}$ and thus cannot be in contact. Such derived information directly improves the bound on the objective function.

However computing the possible distances between two aminoacids is quite complex in general. As a result, our constraint-programming algorithm only generates relevant distances each time a new aminoacid is positioned. More precisely, assuming that aminoacid i has just been positioned on the FCC lattice, the algorithm determines which unassigned aminoacids cannot be in contact with already placed aminoacids (only for H-type aminoacids). The key idea is to compute the shortest path sp_{ij} in the FCC lattice between aminoacid i and an already placed aminoacid j : It then follows that unassigned aminoacids $i + 1, \dots, i + sp_{ij} - 2$ cannot be in contact with j . Formally, after placing aminoacid i , the model is augmented with the constraints

$$\forall 0 \leq j \leq i - 2, i + 1 \leq l \leq i + sp_{ij} - 2 : d_{jl} > 2$$

which ensures that aminoacids j and l cannot be in contact.

Bounding the Number of Contacts. The expression of the objective function also does not take into account how the aminoacids are placed in the FCC lattice. As a result, it typically gives weak bounds on the objective value. This section shows how to bound the objective value at a search node more effectively.

The key idea to bound the objective value is to compute the maximum number of contacts for each unassigned aminoacid independently, thus ignoring their interactions through the self-avoiding walk. Consider a node of search tree where the sequence can be partitioned into the concatenation $A :: U$, where A is the subsequence of already positioned aminoacids in which i is the last assigned one (also, we only consider $a \in A \parallel H_a == 1$ and $k \in U \parallel H_k == 1$). The objective function can then be bounded by

$$obj \leq contact(A) + \sum_{k \in U} \min(maxContact(k), bcontact(k, A) + fcontact(k, U))$$

where $contact(A)$ denotes the number of contacts in subsequence A , $bcontact(k, A)$ bounds the number of contacts of an aminoacid $k \in U$ with those aminoacids in A , and $fcontact(k, U)$ bounds the number of contacts of k with those aminoacids in U occurring later in the sequence. The contacts of each aminoacid $k \in U$, $maxContact(k)$, are bounded by 10, since a point in the FCC lattice has 12 neighbors and there cannot be any contact between two successive aminoacid in the sequence. However, if $k == n - 1$, i.e., if k is the last aminoacid of the sequence then $maxContact(k) == 11$, since that k has no successor aminoacid.

To bound the contact of aminoacid k with A , the idea is to consider the neighbors of each aminoacid $a \in A$ and to find the one maximizing the contacts with k , i.e.,

$$\begin{aligned} bcontact(k, A) &= \max_{a \in A} bcontact(k, a, A) \\ bcontact(k, a, A) &= \#\{j \in A \mid j \in N(a) \wedge j \in R(k, a)\}. \end{aligned}$$

where $N(a)$ denotes the neighbors of aminoacid a and $R(k, a)$ denotes the aminoacid in A reachable from k , i.e., $R(k, A) = \{a \in A \mid sp_{ai} \leq (k - i) + 1\}$. Recall that i is the last aminoacid assigned. Finally, to bound the number of contacts of k with those aminoacids occurring later in the sequence, we use

$$fcontact(k, U) = \sum_{l \in U: l \geq k+2} H_l$$

```

1. LNS.PSP( $\sigma$ )
2.    $limit \leftarrow limit_0$ 
3.    $fraction \leftarrow fraction_0$ 
4.   for  $m$  iterations do
5.     uniform select  $i \in 1..n - 1$ 
6.      $size \leftarrow n \cdot fraction$ 
7.      $j \leftarrow i + size$ 
8.      $\langle \sigma^*, explored \rangle = \text{CPSolve}(\sigma, i..j, limit)$ 
9.     if  $\sigma^* \neq \perp$  then
10.       $\sigma \leftarrow \sigma^*$ 
11.       $limit \leftarrow limit_0$ 
12.       $fraction \leftarrow fraction_0$ 
13.     else if  $explored$  then
14.       $fraction \leftarrow fraction + \Delta fraction$ 
15.     else
16.       $limit \leftarrow limit + \Delta limit$ 
17.   return  $\sigma$ 

```

Fig. 3. LNS for Protein Structure Prediction ($limit_0=500$ failures, $fraction_0 = \frac{3}{100}$, $\Delta fraction = \frac{1}{1000}$ and $\Delta limit=100$ failures).

to count the number of hydrophobic aminoacids occurring later in U that can be in contact with k . This bound can be computed in time $O(n^2)$ and is quite tight when the number of aminoacids in U is reasonably small.

5.4 Large Neighborhood Search

Structure prediction is a highly complex combinatorial optimization problem. As a result, constraint programming search may spend considerable time in suboptimal regions of the search space. To remedy this limitation, our algorithm uses the idea of large neighborhood search (LNS) [38] which focuses on reoptimizing subparts of a solution. Given a feasible walk σ , the idea is to solve the structure prediction problem for a subsequence of the original sequence, assuming that the remaining aminoacids are positioned like in σ . More precisely, given an interval $i..j$, an LNS optimization step consists of solving the original model with the additional constraints

$$\forall k : 0 \leq k < i : x_i = \sigma(x_i) \wedge y_i = \sigma(y_i) \wedge z_i = \sigma(z_i)$$

$$\forall k : j < k < n : x_i = \sigma(x_i) \wedge y_i = \sigma(y_i) \wedge z_i = \sigma(z_i)$$

where $\sigma(x)$ denotes the value of variable x in solution σ .

The complete LNS algorithm is depicted in Figure 3. It receives as input a high-quality solution produced by the tabu-search algorithm described in [13] and uses a subroutine $\text{CPSolve}(\sigma, i..j, l)$ which solves augmented models using constraint programming and terminates after at most $limit$ failures had occurred or when the entire search space has been explored. It returns a pair $\langle \sigma^*, explored \rangle$, where σ^* is either a new best solution or $\perp$ if no such solution was found, and $explored$ is a boolean which is true when the entire search space has been explored for the augmented model. Lines

Table 1. Results for the Harvard instances

Seq.	Lowest LS E	median time	Lowest LNS E	time	Improvement %
H1	-68	114 sec.	-69	5.32 sec.	1.47
H2	-69	265 sec.	Not improv.		0
H3	-68	72 sec.	-71	28.64 sec.	4.41
H4	-66	44 sec.	-69	26.55 sec.	4.55
H5	-66	53 sec.	-67	4.18 sec.	1.52
H6	-70	149 sec.	Not improv.		0
H7	-68	8 sec.	-69	9.86 sec.	1.47
H8	-64	10 milise.	-65	18.3 sec.	1.56
H9	-69	89 sec.	Not improv.		0
H10	-66	30 sec.	-67	9.74 mins.	1.52

2–3 initialize two parameters: the limit on the number of failures and the fraction of the subsequence to (re)-position on the FCC lattice. Line 8 is the call to the constraint-programming solver. After this call there are three possibilities. First, that the search is successful: then the best solution is updated and the parameters are re-initialized (lines 9–12). Second, the search space has been explored entirely with no improvement; the fraction of the sequence to re-position is increased at a certain rate $\Delta fraction$ (lines 13–14). Finally, *CPSolve* reached *limit* without an improvement: the number of failures is increased in $\Delta limit$ to give it more time to succeed in the next trial (lines 15–16).

6 Experimental Results

All the results presented in this section have been produced by a COMET [34,44] implementation of the LNS algorithm, run on a single core of a 60 Intel based, dual-core, dual processor, Dell Poweredge 1855 blade server. Each blade has 8G of memory and a 300G local disk. Each of the considered benchmarks was run for about 48 hours.

The Harvard Instances. Reference [49] contains a comparison of several methods to fold 10 different proteins, called the "Harvard instances", on the cubic lattice. The cubic lattice has been heavily studied as pointed out in the introduction, but the FCC lattice has been shown to admit the tightest packing of spheres [15], indicating that it allows for more complex 3D structures. The first results for these instances on the FCC lattice were presented in [13] and confirmed that the FCC lattice allows for structures with much lower energy than the cubic lattice. Table 1 depicts the results of our hybrid algorithm, starting with a local-search algorithm and improving the result with LNS. Note that the energy shown in the table corresponds to minus the number of HH contacts. The LNS step improves 7 out of 10 solutions quickly. Since no complete search algorithms have been applied to these instances on the FCC lattice, the energy of the optimal structure is not known. However, given the consistency in the energies of all the sequences (which all have 48 aminoacids and 24 hydrophobic aminoacids), it is probably the case that these results are near-optimal.

Table 2. Results for the Will’s instances

Seq.	Native E	Lowest LS E	median time	Lowest LNS E	time	Improvement %
S1	-357	-325	15.98 min.	-346	1.61 hour	6.46
S2	-360	-315	19.18 min.	-343	4.48 hours	8.89
S3	-367	-307	1.14 min.	-341	54.18 mins.	11.07
S4	-370	-318	13.14 min.	-340	7.4 hours	6.92
R1	-384	-284	2.09 min.	-337	1.3 hours	18.66
R2	-383	-290	18.8 min.	-325	7.67 hours	12.07
R3	-385	-282	6.45 min.	-317	2.08 hours	12.41

Other Instances. We also evaluated our algorithm with the only FCC foldings available in the literature. Table 2 depicts a comparison for 7 instances found in [46]. All instances contain 100 H aminoacids, and the R instances have a total of 200 aminoacids, while the S instances range between 130 and 180 aminoacids. Table 2 also shows optimal energies for these instances. The results demonstrate that LNS significantly improves the local search algorithm, with improvements ranging from 6% to 18%. The largest improvements occur on the R instances, which is explained by the lower quality of local search for these instances. The results on the S instances are within 8% of the optimal solution, while the algorithm is within 18% of the optimal solutions on the R instances. Figure 4 depicts a 3D view of the best configuration found for S2 for the local search in [13], the LNS algorithm, and the native state.

It is also important to stress how the optimal solutions were obtained in [46]. Will’s algorithm solves a substantially different problem which consists of threading a sequence into a pre-calculated H core. The algorithm relies on a set of precomputed (optimal and suboptimal) cores and tries to map the protein on these cores. Such threading for the protein may not exist for any of these cores or may not be found within the given time limit, in which case the threading algorithm may not provide any solution. There is thus a fundamental conceptual difference between the algorithm presented in this paper and the hydrophobic-core constraint-programming method of Will and Backofen [8,45], which can be captured using the concepts of Monte-Carlo and Las Vegas algorithms from theoretical computer science [36]. Monte-Carlo algorithms always converge, but have a (small) probability of error in the solution proposed; in contrast, Las Vegas algorithms always return the correct solution, but have a (small) probability

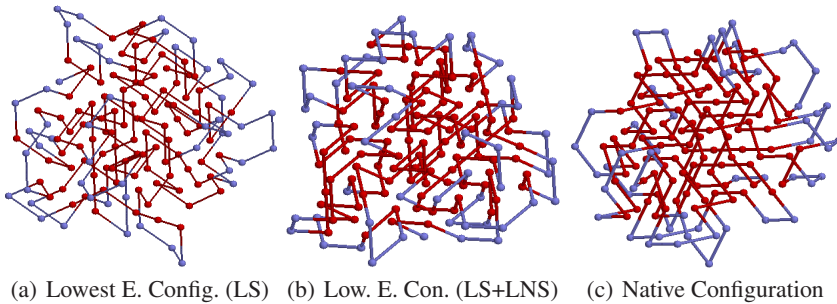


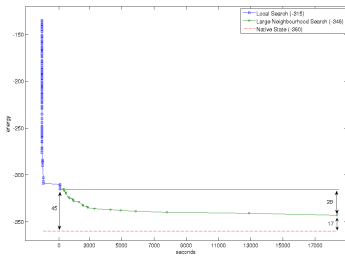
Fig. 4. Lowest Energy Configurations for Instance S2. Native is the Optimal Configuration

Table 3. LS and LNS over unsolved instances by Will's approach

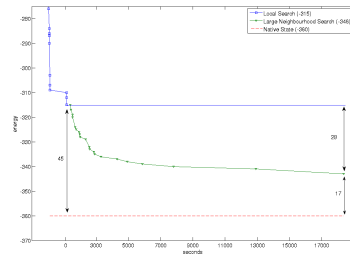
sequence	after 180 sec. of LS	180 sec. of LS + 5 min. of LNS	180 sec. of LS + 2 hours of LNS
F90_1	-127	-149	-157
F90_2	-136	-151	-154
F90_3	-131	-150	-156
F90_4	-134	-153	-157
F90_5	-132	-144	-148
sequence	after 300 sec. of LS	300 sec. of LS + 5 min. of LNS	300 sec. of LS + 2 hours of LNS
F160_1	-216	-231	-272
F160_2	-241	-264	-295
F160_3	-226	-242	-278

of not converging. By analogy, our approach (LNS with constraint programming) is akin to a Monte-Carlo method, in that an approximate solution is always returned. In contrast, hydrophobic-core constraint programming is akin to a Las Vegas method, in that any solution returned is an exact (optimal) solution; however, in many cases, the hydrophobic-core method fails to return any answer. Reference [46, p. 129] includes a table indicating that the threading algorithm only solves 50% of the instances with an H core of size 100 within the given time limit. The instances for which they report a solution are those which can be threaded in an optimal H core. These instances are heavily biased against our algorithm and none of the other sequences are available. Thus, a fair comparison of the algorithms is not possible at this stage, since only the above 7 sequences are available and they belong to the 50% the threading algorithm can solve.

Table 3 shows results on instances for which Will's approach did not yield any solution within the given time limits (180 secs for sequences with 90 aminoacids and 300 secs for sequences with 160 aminoacids). It can be seen how the local search achieves initial solutions which are then quickly improved by the LNS. Running LNS for longer time improves the solutions substantially, demonstrating the potential of this approach. Note also that Will's algorithm relies heavily on the definition of energy and it is hard to generalize to other energy models. Our algorithm solves the problem *ab-initio* and has the potential of obtaining near-optimal solution for general proteins. In addition, our approach is completely general and may encompass different notions of energies at very



(a) Behaviour over 5 hours of LS + LNS



(b) Zoom on LNS behaviour

Fig. 5. Algorithm Behavior over Time for Will's instance S2

small cost of implementation. Moreover, some preliminary results indicate that it can be applied to problems such as RNA structure prediction with minimal modifications.

Finally, figure 5 depicts the improvement of the solutions of our algorithm over time. The algorithm exhibits a steep descent, followed by a long plateau, and then another steep descent. It is interesting to see how the local search, the LNS (on their own) and the complete process (local search + LNS), they all present the same behavior.

7 Conclusions and Future Work

This paper presented an LNS algorithm for finding high-quality self avoiding walk for the Hydrophobic-Polar (HP) energy model on the Face Centred Cubic (FCC) lattice. The algorithm relies on a local search initial solution which is then improved by a constraint-programming LNS strategy. Experimental results on the standard Harvard instances show improvements over previously presented results, while significant improvements are achieved in other larger instances. The result shows that the hybridization of local search and constraint programming has great potential to approach the highly combinatorial problem of structure prediction.

The goal of our paper is to apply CP to compute approximations for solutions to instances of the NP-complete problem of protein structure prediction for the HP-model on the FCC lattice. Experimental results are meant only to benchmark the LNS algorithm. Our long term interest is the application of local search and CP to real biomolecular structure prediction. Bradley et al. [12] argue that protein structure prediction consists of two aspects: (1) a good search strategy (2) adequate fragment library. Skolnick and others have argued that due to the Structural Genome Initiative (high-throughput X-ray diffraction studies of proteins having less than 30% homology to any existent proteins), the fragment library is essentially currently adequate. While most search strategies (including that of Bradley, Misura and Baker) are Monte Carlo (possibly with simulated annealing, possibly with replicate exchange), our goal is to develop algorithms such as LNS that ultimately will play a role in biomolecular structure prediction. This is the ultimate justification of the current work.

Acknowledgements. Thanks to the reviewers for their useful comments and Sebastian Will for sharing his results. Ivan Dotú is supported by a “Fundacion Caja Madrid” grant and Manuel Cebrián by grant TSI 2005-08255-C07-06 of the Spanish Ministry of Education and Science.

References

1. Abagyan, R.A., Totrov, M.M., Kuznetsov, D.A.: ICM: a new method for structure modeling and design: Applications to docking and structure prediction from the distorted native conformation. *J. Comp. Chem.* 15, 488–506 (1994)
2. Anfinsen, C.B.: Principles that govern the folding of protein chains. *Science* 181 (1973)
3. Arnold, K., Bordoli, L., Kopp, J., Schwede, T.: The SWISS-MODEL workspace: a web-based environment for protein structure homology modelling. *Bioinformatics* 22(2) (2006)
4. Backofen, R.: The protein structure prediction problem: A constraint optimization approach using a new lower bound. *Constraints* 6(2-3), 223–255 (2001)

5. Backofen, R., Will, S., Bornberg-Bauer, E.: Application of constraint programming techniques for structure prediction of lattice proteins with extended alphabets. *Bioinformatics* 15(3), 234–242 (1999)
6. Backofen, R., Will, S., Clote, P.: Algorithmic approach to quantifying the hydrophobic force contribution in protein folding. In: *Pacific Symposium on Biocomputing*, vol. 5, pp. 92–103 (2000)
7. Backofen, R.: Using constraint programming for lattice protein folding. In: *Workshop on Constraints and Bioinformatics/Biocomputing* (1997)
8. Backofen, R., Will, S.: A constraint-based approach to structure prediction for simplified protein models that outperforms other existing methods. In: Palamidessi, C. (ed.) *ICLP 2003*. LNCS, vol. 2916, pp. 49–71. Springer, Heidelberg (2003)
9. Berger, B., Leighton, T.: Protein folding in the hydrophobic-hydrophilic (hp) model is NP-complete. *Journal of Computational Biology* 5, 27–40 (1998)
10. Berman, H.M., Battistuz, T., Bhat, T.N., Bluhm, W.F., Bourne, P.E., Burkhardt, K., Feng, Z., Gilliland, G.L., Iype, L., Jain, S., Fagan, P., Marvin, J., Padilla, D., Ravichandran, V., Schneider, B., Thanki, N., Weissig, H., Westbrook, J.D., Zardecki, C.: The Protein Data Bank. *Acta Crystallogr. D. Biol. Crystallogr.* 58(Pt), 899–907 (2002)
11. Bornberg-Bauer, E.: Chain growth algorithms for HP-type lattice proteins. In: *RECOMB*, pp. 47–55. ACM Press, New York (1997)
12. Bradley, P., Misura, K.M., Baker, D.: Toward high-resolution de novo structure prediction for small proteins. *Science* 309(5742), 1868–1871 (2005)
13. Cebrian, M., Dotu, I., Van Hentenryck, P., Clote, P.: Protein Structure Prediction on the Face Centered Cubic Lattice by Local Search. In: *AAAI 2008* (to appear, 2008)
14. Brooks, B.R., Bruccoleri, R.E., Olafson, B.D., States, D.J., Swaminathan, S., Karplus, M.: CHARMM: A program for macromolecular energy, minimization, and dynamics calculations. *J. Comput. Chem.* 4, 187–217 (1983)
15. Cipra, B.: Packing challenge mastered at last. *Science* 281, 1267 (1998)
16. Conway, J.H., Sloane, N.J.A.: *Sphere Packing, Lattices and Groups*. Springer, Heidelberg (1998)
17. Crescenzi, P., Goldman, D., Papadimitriou, C., Piccolboni, A., Yannakakis, M.: On the complexity of protein folding. *J. Comp. Biol.* 5(3), 523–466 (1998)
18. Dal Palu, A., Dovier, A., Fogolari, F.: Constraint Logic Programming approach to protein structure prediction. *BMC. Bioinformatics* 5, 186 (2004)
19. Dalton, J.A., Jackson, R.M.: An evaluation of automated homology modelling methods at low target template sequence similarity. *Bioinformatics* 23(15), 1901–1908 (2007)
20. Duan, Y., et al.: A point-charge force field for molecular mechanics simulations of proteins based on condensed-phase quantum mechanical calculations. *J. Comput. Chem.* 24(16), 1999–2012 (2003)
21. Floudas, C.A.: Computational methods in protein structure prediction. *Biotechnol. Bioeng.* 97(2), 207–213 (2007)
22. Go, N., Taketomi, H.: Respective roles of short- and long-range interactions in protein folding. *Proc. Natl. Acad. Sci. U.S.A.* 75(2), 559–563 (1978)
23. Go, N., Taketomi, H.: Studies on protein folding, unfolding and fluctuations by computer simulation. III. Effect of short-range interactions. *Int. J. Pept. Protein. Res.* 13(3) (1979)
24. Helles, G.: A comparative study of the reported performance of ab initio protein structure prediction algorithms. *J. R. Soc. Interface* 5(21), 387–396 (2008)
25. Holm, L., Sander, C.: Database algorithm for generating protein backbone and side-chain coordinates from a C α trace application to model building and detection of co-ordinate errors. *J. Mol. Biol.* 218(1), 183–194 (1991)
26. John, B., Sali, A.: Comparative protein structure modeling by iterative alignment, model building and model assessment. *Nucleic. Acids. Res.* 31(14), 3982–3992 (2003)

27. Klepeis, J.L., Floudas, C.A.: Prediction of β -sheet topology and disulfide bridges in polypeptides. *Journal of Computational Chemistry* 24(2), 191–208 (2002)
28. Kyte, J., Doolittle, R.F.: A simple method for displaying the hydropathic character of a protein. *J. Mol. Biol.* 157(1), 105–132 (1982)
29. Lam, P.Y., Jadhav, P.K., Eyermann, C.J., Hodge, C.N., Ru, Y., Bacheler, L.T., Meek, J.L., Otto, M.J., Rayner, M.M., Wong, Y.N., et al.: Rational design of potent, bioavailable, non-peptide cyclic ureas as HIV protease inhibitors. *Science* 263(5145), 380–384 (1994)
30. Lathrop, R.H.: The protein threading problem with sequence amino acid interaction preferences is NP-complete. *Protein. Eng.* 7(9), 1059–1068 (1994)
31. Lathrop, R.H., Smith, T.F.: Global optimum protein threading with gapped alignment and empirical pair score functions. *J. Mol. Biol.* 255(4), 641–665 (1996)
32. Lau, K.F., Dill, K.A.: A lattice statistical mechanics model of the conformational and sequence spaces of proteins. *Journal of the American Chemical Society* 22 (1989)
33. Madras, N., Slade, G.: *The Self-Avoiding Walk. Probability and its Applications*, 448 p. Birkhäuser, Boston (1996)
34. 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)
35. Miyazawa, S., Jernigan, R.L.: Self-consistent estimation of inter-residue protein contact energies based on an equilibrium mixture approximation of residues. *Proteins* 34(1) (1999)
36. Papadimitriou, C.: *Computational Complexity*. Addison Wesley, Reading (1994)
37. Pokarowski, P., Kloczkowski, A., Jernigan, R.L., Kothari, N.S., Pokarowska, M., Kolinski, A.: Inferring ideal amino acid interaction forms from statistical protein contact potentials. *Proteins* 59(1), 49–57 (2005)
38. Shaw, P.: Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In: Maher, M.J., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520. Springer, Heidelberg (1998)
39. Siew, N., Fischer, D.: Convergent evolution of protein structure prediction and computer chess tournaments: CASP, Kasparov, and CAFASP. *IBM Systems Journal* 40(2) (2001)
40. Sippl, M.: Calculation of conformation ensembles from potentials of mean force. *J. Mol. Biol.* 213, 859–883 (1990)
41. Skolnick, J., Kolinski, A.: Simulations of the Folding of a Globular Protein. *Science* 250(4984), 1121–1125 (1990)
42. Taketomi, H., Kano, F., Go, N.: The effect of amino acid substitution on protein-folding and -unfolding transition studied by computer simulation. *Biopolymers* 27(4) (1988)
43. Unger, R., Moulton, J.: Genetic algorithms for protein folding simulations. *Journal of Molecular Biology* 231, 75–81 (1993)
44. Van Hentenryck, P., Michel, L.: *Constraint-Based Local Search*. The MIT Press, Cambridge (2005)
45. Will, S.: Constraint-based hydrophobic core construction for protein structure prediction in the face-centered-cubic lattice. In: *Pacific Symposium on Biocomputing* (2002)
46. Will, S.: Exact, Constraint-Based Structure Prediction in Simple Protein Models. In: PhD thesis, Friedrich-Schiller-Universität Jena (April 2005)
47. Wu, S., Skolnick, J., Zhang, Y.: Ab initio modeling of small proteins by iterative TASSER simulations. *BMC. Biol.* 5, 17 (2007)
48. Yue, K., Dill, K.A.: Folding proteins with a simple energy function and extensive conformational searching. *Protein. Sci.* 5(2), 254–261 (1996)
49. Yue, K., Fiebig, K.M., Thomas, P.D., Chan, H.S., Shakhnovich, E.I., Dill, K.A.: A test of lattice protein folding algorithms. *National Academy of Science* 92, 325–329 (1995)
50. Zaki, M.J.: *Protein Structure Prediction*, 2nd edn. Humana Press (2007)
51. Zhang, Y.: I-TASSER server for protein 3D structure prediction. *Bioinformatics* (2008)
52. Zhang, Y., Skolnick, J.: The protein structure prediction problem could be solved using the current PDB library. *Proc. Natl. Acad. Sci. U.S.A.* 102(4), 1029–1034 (2005)

An Application of Constraint Programming to Superblock Instruction Scheduling

Abid M. Malik, Michael Chase, Tyrel Russell, and Peter van Beek

Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
{`ammalik, vanbeek`}@cs.uwaterloo.ca

Abstract. Modern computer architectures have complex features that can only be fully taken advantage of if the compiler schedules the compiled code. A standard region of code for scheduling in an optimizing compiler is called a superblock. Scheduling superblocks optimally is known to be NP-complete, and production compilers use non-optimal heuristic algorithms. In this paper, we present an application of constraint programming to the superblock instruction scheduling problem. The resulting system is both optimal and fast enough to be incorporated into production compilers, and is the first optimal superblock scheduler for *realistic* architectures. In developing our optimal scheduler, the keys to scaling up to large, real problems were in applying and adapting several techniques from the literature including: implied and dominance constraints, impact-based variable ordering heuristics, singleton bounds consistency, portfolios, and structure-based decomposition techniques. We experimentally evaluated our optimal scheduler on the SPEC 2000 benchmarks, a standard benchmark suite. Depending on the architectural model, between 98.29% to 99.98% of all superblocks were solved to optimality. The scheduler was able to routinely solve the largest superblocks, including superblocks with up to 2,600 instructions, and gave noteworthy improvements over previous heuristic approaches.

1 The Problem

Modern computer architectures have complex features that can only be fully taken advantage of if the compiler schedules the compiled code. This instruction scheduling, as it is called, is one of the most important steps for improving the performance of object code produced by a compiler as it can lead to significant speedups [1]. As well, in VLIW (very large instruction word) architectures, instruction scheduling is necessary for correctness as the processor strictly follows the schedule given by the compiler (this is not true in so-called out-of-order processors). In the remainder of this section, we briefly review the necessary background in computer architecture before defining the superblock instruction scheduling problem, the problem that we address in this paper (for more background on these topics see, for example, [1,2,3]).

We consider multiple-issue, pipelined processors. Multiple-issue and pipelining are two techniques for performing instructions in parallel and processors that use these techniques are now standard in desktop and laptop machines. In such processors, there are multiple functional units and multiple instructions can be issued (begin execution) in each clock cycle. Examples of functional units include arithmetic-logic units (ALUs), floating-point units, memory or load/store units that perform address computations and accesses to the memory hierarchy, and branch units that execute branch and call instructions. The number of instructions that can be issued in each clock cycle is called the *issue width* of the processor. On most architectures, including the PowerPC [4] and Intel Itanium [5], the issue width is less than the number of available functional units.

Pipelining is a standard hardware technique for overlapping the execution of instructions on a single functional unit. A helpful analogy is to a vehicle assembly line [2] where there are many steps to constructing the vehicle and each step operates in parallel with the other steps. An instruction is issued on a functional unit (begins execution on the pipeline) and associated with each instruction is a delay or *latency* between when the instruction is issued and when the instruction has completed (exits the pipeline) and the result is available for other instructions that use the result. Also associated with each instruction is an *execution time*, the number of cycles between when the instruction is issued on a functional unit and when any subsequent instruction can be issued on the same functional unit. An architecture is said to be *fully pipelined* if every instruction has an execution time of 1. However, most architectures are not fully pipelined and so there will be cycles in which instructions cannot be issued on a particular functional unit, since the unit will still be executing a previously-issued instruction.

Further, some processors, such as the PowerPC and Intel Itanium, contain *serializing instructions*, instructions that require exclusive access to the processor in the cycle in which they are issued. This can happen when an architecture has only one of a particular resource, such as a condition register, and needs to ensure that only one instruction is accessing that resource at a time. In the cycle in which such instructions are issued, no other instruction can be executing or can be issued—for that one cycle, the instruction has sole access to the processor and its resources.

Example 1. Consider a PowerPC 603e processor [4]. The processor has four functional units—an ALU, a floating-point unit, a load/store unit, and a branch unit—and an issue width of two. On this processor a floating point addition has an execution time of 1 cycle and a latency of 3 cycles. In contrast, a floating point division has an execution time of 18 cycles and also a latency of 18 cycles. Thus, once a floating-point division instruction is issued on the floating-point unit, no other floating point instruction can be issued (because there is only one unit) until 18 cycles have elapsed and no other instruction can use the result of that floating-point division until 18 cycles have elapsed. Finally, on the PowerPC 603e, about 15% of the instructions executed by the processor are serializing instructions.

A compiler needs an accurate architectural model of the target processor that will execute the code in order to schedule the code in the best possible manner. In the rest of the paper, we refer to an architectural model as *idealized* if it assumes that (i) the issue width of the processor is equal to the number of functional units, (ii) the processor is fully pipelined, and (iii) that the processor contains no serializing instructions. An architectural model is referred to as *realistic* if it does not make any of these assumptions.

Instruction scheduling is done on certain regions of a program. All compilers schedule *basic blocks*, where a basic block is a straight-line sequence of code with a single entry point and a single exit point. However, basic blocks alone are considered insufficient for fully utilizing a processor's resources and most optimizing compilers also schedule a generalization of basic blocks called *superblocks*. A superblock is a collection of basic blocks with a unique entrance but multiple exit points [6]. We use the standard labeled directed acyclic graph (DAG) representation of a superblock. Each node corresponds to an instruction and there is an edge from i to j labeled with a non-negative integer $l(i, j)$ if j must not be issued until i has executed for $l(i, j)$ cycles. In particular, if $l(i, j) = 0$, j can be issued in the same cycle as i ; if $l(i, j) = 1$, j can be issued in the next cycle after i has been issued; and if $l(i, j) > 1$, there must be some intervening cycles between when i is issued and when j is subsequently issued. These cycles can possibly be filled by other instructions. Each node or instruction i has an associated execution time $d(i)$. *Exit nodes* are special nodes in a DAG representing the branch instructions. Each exit node i has an associated weight or exit probability $w(i)$ that represents the probability that the flow of control will leave the superblock through this exit point. The probabilities are calculated by running the instructions on representative data, a process known as *profiling*.

Given a labeled dependency DAG for a superblock and a target architectural model, a *schedule* for a superblock is an assignment of a clock cycle to each instruction such that the latency and resource constraints are satisfied. The resource constraints are satisfied if, at every time cycle, the resources needed by all the instructions issued or executing at that cycle do not exceed the limits of the processor.

Definition 1 (Superblock Instruction Scheduling). *The weighted completion time or cost of a superblock schedule is $\sum_{i=1}^n w(i)e(i)$, where n is the number of exit nodes, $w(i)$ is the weight of exit i , and $e(i)$ is the clock cycle in which exit i will be issued in the schedule. The superblock instruction scheduling problem is to construct a schedule with minimum weighted completion time.*

Example 2. Consider the superblock shown in Figure 1. Nodes E and K are branch instructions, with exit probability 0.3 and 0.7, respectively. Consider an idealized processor with two functional units. One functional unit can execute clear instructions and the other can execute shaded instructions. Figure 1(b) shows two possible schedules, S_1 and S_2 . The weighted completion time for schedule S_1 is $0.3 \times 4 + 0.7 \times 15 = 11.7$ cycles and for schedule S_2 is $0.3 \times 5 + 0.7 \times 14 = 11.3$ cycles. Schedule S_2 is an *optimal* solution.

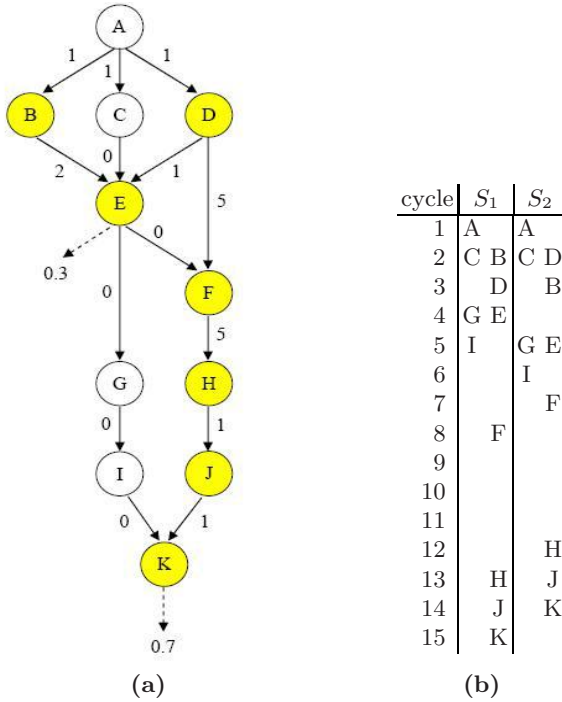


Fig. 1. (a) Superblock representation: nodes E and K are exit nodes with exit probabilities 0.3 and 0.7 respectively; (b) two possible schedules for Example 2

2 Why CP?

Superblock instruction scheduling for realistic multiple-issue processors is NP-complete [7] and currently is done using heuristic approaches in all commercial and open-source research compilers. The most common heuristic approach is a greedy list scheduling algorithm coupled with a priority heuristic. Many sophisticated heuristics have been proposed including critical path [3], dependence height and speculative yield [8], G^* [9], speculative hedge [10], balance scheduling [11], and successive retirement [9]. However, even the best heuristic approaches can produce sub-optimal solutions.

Optimal approaches for instruction scheduling have also been proposed. We first review previous work on basic block scheduling, the easier special case of superblock scheduling where there is only one exit and all of the instructions in the block are always executed. Previous work on optimal approaches to basic block instruction scheduling can be categorized by those approaches that are targeted only towards idealized—i.e., unrealistic—architectural models [12,13,14,15] and those approaches that have been developed for more realistic architectural models [16,17,18]. Broadly speaking, previous work has shown that (i) for an *idealized* multi-issue processor, optimal approaches can scale up to the largest basic blocks

that arise in practice, and (ii) for more realistic architectures, optimal approaches can be used but do not yet scale up beyond 10-40 instructions (the largest blocks that arise in practice have 2,600 instructions). In our work, we present a constraint programming approach that applies to *realistic* architectures and scales up to the largest blocks. Our work builds on a previously developed constraint programming approach for basic block scheduling, which assumed an idealized architecture [15].

In contrast to optimal basic block scheduling, there has been relatively little work on optimal superblock scheduling. Winkel [19] presents an integer linear programming model for instruction scheduling for Itanium processors. However, the approach has two limitations. First, the model is limited to small regions with size up to 200 instructions. Second, and more importantly, the approach minimizes the *length* of the schedule. This measure is appropriate for basic blocks, which consist of straight line code. But it is not appropriate for regions that contain multiple exits and whose paths of execution may rarely fall through to the last instruction. Shobaki and Wilken [20,21] were the first to develop a robust optimal scheduler for superblocks that scaled up to large superblocks. Their approach is based on enumeration. However, their work is targeted to idealized architectures and assumes that the functional units are fully pipelined, the issue width of the processor is equal to the number of functional units, and there are no serializing instructions. It is not at all clear how to successfully extend these previously proposed enumeration and integer programming approaches to realistic architectures and cost functions. In our constraint programming approach, we remove these assumptions and present the first optimal superblock scheduling approach for realistic architectures. Further, even though our target architectures are realistic, our approach scales up to more difficult and larger superblocks than in previous work.

3 How CP?

In this section, we present our constraint programming approach for superblock instruction scheduling. We first present the basic model—a model that is correct but inefficient—followed by the techniques we used to improve our model and solving approach. Our description is at a high-level; see [22] for more details.

3.1 Basic Model

Given a labeled dependency DAG $G = (N, E)$ for a superblock and a target architectural model, we model each instruction or node i by a variable x_i . The domain of each variable $dom(x_i)$ is a subset of $\{1, \dots, m\}$, which are the available time cycles. Assigning a value $t \in dom(x_i)$ to a variable x_i has the intended meaning that instruction i will be issued at time cycle t . The domain $dom(x_i) = \{a, \dots, b\}$ of a variable x_i is represented by the endpoints of the interval $[a, b]$.

To model the latencies of the instructions, for each pair of variables x_i and x_j such that $(i, j) \in E$, a latency constraint of the form $x_i + l(i, j) \leq x_j$ is

added to the constraint model, where $l(i, j)$ is the latency on the edge (i, j) . Global cardinality constraints (GCC) [23] are used to model the resources of the processor. A GCC over a set of variables and values states that the number of variables instantiating to a value must be between a given upper and lower bound. For each type t of functional unit, a GCC over all variables of type t is added to the constraint model, where the lower bound is zero and the upper bound is the number of functional units of type t . As well, a GCC over all variables is added, where the lower bound is zero and the upper bound is the issue width of the processor.

So far, the model assumes an idealized architecture where each unit is fully pipelined and there are no serializing instructions. To model a non-fully pipelined processor, we add auxiliary variables to the constraint model. Recall that in a non-fully pipelined processor, some instructions have execution times greater than 1. Let i be an instruction with execution time $e(i) > 1$ and let x_i be the corresponding variable. The auxiliary variables $p_{i,j}$, $1 \leq j \leq e(i) - 1$, are added into the model, where each variable $p_{i,j}$ is of the same functional unit type as x_i . The constraints $x_i + j = p_{i,j}$, $1 \leq j \leq e(i) - 1$, are also added into the model. Finally, we also add the variables $p_{i,j}$, all of which are of type t , to the GCC functional unit constraint for type t .

Serializing instructions can be modeled in a manner similar. Let i be a serializing instruction and let x_i be the corresponding variable. Let F be the total number of functional units in the processor. The auxiliary variables $s_{i,j}$, $1 \leq j \leq F - 1$, are added into the constraint model. There is one auxiliary variable for every functional unit except for the one on which instruction i is issued; the functional unit type of each auxiliary variable is assigned accordingly. The constraints $x_i = s_{i,j}$, $1 \leq j \leq F - 1$, are also added into the model. Finally, for each type t , we add all auxiliary variables of type t to the corresponding GCC functional unit constraint for type t .

Example 3. Consider again the superblock shown in Figure 1 and assume initially the same idealized processor as in Example 2. The constraint model would have variables $A, \dots, K$, and the constraints,

$$\begin{array}{lll} B \geq A + 1, & \dots & \text{GCC}(B, D, E, F, H, J, K), \\ C \geq A + 1, & K \geq I, & \text{GCC}(A, C, G, I), \\ D \geq A + 1, & K \geq J + 1, & \end{array}$$

where the lower and upper bounds of each GCC constraint are 0 and 1 (the number of functional units of each type), respectively, and the cost function is $0.3 \times E + 0.7 \times K$. Somewhat more realistically, suppose instead that instruction D is not fully pipelined and has an execution time $e(D) = 3$ and that instruction G is a serializing instruction. The auxiliary variables $p_{D,1}$, $p_{D,2}$, and $s_{G,1}$ would be added to the model along with the constraints $D + 1 = p_{D,1}$, $D + 2 = p_{D,2}$, and $G = s_{G,1}$. Finally, one of the GCC constraints would incorporate the auxiliary variables and would become $\text{GCC}(B, D, E, F, H, J, K, p_{D,1}, p_{D,2}, s_{G,1})$.

We have described a correct, but minimal, model for the superblock scheduling problem targeted towards realistic architectures. As is usual in constraint

programming, the minimal model cannot solve all but the smallest instances as it does not scale beyond 40 instructions. We next describe the improvements we made to scale up our constraint programming approach to instances with 2,600 instructions (the largest that we have found in practice).

3.2 Improving the Model and Solving Approach

In developing our optimal scheduler, the keys to scaling up to large, real problems were in applying and adapting several techniques from the literature including: implied and dominance constraints, impact-based variable ordering heuristics, singleton bounds consistency, portfolios, and structure-based decomposition techniques.

Implied constraints do not change the set of solutions while dominance constraints may but preserve an optimal solution. Both types of constraints can increase the amount of constraint propagation and so greatly improve the efficiency of the search for a solution (see, e.g., [24] and references therein). In our work, many instances of each of these constraints are added to the constraint model in an extensive preprocessing stage that occurs once. The extensive preprocessing effort pays off as the model is solved many times.

Two forms of implied constraints are added to the model: $x_i + d(i, j) \leq x_j$ and $x_j \leq x_i + d(i, j)$. Roughly, the first form is added if a pair of nodes i and j in the DAG for a superblock form a region; i.e., there is more than one path from i to j [12]. If the region is small enough, it is solved exactly using a backtracking algorithm; if it is large, the distance $d(i, j)$ is estimated, making sure that the estimate is a lower bound. Again roughly, the second form is added if i and j define a region and are articulation nodes—an articulation node is a node which disconnects the graph once removed—and the region defined by i and j is small enough to be solved quickly and exactly in isolation. It can be shown that the solution to the isolated subproblem can be used to form a tight upper bound on the distance between i and j in any optimal schedule.

Heffernan and Wilken [14] present a set of graph transformations for dependency DAGs for basic blocks and show that optimally scheduling the transformed DAGs using branch-and-bound enumeration is faster and more robust. We adapted these transformations to superblock scheduling and proved under what conditions they preserve optimality. In our context, the transformations add simple dominance constraints to the model of the form $x_i \geq x_j$. Adding dominance constraints requires identifying pairs of disjoint, isomorphic subgraphs A and B in a dependency DAG for a superblock. Subgraphs A and B are isomorphic if there is a mapping from the node set of A to the node set of B such that A and B are identical (identical instruction types, edges, and latencies on the edges). We use a fast heuristic approach to find pairs of disjoint, isomorphic subgraphs adapted from our work on basic block scheduling [15].

Example 4. Consider the DAG shown in Figure 2(a). Nodes H and I are called speculative nodes in the compiler literature as they can be moved across exit node G. The subgraphs with nodes $\{C, E\}$ and $\{H, I\}$ are isomorphic and satisfy

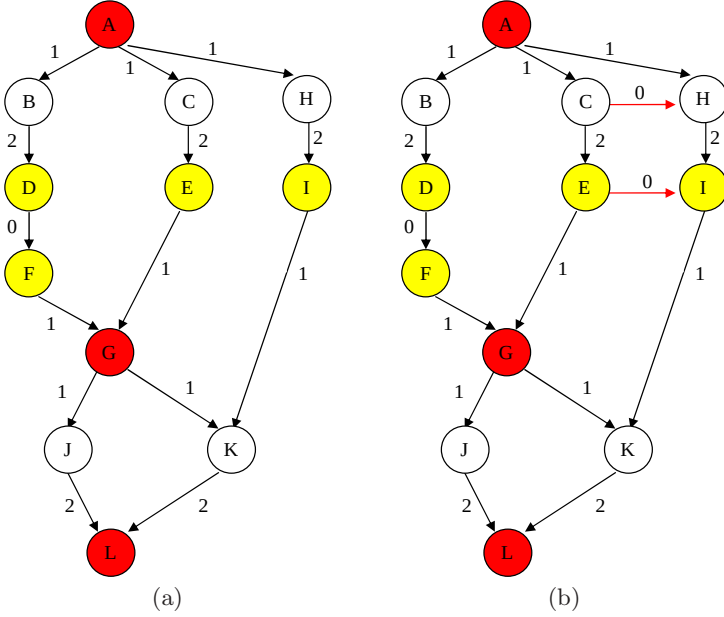


Fig. 2. Example of adding dominance constraints in a superblock: (a) actual DAG; (b) the constraints $C \leq H$ and $E \leq I$ (zero latency edges) would be added to the constraint model. Nodes A, G and L are exit nodes.

the conditions for adding dominance constraints. Hence, the constraints $C \leq H$ and $E \leq I$ can be added to the model. Figure 2(b) shows the DAG with the added constraints. Note that the added constraints do not change the speculative characteristic of exit node G, as nodes H and I still can be moved across—i.e., can be scheduled either before or after—node G. Similarly, the constraint $J \leq K$ can be added.

Once the constraint model has been extensively preprocessed by adding implied and dominance constraints, it is ready to be solved. Recall that the superblock scheduling problem is an optimization problem. To turn it into a satisfaction problem, we first establish an upper bound on the cost function using a fast heuristic scheduling method (a list scheduling algorithm, as discussed in the Experimental evaluation section). Given an upper bound on the cost function, we then prune the cost variables using singleton bounds consistency and enumerate the possible solutions to the cost function using techniques adapted from [25]. The solutions to the cost function are then stepped through in increasing order of cost until one is found that can be extended to a solution to the entire constraint model. Testing whether a solution to the cost function can be extended is done using a backtracking search algorithm. Of course, once a solution to the entire constraint model is found it is a provably optimal solution.

To reduce the brittleness or variability in performance of our backtracking search algorithm, we use a portfolio approach. Portfolios of multiple algorithms

have been proposed and shown to dramatically improve performance on some instances (see, e.g., [26]). In instruction scheduling, thousands of superblocks arise each time a compiler is invoked on some software project and a limit needs to be placed on the time given for solving each instance in order to keep the total compile time to an acceptable level. Given a set of possible backtracking algorithms $\{\mathcal{A}_1, \mathcal{A}_2, \dots\}$ and a time deadline d , a *portfolio* P for a single processor is a sequence of pairs, $P = [(\mathcal{A}_{k_1}, t_1), (\mathcal{A}_{k_2}, t_2), \dots, (\mathcal{A}_{k_m}, t_m)]$, where each $\mathcal{A}_{k_i}$ is a backtracking algorithm, each t_i is a positive integer, and $\sum_{i=1}^m t_i = d$. To apply a portfolio to an instance, algorithm $\mathcal{A}_{k_1}$ is run for t_1 steps. If no solution is found within t_1 steps, algorithm $\mathcal{A}_{k_1}$ is terminated and algorithm $\mathcal{A}_{k_2}$ is run for t_2 steps, and so on until either a solution is found or the sequence is exhausted as the time deadline d has been reached.

In contrast to previous work, where the differences in the possible backtracking algorithms $\{\mathcal{A}_1, \mathcal{A}_2, \dots\}$ often involves the variable ordering heuristic, we created variability in solving performance by increasing levels of constraint propagation from light-weight to heavy-weight. For our approach, we used a deterministic backtracking algorithm capable of performing three levels of constraint propagation,

- Level = 1 bounds consistency,
- Level = 2 singleton bounds consistency, and
- Level = 3 singleton bounds consistency to a depth of two.

and the portfolio involved three phases in increasing order. We chose bounds consistency—instead of the more usual arc consistency—as in our problem it is equivalent but more efficient. In bounds consistency, one ensures that each upper and lower bound of the domain of a variable is consistent with each constraint (see, e.g., [27] and references therein). In singleton bounds consistency, one temporarily assigns a value to a variable and then performs bounds consistency. In singleton bounds consistency to a depth of two, one temporarily assigns a value to a variable and then performs singleton consistency. In each, if the value is found to be inconsistent it is not part of any solution and can be removed from the domain of the variable.

During phase one, a standard dynamic variable ordering heuristic based on minimum domain size is used. However, in the next two phases which involve singleton consistency a variation of an impact-based variable ordering heuristic [28] is used. The idea in impact-based heuristics is to measure the importance of a variable for reducing the search space. Here, we record the number of changes that are made due to each variable during the singleton consistency propagation. This information is then used to select the next variable to branch on with the goal being to branch on a variable that causes the most reductions in the domains of the other variables. The impact-based heuristic is very effective and essentially comes for free as a side-effect of enforcing singleton consistency.

As a final technique for scaling up our constraint programming approach to the largest instances, we also adapted a structure-based decomposition technique [29]. For some of the largest superblock instances, all of the exit nodes were articulation nodes. We showed that such instances could be solved optimally by

solving them progressively. Let $e_1, \dots, e_n$ be the exit nodes. We first solve the subproblem consisting of e_1 and all of its predecessor nodes. Variable e_1 is then fixed using the optimal solution to the subproblem and we then in turn solve the subproblem consisting of e_2 and all of its predecessor nodes, and so on. The proof that this procedure preserves optimality requires careful attention to the resource contention at each exit node.

3.3 Experimental Evaluation

The constraint programming model was implemented and evaluated on all of the 154,651 superblocks from the SPEC 2000 integer and floating point benchmarks (www.spec.org). This benchmark suite consists of source code for software packages that are chosen to be representative of a variety of programming languages and types of applications. The benchmarks were compiled using IBM's Tobey compiler [30] targeted towards the IBM PowerPC processor [4], and the superblocks were captured as they were passed to Tobey's instruction scheduler. The Tobey compiler performs instruction scheduling before register allocation and once again afterward, and our test suite contains both versions of the superblocks. The compilations were done using Tobey's highest level of optimization, which includes aggressive optimization techniques such as software pipelining and loop unrolling.

The following table shows the four realistic architectural models we used in our evaluation. In these architectures, the functional units are not fully pipelined, the issue width of the processor is not equal to the number of functional units, and there are serializing instructions.

architecture	issue width	simple int. units	complex int. units	memory units	branch units	floating pt. units
1r-issue	1	1				
PowerPC 603e (ppc603e)	2	1		1	1	1
PowerPC 604 (ppc604)	4	2	1	1	1	1
6r-issue	6	2		2	3	2

The following table shows the total time (hh:mm:ss) to schedule all superblocks in the SPEC 2000 benchmark suite and the percentage of superblocks that were solved to optimality, for various realistic architectural models and time limits for solving each superblock.

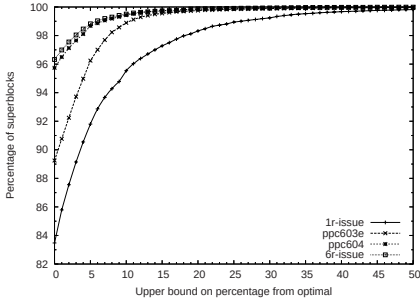
	1 sec.		10 sec.		1 min.		10 min.	
	time	%	time	%	time	%	time	%
1r-issue	1:30:20	97.34	7:15:46	99.38	10:22:36	99.96	15:08:44	99.98
ppc603e	3:57:13	91.83	30:53:56	93.90	108:50:01	97.18	665:31:00	97.70
ppc604	2:17:44	95.47	17:09:48	96.60	61:29:31	98.43	343:04:46	98.87
6r-issue	3:04:18	93.59	25:03:44	94.76	87:04:34	97.78	511:19:14	98.29

Table 1. *Superblock scheduling after register allocation.* For the SPEC 2000 benchmark suite, number of cycles saved ($\times 10^9$) by the optimal scheduler over a list scheduler using the dependence height and speculative yield heuristic and using the critical path heuristic, and the percentage reduction (%), for various realistic architectural models. The time limit for solving each superblock was 10 minutes.

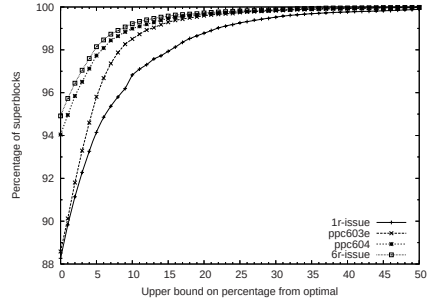
	DHASY heuristic								critical path heuristic							
	1r-issue		ppc603e		ppc604		6r-issue		1r-issue		ppc603e		ppc604			
benchmark	$\times 10^9$	%	$\times 10^9$	%	$\times 10^9$	%	$\times 10^9$	%	$\times 10^9$	%	$\times 10^9$	%	$\times 10^9$	%		
ammp	47.4	0.2	669.3	3.2	225.9	1.1	221.2	1.3	457.5	1.8	949.9	4.5	243.9	1.2		
applu	5.2	0.4	0.3	0.0	0.6	0.1	0.1	0.0	23.7	1.9	4.1	0.4	0.5	0.0		
apsi	52.4	1.1	43.6	1.0	45.8	1.1	31.8	1.0	341.5	7.1	89.0	2.0	84.9	2.0		
art	0.4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	2.7	0.1	1.1	0.0	1.1	0.0		
bzip2	51.8	0.3	282.1	1.8	281.6	1.9	137.5	0.9	300.1	1.5	405.0	2.5	356.7	2.3		
crafty	50.9	0.7	67.8	1.2	30.1	0.6	32.0	0.6	162.1	2.3	119.6	2.1	53.9	1.0		
eon	303.1	2.7	65.7	0.7	47.1	0.5	124.7	1.5	610.6	5.5	127.0	1.3	150.1	1.6		
equake	22.4	0.5	12.1	0.3	11.8	0.3	0.1	0.0	20.6	0.5	1.5	0.0	1.1	0.0		
facerec	19.4	0.3	27.0	0.5	3.6	0.1	1.6	0.0	28.7	0.5	32.9	0.7	3.6	0.1		
fma3d	36.4	0.4	48.4	0.6	86.2	1.1	18.6	0.3	51.8	0.5	49.2	0.6	73.4	1.0		
galgel	1.6	0.1	0.7	0.1	0.5	0.1	0.0	0.0	4.9	0.5	2.2	0.2	1.3	0.1		
gap	18.9	0.0	67.6	0.0	43.2	0.0	31.0	0.0	99.9	0.0	69.3	0.0	38.8	0.0		
gcc	28.7	0.6	33.4	0.8	18.8	0.5	16.6	0.4	65.9	1.3	51.2	1.2	26.3	0.6		
gzip	11.2	0.1	36.8	0.3	22.1	0.2	29.9	0.2	158.1	1.0	50.6	0.4	22.2	0.2		
lucas	0.0	0.0	0.2	0.1	0.0	0.0	0.0	0.0	4.5	1.2	0.5	0.2	0.0	0.0		
mcf	54.1	1.5	43.9	1.4	38.9	1.2	0.0	0.0	89.0	2.5	93.7	2.9	94.9	3.0		
mesa	34.0	0.2	53.3	0.4	31.0	0.3	306.5	3.3	85.3	0.6	32.2	0.3	31.0	0.3		
mgrid	1.7	0.5	0.3	0.1	0.0	0.0	0.0	0.0	3.1	0.9	0.5	0.2	0.0	0.0		
parser	483.1	1.9	530.8	2.6	507.9	2.6	277.5	1.5	956.8	3.8	808.8	3.9	526.0	2.7		
perlbmk	67.8	0.2	386.4	1.5	117.6	0.5	76.6	0.3	181.7	0.6	439.6	1.7	141.3	0.6		
sixtrack	122.6	3.5	6.3	0.2	4.0	0.1	1.5	0.0	655.4	18.6	19.9	0.6	5.0	0.1		
swim	0.0	0.2	0.1	1.7	0.1	1.9	0.0	0.0	8.5	99.8	6.6	102.3	3.2	58.1		
twolf	288.5	1.5	43.9	0.3	88.6	0.6	69.7	0.5	689.0	3.5	378.3	2.3	198.1	1.3		
vortex	41.7	0.4	252.9	3.1	274.6	3.7	220.3	3.3	212.3	2.0	310.6	3.9	306.5	4.1		
vpr	57.6	0.5	26.8	0.3	41.1	0.5	6.1	0.1	227.1	2.1	64.4	0.7	38.2	0.4		
wupwise	83.5	1.0	30.8	0.4	20.8	0.3	22.6	0.4	523.6	6.3	211.5	2.9	69.3	1.0		

We also evaluated our optimal scheduler with respect to how much it improves on previous heuristic approaches. Most production compilers use a greedy list scheduling algorithm coupled with a heuristic priority function for scheduling. Here we compare against a list scheduler with a realistic resource model using the dependence height and speculative yield (DHASY) heuristic [8] and a critical path heuristic [3]. We chose the former heuristic as it is considered one of the best available (it is the default heuristic used in the Trimaran compiler [31], for example) and the latter heuristic because it is a standard reference point.

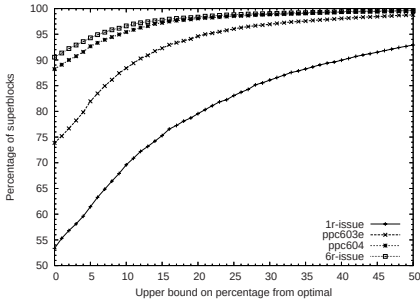
Table 1 gives the number of cycles saved ($\times 10^9$) by the optimal scheduler over the list scheduler using the dependence height and speculative yield heuristic and using the critical path heuristic, and the percentage reduction (%), for various



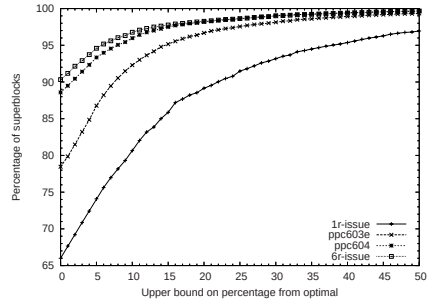
(a)



(b)



(c)



(d)

Fig. 3. Performance of optimal scheduler versus list scheduler, for various realistic architectures: (a) before register allocation using DHASY heuristic; (b) after register allocation using DHASY heuristic; (c) before register allocation using critical path heuristic; and (d) after register allocation using critical path heuristic

realistic architectural models after register allocation. We compiled the SPEC 2000 benchmark with the training data set associated with the benchmark using the Tobey compiler. The compiler uses the training data to construct a profile for each branch instruction. The profile is used to calculate the information regarding the number of times each instruction is executed.

Figures 3(a–d) summarizes the performance of the optimal scheduler versus the list scheduler, for various architectures. For example, consider the 1r-issue architecture and the DHASY heuristic. The list scheduler finds an optimal schedule (i.e. is within 0% of optimal) for approximately 84% of all superblocks before register allocation and approximately 88% of all superblocks after register allocation. In other words, the optimal scheduler improves on 16% and 12% of superblocks, respectively. Further, the list scheduler is within 10% of optimal for approximately 95% of all superblocks before register allocation and approximately 97% of all superblocks after register allocation, for this architecture. The graph also shows that, although quite rare, there exists superblocks for which the optimal scheduler finds improvements of up to 50% over the DHASY heuristic. As a second example, consider the 1r-issue architecture and the critical path

heuristic. The list scheduler finds an optimal schedule (i.e. is within 0% of optimal) for approximately 54% of all superblocks before register allocation and approximately 65% of all superblocks after register allocation. Further, the list scheduler is within 10% of optimal for approximately 70% of all superblocks before register allocation and approximately 80% of all superblocks after register allocation, for this architecture.

4 Added Value of CP?

Using constraint programming brought added value in two ways.

The first value added by constraint programming is that it allowed us to achieve the primary goal of our project, which was to develop a superblock instruction scheduler that was realistic yet fast enough to be incorporated into a production compiler. Using constraint programming, it was relatively easy to add additional constraints to model realistic architectures and it is not clear how to similarly extend previously proposed enumeration and integer programming approaches. As well, we had previously shown that constraint programming could be much faster than integer programming on a restricted form of these types of problems [13]. But perhaps the most important reason we were able to achieve our goal is that constraint programming allows and facilitates *programming* in the computer science sense of the word. This was crucial to scaling up to large instances, as it allowed us to design and implement domain-specific structure-based decomposition techniques and to incorporate and fine-tune ideas such as portfolios and impact-based variable ordering heuristics into our solver.

The second value added by constraint programming is that it allowed us to find optimal solutions. Although heuristic approaches have the advantage that they are very fast, a scheduler that finds optimal schedules can be useful in practice when longer compiling times are tolerable such as when compiling for software libraries, digital signal processing or embedded applications [1]. As well, an optimal scheduler can be used to evaluate the performance of heuristic approaches. Such an evaluation can tell whether there is a room for improvement in a heuristic or not.

5 Conclusions

We presented a constraint programming approach to superblock instruction scheduling for realistic architectural models. Our approach is optimal and robust on large, real instances. The keys to scaling up to large, real problems were in applying and adapting several techniques from the literature including: implied and dominance constraints, impact-based variable ordering heuristics, singleton bounds consistency, portfolios, and structure-based decomposition techniques. We experimentally evaluated our optimal scheduler on the SPEC 2000 integer and floating point benchmarks. On this benchmark suite, the optimal scheduler scaled to the largest superblocks. Depending on the architectural model, between 98.23% to 99.98% of all superblocks were solved to optimality. The scheduler

was able to routinely solve the largest superblocks, including blocks with up to 2,600 instructions. The schedules produced by the optimal schedule showed an improvement of 0%-3.8% on average over a list scheduler using the dependence height and speculative yield heuristic, considered one of the best heuristics available, and an improvement of 0%-102% on average over a critical path heuristic. One final conclusion we draw from our work is that constraint programming can be a fruitful approach for solving NP-hard compiler optimization problems.

Acknowledgments

This research was supported by an IBM Center for Advanced Studies (CAS) Fellowship, an NSERC Postgraduate Scholarship, and an NSERC CRD Grant.

References

1. Govindarajan, R.: Instruction scheduling. In: Srikant, Y.N., Shankar, P. (eds.) *The Compiler Design Handbook*, pp. 631–687. CRC Press, Boca Raton (2003)
2. Hennessy, J., Patterson, D.: *Computer Architecture: A Quantitative Approach*, 3rd edn. Morgan Kaufmann, San Francisco (2003)
3. Muchnick, S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco (1997)
4. Hoxey, S., Karim, F., Hay, B., Warren, H.: *The PowerPC Compiler Writer's Guide*. Warthman Associates (1996)
5. Intel: *Intel Itanium Architecture Software Developer's Manual, Volume 2: System Architecture* (2002)
6. Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm, J.G., Lavery, D.M.: The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing* 7(1), 229–248 (1993)
7. Hennessy, J., Gross, T.: Postpass code optimization of pipeline constraints. *ACM Transactions on Programming Languages and Systems* 5(3), 422–448 (1983)
8. Bringmann, R.A.: *Enhancing Instruction Level Parallelism through Compiler-Controlled Speculation*. PhD thesis, U. of Illinois at Urbana-Champaign (1995)
9. Chekuri, C., Johnson, R., Motwani, R., Natarajan, B., Rau, B.R., Schlansker, M.: Profile-driven instruction level parallel scheduling with application to superblocks. In: *Proc. of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-29)*, Paris, pp. 58–67 (1996)
10. Deitrich, B., Hwu, W.: Speculative hedge: Regulating compile-time speculation against profile variations. In: *Proc. of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-29)*, Paris (1996)
11. Eichenberger, A.E., Meleis, W.M.: Balance scheduling: Weighting branch tradeoffs in superblocks. In: *Proc. of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture (Micro-32)*, Haifa, Israel (1999)
12. Wilken, K., Liu, J., Heffernan, M.: Optimal instruction scheduling using integer programming. In: *Proc. of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, Vancouver, pp. 121–133 (2000)
13. van Beek, P., Wilken, K.: Fast optimal instruction scheduling for single-issue processors with arbitrary latencies. In: *Proc. of the 7th Int'l Conf. on Principles and Practice of Constraint Programming*, Paphos, Cyprus, pp. 625–639 (2001)

14. Heffernan, M., Wilken, K.: Data-dependency graph transformations for instruction scheduling. *Journal of Scheduling* 8, 427–451 (2005)
15. Malik, A.M., McInnes, J., van Beek, P.: Optimal basic block instruction scheduling for multiple-issue processors using constraint programming. In: *Proc. of the 18th IEEE Int'l Conf. on Tools with AI*, Washington, DC, pp. 279–287 (2006)
16. Ertl, M.A., Krall, A.: Optimal instruction scheduling using constraint logic programming. In: *Proc. of 3rd International Symposium on Programming Language Implementation and Logic Programming*, Passau, Germany, pp. 75–86 (1991)
17. Kästner, D., Winkel, S.: ILP-based instruction scheduling for IA-64. In: *Proc. of the SIGPLAN 2001 Workshop on Languages Compilers, and Tools for Embedded Systems*, Snowbird, Utah, pp. 145–154 (2001)
18. Liu, J., Chow, F.: A near-optimal instruction scheduler for a tightly constrained, variable instruction set embedded processor. In: *Proc. of the Int'l Conf. on Compilers, Architectures, and Synthesis for Embedded Systems*, Grenoble, pp. 9–18 (2002)
19. Winkel, S.: Exploring the performance potential of Itanium processors with ILP-based scheduling. In: *2nd IEEE/ACM International Symposium on Code Generation and Optimization*, pp. 189–200 (2004)
20. Shobaki, G., Wilken, K.: Optimal superblock scheduling using enumeration. In: *Proc. of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (Micro-37)*, Portland, Oregon, pp. 283–293 (2004)
21. Shobaki, G.: Optimal Global Instruction Scheduling Using Enumeration. PhD thesis, University of California, Davis (2006)
22. Malik, A.M.: Constraint Programming Techniques for Optimal Instruction Scheduling. PhD thesis, University of Waterloo (2008)
23. Régim, J.C.: Generalized arc consistency for global cardinality constraint. In: *Proc. of the 13th National Conference on AI*, Portland, Oregon, pp. 209–215 (1996)
24. Smith, B.M.: Modelling. In: Rossi, F., van Beek, P., Walsh, T. (eds.) *Handbook of Constraint Programming*. Elsevier, Amsterdam (2006)
25. Trick, M.: A dynamic programming approach for consistency and propagation of knapsack constraints. In: *Proc. of Third International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems* (2001)
26. Gomes, C., Selman, B., Crato, N.: Heavy-tailed distributions in combinatorial search. In: *Proc. of the 3rd Int'l Conf. on Principles and Practice of Constraint Programming*, Linz, Austria, pp. 121–135 (1997)
27. Bessiere, C.: Constraint propagation. In: Rossi, F., van Beek, P., Walsh, T. (eds.) *Handbook of Constraint Programming*. Elsevier, Amsterdam (2006)
28. Refalo, P.: Impact-based search strategies for constraint programming. In: *Proc. of the 10th Int'l Conf. on Principles and Practice of Constraint Programming*, Toronto, pp. 557–571 (2004)
29. Freuder, E.C.: Exploiting structure in constraint satisfaction problems. In: Mayoh, B., Tyugo, E., Penjam, J. (eds.) *Constraint Programming*. Springer, Heidelberg (1994)
30. Blainey, R.J.: Instruction scheduling in the TOBEY compiler. *IBM J. Res. Develop.* 38(5), 577–593 (1994)
31. Chakrapani, L.N., Gyllenhaal, J., Hwu, W.W., Mahlke, S.A., Palem, K.V., Rabbah, R.M.: Trimaran: An infrastructure for research in instruction-level parallelism. In: *Proc. of the 17th International Workshop on Languages and Compilers for High Performance Computing*, West Lafayette, Indiana, USA, pp. 32–41 (2005)

Classes of Submodular Constraints Expressible by Graph Cuts

Stanislav Živný and Peter G. Jeavons

Computing Laboratory, University of Oxford,
Wolfson Building, Parks Road, Oxford OX1 3QD, United Kingdom
{stanislav.zivny,peter.jeavons}@comlab.ox.ac.uk

Abstract. Submodular constraints play an important role both in theory and practice of valued constraint satisfaction problems (VCSPs). It has previously been shown, using results from the theory of combinatorial optimisation, that instances of VCSPs with submodular constraints can be minimised in polynomial time. However, the general algorithm is of order $O(n^6)$ and hence rather impractical. In this paper, by using results from the theory of pseudo-Boolean optimisation, we identify several broad classes of submodular constraints over a Boolean domain which are expressible using binary submodular constraints, and hence can be minimised in cubic time. We also discuss the question of whether all submodular constraints of bounded arity over a Boolean domain are expressible using only binary submodular constraints, and can therefore be minimised efficiently.

1 Introduction

The CONSTRAINT SATISFACTION PROBLEM (CSP) is a general framework which can be used to model many different problems [11,17,21]. However, the CSP model considers only the feasibility of satisfying a collection of simultaneous requirements (so-called *hard constraints*).

Various extensions have been proposed to this model to allow it to deal with different kinds of optimisation criteria, or preferences, between different feasible solutions (so-called *soft constraints*). Two very general extended frameworks that have been proposed are the SCSP (semi-ring CSP) framework and the VCSP (valued CSP) framework [2]. The SCSP framework is slightly more general¹, but the VCSP framework is simpler, and yet sufficiently powerful to model a wide range of optimisation problems [2,21,22].

Informally, in the VALUED CONSTRAINT SATISFACTION PROBLEM (VCSP) framework, an instance consists of a set of variables, a set of possible values, and a set of (soft) constraints. Each constraint has an associated cost function which assigns a cost (or a degree of violation) to every possible tuple of values for the

¹ The main difference is that costs in VCSPs represent violation levels and have to be totally ordered, whereas costs in SCSPs represent preferences and might be ordered only partially.

variables in the scope of the constraint. The goal is to find an assignment of values to all of the variables which has the minimum total cost. We remark that infinite costs can be used to indicate infeasible assignments (hard constraints), and hence the VCSP framework includes the standard CSP framework as a special case and is equivalent to the CONSTRAINT OPTIMISATION PROBLEM (COP) framework [21], which is widely used in practice.

One significant line of research on the VCSP is to identify restrictions which ensure that instances are solvable in polynomial time. There are two main types of restrictions that have been studied in the literature. Firstly, we can limit the *structure* of the instances. We will not deal with this approach in this paper.

Secondly, we can restrict the *forms* of the valued constraints which are allowed in the problem, giving rise to so-called *language restrictions*. Several language restrictions which ensure tractability have been identified in the literature, (see e.g., [8]). One important and well-studied restriction on valued constraints is *submodularity*. In fact the class of submodular constraints is the only non-trivial tractable case in the dichotomy classification of the Boolean VCSP [8].

The concept of submodularity not only plays an important role in theory, but is also very important in practice. For example, many of the problems that arise in computer vision can be expressed in terms of energy minimisation [16]. The problem of energy minimisation is NP-hard in general, and therefore a lot of research has been devoted to identifying instances which can be solved more efficiently. Kolmogorov and Zabih identified classes of instances for which the energy minimisation problem can be solved efficiently [16], and which are applicable to a wide variety of vision problems, including image restoration, stereo vision and motion tracking, image synthesis, image segmentation, multi-camera scene reconstruction and medical imaging. The so-called *regularity* condition, which specifies the efficiently solvable classes in [16], is equivalent to submodularity.

The notion of submodularity originally comes from combinatorial optimisation where submodular functions are defined on subsets of a given base set [14,18]. The time complexity of the fastest known algorithm for the problem of SUBMODULAR FUNCTION MINIMISATION (SFM) is roughly $O(n^6)$ [19]. However, there are several known special classes of SFM that can be solved more efficiently than the general case (see [3] for a survey).

Cohen et al. showed that VCSPs with submodular constraints over an arbitrary finite domain can be reduced to the SFM problem over a special family of sets known as a ring family [8]. This problem is equivalent to the general SFM problem [23], thus giving an algorithm of order $O(n^6 + n^5L)$, where L is the look-up time (needed to evaluate an assignment to all variables), for any VCSP with submodular constraints. This tractability result has since been generalised to a wider class of valued constraints over arbitrary finite domains known as tournament-pair constraints [6]. An alternative approach can be found in [9].

In this paper we focus on submodular constraints over a Boolean domain $\{0,1\}$, which correspond precisely to submodular set functions [8]. We describe an algorithm based on graph cuts which can be used to solve certain VCSPs with submodular constraints over a Boolean domain much more efficiently than

the general case. Some of our results are closely related to known efficient cases of SFM, and other previous results from different areas of computer science, but we present them here in a unified and constraints-based framework which allows us to make the proofs more consistent and often simpler. Moreover, we explicitly discuss for the first time the *expressive power* of binary submodular constraints, and use this powerful idea in a consistent way to identify new classes of submodular constraints which can be solved efficiently.

The paper is organised as follows. In Section 2, we define the VCSP framework and submodular constraints, and note that submodular constraints over a Boolean domain can be represented by polynomials. In Section 3, we show that the standard (s, t) -MIN-CUT problem can be expressed in the VCSP framework with a restricted constraint language Γ_{cut} , and that any instance of $\text{VCSP}(\Gamma_{\text{cut}})$ is solvable in cubic time. Moreover, we show that Γ_{cut} can express all *binary* submodular constraints. In Section 4, we show that any instance of the VCSP with constraints whose corresponding polynomials have only non-positive coefficients for terms of degree ≥ 2 can be expressed in $\text{VCSP}(\Gamma_{\text{cut}})$. We show the same for all $\{0, 1\}$ -valued submodular constraints, and also for all *ternary* submodular constraints. In Section 5, we present a necessary condition for a quartic polynomial to be submodular. Moreover, for every $k \geq 4$, we identify new classes of k -ary submodular constraints which can be expressed over $\text{VCSP}(\Gamma_{\text{cut}})$, and thus solved efficiently. We then discuss the question of whether *all* submodular constraints of bounded arity can be expressed over Γ_{cut} . Finally, in Section 6, we summarise our work and discuss related and future work.

2 Definitions

2.1 Valued Constraint Satisfaction and Expressibility

In this section we define the VALUED CONSTRAINT SATISFACTION PROBLEM (VCSP). In the original definition of this problem, given in [22], costs were allowed to lie in any positive totally ordered monoid called a *valuation structure*. For our purposes, it is sufficient to consider costs which lie in the set $\overline{\mathbb{Q}}_+$ consisting of all non-negative rational numbers together with infinity².

Given a fixed set D , a function from D^k to $\overline{\mathbb{Q}}_+$ will be called a *cost function*. If the range of ϕ is $\{0, \infty\}$, then ϕ is called a *crisp* cost function. Note that crisp cost functions correspond precisely to relations, so we shall use these terms interchangeably. If the range of ϕ lies entirely within $\mathbb{Q}_+$, the set of non-negative rationals, then ϕ is called a *finite-valued* cost function.

Definition 1. *An instance $\mathcal{P}$ of VCSP is a triple $\langle V, D, \mathcal{C} \rangle$, where V is a finite set of variables, which are to be assigned values from the set D , and $\mathcal{C}$ is a set of valued constraints. Each $c \in \mathcal{C}$ is a pair $c = \langle \sigma, \phi \rangle$, where σ is a tuple of variables of length $|\sigma|$, called the scope of c , and $\phi : D^{|\sigma|} \rightarrow \overline{\mathbb{Q}}_+$ is a cost function. An*

² See [10] for a discussion of why limiting ourselves to the $\overline{\mathbb{Q}}_+$ valuation structure is not a severe restriction.

assignment for the instance $\mathcal{P}$ is a mapping s from V to D . The cost of an assignment s is defined as follows:

$$\text{Cost}_{\mathcal{P}}(s) = \sum_{\langle \{v_1, v_2, \dots, v_m\}, \phi \rangle \in \mathcal{C}} \phi(\langle s(v_1), s(v_2), \dots, s(v_m) \rangle).$$

A solution to $\mathcal{P}$ is an assignment with minimum cost.

Any set Γ of cost functions is called a *valued constraint language*. The class $\text{VCSP}(\Gamma)$ is defined to be the class of all VCSP instances where the cost functions of all valued constraints lie in Γ .

In any VCSP instance, the variables listed in the scope of each valued constraint are explicitly constrained, in the sense that each possible combination of values for those variables is associated with a given cost. Moreover, if we choose *any* subset of the variables, then their values are constrained *implicitly* in the same way, due to the combined effect of the valued constraints. This motivates the concept of *expressibility* for cost functions, which is defined as follows:

Definition 2. For any VCSP instance $\mathcal{I} = \langle V, D, \mathcal{C} \rangle$, and any list of variables of $\mathcal{I}$, $l = \langle v_1, \dots, v_m \rangle$, the projection of $\mathcal{I}$ onto l , denoted $\pi_l(\mathcal{I})$, is the m -ary cost function defined as follows:

$$\pi_l(\mathcal{I})(x_1, \dots, x_m) = \min_{\{s: V \rightarrow D \mid \langle s(v_1), \dots, s(v_m) \rangle = \langle x_1, \dots, x_m \rangle\}} \text{Cost}_{\mathcal{I}}(s).$$

We say that a cost function ϕ is expressible over a valued constraint language Γ if there exists an instance $\mathcal{I} \in \text{VCSP}(\Gamma)$ and a list l of variables of $\mathcal{I}$ such that $\pi_l(\mathcal{I}) = \phi$. We call the pair $\langle \mathcal{I}, l \rangle$ a gadget for expressing ϕ over Γ . Variables from $V \setminus l$ are called *extra* or *hidden variables*.

Note that in the special case of relations (crisp cost functions) this notion of expressibility corresponds to the standard notion of expressibility using conjunction and existential quantification (*primitive positive formulas*) [4].

We denote by $\langle \Gamma \rangle$ the *expressive power* of Γ which is the set of all cost functions expressible over Γ up to additive and multiplicative constants.

2.2 Submodular Functions and Polynomials

A function $\psi : 2^V \rightarrow \mathbb{Q}$ defined on subsets of a set V is called a *submodular function* [18] if, for all subsets S and T of V , $\psi(S \cap T) + \psi(S \cup T) \leq \psi(S) + \psi(T)$. The problem of SUBMODULAR FUNCTION MINIMISATION (SFM) consists in finding a subset S of V for which the value of $\psi(S)$ is minimal.

For any lattice-ordered set D , a cost function $\phi : D^k \rightarrow \overline{\mathbb{Q}}_+$ is called *submodular* if for every $u, v \in D^k$, $\phi(\min(u, v)) + \phi(\max(u, v)) \leq \phi(u) + \phi(v)$ where both $\min$ and $\max$ are applied coordinate-wise on tuples u and v . Note that expressibility preserves submodularity: if every $\phi \in \Gamma$ is submodular, and $\phi' \in \langle \Gamma \rangle$, then ϕ' is also submodular.

Using results from [8] and [24], it can be shown that any submodular cost function ϕ can be expressed as the sum of a finite-valued submodular cost function

ϕ_{fin} , and a submodular relation ϕ_{crisp} , that is, $\phi = \phi_{fin} + \phi_{crisp}$. Moreover, it is known that all submodular *relations* are binary decomposable [15], and hence expressible using only binary submodular relations. Therefore, when considering which cost functions are expressible over binary submodular cost functions, we can restrict our attention to *finite-valued* cost functions without any loss of generality.

In this paper we focus on problems over Boolean domains. We denote by $\Gamma_{\text{sub},k}$ the set of all finite-valued submodular cost functions of arity at most k on a Boolean domain $D = \{0, 1\}$, and we set $\Gamma_{\text{sub}} = \bigcup_k \Gamma_{\text{sub},k}$. We will show below that $\text{VCSP}(\Gamma_{\text{sub},2})$ can be solved in cubic time, and hence we will be concerned with what other cost functions are expressible over $\Gamma_{\text{sub},2}$, and so can also be solved efficiently.

A cost function of arity k can be represented as a table of values of size D^k . Alternatively, a (finite-valued) cost function $\phi : D^k \rightarrow \mathbb{Q}_+$ on a Boolean domain $D = \{0, 1\}$ can be uniquely represented as a *polynomial* in k (Boolean) variables with coefficients from $\mathbb{Q}$ [3] (such functions are sometimes called *pseudo-Boolean functions*). Hence, in what follows, we will often represent a finite-valued cost function on a Boolean domain by a polynomial.

Note that if Γ is a set of cost functions on a Boolean domain, with arity at most k , then any instance of $\text{VCSP}(\Gamma)$ with n variables can be uniquely represented as a polynomial p in n Boolean variables, of degree at most k . Conversely, any such polynomial represents an n -ary cost function which can be expressed over a set of cost functions on a Boolean domain, with arity at most k . Note that $x^2 = x$, so p has at most 2^n terms which correspond to subsets of variables.

For polynomials over Boolean variables there is a standard way to define *derivatives* of each order (see [3]). For example, the second order derivative of a polynomial p , with respect to the first two indices, denoted $\delta_{1,2}(\mathbf{x})$, is defined as $p(1, 1, \mathbf{x}) - p(1, 0, \mathbf{x}) - p(0, 1, \mathbf{x}) + p(0, 0, \mathbf{x})$. Analogously for all other pairs of indices.

Proposition 3 ([3]). *A polynomial $p(x_1, \dots, x_n)$ over Boolean variables $x_1, \dots, x_n$ represents a submodular cost function if and only if its second order derivatives $\delta_{i,j}(\mathbf{x})$ are non-positive for all $1 \leq i < j \leq n$ and all $\mathbf{x} \in D^{n-2}$.*

Corollary 4. *A quadratic polynomial $a_0 + \sum_{i=1}^n a_i x_i + \sum_{1 \leq i < j \leq n} a_{ij} x_i x_j$ over Boolean variables $x_1, \dots, x_n$, represents a submodular cost function if and only if $a_{ij} \leq 0$ for every $1 \leq i < j \leq n$.*

3 Binary Submodular Constraints

In this section we show that a constraint language Γ_{cut} , consisting of certain simple binary and unary cost functions over a Boolean domain, has cubic time complexity. We also show that Γ_{cut} can express any binary submodular cost function over a Boolean domain, that is, $\Gamma_{\text{sub},2} \subseteq \langle \Gamma_{\text{cut}} \rangle$. It follows that any instance of $\text{VCSP}(\Gamma_{\text{sub},2})$ can also be solved in cubic time.

For any $w \in \overline{\mathbb{Q}}_+$, we define the binary cost function χ^w as follows:

$$\chi^w(x, y) = \begin{cases} w & \text{if } (x, y) = (0, 1), \\ 0 & \text{otherwise.} \end{cases}$$

For any $d \in D$ and $c \in \overline{\mathbb{Q}}_+$, we define the unary cost function μ_d^c as follows:

$$\mu_d^c = \begin{cases} c & \text{if } x \neq d, \\ 0 & \text{if } x = d. \end{cases}$$

It is straightforward to check that all χ^w and μ_d^c are submodular.

We define the constraint language Γ_{cut} to be the set of all cost functions χ^w and μ_d^c over a Boolean domain, for $c, w \in \overline{\mathbb{Q}}_+$ and $d \in \{0, 1\}$.

Theorem 5. *The problems (s, t) -MIN-CUT and $\text{VCSP}(\Gamma_{\text{cut}})$ are linear-time equivalent.*

Proof. Consider any instance of (s, t) -MIN-CUT with (directed) graph $G = \langle V, E \rangle$ and weight function $w : E \rightarrow \overline{\mathbb{Q}}_+$. Define a corresponding instance $\mathcal{I}$ of $\text{VCSP}(\Gamma_{\text{cut}})$ as follows:

$$\mathcal{I} = \langle V, \{0, 1\}, \{ \langle \langle i, j \rangle, \chi^{w(i, j)} \rangle \mid \langle i, j \rangle \in E \} \cup \{ \langle s, \mu_0^\infty \rangle, \langle t, \mu_1^\infty \rangle \} \rangle.$$

Note that in any solution to $\mathcal{I}$ the source and target nodes, s and t , must take the values 0 and 1, respectively. Moreover, the weight of any cut containing s and not containing t is equal to the cost of the corresponding assignment to $\mathcal{I}$. Hence we have shown that (s, t) -MIN-CUT can be reduced to $\text{VCSP}(\Gamma_{\text{cut}})$ in linear time.

On the other hand, given an instance $\mathcal{I} = \langle V, D, \mathcal{C} \rangle$ of $\text{VCSP}(\Gamma_{\text{cut}})$, construct a graph on $V \cup \{s, t\}$ as follows: any unary constraint on variable v with cost function μ_0^c (respectively μ_1^c) is represented by an edge of weight c from the source node s to node v (respectively, from node v to the target node t). Any binary constraint on variables $\langle v_1, v_2 \rangle$ with cost function χ^w is represented by an edge of weight w from node v_1 to v_2 . It is straightforward to check that a solution to $\mathcal{I}$ corresponds to a minimum (s, t) -cut of this graph. $\square$

Corollary 6. *$\text{VCSP}(\Gamma_{\text{cut}})$ can be solved in cubic time.*

Proof. By Theorem 5, $\text{VCSP}(\Gamma_{\text{cut}})$ has the same time complexity as (s, t) -MIN-CUT, which is known to be solvable in cubic time [13]. $\square$

Using a standard reduction (see, for example, [3]), we now show that all *binary* submodular cost functions over a Boolean domain can be expressed over Γ_{cut} .

Theorem 7. $\Gamma_{\text{sub}, 2} \subseteq \langle \Gamma_{\text{cut}} \rangle$.

Proof. By Corollary 4, any cost function from $\Gamma_{\text{sub}, 2}$ can be represented by a quadratic Boolean polynomial $p(x_1, x_2) = a_0 + a_1x_1 + a_2x_2 + a_{12}x_1x_2$ where $a_{12} \leq 0$. This can then be re-written as

$$p(x_1, x_2) = a'_0 + \sum_{i \in P} a'_i x_i + \sum_{j \in N} a'_j (1 - x_j) + a'_{12} (1 - x_1) x_2,$$

where $P \cap N = \emptyset$, $P \cup N = \{1, 2\}$, $a'_{12} = -a_{12}$, and $a'_i, a'_j, a'_{12} \geq 0$. (This is known as a *posiform* [3].)

Hence p can be expressed over Γ_{cut} (up to the constant a'_0) by the gadget $\langle \mathcal{I}, \langle x_1, x_2 \rangle \rangle$, where $\mathcal{I}$ is the instance $\langle \{x_1, x_2, s, t\}, \{0, 1\}, \mathcal{C} \rangle$ of $\text{VCSP}(\Gamma_{\text{cut}})$ and

$$\begin{aligned} \mathcal{C} = \{ \langle \langle s, x_i \rangle, \chi^{a'_i} \rangle \mid i \in P \} \cup \{ \langle \langle x_j, t \rangle, \chi^{a'_j} \rangle \mid j \in N \} \\ \cup \{ \langle \langle s \rangle, \mu_0^\infty \rangle, \langle \langle t \rangle, \mu_1^\infty \rangle, \langle \langle x_1, x_2 \rangle, \chi^{a'_{12}} \rangle \}. \quad \square \end{aligned}$$

Corollary 8. $\text{VCSP}(\Gamma_{\text{sub},2})$ can be solved in cubic time.

Proof. By Theorem 7, any instance of $\text{VCSP}(\Gamma_{\text{sub},2})$ can be reduced to $\text{VCSP}(\Gamma_{\text{cut}})$ in linear time by replacing each constraint with a suitable gadget of fixed size. The result then follows from Corollary 6. (Note that we can use the same vertices s and t for all constraints.) $\square$

4 Negative Higher Degree Terms, $\{0, 1\}$ -Valued and Ternary Submodular Constraints

In this section we extend the results from Section 3 to three further classes of constraints over a Boolean domain: submodular constraints whose corresponding polynomials have negative coefficients for all terms of degree ≥ 2 ; $\{0, 1\}$ -valued submodular constraints; and ternary submodular constraints. We show that the cost functions for these three classes of submodular constraints can all be expressed over $\Gamma_{\text{sub},2}$, and hence can be minimised in cubic time in the number of variables plus the number of higher-order (non-binary) constraints.

Define $\Gamma_{\text{neg},k}$ to be the set of all cost functions over a Boolean domain, of arity at most k , whose corresponding polynomials have negative coefficients for all terms of degree greater than or equal to 2. It is easy to check that these cost functions, sometimes called *negative-positive*, are submodular. Set $\Gamma_{\text{neg}} = \bigcup_k \Gamma_{\text{neg},k}$. The minimisation of cost functions chosen from Γ_{neg} using min-cuts was first studied in [20].

Theorem 9. $\Gamma_{\text{neg}} \subseteq \langle \Gamma_{\text{sub},2} \rangle$.

Proof. Consider the following polynomial:

$$p_0(x_1, \dots, x_n) = \min_{y \in \{0,1\}} \{-y + y \sum_{i \in A} (1 - x_i)\}.$$

It is straightforward to check that for a given $A \subseteq \{1, \dots, n\}$, $p_0(\mathbf{x}) = -1$ if $A \subseteq \mathbf{x}$ and $p_0(\mathbf{x}) = 0$ otherwise (where $A \subseteq \mathbf{x}$ means $\forall i \in A, x_i = 1$).

Now, given any polynomial of the form $p_1(x_1, \dots, x_n) = -a_{klm} x_k x_l x_m + Q$, where Q consists of terms of degree ≤ 2 , we can use a similar construction to p_0 to obtain

$$p_1(x_1, \dots, x_n) = \min_{y \in \{0,1\}} \{Q + a_{klm}(-y + y \sum_{i \in \{k,l,m\}} (1 - x_i))\}.$$

Given any polynomial p , we can use a similar construction to replace each term of degree ≥ 3 in turn, introducing a distinct new variable y each time.

Proceeding in this way, we can express any polynomial p representing a cost function in Γ_{neg} as a quadratic polynomial with non-positive quadratic coefficients, introducing k new variables, where k is the total number of terms of degree ≥ 3 . Such a quadratic polynomial can be expressed over $\Gamma_{\text{sub},2}$, by Corollary 4. $\square$

Corollary 10. *For any fixed k , $\text{VCSP}(\Gamma_{\text{neg},k})$ can be solved in cubic time.*

Proof. By Theorem 9, any instance of $\text{VCSP}(\Gamma_{\text{neg},k})$ can be reduced to $\text{VCSP}(\Gamma_{\text{sub},2})$ in linear time by replacing each constraint with a suitable gadget. For any fixed k , the number of new variables introduced in any of these gadgets is bounded by a constant. The result then follows from Corollary 8.

Next we consider the class of submodular constraints over a Boolean domain which take only the cost values 0 and 1. (Such constraints can be used to model optimisation problems such as MAX-CSP, see [7].) Define $\Gamma_{\{0,1\},k}$ to be the set of all $\{0,1\}$ -valued submodular cost functions over a Boolean domain, of arity at most k , and set $\Gamma_{\{0,1\}} = \cup_k \Gamma_{\{0,1\},k}$. The minimisation of submodular cost functions from $\Gamma_{\{0,1\}}$ was studied in [11], where they were called *2-monotone* functions. The equivalence of 2-monotone and submodular cost functions and a generalisation of 2-monotone functions to non-Boolean domains was shown in [7].

Definition 11. *A cost function ϕ is called 2-monotone if there exist two sets $A, B \subseteq \{1, \dots, n\}$ such that $\phi(\mathbf{x}) = 0$ if $A \subseteq \mathbf{x}$ or $\mathbf{x} \subseteq B$ and $\phi(\mathbf{x}) = 1$ otherwise (where $A \subseteq \mathbf{x}$ means $\forall i \in A, x_i = 1$ and $\mathbf{x} \subseteq B$ means $\forall i \notin B, x_i = 0$).*

Theorem 12. $\Gamma_{\{0,1\}} \subseteq \langle \Gamma_{\text{sub},2} \rangle$.

Proof. Any 2-monotone cost function ϕ can be expressed over $\Gamma_{\text{sub},2}$ using 2 extra variables, y_1, y_2 :

$$\phi(\mathbf{x}) = \min_{y_1, y_2 \in \{0,1\}} \left\{ (1 - y_1)y_2 + y_1 \sum_{i \in A} (1 - x_i) + (1 - y_2) \sum_{i \notin B} x_i \right\}. \quad \square$$

Corollary 13. *For any fixed k , $\text{VCSP}(\Gamma_{\{0,1\},k})$ can be solved in cubic time.*

Finally, we consider the class $\Gamma_{\text{sub},3}$ of ternary submodular cost functions over a Boolean domain. This class was studied in [1], from where we obtain the following useful characterisation of cubic submodular polynomials.

Lemma 14 ([1]). *A cubic polynomial $p(x_1, \dots, x_n)$ over Boolean variables represents a submodular cost function if and only if it can be written as*

$$\begin{aligned} p(x_1, \dots, x_n) = & a_0 + \sum_{\{i\} \in C_1^+} a_i x_i - \sum_{\{i\} \in C_1^-} a_i x_i - \sum_{\{i,j\} \in C_2} a_{ij} x_i x_j \\ & + \sum_{\{i,j,k\} \in C_3^+} a_{ijk} x_i x_j x_k - \sum_{\{i,j,k\} \in C_3^-} a_{ijk} x_i x_j x_k, \end{aligned}$$

where

1. $a_i, a_{ij}, a_{ijk} \geq 0$ ($\{i\} \in C_1^+ \cup C_1^-, \{i, j\} \in C_2, \{i, j, k\} \in C_3^+ \cup C_3^-$),
2. $\forall \{i, j\} \in C_2, a_{ij} + \sum_{k|\{i,j,k\} \in C_3^+} a_{ijk} \leq 0$.

Theorem 15. $\Gamma_{\text{sub},3} \subseteq \langle \Gamma_{\text{sub},2} \rangle$.

Proof. Let p be a polynomial representing an arbitrary cost function in $\Gamma_{\text{sub},3}$. By Lemma 14, the quadratic terms in p are non-positive. We already know how to express a negative cubic term using a gadget over $\Gamma_{\text{sub},2}$ (Theorem 9). To express a positive cubic term, consider the following identity:

$$x_i x_j x_k - x_i x_j - x_i x_k - x_j x_k = \min_{y \in \{0,1\}} \{(1 - x_i - x_j - x_k)y\}.$$

We can replace a positive cubic term $a_{ijk}x_i x_j x_k$ in p with

$$\min_{y \in \{0,1\}} \{a_{ijk}(1 - x_i - x_j - x_k)y + a_{ijk}(x_i x_j + x_i x_k + x_j x_k)\}.$$

It remains to check that all quadratic coefficients of the resulting polynomial are non-positive. However, this is ensured by the second condition from Lemma 14. $\square$

Corollary 16. $\text{VCSP}(\Gamma_{\text{sub},3})$ can be solved in cubic time.

5 Submodular Constraints of Arity 4 and Higher

In this section we investigate the question of which submodular constraints of arity 4 or higher can be expressed by binary submodular constraints. We derive a necessary condition for a 4-ary constraint over a Boolean domain to be submodular. We also present some sufficient conditions, which give rise to new classes of submodular constraints which can be expressed over $\Gamma_{\text{sub},2}$, and hence minimised efficiently. First, we prove the sufficient condition for 4-ary submodular cost functions. Next, we generalise it to k -ary submodular cost functions for every $k \geq 4$. Finally, we discuss the general question of which submodular cost functions over a Boolean domain can be expressed with binary submodular cost functions.

5.1 4-ary Constraints

One might hope to obtain a nice characterisation of 4-ary submodular cost functions over a Boolean domain similar to Lemma 14. However, it has been shown that testing whether a given quartic Boolean polynomial is submodular is co-NP-complete [12]. Hence, one is unlikely to find a “simple” characterisation; any characterisation is likely to involve an exponential blow-up (e.g., quantification over a non-constant number of variables). However, we can obtain the following necessary condition.

Lemma 17. *If a quartic polynomial $p(x_1, \dots, x_n)$ over Boolean variables represents a submodular cost function, then it can be written such that, for all $\{i, j\} \in C_2$:*

1. $a_{ij} \leq 0$, and
2. $a_{ij} + \sum_{k|\{i,j,k\} \in C_3^+} a_{ijk} + \sum_{k,l|\{i,j,k,l\} \in C_4^+} a_{ijkl} + F_{ij} \leq 0$

where $F_{i,j}$ is a non-positive value which is equal to the sum of the coefficients of certain non-positive cubic and quartic terms, C_2 denotes the set of quadratic terms, and C_i^+ denotes the set of terms of degree i with positive coefficients, for $i = 3, 4$.

Proof. Let p be a quartic submodular polynomial and let i and j be given, then p can always be put in a form so that the second order derivative is:

$$\begin{aligned} \delta_{i,j} = a_{i,j} &+ \sum_{k|\{i,j,k\} \in C_3^+} a_{ijk} x_k + \sum_{k,l|\{i,j,k,l\} \in C_4^+} a_{ijkl} x_k x_l \\ &- \sum_{k|\{i,j,k\} \in C_3^-} a_{ijk} x_k - \sum_{k,l|\{i,j,k,l\} \in C_4^-} a_{ijkl} x_k x_l. \end{aligned}$$

Consider an assignment which sets $x_i = x_j = 1$ and $x_k = 0 \ \forall k \neq i \neq j$. By Proposition 3, $a_{ij} \leq 0$, which proves the first condition. By setting $x_k = 1$ for all k such that $\{i, j, k\} \in C_3^+$ and $x_k = x_l = 1$ for all k, l such that $\{i, j, k, l\} \in C_4^+$, we get the second condition. We set to 1 all variables which occur in some positive cubic or quartic term. The second condition then says that the sum of all these positive coefficients minus those which are forced, by our setting of variables, to be 1 (F_{ij}), is at most 0. (Note that this also proves Lemma 14.) $\square$

Next we show a useful example of a 4-ary submodular cost function which can be expressed over the binary submodular cost functions using one extra variable.

Example 18. Let ϕ be the 4-ary cost function defined as follows: $\phi(\mathbf{x}) = \min\{2k, 5\}$, where k is the number of 0s in $\mathbf{x} \in \{0, 1\}^4$. The corresponding quartic polynomial representing ϕ is

$$p(x_1, x_2, x_3, x_4) = 5 + x_1 x_2 x_3 x_4 - x_1 x_2 - x_1 x_3 - x_1 x_4 - x_2 x_3 - x_2 x_4 - x_3 x_4.$$

Is is easy to check that p is submodular. It can be shown by simple case analysis that p cannot be expressed as a quadratic polynomial with non-positive quadratic coefficients (from the definition of p , the polynomial would have to be $5 - x_1 x_2 - x_1 x_3 - x_1 x_4 - x_2 x_3 - x_2 x_4 - x_3 x_4$ which is not equal to p on $x_1 = x_2 = x_3 = x_4 = 1$).

However, p can be expressed over $\Gamma_{\text{sub},2}$ using just one extra variable, via the following gadget:

$$p(x_1, x_2, x_3, x_4) = \min_{y \in \{0,1\}} \{5 + (3 - 2x_1 - 2x_2 - 2x_3 - 2x_4)y\}.$$

Using the same notation as in Lemma 17, define $\Gamma_{\text{new},4}$ to be the set of all 4-ary submodular cost functions over a Boolean domain whose corresponding quartic polynomials satisfy, for every $i < j$,

$$a_{ij} + \sum_{k|\{i,j,k\} \in C_3^+} a_{ijk} + \sum_{k,l|\{i,j,k,l\} \in C_4^+} a_{ijkl} \leq 0. \quad (1)$$

Theorem 19. $\Gamma_{\text{new},4} \subseteq \langle \Gamma_{\text{sub},2} \rangle$.

Proof. Let $\phi \in \Gamma_{\text{new},4}$ and let p be the corresponding polynomial which represents ϕ . First, replace all negative cubic and quartic terms using the construction in Theorem 9. As in the proof of Theorem 15, replace every positive cubic term $a_{ijk}x_i x_j x_k$ in p with

$$\min_{y \in \{0,1\}} \{a_{ijk}(1 - x_i - x_j - x_k)y + a_{ijk}(x_i x_j + x_i x_k + x_j x_k)\}.$$

Using the same construction as in Example 18, replace every positive quartic term $a_{ijkl}x_i x_j x_k x_l$ with

$$\begin{aligned} \min_{y \in \{0,1\}} \{ & a_{ijkl}(3 - 2x_i - 2x_j - 2x_k - 2x_l)y \\ & + a_{ijkl}(x_i x_j + x_i x_k + x_i x_l + x_j x_k + x_j x_l + x_k x_l)\}. \end{aligned}$$

It only remains to check that all quadratic coefficients in the resulting polynomial are non-positive. However, this is ensured by the definition of $\Gamma_{\text{new},4}$ and by the choice of the gadgets. $\square$

Corollary 20. $\text{VCSP}(\Gamma_{\text{new},4})$ can be solved in cubic time.

Unfortunately, our next example shows that $\Gamma_{\text{new},4} \subsetneq \Gamma_{\text{sub},4}$, and it remains an open question whether *all* 4-ary submodular constraints over a Boolean domain can be expressed over $\Gamma_{\text{sub},2}$.

Example 21. Define a 4-ary submodular cost function ϕ as follows: $\phi(\mathbf{x}) = \min(3k, 7) + 2y + z$, where k is the number of 0s in $\mathbf{x} \in \{0, 1\}^4$, $y = 1$ if and only if $\mathbf{x} = \langle 1, 1, 1, 0 \rangle$, and $z = 1$ if and only if $\mathbf{x} = \langle 1, 1, 0, 0 \rangle$. The corresponding polynomial representing ϕ is

$$\begin{aligned} p(x_1, x_2, x_3, x_4) = & 7 + 2x_1 x_2 x_3 x_4 - 2x_1 x_2 x_4 - x_1 x_3 x_4 - x_2 x_3 x_4 \\ & - x_1 x_3 - x_1 x_4 - x_2 x_3 - x_2 x_4 - x_3 x_4. \end{aligned}$$

It is easy to check that ϕ is submodular, but $\phi \notin \Gamma_{\text{new},4}$ (e.g., for $i = 1$ and $j = 2$, the expression in Equation 1 gives 2), so Theorem 19 does not apply.

As in Example 18, by a simple case analysis, it can be shown that ϕ cannot be expressed over $\Gamma_{\text{sub},2}$ without extra variables. However, the following gadget shows that ϕ is in fact expressible over $\Gamma_{\text{sub},2}$ using just two extra variables:

$$\begin{aligned} p(x_1, x_2, x_3, x_4) = & 7 - x_1 x_4 - x_2 x_4 - x_3 x_4 \\ & + \min_{y_1, y_2 \in \{0,1\}} \{2y_1 + 3y_2 - y_1 y_2 - y_1(x_1 + x_2 + 2x_3) - y_2(x_1 + x_2 + 2x_4)\}. \end{aligned}$$

5.2 The General Case

We now generalise the result from the previous section to subclasses of submodular constraints of arbitrary arities. We define $\Gamma_{\text{new},k}$ to be the set of all k -ary submodular cost functions over a Boolean domain whose corresponding polynomials satisfy, for every $1 \leq i < j \leq k$,

$$a_{ij} + \sum_{s=1}^{k-2} \sum_{\{i,j,i_1,\dots,i_s\} \in C_{s+2}^+} a_{i,j,i_1,\dots,i_s} \leq 0.$$

In other words, for any $1 \leq i < j \leq k$, the sum of a_{ij} and all positive coefficients of cubic and higher degree terms which include x_i and x_j is non-positive.

Theorem 22. *For every $k \geq 4$, $\Gamma_{\text{new},k} \subseteq \langle \Gamma_{\text{sub},2} \rangle$.*

Proof. Note that the case $k = 4$ is proved by Theorem 19. First we show that in order to prove the statement of the theorem, it is sufficient to have a uniform way of generating gadgets over $\Gamma_{\text{sub},2}$ for polynomials of the following type:

$$p_k(x_1, \dots, x_k) = \prod_{i=1}^k x_i - \sum_{1 \leq i < j \leq k} x_i x_j.$$

Note that $p_k(\mathbf{x}) = -\binom{m}{2}$, where m is the number of 1s in $\mathbf{x}$, and $\binom{0}{2} = \binom{1}{2} = 0$, unless $m = k$ ($\mathbf{x}$ consists of 1s only), in which case $p_k(\mathbf{x}) = -\binom{m}{2} + 1$.

Assume that for any $k \geq 5$, we can construct a gadget $\mathcal{P}_k$ for p_k over $\Gamma_{\text{sub},2}$. Given a cost function $\phi \in \Gamma_{\text{new},k}$, let p be the corresponding polynomial which represents ϕ . By the construction in Theorem 9, we can replace all negative terms of degree ≥ 3 . By the constructions in Theorem 15 and Theorem 19, we can replace all positive cubic and quartic terms. Now for any positive term of degree d , $5 \leq d \leq k$, we replace it with the gadget $\mathcal{P}_d$. This construction works if all quadratic coefficients of the resulting polynomial are non-positive. However, this is ensured by the definition of $\Gamma_{\text{new},k}$ and by the choice of the gadgets.

It remains to show how to uniformly generate gadgets for p_k , where $k \geq 5$. We claim, that for any $k \geq 4$, the following, denoted by $\mathcal{P}_k$, is a gadget for p_k :

$$p_k(x_1, \dots, x_k) = \min_{y_0, \dots, y_{k-4} \in \{0,1\}} \left\{ y_0 \left(3 - 2 \sum_{i=1}^k x_i \right) + \sum_{j=1}^{k-4} y_j \left(2 + j - \sum_{i=1}^k x_i \right) \right\}.$$

Notice that in the case of $k = 4$, the gadget corresponds to the gadget used in the proof of Theorem 19, and therefore the base case is proved. We proceed by induction in k . Assume that $\mathcal{P}_i$ is a gadget for p_i for every $i \leq k$. We prove that $\mathcal{P}_{k+1}$ is a gadget for p_{k+1} .

Firstly, take the gadget $\mathcal{P}_k$ for p_k , and replace every sum $\sum_{i=1}^k x_i$ with $\sum_{i=1}^{k+1} x_i$. We denote the new gadget $\mathcal{P}'$. It is not difficult to see that $\mathcal{P}'$ is a valid gadget for p_{k+1} on all assignments with at most $k - 1$ 1s. Also, on any

assignment with exactly k 1s, $\mathcal{P}'$ returns $-\binom{k}{2} + 1$. On the assignment consisting of 1s only, $\mathcal{P}'$ returns: $-\binom{k}{2} + 1 - 2 - 1(k - 4)$. This can be simplified as follows: $-\binom{k}{2} + 1 - 2 - k + 4 = -\binom{k}{2} + 1 - k + 2 = -(\binom{k}{2} + \binom{k}{1}) + 1 + 2 = -\binom{k+1}{2} + 1 + 2$. Hence $\mathcal{P}'$ is *almost* a gadget for p_{k+1} : we only need to subtract 1 on an assignment which has exactly k 1s, and subtract 2 on the assignment consisting of 1s only. But this is exactly what $\min_{y_{k-3} \in \{0,1\}} \{y_{k-3}(2 + (k-3) - \sum_{i=1}^{k+1} x_i)\}$ does. Therefore, we have established that $\mathcal{P}_{k+1}$ is a gadget for p_{k+1} , which finishes the proof of the theorem. $\square$

Corollary 23. *For any $k \geq 4$, $\text{VCSP}(\Gamma_{\text{new},k})$ can be solved in cubic time.*

The general question we are investigating is what can be expressed over $\Gamma_{\text{sub},2}$. Denote by $\langle \Gamma_{\text{sub},2} \rangle^m$ the set of all (submodular) cost functions expressible over $\Gamma_{\text{sub},2}$ with at most m extra variables. Clearly, $\langle \Gamma_{\text{sub},2} \rangle^m \subseteq \langle \Gamma_{\text{sub},2} \rangle^{m+1}$ for every $m \geq 0$.

In the proof of Theorem 22, we proved that, for any $k \geq 4$, $\Gamma_{\text{new},k} \subseteq \langle \Gamma_{\text{sub},2} \rangle^m$ where $m = k - 3$. We have since found out that a slightly stronger result was obtained independently by Zalesky. He has shown that $\Gamma_{\text{new},k} \subseteq \langle \Gamma_{\text{sub},2} \rangle^m$ where $m = \lfloor \frac{k-1}{2} \rfloor$ (see the unpublished manuscript [25]). This result yields the same cubic time complexity for $\text{VCSP}(\Gamma_{\text{new},k})$.

We saw in Section 5 that $\langle \Gamma_{\text{sub},2} \rangle^1$ is strictly larger than $\langle \Gamma_{\text{sub},2} \rangle^0$ (see Example 18). In other words, allowing a single hidden variable strictly increases the expressive power of $\Gamma_{\text{sub},2}$. On the other hand, we do not know whether allowing further hidden variables increases the expressive power any further. In other words, it is an open question whether $\langle \Gamma_{\text{sub},2} \rangle^m \subsetneq \langle \Gamma_{\text{sub},2} \rangle^{m+1}$ for any $m \geq 1$. We suspect that some of these inclusions are strict (see Example 21), as we carried out a computer-assisted search for gadgets, using the constraint-solver MINION³, and for some cost function we were not able to find a gadget with a given number of extra variables.

However, we do know that there is a limit to the additional expressive power that can be gained by allowing an arbitrary number of hidden variables. This is a consequence of the following result, which is a general result about expressibility, and not specific to submodular constraints or Boolean domains.

Proposition 24. *If a cost function $\phi : D^k \rightarrow \overline{\mathbb{Q}}_+$ is expressible over Γ , then ϕ is expressible over Γ using at most $|D|^{|D|^k}$ hidden variables.*

Proof. If $\phi \in \langle \Gamma \rangle$, then by Definition 2, there is a gadget $\langle \mathcal{P}, l \rangle$, where $l = \langle v_1, \dots, v_k \rangle$, for expressing ϕ over Γ . For the gadget $\langle \mathcal{P}, l \rangle$ to express ϕ , it has to define ϕ on each of the $|D|^k$ different assignments to l . Let each of these $|D|^k$ assignments be extended to a complete assignment to all variables of $\mathcal{P}$ (including hidden variables) in a way that minimises the total cost. For each hidden variable v of $\langle \mathcal{P}, l \rangle$, we can use the list of $|D|^k$ values assigned to v by these complete assignments to label the variable v . If there are more than $|D|^{|D|^k}$ hidden variables, then two of them will receive the same label. However,

³ Available from <http://minion.sourceforge.net/>

this implies that one of the two is redundant, as all constraints involving that variable can replace it with the other variable without changing the overall cost. Hence we require at most $|D|^{|D|^k}$ distinct hidden variables to express ϕ . $\square$

6 Conclusion

In this paper we first considered binary submodular constraints over a Boolean domain, and showed that they can be minimised in cubic time via a reduction to the minimum cut problem for graphs. We then investigated which other submodular constraints are *expressible* using binary submodular constraints over a Boolean domain, and hence can also be minimised efficiently using minimum cuts.

Using known results from combinatorial optimisation, we identified several such classes of constraints, including all ternary submodular constraints, and all $\{0, 1\}$ -valued submodular constraints of any arity. By constructing suitable gadgets, we identified certain new classes of k -ary submodular constraints, where $k \geq 4$, which can also be expressed by binary submodular constraints.

The main open problem raised by this paper is whether *all* bounded-arity submodular constraints over a Boolean domain can be expressed by binary submodular constraints, and hence solved in cubic time. In terms of polynomials, this is equivalent to the following problem: can any Boolean polynomial with non-positive second order derivatives be expressed as the projection of a quadratic polynomial with non-positive quadratic coefficients?

The results presented in this paper provide a partial answer to this question using constructive methods which can be used to obtain concrete reductions to problems such as (s, t) -MIN-CUT. We note that an alternative general approach to the problem of determining the expressive power of valued constraints was developed in [5]. It was shown there that the expressive power of any valued constraint language is characterised by a collection of algebraic properties called *fractional polymorphisms* [5]. In order to show that $\Gamma_{\text{sub},k} \subseteq \langle \Gamma_{\text{sub},2} \rangle$ it would therefore be sufficient to show that $\Gamma_{\text{sub},2}$ and $\Gamma_{\text{sub},k}$ have the same fractional polymorphisms. However, this algebraic approach is *non-constructive*, and hence has certain limitations: even if it could be established in this way that $\Gamma_{\text{sub},k} \subseteq \langle \Gamma_{\text{sub},2} \rangle$, this would not directly provide us with a gadget for any given problem (and hence an efficient algorithm). Conversely, if it could be established using the algebraic approach that $\Gamma_{\text{sub},k} \not\subseteq \langle \Gamma_{\text{sub},2} \rangle$, that would still leave open the question of identifying which subclasses of $\Gamma_{\text{sub},k}$ *can* be expressed over $\Gamma_{\text{sub},2}$, and hence solved efficiently. This paper provides a first step in answering that question using constructive techniques that could be implemented in valued constraint solvers.

Acknowledgements. The authors would like to thank David Cohen and Martin Cooper for fruitful discussions on submodular constraints and Chris Jefferson for help with using the constraint-solver MINION, which helped us to find and simplify some of the gadgets presented in this paper. Stanislav Živný gratefully acknowledges the support of EPSRC grant EP/F01161X/1.

References

1. Billionet, A., Minoux, M.: Maximizing a supermodular pseudo-boolean function: a polynomial algorithm for cubic functions. *D. App. Mathematics* 12, 1–11 (1985)
2. Bistarelli, S., Fargier, H., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G.: Semiring-based CSPs and valued CSPs: Frameworks, properties, and comparison. *Constraints* 4, 199–240 (1999)
3. Boros, E., Hammer, P.L.: Pseudo-boolean optimization. *Discrete Applied Mathematics* 123(1-3), 155–225 (2002)
4. Bulatov, A., Krokhin, A., Jeavons, P.: Classifying the complexity of constraints using finite algebras. *SIAM Journal on Computing* 34(3), 720–742 (2005)
5. Cohen, D., Cooper, M., Jeavons, P.: An algebraic characterisation of complexity for valued constraints. In: Benhamou, F. (ed.) *CP 2006. LNCS*, vol. 4204, pp. 107–121. Springer, Heidelberg (2006)
6. Cohen, D., Cooper, M., Jeavons, P.: Generalising submodularity and Horn clauses: Tractable optimization problems defined by tournament pair multimorphisms. *Theoretical Computer Science* (in press, 2008)
7. Cohen, D., Cooper, M., Jeavons, P., Krokhin, A.: Supermodular functions and the complexity of Max-CSP. *Discrete Applied Mathematics* 149, 53–72 (2005)
8. Cohen, D., Cooper, M., Jeavons, P., Krokhin, A.: The complexity of soft constraint satisfaction. *Artificial Intelligence* 170, 983–1016 (2006)
9. Cooper, M.C.: Minimization of locally defined submodular functions by optimal soft arc consistency. *Constraints* 13 (2008)
10. Cooper, M.: High-order consistency in valued constraint satisfaction. *Constraints* 10, 283–305 (2005)
11. Creignou, N., Khanna, S., Sudan, M.: Complexity Classification of Boolean Constraint Satisfaction Problems. *SIAM Monographs on Discrete Mathematics and Applications*, vol. 7. SIAM, Philadelphia (2001)
12. Gallo, G., Simeone, B.: On the supermodular knapsack problem. *Mathematical Programming* 45, 295–309 (1988)
13. Goldberg, A., Tarjan, R.: A new approach to the maximum flow problem. *Journal of the ACM* 35, 921–940 (1988)
14. Iwata, S.: Submodular function minimization. *Math. Progr.* 112, 45–64 (2008)
15. Jeavons, P., Cohen, D., Cooper, M.: Constraints, consistency and closure. *Artificial Intelligence* 101(1-2), 251–265 (1998)
16. Kolmogorov, V., Zabih, R.: What energy functions can be minimized via graph cuts? *IEEE Trans. Pattern Anal. Mach. Intell.* 26(2), 147–159 (2004)
17. Montanari, U.: Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences* 7, 95–132 (1974)
18. Nemhauser, G., Wolsey, L.: *Integer and Combinatorial Optimization* (1988)
19. Orlin, J.B.: A faster strongly polynomial time algorithm for submodular function minimization. In: Fischetti, M., Williamson, D.P. (eds.) *IPCO 2007. LNCS*, vol. 4513, pp. 240–251. Springer, Heidelberg (2007)
20. Rhys, J.: A selection problem of shared fixed costs and network flows. *Management Science* 17(3), 200–207 (1970)
21. Rossi, F., van Beek, P., Walsh, T. (eds.): *The Handbook of CP*. Elsevier, Amsterdam (2006)

22. Schiex, T., Fargier, H., Verfaillie, G.: Valued constraint satisfaction problems: hard and easy problems. In: IJCAI 1995, pp. 631–639 (1995)
23. Schrijver, A.: A combinatorial algorithm minimizing submodular functions in strongly polynomial time. *J. of Combinatorial Theory, Series B* 80, 346–355 (2000)
24. Schrijver, A.: *Combinatorial Optimization: Polyhedra and Efficiency. Algorithms and Combinatorics*, vol. 24. Springer, Heidelberg (2003)
25. Zalesky, B.: Efficient determination of Gibbs estimators with submodular energy functions. [arXiv:math/0304041v1](https://arxiv.org/abs/math/0304041v1) (February 2008)

Optimization of Simple Tabular Reduction for Table Constraints

Christophe Lecoutre

CRIL – CNRS UMR 8188,
Université Lille-Nord de France, Artois
rue de l’université, SP 16, F-62307 Lens
lecoutre@cril.fr

Abstract. Table constraints play an important role within constraint programming. Recently, many schemes or algorithms have been proposed to propagate table constraints or/and to compress their representation. We show that simple tabular reduction (STR), a technique proposed by J. Ullmann to dynamically maintain the tables of supports, is very often the most efficient practical approach to enforce generalized arc consistency within MAC. We also describe an optimization of STR which allows limiting the number of operations related to validity checking or search of supports. Interestingly enough, this optimization makes STR potentially r times faster where r is the arity of the constraint(s). The results of an extensive experimentation that we have conducted with respect to random and structured instances indicate that the optimized algorithm we propose is usually around twice as fast as the original STR and can be up to one order of magnitude faster than previous state-of-the-art algorithms on some series of instances.

1 Introduction

Arc Consistency (AC) plays a central role in Constraint Programming (CP). It is a property of constraint networks which can be exploited to identify and remove some inconsistent values, i.e. values which cannot lead to any solution. It is an essential component of the Maintaining Arc Consistency (MAC) algorithm, which is commonly used to solve binary instances of the Constraint Satisfaction Problem (CSP). It is also at the heart of stronger consistencies that have recently received some attention such as, e.g., singleton arc consistency [3,11], weak k -singleton arc consistency [19] and conservative dual consistency [12].

For non-binary constraints, which arise naturally in many applications, Generalized AC (GAC) replaces AC. Instead of using GAC extensions of generic AC algorithms, efficiency may be improved by exploiting the semantics/structure of the constraints. Indeed, enforcing GAC is NP-hard [4] and the best worst-case time complexity [2] that can be obtained with a generic GAC algorithm is $O(erd^r)$ where e denotes the number of constraints, d the greatest domain size and r the greatest constraint arity.

This paper is concerned with GAC algorithms for positive table constraints. A positive (resp. negative) table constraint is a constraint that is defined in extension by a set of allowed (resp. disallowed) tuples. Table constraints are commonly used in configuration applications or applications related to databases. Moreover, table constraints play

a particular role in constraint programming since they are easily handled by end-users of CP systems. Because any constraint can be theoretically translated into a table one (except that, in practice, this can lead to a time and space explosion), tables can be considered as the universal way of representing constraints.

In the last few years, many works have been devoted to table constraints. Many schemes or algorithms have been proposed to enforce GAC on table constraints or/and to compress their representation. In particular, one line of research (see [8,13,15]) aims to combine the two concepts of validity and acceptability of tuples of values. Significant formal and practical results have been obtained with respect to the very classical schemes [5]. Another recent proposal, called simple tabular reduction (STR) [18], significantly differs from previous methods: the principle is to dynamically maintain tables in order to only keep supports.

Our contribution in this paper is two-fold. First, we present the results of an extensive experimentation including both random and structured CSP instances which show that STR is quite competitive with respect to state-of-the-art algorithms (results in [18] were essentially given for random instances in the context of partition search). We can conclude that when GAC is maintained on table constraints during search, very often, STR is the most efficient approach. Second, we present an optimization of STR which allows limiting the number of validity checking and support search operations. Interestingly, we show that this optimization makes STR potentially r times faster where r is the arity of the constraint(s). It means that the algorithm we propose is particularly adapted to table constraints of large arity.

The paper is organized as follows. After introducing some technical background and related work, we present STR. Then, we describe how STR can be optimized. Before concluding, we present the results of an extensive experimentation.

2 Technical Background

A Constraint Network (CN) P is a pair $(\mathcal{X}, \mathcal{C})$ where $\mathcal{X}$ is a finite set of n variables and $\mathcal{C}$ a finite set of e constraints. Each variable $X \in \mathcal{X}$ has an associated domain, denoted $dom(X)$, that contains the set of values allowed for X . Each constraint $C \in \mathcal{C}$ involves an ordered subset of variables of $\mathcal{X}$ and has an associated relation, denoted $rel(C)$, which is the set of tuples allowed for this subset of variables. This subset of variables is the *scope* of C and is denoted $scp(C)$. The *arity* of a constraint is the number of variables in its scope. A *binary* constraint has arity 2.

A solution to a CN is an assignment of a value to each variable such that all the constraints are satisfied. A CN is said to be *satisfiable* iff it admits at least one solution. The Constraint Satisfaction Problem (CSP) is the NP-hard task of determining whether a given CN is satisfiable or not. A CSP instance is defined by a CN which is solved either by finding a solution or by proving unsatisfiability. In many cases, a CSP instance can be solved by using a combination of search and inferential simplification.

A central example of inferential simplification is enforcement of GAC, which removes inconsistent values from domains without reducing the set of solutions of the CN. Values are inconsistent if they cannot occur in any solution. In some cases, enforcement of GAC can yield a solution directly, without any search.

Before giving a technical definition of GAC, we introduce the notion of support. Given an ordered set $\{X_1, \dots, X_i, \dots, X_k\}$ of k variables and a k -tuple $\tau = (a_1, \dots, a_i, \dots, a_k)$ of values, the individual value a_i will be denoted by $\tau[i]$ and also $\tau[X_i]$ by abuse of notation. If C is a k -ary constraint, then the k -tuple τ is said to be allowed by C iff $\tau \in \text{rel}(C)$; a valid tuple of C iff $\forall X \in \text{scp}(C), \tau[X] \in \text{dom}(X)$; a support of C iff τ is a valid tuple of C which is allowed by C .

A pair (X, a) with $X \in \mathcal{X}$ and $a \in \text{dom}(X)$ will be called a *value* (of P). A tuple τ is a support for a value (X, a) in C iff $X \in \text{scp}(C)$ and τ is a support of C such that $\tau[X] = a$. Determining if a tuple is allowed is called a *constraint check* and determining if a tuple is valid is called a *validity check*. A value (X, a) of P is generalized arc-consistent (GAC) iff for every constraint C involving X , there exists a support for (X, a) in C . A variable X of P is GAC iff $\text{dom}(X) \neq \emptyset$ and $\forall a \in \text{dom}(X), (X, a)$ is GAC. P is GAC iff every variable of P is GAC.

It is easy to see that a value (X, a) of P which is not GAC cannot be included in any solution of P and is therefore inconsistent. Enforcing GAC means making the CN GAC by removing inconsistent values from domains. Many algorithms are available for enforcing GAC (or for enforcing AC when the constraints are binary) [2].

In this paper we use MAC which is a complete algorithm that is considered to provide the most efficient generic approach to the solution of CSP instances. MAC explores the search space depth-first, backtracks when dead-ends occur, and enforces (generalized) arc consistency after each decision taken (variable assignment or value refutation) during search. The *depth* of a node ν is the number of variable assignments performed along the path leading from the root of the search tree to ν . A *past* variable is (explicitly) assigned whereas a *future* variable is not (explicitly) assigned. $\text{fut}(C)$ is the set of future variables belonging to $\text{scp}(C)$. Finally, we emphasise that when GAC is enforced at a given step of the search, values can be removed only from domains of future variables.

3 Related Work on GAC for Table Constraints

A positive table constraint is a constraint given in extension and defined by a set of allowed tuples. Such constraints arise in practice in configuration problems, and more generally, in problems whose data come from databases. The set of allowed tuples associated with any constraint C is a table denoted by $C.\text{table}$. The worst-case space complexity of this table is $O(tr)$ where $t = |C.\text{table}|$ denotes the size of the table (i.e. the number of allowed tuples) and r denotes the arity of C .

GAC can be enforced by focusing in turn on each value in each domain; a value (X, a) is removed from its domain unless it is included in a support in every constraint that involves X . The classical generic GAC-valid scheme [5] seeks support by iterating over valid tuples (i.e. tuples that can be built from the current domains of constraint variables) until one is found that is allowed (i.e. accepted by the constraint). When working with table constraints we can instead seek support by iterating over allowed tuples until one is found to be valid [5]. From now on, these two schemes will be denoted by GACv and GACa, respectively. The efficiency of both approaches highly depends on the size of visited lists.

Recently, new schemes have been proposed, combining the exploitation of both valid and allowed tuples. In [15], it is shown that it is possible to skip over an exponential (in the arity) number of allowed tuples by reasoning from the current domains of variables. The key point is to know for each allowed tuple and each value the next tuple of the table that contains this value. To limit the space complexity of this approach which is in $O(trd)$, a sophisticated data structure, called hologram [14], can be used. To further improve the method, the authors propose to exploit lower bounds on supports such as, e.g., the *last* structure employed in AC2001/3.1.

Another approach, proposed in [13], involves visiting, in turn, lists of valid and allowed tuples. The principle is to avoid considering irrelevant tuples (when a support is looked for) by jumping over sequences of valid tuples containing no allowed tuple and over sequences of allowed tuples containing no valid tuple. This approach admits on some instances a behaviour quadratic in the arity of the constraints whereas classical schemes (GACv and GACa) admit an exponential behaviour. Interestingly, this approach whose worst-case space complexity is $O(tr)$ can be easily grafted to any generic GAC algorithm.

More recently, two additional data structures have been introduced [8] for table constraints. The most promising one corresponds to the tree structure called trie. A trie is a multi-way tree structure useful for storing strings over an alphabet. It can then be used to store large dictionaries of words. The original proposal in [8] is to represent the set of tuples of a constraint by a trie, and to explore this trie when looking for supports. In order to keep cheap the search of supports, the authors suggest to build one trie per variable (of the scope of the constraint), the first level being dedicated to it. The worst-case space complexity of the trie approach is $O(tr^2)$ but as tuples are compressed at the top of each trie, one can expect a better memory usage in practice.

Finally, in a recent work [10], an algorithm has been proposed to compress table constraints. The principle is to represent the initial set of tuples by subsets of the Cartesian product of the domains of the variables involved in the constraint. Interestingly, this approach is also suitable to negative table constraints. Indeed, from an initial set of disallowed tuples, the authors show it is possible to build a set of compressed allowed tuples, whose size is at most nd times the size of the original set. Among related approaches, one can cite the use of directed acyclic graphs (DAGs) [6] and binary decision diagrams (BDDs) [7].

4 Simple Tabular Reduction

Simple tabular reduction (STR) [18] is another original approach to enforce GAC on positive table constraints. The principle of STR is to dynamically maintain the tables of allowed tuples. More precisely, whenever a value is removed from the domain of a variable, the table associated with a constraint involving this variable is updated, removing all tuples that have become invalid. Values which are no more GAC are then (easily) identified and removed. To summarize, GAC is enforced while removing invalid tuples, and consequently, only supports are kept in tables. One related work [16] is the AC algorithm proposed for the hidden variable encoding.

We now formulate in more detail an implementation of STR in which each constraint is an object. Recall that *C.table* contains the initial set of tuples allowed by the

constraint C . Without any loss of generality, we hold this set within an array. The tuple that is the i^{th} element of $C.table$, denoted by $C.table[i]$, can be accessed in constant time. Within $C.table$, every tuple is a member of exactly one of a set of linked lists of tuples. One of these lists links all tuples that are currently valid (and consequently are supports): tuples in this list constitute the contents of the *current table* of C . Any tuple of the current table of C will be called a *current tuple*. To provide access to the disjoint lists of tuples within $C.table$ we introduce the following additional fields for each constraint object C :

- $C.first$ is the position (i.e. the subscript) of the first current tuple in $C.table$. $C.first = -1$ if the current table of C is empty.
- $removedHead$ is an array of size n such that $C.removedHead[d]$ is the position of a first invalid tuple of $C.table$ that was removed when the search was at depth d . $C.removedHead[d] = -1$ if none was removed at depth d .
- $removedTail$ is an array of size n such that $C.removedTail[d]$ gives the position of a last invalid tuple removed at depth d . $C.removedTail[d]$ is relevant only if $C.removedHead[d] \neq -1$.
- $next$ is an array of size $t = |C.table|$ that is used for linking lists of tuples. More precisely, if i is the position of a current tuple of C , then $C.next[i]$ indicates the position of the next tuple in the current table. $C.next[i] = -1$ if i is the position of the last current tuple. Similarly, if i is the position of a tuple removed at depth d , then $C.next[i]$ indicates the position of the next tuple removed at depth d , except that $C.next[i] = -1$ if i is now the position of the last invalid tuple removed at depth d .

Besides, corresponding to each variable X , we provide a set $gacValues[X]$ [17] that will contain all values in $dom(X)$ which are proved to have a support when enforcing GAC on a constraint C . With a $O(d)$ space consumption per variable, one can guarantee that all elementary operations (determining if a value is present, adding/removing a value, etc.) are performed in constant time (see [9,13]).

To enforce GAC at a given depth on a (positive table) constraint C using STR, Algorithm 1 is called. The loops at lines 1, 7 and 15 only iterate over future variables because it is only for these variables that it is possible to remove values from domains. This is an optimization wrt the original algorithm given in [18]. The sets $gacValues$ are emptied at lines 1 and 2 of Algorithm 1 because no value is initially guaranteed to be GAC. Then, all current tuples of the table of C are considered in turn by the loop at lines 4 – 14. When a tuple τ is proved to be valid (see Algorithm 2), we know that it is necessarily a support since it is (by definition) allowed. Values that have been proved to be GAC are collected at lines 7 to 9. In constant time (see Algorithm 3), at line 13 an invalid tuple τ is removed from the current table and put (at first position) into the list of invalid tuples that were removed at the current depth d . Note that τ is removed without actually being moved in memory. Once all current tuples have been considered, unsupported values are removed (lines 15 to 20): these are the values in $dom(X) \setminus gacValues[X]$. If a domain becomes empty then *false* is returned at line 18 because of inconsistency.

To enforce GAC on the constraint network, some events must be recorded: here, a variable is put in a queue dedicated to propagation (see line 20 of Algorithm 1)

Algorithm 1. $\text{GACstr}(C: \text{Constraint}, \text{depth}: \text{Integer}): \text{Boolean}$

```

1 foreach variable  $X \in \text{fut}(C)$  do
2    $\text{gacValues}[X] \leftarrow \emptyset$ 
3  $\text{prev} \leftarrow -1$  ;  $\text{curr} \leftarrow C.\text{first}$ 
4 while  $\text{curr} \neq -1$  do
5    $\tau \leftarrow C.\text{table}[\text{curr}]$ 
6   if  $\text{isValid}(C, \tau)$  then
7     foreach variable  $X \in \text{fut}(C)$  do
8       if  $\tau[X] \notin \text{gacValues}[X]$  then
9          $\text{add } \tau[X] \text{ to } \text{gacValues}[X]$ 
10     $\text{prev} \leftarrow \text{curr}$  ;  $\text{curr} \leftarrow C.\text{next}[\text{curr}]$ 
11  else
12     $\text{next} \leftarrow C.\text{next}[\text{curr}]$ 
13     $\text{removeTuple}(C, \text{prev}, \text{curr}, \text{depth})$ 
14     $\text{curr} \leftarrow \text{next}$ 
15 foreach variable  $X \in \text{fut}(C)$  do
16   if  $|\text{gacValues}[X]| \neq |\text{dom}(X)|$  then
17     if  $\text{gacValues}[X] = \emptyset$  then
18       return false
19      $\text{dom}(X) \leftarrow \text{gacValues}[X]$ 
20      $\text{add } X \text{ to } \text{propagationQueue}$ 
21 return true

```

Algorithm 2. $\text{isValid}(C: \text{Constraint}, \tau: \text{Tuple}): \text{Boolean}$

```

1 foreach variable  $X \in \text{scp}(C)$  do
2   if  $\tau[X] \notin \text{dom}(X)$  then
3     return false
4 return true

```

Algorithm 3. $\text{removeTuple}(C: \text{Constraint}, \text{prev}, \text{curr}, \text{depth}: \text{Integers})$

```

1 if  $\text{prev} = -1$  then  $C.\text{first} \leftarrow C.\text{next}[\text{curr}]$ 
2 else  $C.\text{next}[\text{prev}] \leftarrow C.\text{next}[\text{curr}]$ 
3  $C.\text{next}[\text{curr}] \leftarrow C.\text{removedHead}[\text{depth}]$ 
4 if  $C.\text{removedHead}[\text{depth}] = -1$  then  $C.\text{removedTail}[\text{depth}] \leftarrow \text{curr}$ 
5  $C.\text{removedHead}[\text{depth}] \leftarrow \text{curr}$ 

```

Algorithm 4. $\text{restoreTuples}(C: \text{Constraint}, \text{depth}: \text{Integer})$

```

1 if  $C.\text{removedHead}[\text{depth}] \neq -1$  then
2    $C.\text{next}[C.\text{removedTail}[\text{depth}]] \leftarrow C.\text{first}$ 
3    $C.\text{first} \leftarrow C.\text{removedHead}[\text{depth}]$ 
4    $C.\text{removedHead}[\text{depth}] \leftarrow -1$ 

```

whenever its domain is reduced. Later, this variable will be picked from the queue, and all constraints involving this variable will be enforced to be GAC (a call to GACstr will be performed for a positive table constraint). Also, the code given here can be easily adapted to take into account finer propagation events.

The worst-case time complexity of GACstr (Algorithm 1) is $O(r'd + rt')$ where, for a given constraint C , $r' = |fut(C)|$ denotes the number of future variables in C and t' the size of the current table of C . Indeed, loops at lines 1, 4 and 15 are $O(r')$, $O(rt')$ and $O(r'd)$, respectively. The worst-case space complexity of GACstr is $O(n + rt)$ per constraint since *removedHead* and *removedTail* are $O(n)$, *table* is $O(rt)$ and *next* is $O(t)$.

Importantly, when backtracking occurs, values must be restored to domains, as is well known, and because of domain restoration, tuples that were invalid may now be valid. If a tuple τ was removed from the current table of C at depth d , then τ must be restored to the current table of C when the search backtracks to depth d or assigns a new value at depth d . In our implementation, tuples are restored by calling Algorithm 4. This algorithm puts the list of invalid tuples removed at the given depth at the head of the current table. Restoration is achieved in constant time (for each constraint) without traversing either list and without moving any tuple in memory [18].

5 Optimizing STR

It is possible to improve STR in two directions. First, as soon as all values in the domain of a variable have been detected GAC, it is futile to continue to seek supports for values of this variable. We therefore introduce a set, S^{sup} , of variables in $fut(C)$ whose domain contains at least one value for which a support has not yet been found. In GACstr2 (Algorithm 5), which is an optimized version of GACstr, lines 1, 6 and 8 initialize S^{sup} to be the same as $fut(C)$. Whenever a support is found for the last unsupported value in the domain of a variable X , line 20 removes X from S^{sup} . If $|gacValues[X]| = |dom(X)|$ at line 19 then all values of $dom(X)$ are supported. Efficiency is gained by iterating only over variables in S^{sup} at lines 16 and 26.

The second direction of improvement avoids unnecessary validity operations. At the end of an invocation of GACstr for constraint C , every tuple τ such that $\tau[X] \notin dom(X)$ (with $X \in scp(C)$) has been removed from the current table of C . If there is no backtrack and $dom(X)$ does not change between this invocation and the next invocation, then at the time of this next invocation it is certainly true that $\tau[X] \in dom(X)$ for every tuple τ in the current table of C . In this case, there is no need to check whether $\tau[X] \in dom(X)$; efficiency is gained by omitting this check. We implement this optimization by means of a set S^{val} , which is the set of future variables whose domain has been reduced since the previous invocation of GACstr2. Initially, this set also contains the last assigned variable (denoted by *lastAssignedVariable* here) if it belongs to the scope of the constraint C . Indeed, after any variable assignment $X = a$, some tuples may become invalid due to the removal of some values in $dom(X)$. This is the only past variable for which validity operations must be performed. Algorithm 6 checks validity only for variables in S^{val} . The set S^{val} is first initialized at lines 2 through 5 of Algorithm 5. At line 9 of this algorithm, $dom(X).getLastRemovedValue()$ is the value that was most recently removed from $dom(X)$ whilst processing this or any

other constraint. A special value *nul* must be used when no value has been removed. $C.lastRemoved[X]$ is the value that was most recently removed from $dom(X)$ whilst processing the specific constraint C (see lines 11 and 30). If these two values differ at line 9 then $dom(X)$ has changed since the previous invocation of Algorithm 5 for the specific constraint C . In this case, X is included in S^{val} at line 10. This is how the membership of S^{val} is determined.

The worst-case time complexity of GACstr2 is $O(r'(d + t'))$. Indeed, performing a validity check is now $O(r')$ instead of $O(r)$, as it can be observed from Algorithm 6. Moreover, the loop starting at line 13 is in $O(r't')$. Like GACstr, the worst-case space complexity of GACstr2 is $O(n + rt)$ per constraint since data structures inherited from GACstr are $O(n + rt)$ and $lastRemoved$ is $O(r)$; S^{sup} and S^{val} are also $O(r)$ but may be shared by all constraints.

The worst case scenariï used to develop the worst-case time complexities of both GACstr and GACstr2 do not entirely characterize the difference in behaviour that may occur, in practice, between the two algorithms. Let us consider a positive table constraint C such that $scp(C) = \{X_1, \dots, X_r\}$ and the table initially includes:

```
(0, 0, . . . , 0)
(1, 1, . . . , 1)
. . .
(d-2, d-2, . . . , d-2)
(d-2, d-1, . . . , d-1)
. . .
```

In this example, the domain of each variable involved in C comprises all digits from 0 to $d - 1$. In the table, the first $d - 1$ tuples are sequences formed with the same digit (from 0 to $d - 2$), while the d^{th} tuple consists of the digit $d - 2$ followed by a sequence of $d - 1$. Assume that all variables are future, that STR (either of the two algorithms) is applied to this constraint and that no value is removed because all values are present in domains and there also exists a support for $(X_1, d - 1)$ in C (although this is not shown above). Now, imagine that $(X_1, d - 1)$ is deleted while propagating some other constraints, whereas all other values remain valid. If STR is applied again to this constraint, no value will be removed (since the constraint is still GAC), but some tuples (at least one) will be eliminated. Interestingly, calling GACstr requires $O(r)$ constant time operations to deal with *gacValues* structures (loops starting at line 1 and 15), $O(rt)$ operations to perform validity checks, $O(rt)$ operations to check GAC values and $O(rd)$ operations to collect GAC values. On the other hand, calling GACstr2 requires $O(r)$ operations to deal with *gacValues* structures, $O(t)$ operations to perform validity checks (since $S^{val} = \{X_1\}$), $O(rd)$ operations to check GAC values (since $S^{sup} = \emptyset$ after the treatment of the first d tuples) and $O(rd)$ operations to collect GAC values. This is summarized as follows:

Observation 1. *There exist situations where applying GACstr to a r -ary constraint C is $O(rt + rd)$ whereas applying GACstr2 is $O(t + rd)$.*

Most of the time, $d \ll t$ since $t \in O(d^r)$. In this case, Observation 1 shows that GACstr2 is potentially r times faster than GACstr. The higher the arity, the greater the

Algorithm 5. GACstr2(C : Constraint, $depth$: Integer): Boolean

```

1   $S^{sup} \leftarrow \emptyset$ 
2  if  $lastAssignedVariable \notin scp(C)$  then
3     $S^{val} \leftarrow \emptyset$ 
4  else
5     $S^{val} \leftarrow \{lastAssignedVariable\}$ 
6  foreach variable  $X \in fut(C)$  do
7     $gacValues[X] \leftarrow \emptyset$ 
8     $S^{sup} \leftarrow S^{sup} \cup \{X\}$ 
9    if  $dom(X).getLastRemovedValue() \neq C.lastRemoved[X]$  then
10      $S^{val} \leftarrow S^{val} \cup \{X\}$ 
11      $C.lastRemoved[X] \leftarrow dom(X).getLastRemovedValue()$ 
12  $prev \leftarrow -1$  ;  $curr \leftarrow C.first$ 
13 while  $curr \neq -1$  do
14    $\tau \leftarrow C.table[curr]$ 
15   if  $isValid(C, \tau)$  then
16     foreach variable  $X \in S^{sup}$  do
17       if  $\tau[X] \notin gacValues[X]$  then
18         add  $\tau[X]$  to  $gacValues[X]$ 
19         if  $|gacValues[X]| = |dom(X)|$  then
20            $S^{sup} \leftarrow S^{sup} \setminus \{X\}$ 
21      $prev \leftarrow curr$  ;  $curr \leftarrow C.next[curr]$ 
22   else
23      $next \leftarrow C.next[curr]$ 
24      $removeTuple(C, prev, curr, depth)$ 
25      $curr \leftarrow next$ 
26 foreach variable  $X \in S^{sup}$  do
27   if  $gacValues[X] = \emptyset$  then
28     return false
29    $dom(X) \leftarrow gacValues[X]$ 
30    $C.lastRemoved[X] \leftarrow dom(X).getLastRemovedValue()$ 
31   add  $X$  to  $propagationQueue$ 
32 return true

```

Algorithm 6. isValid(C : Constraint, τ : Tuple): Boolean

```

1  foreach variable  $X \in S^{val}$  do
2    if  $\tau[X] \notin dom(X)$  then
3      return false
4  return true

```

benefit of using GACstr2 may be. Finally, one may wonder about backtracking issues. A first solution, when backtracking occurs, is to reinitialize all arrays *lastRemoved*, filling them with the special value *nul*. Alternatively, one may record the content of

such arrays at each depth of search. Upon backtracking, one can then benefit from the original state of the arrays. This approach, which requires an additional structure that is $O(nr)$ per constraint, will be denoted by GACstr2+.

6 Experimental Results

In order to show the practical interest of simple tabular reduction, and in particular the optimization we propose, we have experimented using a cluster of Xeon 3.0GHz with 1GiB of RAM under Linux, employing MAC with *dom/dddeg* and *lexico* as variable¹ and value ordering heuristics, respectively. We have compared classical schemes to enforce GAC on (positive) table constraints with STR. More precisely, we have implemented GACv and GACa (see Section 3) as well as the scheme described in [13], denoted by GACva here. We do believe that GACva is a representative state-of-the-art algorithm for table constraints. Our own experience confirms the results reported in [8]: GACva and the trie approach are quite robust and close in terms of performance.

We have performed a first experimentation with random CSP instances. We have generated different classes of instances from Model RD [20]. Each generated class $\langle r, 60, 2, 20, t \rangle$ contains 20 CSP instances involving 60 Boolean variables and 20 r -ary constraints of tightness t . Whatever the arity $r \geq 8$ is, Theorem 2 [20] holds: an asymptotic phase transition is guaranteed at the threshold point $t_{cr} = 0.875$. It means that the hardest instances are generated when the tightness t is close to t_{cr} . Figure 1 shows the mean cpu time required to solve 20 instances of each class $\langle 13, 60, 2, 20, t \rangle$ where t ranges from 0.8 to 0.96. On these instances of intermediate difficulty, we can observe that STR is far more efficient than classical schemes (including GACva). When

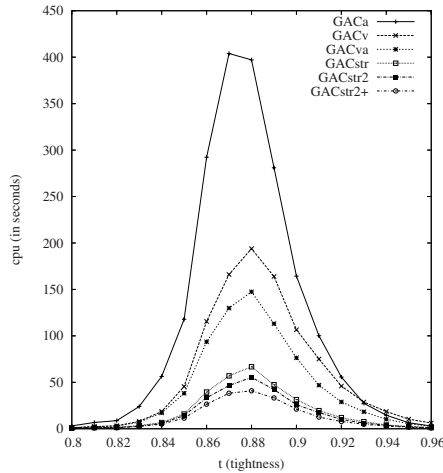


Fig. 1. Mean search cost of solving instances in classes $\langle 13, 60, 2, 20, t \rangle$ with MAC

¹ In our implementation, using *dom/wdeg* does not guarantee exploring the same search tree with classical and STR schemes.

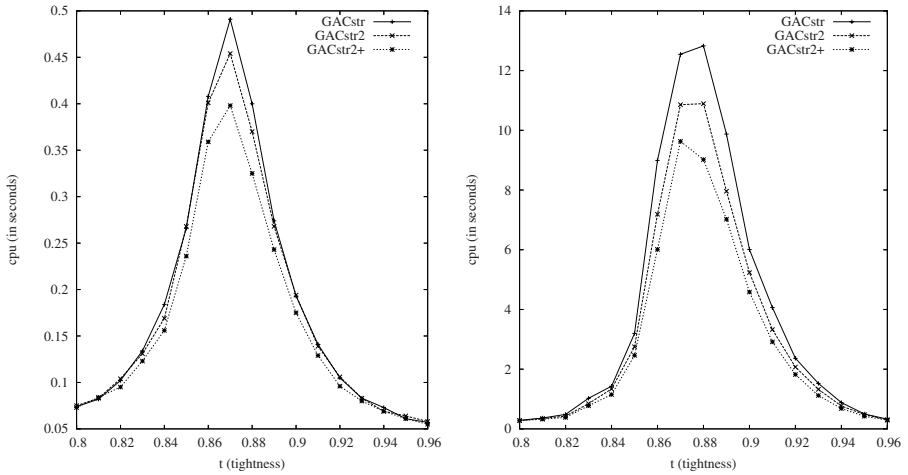


Fig. 2. Mean search cost for classes $\langle 10, 60, 2, 20, t \rangle$ (left) and $\langle 12, 60, 2, 20, t \rangle$ (right)

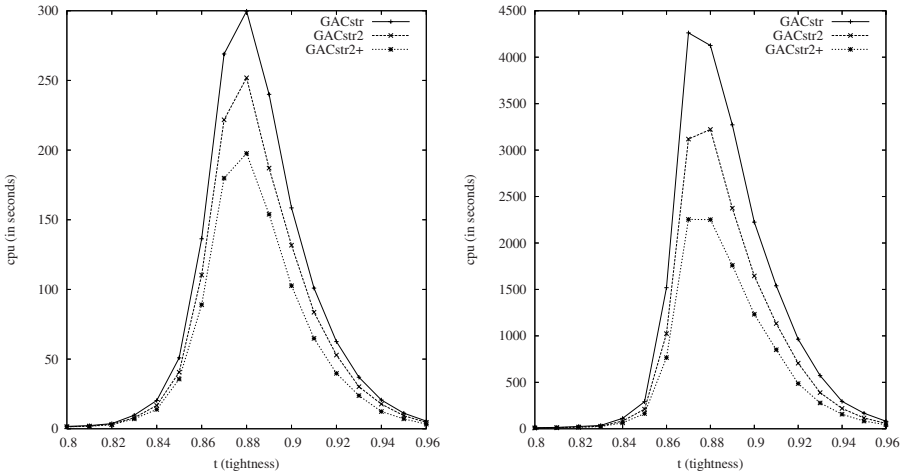


Fig. 3. Mean search cost for classes $\langle 14, 60, 2, 20, t \rangle$ (left) and $\langle 16, 60, 2, 20, t \rangle$ (right)

focusing on STR algorithms, Figures 2 and 3 clearly confirm the general observation made in Section 5 about the increasing interest of using GACstr2(+) when the arity of the constraints increases. Indeed, while GACstr2+ is about 20% faster than GACstr (at the threshold) when the arity of constraints is 10, it becomes two times faster when the arity of constraints is 16. Similar results have been obtained with larger domains.

Next, we have experimented on series of (random and structured) CSP instances involving table constraints, that are available from <http://www.cril.univ-artois.fr/~lecoutre/>. These series represent a large spectrum of instances, and importantly,

Table 1. Mean cpu time (in seconds) to solve instances of different series (a time-out of 1, 200 seconds was set per instance)

Series	#Inst	Classical GAC schemes			Simple Tabular Reduction		
		<i>GAC_v</i>	<i>GAC_a</i>	<i>GAC_{va}</i>	<i>GAC_{str}</i>	<i>GAC_{str2}</i>	<i>GAC_{str2+}</i>
bdd-21-2713-15	35	69.3	386	58.8	164	94.5	52.1
bdd-21-133-18	35	37.3	(23 out)	36.0	66.1	38.3	26.2
renault-mod	45	83.8	45.7	48.0	61.6	54.9	45.4
tsp-20	15	28.4	23.3	14.9	8.80	8.95	8.35
tsp-25	15	254	273	196	119	122	118
rand-8-20-5-18	20	107	(16 out)	119	108	81.2	65.6
rand-10-20-10-5	20	(20 out)	4.49	5.61	1.00	0.77	0.53

allow anyone to easily reproduce our experimentation. The two first series [7] bdd-21-2713-15 and bdd-21-133-18 contain 35 instances each, involving 21 Boolean variables and large and small BDD constraints of arity 15 and 18, respectively. The series renault-mod contains 45 real-world instances (we were unable to solve 5 of them with the selected heuristics within a reasonable amount of time) involving domains containing up to 42 values and constraints of various arity defined by large tables (the greatest one contains about 50, 000 6-tuples). The two series tsp-20 and tsp-25 contain 15 instances of the Travelling Salesperson Problem each, involving domains containing up to 1, 000 values and ternary constraints defined by large tables (about 20, 000 3-tuples). Finally, the two series rand-8-20-5-18 and rand-10-20-10-5 contain 20 random instances each involving 20 variables. Each instance of the series rand-8-20-5-18 (resp., rand-10-20-10-5) involves domains containing 5 (resp., 10) values and 18 (resp., 5) constraints of arity 8 (resp., 10). Tables contain about about 78, 000 and 10, 000 tuples, respectively.

Table 1 indicates the mean cpu time required to solve the instances of these different series. Overall, we can observe that *GAC_{str2+}* is always the most efficient approach. In particular, *GAC_{str2+}* is 3 times faster than *GAC_{str}* on the bdd-21-2713-15 series and 10 times faster than *GAC_{va}* on the rand-10-20-10-5 series. Table 2 presents the results obtained on some representative instances. Here, for each series, we show the results for 2 or 3 instances of various difficulty. For example, the instance bdd-21-133-18-10 only requires visiting 21 nodes (to be solved) whereas the instance bdd-21-133-18-11 requires visiting 14, 716 nodes. Typically, when the instance is easy, using STR is rather penalising. This is not really surprising since managing dynamic tables is then just an overhead. This is particularly visible for easy instances of series bdd-21-2713-15 and bdd-21-133-18. In terms of memory, the difference of memory consumption between all algorithms is at most by a factor 2. Note that the additional structure in $O(nd)$ required by *GAC_{str2+}* has a very limited practical impact on all these series.

Finally, we have tested the series of Crossword puzzles that have been recently posted at the web page mentioned earlier. For each grid, there is a variable per white square which can be assigned any of the 26 letters of the Latin alphabet, and a constraint for any sequence of white squares which corresponds to a word that we must put in the grid. Such constraints are defined by a table which contains all words of the right length. The series prefixed by cw-m1c are defined from blank grids and only contain positive table constraints (contrary to model m1 in [1] where no two identical words

Table 2. Representative results obtained on various structured and random instances. Cpu time is given in seconds and mem(ory) in MiB. The number of nodes (#nodes) explored by MAC is given below the name of each instance.

Instance		Classical GAC schemes			Simple Tabular Reduction		
		<i>GAC_v</i>	<i>GAC_a</i>	<i>GAC_{va}</i>	<i>GAC_{str}</i>	<i>GAC_{str2}</i>	<i>GAC_{str2+}</i>
bdd-21-133-18-10 #nodes=21	<i>cpu</i>	0.82	0.93	0.93	7.18	3.62	3.57
	<i>mem</i>	39M	43M	43M	63M	63M	63M
bdd-21-133-18-2 #nodes=10, 660	<i>cpu</i>	38.7	> 1, 200	38.7	68.1	38.6	25.9
	<i>mem</i>	61M		72M	127M	127M	126M
bdd-21-133-18-11 #nodes=14, 716	<i>cpu</i>	58.4	> 1, 200	53.6	104	61.1	43.9
	<i>mem</i>	46M		59M	100M	101M	101M
bdd-21-2713-15-22 #nodes=21	<i>cpu</i>	0.81	0.74	0.81	13.8	5.85	5.97
	<i>mem</i>	91M	93M	93M	165M	166M	175M
bdd-21-2713-15-32 #nodes=1, 140	<i>cpu</i>	61.5	357	55.1	145	82.7	44.5
	<i>mem</i>	73M	74M	74M	166M	167M	176M
bdd-21-2713-15-35 #nodes=1, 465	<i>cpu</i>	78.6	372	71.9	193	121	66.1
	<i>mem</i>	73M	74M	74M	167M	168M	177M
renault-mod-0 #nodes=287	<i>cpu</i>	11.1	1.05	1.04	1.05	0.99	1.04
	<i>mem</i>	36M	41M	41M	34M	34M	34M
renault-mod-12 #nodes=415K	<i>cpu</i>	149	92.2	88.9	92.4	83.7	77.6
	<i>mem</i>	39M	52M	52M	49M	49M	50M
renault-mod-14 #nodes=1, 135K	<i>cpu</i>	411	321	318	384	359	302
	<i>mem</i>	40M	51M	51M	66M	66M	68M
tsp-20-190 #nodes=7, 738	<i>cpu</i>	6.02	6.91	5.56	4.89	4.98	4.59
	<i>mem</i>	12M	12M	12M	10M	10M	10M
tsp-20-366 #nodes=31, 701	<i>cpu</i>	37.0	41.6	32.9	25.2	25.7	23.5
	<i>mem</i>	10M	10M	9, 731K	9, 115K	9, 124K	9, 261K
tsp-20-193 #nodes=80, 849	<i>cpu</i>	291	207	146	99.2	101	91.6
	<i>mem</i>	16M	17M	16M	17M	17M	17M
tsp-25-13 #nodes=2, 421	<i>cpu</i>	4.23	3.2	3.03	3.03	3.07	2.86
	<i>mem</i>	20M	20M	20M	17M	17M	17M
tsp-25-163 #nodes=89, 883	<i>cpu</i>	178	205	140	108	105	105
	<i>mem</i>	15M	15M	14M	15M	15M	16M
tsp-25-456 #nodes=686K	<i>cpu</i>	1, 060	1, 140	813	643	642	683
	<i>mem</i>	28M	28M	26M	40M	40M	42M
rand-10-20-10-5-10 #nodes=794	<i>cpu</i>	> 1, 200	3.86	2.59	0.58	0.51	0.43
	<i>mem</i>		20M	20M	16M	16M	16M
rand-10-20-10-5-0 #nodes=826	<i>cpu</i>	> 1, 200	4.59	3.22	1.19	1.36	0.83
	<i>mem</i>		23M	23M	20M	20M	20M
rand-8-20-5-18-10 #nodes=57, 579	<i>cpu</i>	42.7	> 1, 200	51.8	50.9	39.4	31.8
	<i>mem</i>	193M		283M	205M	205M	205M
rand-8-20-5-18-13 #nodes=569K	<i>cpu</i>	420	> 1, 200	403	241	186	153
	<i>mem</i>	196M		291M	221M	221M	221M

Table 3. Representative results obtained on series of Crossword puzzles using dictionaries of different length. Cpu time is given in seconds and mem(ory) in MiB. The number of nodes (#nodes) explored by MAC is given below the name of each instance.

		Classical GAC schemes			Simple Tabular Reduction		
		<i>GAC_v</i>	<i>GAC_a</i>	<i>GAC_{va}</i>	<i>GAC_{str}</i>	<i>GAC_{str2}</i>	<i>GAC_{str2+}</i>
Crossword puzzles with dictionary lex (24, 974 words)							
cw-m1c-lex-vg5-6	<i>cpu</i>	> 1, 200	38.8	54.2	14.3	12.4	10.7
#nodes=26, 679	<i>mem</i>		2, 889K	2, 928K	2, 932K	2, 935K	2, 968K
cw-m1c-lex-vg5-7	<i>cpu</i>	> 1, 200	357	875	134	114	96.3
#nodes=171K	<i>mem</i>		4, 134K	4, 173K	8, 005K	8, 055K	8, 059K
cw-m1c-lex-vg6-6	<i>cpu</i>	> 1, 200	2.98	4.29	1.28	1.05	0.91
#nodes=1, 602	<i>mem</i>		4, 422K	4, 344K	4, 226K	4, 203K	4, 296K
cw-m1c-lex-vg6-7	<i>cpu</i>	> 1, 200	436	1, 174	176	143	118
#nodes=152K	<i>mem</i>		5, 887K	5, 692K	9, 458K	9, 437K	9, 555K
Crossword puzzles with dictionary words (45, 371 words)							
cw-m1c-words-vg5-5	<i>cpu</i>	> 1, 200	0.04	0.05	0.05	0.05	0.04
#nodes=38	<i>mem</i>		4, 969K	4, 987K	4, 823K	4, 791K	4, 809K
cw-m1c-words-vg5-6	<i>cpu</i>	> 1, 200	1.19	1.46	0.48	0.37	0.33
#nodes=718	<i>mem</i>		6, 508K	6, 526K	6, 348K	6, 273K	6, 348K
cw-m1c-words-vg5-7	<i>cpu</i>	> 1, 200	18.6	36.0	6.61	5.21	4.03
#nodes=6, 957	<i>mem</i>		8, 470K	8, 489K	8, 276K	8, 145K	8, 237K
cw-m1c-words-vg5-8	<i>cpu</i>	> 1, 200	866	> 1, 200	273	229	187
#nodes=256K	<i>mem</i>		4, 604K		10M	10M	10M
Crossword puzzles with dictionary uk (225, 349 words)							
cw-m1c-uk-vg5-5	<i>cpu</i>	> 1, 200	0.05	0.05	0.1	0.07	0.07
#nodes=28	<i>mem</i>		12M	12M	12M	12M	12M
cw-m1c-uk-vg5-6	<i>cpu</i>	> 1, 200	0.55	0.5	0.21	0.17	0.17
#nodes=145	<i>mem</i>		17M	17M	16M	16M	16M
cw-m1c-uk-vg5-7	<i>cpu</i>	> 1, 200	2.97	5.18	0.51	0.37	0.34
#nodes=408	<i>mem</i>		22M	22M	22M	22M	22M
cw-m1c-uk-vg5-8	<i>cpu</i>	> 1, 200	82.5	71.9	7.08	5.71	4.78
#nodes=8, 148	<i>mem</i>		12M	12M	11M	11M	11M
Crossword puzzles with dictionary ogd (435, 705 words)							
cw-m1c-ogd-vg6-6	<i>cpu</i>	> 1, 200	0.37	0.31	0.23	0.17	0.15
#nodes=98	<i>mem</i>		46M	47M	46M	46M	48M
cw-m1c-ogd-vg6-7	<i>cpu</i>	> 1, 200	95.3	56.1	12.0	8.01	6.81
#nodes=9, 522	<i>mem</i>		11M	11M	11M	11M	11M
cw-m1c-ogd-vg6-8	<i>cpu</i>	> 1, 200	53.0	6.44	2.91	2.0	1.72
#nodes=2, 806	<i>mem</i>		24M	23M	22M	22M	24M
cw-m1c-ogd-vg6-9	<i>cpu</i>	> 1, 200	727	214	35.1	25.1	19.1
#nodes=23, 283	<i>mem</i>		42M	41M	39M	37M	40M

can be put in the grid, which is then naturally expressed in intension). The arity of the constraints is given by the size of the grids: for example, cw-m1c-lex-vg5-6 involves table constraints of arity 5 and 6 (the grid being 5 by 6). The results that we have obtained (see Table 3) with respect to 4 dictionaries (lex, words, uk, ogd) of different

length confirm our previous results. On the most difficult instances, GACstr2+ is about two times faster than GACstr and one order of magnitude faster than GACva. Note that we do not provide mean results for these series because many instances cannot be solved within 1, 200 seconds.

7 Conclusion

Simple tabular reduction (STR) [18] is a simple and effective GAC algorithm for positive table constraints. In this paper, we have proposed an optimization of this algorithm which significantly improves its efficiency. This new algorithm (GACstr2+) appears among state-of-the-art GAC algorithms for non-binary table constraints.

Acknowledgments

We would like to thank Julian Ullmann for fruitful discussions and helpful comments.

References

1. Beacham, A., Chen, X., Sillito, J., van Beek, P.: Constraint programming lessons learned from crossword puzzles. In: *Proceedings of Canadian Conference on AI*, pp. 78–87 (2001)
2. Bessière, C.: Constraint propagation. In: *Handbook of Constraint Programming* (2006)
3. Bessière, C., Debruyne, R.: Optimal and suboptimal singleton arc consistency algorithms. In: *Proceedings of IJCAI 2005*, pp. 54–59 (2005)
4. Bessière, C., Hebrard, E., Hnich, B., Walsh, T.: The complexity of reasoning with global constraints. *Constraints* 12(2), 239–259 (2007)
5. Bessière, C., Régin, J.: Arc consistency for general constraint networks: preliminary results. In: *Proceedings of IJCAI 1997*, pp. 398–404 (1997)
6. Carlsson, M.: Filtering for the case constraint. Talk given at Advanced School on Global constraints. Samos, Greece (2006)
7. Cheng, K., Yap, R.: Maintaining generalized arc consistency on ad-hoc n-ary Boolean constraints. In: *Proceedings of ECAI 2006*, pp. 78–82 (2006)
8. Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. In: *Proceedings of AAAI 2007*, pp. 191–197 (2007)
9. Van Hentenryck, P., Deville, Y., Teng, C.M.: A generic arc-consistency algorithm and its specializations. *Artificial Intelligence* 57, 291–321 (1992)
10. Katsirelos, G., Walsh, T.: A compression algorithm for large arity extensional constraints. In: Bessière, C. (ed.) *CP 2007*. LNCS, vol. 4741, pp. 379–393. Springer, Heidelberg (2007)
11. Lecoutre, C., Cardon, S.: A greedy approach to establish singleton arc consistency. In: *Proceedings of IJCAI 2005*, pp. 199–204 (2005)
12. Lecoutre, C., Cardon, S., Vion, J.: Conservative dual consistency. In: *Proceedings of AAAI 2007*, pp. 237–242 (2007)
13. Lecoutre, C., Szymanek, R.: Generalized arc consistency for positive table constraints. In: Benhamou, F. (ed.) *CP 2006*. LNCS, vol. 4204, pp. 284–298. Springer, Heidelberg (2006)
14. Lhomme, O.: Arc-consistency filtering algorithms for logical combinations of constraints. In: Régin, J.-C., Rueher, M. (eds.) *CPAIOR 2004*. LNCS, vol. 3011, pp. 209–224. Springer, Heidelberg (2004)

15. Lhomme, O., Régin, J.C.: A fast arc consistency algorithm for n-ary constraints. In: Proceedings of AAAI 2005, pp. 405–410 (2005)
16. Samaras, N., Stergiou, K.: Binary encodings of non-binary constraint satisfaction problems: algorithms and experimental results. *JAIR* 24, 641–684 (2005)
17. Ullmann, J.R.: A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words. *Computer Journal* 20(2), 141–147 (1977)
18. Ullmann, J.R.: Partition search for non-binary constraint satisfaction. *Information Science* 177, 3639–3678 (2007)
19. van Dongen, M.R.C.: Beyond singleton arc consistency. In: ECAI 2006, pp. 163–167 (2006)
20. 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)

Universal Booleanization of Constraint Models

Jinbo Huang

National ICT Australia and Australian National University
jinbo.huang@nicta.com.au

Abstract. While the efficiency and scalability of modern SAT technology offers an intriguing alternative approach to constraint solving via translation to SAT, previous work has mostly focused on the translation of specific types of constraints, such as pseudo Boolean constraints, finite integer linear constraints, and constraints given as explicit listings of allowed tuples. By contrast, we present a translation of constraint models to SAT at language level, using the recently proposed constraint modeling language MiniZinc, such that any satisfaction or optimization problem written in the language (not involving floats) can be automatically Booleanized and solved by one or more calls to a SAT solver. We discuss the strengths and weaknesses of such a universal constraint solver, and report on a large-scale empirical evaluation of it against two existing solvers for MiniZinc: the finite domain solver distributed with MiniZinc and one based on the Gecode constraint programming platform. Our results indicate that Booleanization indeed offers a competitive alternative, exhibiting superior performance on some classes of problems involving large numbers of constraints and complex integer arithmetic, in addition to, naturally, problems that are already largely Boolean.

1 Introduction

General constraint satisfaction and optimization problems are often solved with finite domain (FD) or linear programming (LP) techniques. When variables are all Boolean, however, satisfiability (SAT) solvers offer a natural solution whose efficiency and scalability in practice remain largely unmatched to date.

Booleanization of constraints has thus remained an ongoing quest in the constraint programming (CP) community. For example, pseudo-Boolean constraints (integer linear constraints over the domain $\{0, 1\}$), including the special case of Boolean cardinality constraints, have come under continual scrutiny due to their resemblance to their Boolean cousins [1,2,3,4,5]. Restricted to finite domains, general integer linear constraints have also been Booleanized by transforming all constraints to *primitive comparisons*, of the form $x \leq c$, and encoding each of these by a distinct Boolean variable [6]. Set constraints over a finite universe have been Booleanized by creating a Boolean variable for each possible element of a set [7]. For other types of constraints over finite domain variables, particularly extensional constraints (those given as explicit listings of allowed tuples), it's sometimes effective to use the well-known technique of encoding each value of a variable's domain with a distinct Boolean variable.

```

0  int: z = 10;
1  array [0..z] of 0..z*z: sq = [x*x | x in 0..z];
2  array [0..z] of var 0..z: s;
3  var 0..z: k;
4  var 1..z: j;
5  constraint forall ( i in 1..z ) ( s[i] > 0 -> s[i - 1] > s[i] );
6  constraint s[0] < k;
7  constraint sum ( i in 0..z ) ( sq[s[i]] ) = sq[k];
8  constraint s[j] > 0;
9  solve maximize j;

```

Fig. 1. A “perfect square” problem in MiniZinc

These previous lines of work have allowed CP to directly benefit from the great advances in SAT and related technologies over the past decade. In attempting to further push the envelope, however, one identifies two major limitations in this body of work. First, the types of constraints dealt with are limited; in particular, many nonlinear operations (such as multiplication/division, min/max, and the example given below involving array access) frequently required in modeling are left unsupported. Second, the techniques proposed, being specific to the respective types of constraints they target, are not necessarily compatible and are implemented for different problem specification formats, making the Booleanization of heterogeneous constraint models a difficult task both in theory and practice.

Both these limitations can be removed at once by a procedure that systematically Booleanizes a general constraint *language*, rather than special constraint types, which is the subject of the present paper. Although the techniques we shall present will apply as well to similar languages, our actual Booleanization procedure has been developed and implemented for MiniZinc [8], a recently proposed simple but expressive constraint modeling language. This choice of language is ideal for us as two existing solvers for MiniZinc are available for comparison, and the public distribution of MiniZinc [9] contains a large number of examples and benchmarks of different types, both of which facilitate the empirical evaluation of our new solver. An additional benefit of using MiniZinc, as we discuss in Section 2, is that it comes with a tool that “flattens” things into a more convenient subset of the language, without compromising its expressiveness.

Fig. 1 gives an example constraint model expressed in MiniZinc taken from the MiniZinc distribution [9], modeling the size-10 instance of the “perfect square” problem—finding a largest set of integers from $\{1, \dots, z\}$ the sum of whose squares is itself a square. While this example should be largely self-explanatory,¹ a detailed explanation of the syntax of MiniZinc can be found in [8]. It’s worth

¹ Line 0 declares a constant $z = 10$; line 1 declares an initialized array $sq = [0, 1, 2, 4, \dots, z^2]$; line 2 declares an array s of integers from domain $\{0, \dots, z\}$; lines 3 and 4 declare integers k and j from domains $\{0, \dots, z\}$ and $\{1, \dots, z\}$, respectively; line 5 asserts the logical condition $(s[i] > 0) \rightarrow (s[i - 1] > s[i])$ for all i in $\{1, \dots, z\}$; line 7 asserts $\sum_{i \in \{0, \dots, z\}} sq[s[i]] = sq[k]$.

noting here, though, that this small model, written in a natural way, already contains a type of constraint not handled by previous Booleanization methods: On line 7, we are summing over elements of an array ($sq[]$) using indices ($s[i]$) that are themselves variables, something that cannot be readily turned into a linear constraint, nor an explicit listing of allowed tuples of a reasonable length.

We shall now begin our journey toward a Booleanization of MiniZinc, which will allow problems such as this one to be automatically translated and solved by a SAT solver.

2 Basis for Universal Booleanization

Booleanization of constraint models at language level naturally consists of two parts: Boolean encoding of variables and Boolean encoding of constraints.

MiniZinc provides three scalar types: Booleans, integers, and floats; and two compound types: sets and arrays. In this work we do not consider floats. Hence we need to handle the following variable types: (1) integers (both bounded and unbounded), (2) sets of integers (must be bounded, a requirement in MiniZinc), (3) arrays of Booleans, (4) arrays of integers (both bounded and unbounded), and (5) arrays of sets of integers. We need not directly handle multi-dimensional arrays as MiniZinc comes with a tool that automatically flattens those, among other things, so that the model is rephrased in a subset of MiniZinc called FlatZinc [8], where the above five categories are the only possible types.

The basis for our Booleanization of an integer is to use k Boolean variables to represent the bits of the number in binary. This ensures that the encoding will be adequate for all possible types of constraints—as long as k is sufficiently large (we address this later). Sets are Booleanized as in [7], by using a Boolean variable for each possible element of the set (the boundedness helps here). Arrays are decomposed into individual variables, one for each index, which is feasible as MiniZinc requires that array index ranges be fixed at compile time.

The second, more substantial part of the task is to Booleanize all types of constraints that can be written in MiniZinc. This is facilitated again by the tool that converts MiniZinc to FlatZinc, where all user defined predicates have been inlined, compound operators (**forall**, **sum**, and **product**, which range over elements of an array) unrolled, and all constraints normalized as necessary to conform to a pre-defined set of *operators* (i.e., constraint types). It then remains for us to provide a translation for each of these operators.

The next section presents our Booleanization procedure in more detail, assuming that MiniZinc models have been converted to FlatZinc in a preprocessing step (note that this makes our Booleanization applicable not only to MiniZinc, but potentially any language that can be translated to FlatZinc). A FlatZinc model consists of a list of variables declarations followed by a list of constraints and finally a specification of the nature of the problem (satisfaction/optimization) and the desired output—all of these are known as *items* of the model, and we will describe the translation of them in that order.

3 An Itemized Procedure

It's perhaps helpful to interpose a description of our “finished product” here, so that the goal will be clear when we go through our translation procedure. This is a program called FZNTINI, as it takes FlatZinc models as input and solves them using an interface to the SAT solver TINISAT [10]. It also has the option of printing a Boolean translation of the model (for a given k —the number of bits used to encode an integer) without solving it, in one of two formats: Boolean FlatZinc (the subset of FlatZinc where all variables are Boolean and all constraints are basic Boolean operations; this is to allow the integration of the Booleanization into the G12 platform [11] which understands FlatZinc) and the DIMACS CNF format widely accepted by SAT solvers. For optimization problems, the translation encodes all the constraints, and information on optimization (i.e., which Boolean variables correspond to the integer variable to be maximized/minimized, and its lowerbound and upperbound if known) is coded as an annotation item in the case of Boolean FlatZinc and a special comment line in the case of DIMACS, so that solvers accepting these translations can reconstruct the original optimization version of the problem and solve it accordingly.

Going from Boolean FlatZinc to DIMACS will be straightforward, as it involves simply converting basic Boolean operations to CNF; hence we will describe our translation using Boolean FlatZinc as the target language. Language syntax will be explained as we go along.

3.1 Booleanization of Variable Declarations

Declarations of Boolean variables are left untouched. Hence we have the following five cases to handle as explained earlier.

Integers. An integer declaration in FlatZinc has the following form:

```
var int: x;
```

which translates into declarations of k Boolean variables:

```
var bool: x_1; ... ; var bool: x_k;
```

Note that the symbols x_1 through x_k are meant to represent k fresh identifiers that are not used in the source model or the translation of previous items. This convention applies through the rest of the paper.

A bounded integer is declared as follows:

```
var <g>..<h>: x;
```

where $\langle g \rangle$ and $\langle h \rangle$ are two integer constants giving the domain (e.g., 0..10). This translates into the same k Boolean variables as above, but we also add the following two (less-than-or-equal-to) integer comparison constraints, which are then Booleanized along with other constraints in the source model, as will be described in Section 3.2 (note that depending on the values of g and h , these constraints may fix some of the bits of x to constants, effectively reducing the actual number of bits required to encode a bounded integer):

```
int_le(<g>, x); int_le(x, <h>);
```

Sets of integers. A set of (bounded) integers is declared as follows:

```
var set of <g>..

```

which means that the value of S must be a set whose elements are from the universe $\{g, \dots, h\}$. This translates into declarations of $h - g + 1$ Boolean variables:

```
var bool: S_g; ...; var bool: S_h;
```

where each S_i encodes the proposition “ $i \in S$.”

Arrays of Booleans. Arrays in FlatZinc are always 0-indexed.² A declaration of an array of Booleans has the following form:

```
array[0..] of var bool: X;
```

which translates into declarations of $m + 1$ Boolean variables:

```
var bool: X_0; ...; var bool: X_m;
```

Arrays of integers. Each declaration of an array of integers

```
array[0..] of var int: X;
```

is first decomposed (conceptually) into declarations of $m + 1$ integer variables:

```
var int: X_0; ...; var int: X_m;
```

which are then Booleanized the same way as other integers in the source model (in practice, of course, the intermediate step need not take place explicitly).

Each declaration of an array of bounded integers

```
array[0..] of var <g>..

```

translates into the same Boolean variables as in the unbounded case, plus the translation of additional integer comparison constraints as in the case of bounded scalar integers.

Arrays of sets of integers. Each declaration of an array of sets of integers:

```
array[0..] of var set of <g>..

```

is first decomposed (conceptually) into declarations of $m + 1$ set variables:

```
var set of <g>..

```

which are then Booleanized the same way as other set variables in the source model (again, the intermediate step need not actually take place).

Finally, we note that these five types of variable declarations can all take an (initialization) assignment, in which case the “variables” effectively become

² Subsequent to the completion of this work, the Zinc family of languages and the MiniZinc benchmarks and examples have been modified to use 1-indexed arrays.

constants and the `var` keyword in the declaration can be omitted (as in lines 0–1 of Fig. 1). These cases can be handled in one of two ways: (1) We can simply skip the declaration, storing the value of the variable in a look-up table, and plug in the value when later translating a constraint involving that variable. This method can be easily implemented for Booleans, integers, arrays of Booleans, and arrays of integers. (2) We can Booleanize the variables the same way as if they weren’t initialized, and then add constraints equating the resulting Boolean variables to appropriate Boolean constants corresponding to the assignment. This method has the advantage of uniformity, in that when we later translate the constraints, initialized and uninitialized variables need not be distinguished. This makes it easier to implement for the more complex variables types of sets and arrays of sets. This is a relatively unimportant choice, after all, and our implementation uses a combination of both methods.

3.2 Booleanization of Constraints

Booleanization of constraints involving integers will depend on how an integer is represented by the k bits. We assume the *two’s complement* representation commonly used in computers, where a positive number has the usual representation and flipping all its bits and adding 1 gives its negation. For example, if $k = 4$, then 3 would be 0011 and -3 would be 1101. We also assume that $\mathbf{x}_k$ represents the most significant, and $\mathbf{x}_1$ the least significant bit of $\mathbf{x}$.

Constraints in FlatZinc are instances of a pre-defined set of operators, grouped into several categories. Below we will use these as headings to present translations of constraints. In FlatZinc, arguments to a constraint can be either a variable, array access, or constant. For simplicity, we will represent arguments as variables in all operators (except in linear constraints where the coefficients must be constants). The other two cases can be handled in the following straightforward way: (1) Any argument in the form of $\mathbf{x}[\mathbf{i}]$ (array access with a constant index) is replaced with $\mathbf{x}_{\mathbf{i}}$ before applying the relevant translation procedure (recall that these individual variables have been invented in translating the array declaration); note that array access with a variable index would not appear as such in FlatZinc, but will have been normalized during flattening into *array element operators*, which we will cover below. (2) Constants can be handled by (conceptually) inventing a temporary variable in its place and plugging in appropriate Boolean constants at the end.

Comparison operators. There are three sets of comparison operators in FlatZinc, for Booleans, integers, and sets, respectively. Each set contains six operators corresponding to $=$, $\neq$, $\leq$, $<$, $\geq$, and $>$. The Boolean operators need not be touched. Hence we consider the Booleanization of integer and set comparisons. For both of these, we use the $\leq$ as a representative as the cases of $<$, $\geq$, and $>$ are similar and the cases of $=$ and $\neq$ are simpler. All these operators have reified versions, which we also omit as the modifications required are straightforward.

The $\leq$ comparison of two integer variables in FlatZinc, `int_le(x, y)`, can be translated by simulating Algorithm 1, which performs the comparison of two

Algorithm 1. Comparison ($\leq$) of two k -bit integers (in two's complement)

`int_le( $x, y, k$ ): assuming  $x_k \dots x_1$  ( $y_k \dots y_1$ ) are the bits of  $x$  ( $y$ )`

`1: return  $(x_k > y_k) \vee ((x_k = y_k) \wedge \text{unsigned\_int\_le}(x, y, k - 1))$`
`unsigned_int_le( $x, y, k$ )`
`2: if  $k = 1$  return  $x_1 \leq y_1$`
`3: return  $(x_k < y_k) \vee ((x_k = y_k) \wedge \text{unsigned\_int\_le}(x, y, k - 1))$`

```

bool_gt_reif(x_k, y_k, a_1);      % a_1 = (x_k > y_k)
bool_eq_reif(x_k, y_k, a_2);      % a_2 = (x_k = y_k)

bool_lt_reif(x_{k-1}, y_{k-1}, b_1) % b_1 = (x_{k-1} < y_{k-1})
bool_eq_reif(x_{k-1}, y_{k-1}, b_2) % b_2 = (x_{k-1} = y_{k-1})
...
bool_lt_reif(x_2, y_2, b_{2k-5}) % b_{2k-5} = (x_2 < y_2)
bool_eq_reif(x_2, y_2, b_{2k-4}) % b_{2k-4} = (x_2 = y_2)

bool_le_reif(x_1, y_1, b_{2k-3}) % b_{2k-3} = (x_1 <= y_1)

bool_or(a_1, c_1, true);          % true = (a_1 or c_1)
bool_and(a_2, c_2, c_1);          % c_1 = (a_2 and c_2)
                                  % rest is recursive definition of c_i

bool_or(b_1, c_3, c_2);
bool_and(b_2, c_4, c_3);
...
bool_or(b_{2k-5}, c_{2k-3}, c_{2k-4});
bool_and(b_{2k-4}, c_{2k-2}, c_{2k-3});

bool_eq(c_{2k-2}, b_{2k-3});      % base case for recursion

```

Fig. 2. Comparison ($\leq$) of two k -bit integers in Boolean FlatZinc

integers at bit level. Recall that in two's complement $x_k = 1$ signifies a negative number and $x_k = 0$ signifies a nonnegative number. Hence line 1 of Algorithm 1 says that $x \leq y$ if $x < 0$ and $y \geq 0$, or if x and y have the same sign bit and the remaining bits of x are $\leq$ those of y , both taken as unsigned numbers. The comparison of unsigned numbers is then implemented as a recursive function on lines 2–3, where the logic should be straightforward.

Now to convert this into a set of Boolean constraints, we unroll the recursion, introduce auxiliary variable a_1, a_2 , and b_i for $i \in \{1, \dots, 2k - 3\}$ to encode the various bit comparisons on lines 1 and 3 of the algorithm, and introduce auxiliary variables c_{2i} to encode the value of `unsigned_int_le( $x, y, i$ )` and c_{2i-1} to encode the conjunction on line 3, for each $i \in \{1, \dots, k - 1\}$. Fig. 2 shows the resulting translation in Boolean FlatZinc, where comments have been added (after the % signs) to explain the meanings of the operators (the `constraint` keyword has been omitted from the beginning of each line).

The conversion of Algorithm 1 into the Boolean FlatZinc in Fig. 2 illustrates how such conversions may be done in general. Hence in the rest of the section we will refrain from giving actual Boolean FlatZinc code (which would be space-consuming) and focus on describing the bit-level algorithms or logical formulas, and sometimes just the general ideas, behind the Booleanization.

Algorithm 2. Comparison ($\leq$) of two sets of integers over the universe $\{g, \dots, h\}$

`set_le( $X, Y, g, h$ ):` assuming $X_g \dots X_h$ ($Y_g \dots Y_h$) are the Boolean variables created in translating the declaration of set X (Y) as described in Section 3.1

```

1: if  $g = h$  return  $X_g \leq Y_g$ 
2: return  $((X_g < Y_g) \wedge (\bigwedge_{i=g+1}^h \neg X_i))$ 
3:    $\vee((X_g > Y_g) \wedge (\bigvee_{i=g+1}^h X_i))$ 
4:    $\vee((X_g = Y_g) \wedge \text{set\_le}(X, Y, g + 1, h))$ 

```

We now turn to the $\leq$ comparison of sets, which in FlatZinc is defined lexicographically as if a set is a “string” made up of its elements (in increasing order). For example: $\{1, 2, 3\} \leq \{2\} \leq \{2, 3\}$. Note that this differs from the lexicographical comparison of the characteristic bit strings of the sets (which would be 111, 010, and 011 in this example, assuming a common universe of $\{1, 2, 3\}$), or the “unsigned_int_le” function from Algorithm 1 would have sufficed.

Algorithm 2 gives a recursive bit-level implementation of the $\leq$ comparison of sets over the common universe $\{g, \dots, h\}$, assuming that the set variables have themselves been Booleanized as described in Section 3.1. Line 1 is the straightforward base case, where there is only one possible element in both sets. Otherwise we examine the first pair of characteristic bits (X_g and Y_g), and there are three cases. If $X_g < Y_g$ (line 2), then $g \notin X$ and $g \in Y$, and hence $X \leq Y$ iff X is empty (the second conjunct on line 2), because any other element will be greater than g and thus make $X > Y$. The second case (line 3) can be similarly analyzed and finally if the pair are equal then we recurse (line 4).

The case of set variables defined over different universes can be handled using the trick of inventing temporary variables. Specifically, we need only let $\{g, \dots, h\}$ be the smallest range containing the universes of both sets, and continue to use Algorithm 2, with all X_i and Y_j replaced with false for all i outside X ’s universe and j outside Y ’s.

Arithmetic operators. FlatZinc provides the following arithmetic operators: negation, addition, subtraction, multiplication, division, modulo, absolute value, min, and max. These can all be Booleanized using standard algorithms that perform the corresponding operations on binary numbers. We will not go into their details, but let us use the space here to mention the more critical issue of overflow protection.

In computers, arithmetic overflow can lead to an incorrect result; in the Booleanization of a constraint model, it can lead to spurious solutions if not guarded against. Our solution is analogous to what’s implemented in computer hardware. For addition, both operands as well as the sum are extended by one bit (at the high end) and a constraint is added requiring the leading two bits of the sum to be identical. For multiplication, the product will temporarily have $2k$ bits, and we add constraints to ensure the leading $k + 1$ bits are identical (the lower k bits then correctly represent the result). The other cases can be similarly handled where necessary.

Linear equality and inequalities. An integer linear equality constraint comes in the following form in FlatZinc:

```
int_lin_eq([C1, ..., Cn], [X1, ..., Xn], rhs);
```

where $C1, \dots, Cn$ and rhs are integer constants and $X1, \dots, Xn$ are integer variables (can also be constants or array accesses; see paragraph 2 of Section 3.2).

This encodes the equality $\sum_i C_i X_i = rhs$.

Inequality constraints have the same form, with the “eq” in the name of the operator replaced by “ne, le, lt, ge, gt” respectively for $\neq, \leq, <, \geq, >$. All of them have a reified version taking a Boolean as the fourth argument.

We handle these linear constraints by breaking them down to individual multiplications and additions, followed by an integer comparison, all of which we already know how to Booleanize (introducing auxiliary variables to encode intermediate results).

Set operators. FlatZinc provides the following set operators: membership, cardinality, subset, superset, union, intersection, difference, and symmetric difference. Again, for simplicity we assume that sets involved in the same operator are defined over the same universe $\{g, \dots, h\}$ (otherwise the trick of inventing temporary variables can be applied as described earlier).

The set membership operator, `set_in(x, Y)`, which encodes $x \in Y$, translates into $\bigvee_{i=g}^h ((x = i) \wedge Y_i)$. Note that the comparison $x = i$ is on integers, but we already know how to Booleanize those and can easily expand the expression into a purely Boolean one. Set cardinality, `set_card(X, y)`, can be translated by adding up the Booleans X_i as integers, and equating the sum to y , both of which we know how to Booleanize (if y is a constant, then a unary representation of integers can offer more propagation power in this case [1]). Set inclusion operators, `set_subset(X, Y)` and `set_superset(X, Y)`, are the simplest case, translating into $\bigwedge_{i=g}^h (X_i \leq Y_i)$ and $\bigwedge_{i=g}^h (X_i \geq Y_i)$, respectively.

The remaining operators all take three arguments, with the final one holding the result of the operation. The union operator, `set_union(X, Y, Z)`, for example, encodes $X \cup Y = Z$, and translates into $\bigwedge_{i=g}^h ((X_i \vee Y_i) = Z_i)$. Replacing the disjunction ($\vee$) in this formula with $\wedge, >, \text{ and } \neq$, respectively, gives a translation for the other operators, intersection, difference, and symmetric difference.

Array element operators. Array access with a variable index is expressed in FlatZinc indirectly, by first equating the array element with a new variable, via *array element operators*, and then using that variable in other constraints where the array access is required. The following operator applies to an array of Booleans, encoding $Y[x] = z$:

```
array_var_bool_element(x, Y, z);
```

Let m be the highest index of the array. This operation then translates into $\bigvee_{i=0}^m ((x = i) \wedge (Y_i = z))$. Again, the integer comparisons $x = i$ are meant to be further Booleanized and the results plugged in.

The corresponding operators for arrays of integers and arrays of sets translate into the same formula as above, except that $Y_i = z$ will be an integer comparison, and set comparison, respectively, which we already know how to Booleanize.

Global constraints. In addition to the above groups of operators, a FlatZinc model can also use global constraints. At the moment, `all_different` (over an integer array) is the only type of global constraints that is supported by MiniZinc but not implemented by the official MiniZinc-to-FlatZinc translator. We Booleanize an `all_different` constraint over n elements by turning it into $n(n-1)/2$ inequalities and Booleanizing them as we do comparison operators.

3.3 Booleanization of Solve and Output Items

FlatZinc provides three types of solve items: `solve satisfy`, `solve minimize x`, and `solve maximize x`, where x is an integer variable. Optimization of an expression is provided for indirectly: One introduces constraints equating the expression with a new variable, and optimizes that variable instead.

In Booleanization a `solve satisfy` item is left untouched. In the case of `solve minimize x` and `solve maximize x`, it's not possible for a single (ordinary) Boolean translation to encode the original optimization problem. Our solution is to turn them both into a `solve satisfy`, and, as mentioned in the beginning of the section, use an annotation or comment depending on the target format (Boolean FlatZinc or DIMACS CNF) to encode the information necessary for the optimization version of the problem to be reconstructed. Such translations will be complete and solvers accepting them are then free to decide how to handle the optimization.

FZNTINI solves optimization problems by an uninformed binary search, reducing the domain of the objective variable by at least half at each step. Since integers are encoded using k bits, any optimization problem can be solved this way by at most $k+1$ calls to the SAT solver. Note that these successive calls will involve the original CNF formula plus different sets of new clauses encoding the bounds that are being tested. SAT solvers that support incremental addition and removal of clauses will hence be particularly suitable for these tasks.

For both satisfaction and optimization problems, FZNTINI starts with a k sufficient to encode all constants in the problem, and automatically increases it until either a solution is found, or k reaches the size of a C++ int (typically 32) on the machine on which it's run (this exceeds the capacity of G12/FD, which is fixed at 22 bits). Note that since overflow protection is in place, a solution found under any k is guaranteed to correspond to a correct solution for the original problem, while an “unsatisfiable” answer could just mean that k is not large enough. By the same token, for optimization problems the guarantee of the optimality of a solution returned by FZNTINI is conditioned on the k used. In many problems, however, bounds on the objective variable are given either implicitly or explicitly, in which case the guarantee of optimality provided by FZNTINI will be absolute.

Lastly, output items are also encoded as annotations or comments so solvers can print output back in ordinary decimal.

3.4 Complexity of the Encoding

The size of the Boolean encoding described in this section is quadratic in k , the number of bits used for an integer, for the multiplication, division, and modulo operators, and linear in k for other arithmetic operators and integer comparison operators. Where arrays or sets are involved, it's also linear in the size of the array or the size of the universe of the set, or both in the case of arrays of sets. For linear constraints, it's again quadratic in k as multiplications are involved.

In practice the size of the Booleanization is usually large compared with that of the MiniZinc or FlatZinc model (as one would expect), although not necessarily so from a SAT solver's point of view. For example, the little “perfect square” problem in Fig. 1 grows into 37 variables and 46 constraints in FlatZinc, and after Booleanization has 5441 variables and 5364 constraints in Boolean FlatZinc, or 5441 variables and 14576 clauses in CNF. However, it was solved in just 0.05 second by FZNTINI including translation and SAT solving time.

4 Weaknesses and Strengths

Needless to say, the goal of our universal Booleanization is *not* to solve CP at one fell swoop. Rather we are interested to see how far the SAT-based approach can be pushed, and on what types of problems it might suffer, or excel, so that one can be better informed when designing hybridizations of different techniques. In this light it's perhaps fitting to reflect for a moment on some possible weaknesses and strengths that might be inherent in this approach, before examining concrete experimental results.

The single most apparent weakness of such a solver is its remarkable “blindness.” All explicit domain knowledge and structure is lost when everything boils down to Booleans. This is particularly acerbated in cases where the MiniZinc models contain annotations written by the user giving hints on the nature of the variables and constraints, what techniques might suit which constraints, what variable orderings might work best, etc., which the G12/FD solver is designed to read and make use of. Information contained in these annotations is all lost (in fact, ignored) through Booleanization, and all we can rely on is the generic heuristics of whichever SAT solver we use.

The binary search used by FZNTINI to optimize an integer variable is also blind in that it cannot directly benefit from techniques that may make an informed search efficient in the original search space. One type of heuristic may still be possible though: When the SAT solver is looking for a solution during one of the steps of the binary search, it can be helpful to have the solver prefer “larger” solutions in the case of maximization and the opposite in the case of minimization, where “larger” generally means false instead of true for the sign bit of the objective integer variable, and the opposite for its other bits. It's clear that an intermediate solution closer to the goal helps cut the domain of the objective variable faster, reducing the number of steps required in the binary search. However, such a heuristic can interfere with those of the SAT solver's, and combining them to the best advantage is nontrivial. Our preliminary

experiments have not identified a way to achieve a consistent improvement; hence FZNTINI does not currently use such a heuristic.

On the other hand, universal Booleanization offers an efficient way to seamlessly combine the propagators of all constraints, through the unit propagation of a SAT solver (an analysis of the propagation power of our Booleanization of the constraints is beyond the scope of this paper). In particular, queueing of propagators becomes irrelevant as all constraints are always propagated at once and the propagation iterated to saturation. This feature is especially interesting when one considers that the types of constraints in a model can be quite varied and an advantageous integration of their respective propagators may otherwise have been nontrivial.

The second major strength of universal Booleanization, which also applies to previous work that translated specific types of constraint problems to SAT, lies in the general efficiency and scalability of modern SAT solvers. In our case a clause learning SAT solver has been used, as clause learning is currently known as the best technique for SAT problems arising from real-world applications [12]. Particularly of relevance here is the fact that clause learning is known to be more general and potentially more powerful than traditional nogood learning in constraint solvers (the basic reason is that learned clauses can involve any variables, while traditional nogoods involve decision variables only) [13]. Also, while SAT solvers cannot directly take advantage of domain knowledge and the original problem structure, their heuristics are often good at exploiting the hidden structure of the CNF formula and quickly focusing the search toward solutions or toward early detection of unsatisfiability.

5 Experimental Results

We now present an empirical evaluation of FZNTINI. To obtain a comprehensive picture, we enlist the entire set of benchmarks and examples distributed with MiniZinc [9]. This amounts to the 21 groups of problems listed in Table 1. The “perfsq” (perfect square) and “warehouses” problems in the examples group are scalable, and we have created additional, progressively larger instances and placed these two problems in their own groups of 10 instances each. Also, “2DBinPacking” contains 500 instances divided into 10 classes, and “nsp” (nurse scheduling problem) contains 400 instances divided into 4 classes; due to limited computing resources, we only use the first class of each.

Apart from “examples,” which are a mixture of various types of problems, 12 of the groups are satisfaction problems, and 8 are optimization problems. Many of these problems involve a large number of constraints and complex integer arithmetic. For example, the largest “curriculum” instance involves 66 courses with various numbers of credits to be assigned over 12 periods “in a way that the minimum and maximum academic load for each period, the minimum and maximum number of courses for each period, and the prerequisite relationships are satisfied,” and the maximum academic load for all periods is minimized (this instance was solved by FZNTINI in 10.98 seconds, and could not be solved by the other solvers in 4 hours; see below).

Table 1. Performance of FZNTINI vs. G12/FD and Gecode (4-hour time limit; times on solved instances only, in seconds except in bottom row)

Problem	No. of Inst.	FZNTINI		G12/FD		Gecode	
		Solved	Time	Solved	Time	Solved	Time
2DBinPacking-1	50	9	2843.46	7	2447.65	7	44.13
alpha	1	1	1.65	1	0.10	1	239.20
areas	4	4	0.69	4	0.71	4	0.04
bibd	9	8	16.17	8	757.72	6	197.48
cars	79	1	3.34	1	0.15	1	0.01
carseq	82	44	99403.70	2	3.58	82	66.65
curriculum	3	3	12.76	2	13.17	0	—
eq	1	1	49.92	1	0.18	1	0.00
examples	18	18	2076.74	18	1557.62	18	2.87
golfers	9	3	6278.30	4	12.88	6	1297.26
golomb	5	4	2030.23	5	323.54	5	10.35
jobshop	73	19	50294.40	2	1764.65	2	31.6
kakuro	6	6	0.17	6	1.10	6	0.01
knights	4	4	0.78	4	390.79	4	1.01
magicseq	7	4	9939.32	7	172.12	7	9.19
nsp-1-14	100	99	1800.36	1	3.97	0	—
perfsq	10	10	548.41	4	4350.19	5	2024.85
photo	1	1	0.08	1	0.20	1	0.00
queens	6	5	4168.79	6	90.68	3	0.33
trucking	10	9	14747.10	10	196.48	10	86.52
warehouses	10	10	671.71	9	2266.44	9	221.83
Total	488	263	54.14 hrs	103	3.99 hrs	178	1.18 hrs

The two solvers used for comparison are G12/FD, the FD solver developed by the G12 project [11] and distributed with MiniZinc, and a solver based on translating FlatZinc models to Gecode programs, described in [8] and available for download at the Gecode Web site [14]. Like FZNTINI, both these solvers assume that MiniZinc models have been converted to FlatZinc. We also note that G12/FD determines its search strategy based on annotations given by the user in the model files, and uses its default strategy (first-fail) in their absence. Experiments were run on a computer cluster featuring two types of CPUs, Intel Core Duo and AMD Athlon 64 X2 Dual Core Processor 4600+, both running Linux at 2.4GHz with 4GB of RAM. Each run of a solver on an instance was given a 4-hour time limit.

The overall results are shown in Table 1. For each solver and benchmark group we report the number of instances solved and the time spent on the solved instances. Time for converting MiniZinc to FlatZinc is common to all solvers, and not included (it ranges from a split second to a few seconds per instance).

It’s clear that FZNTINI solves significantly more instances than the other two solvers, 263 vs. 103 and 178 out of a total of 488, indicating the robustness and versatility of universal Booleanization. It’s also interesting to note that each solver appears to have its own “specialties”: FZNTINI was good at curriculum design (curriculum), nurse scheduling (nsp-1-14), warehouse planning (warehouses), and some mathematical puzzles (bibd, perfsq), and relatively good at job shop scheduling (jobshop); G12/FD was good at linear equations under

an all-different constraint (alpha) and the n-queens problem (queens); Gecode was good at car sequencing (carseq),³ linear equations (eq), truck scheduling (trucking), and some other mathematical puzzles (golomb, magicseq).

Overall, these results suggest that universal Booleanization offers a relatively well-rounded, competitive solution to constraint solving, and is a viable alternative where other techniques might fail. The question we are interested to explore—on what types of problems SAT might excel—now has an empirical answer in these results, but we look forward to a continued exploration, where an analytical answer may be sought in detailed analyses of the structure of the problems and their actual constraints.

Finally, it's worth mentioning that the examples group contains a purely Boolean instance, “wolf-goat-cabbage,” where the results confirm that FZNTINI does retain the advantage of SAT on problems of a Boolean nature: This instance was solved by FZNTINI in 0.02 second, G12/FD in 1553.18 seconds, and Gecode in 2.47 seconds.

6 Related Work and Conclusion

A different approach to homogeneous treatment of constraints has been recently explored, also with MiniZinc as the source language but with linear programs as the target language for translation [15]. Interestingly, this approach attempts to do almost the opposite of Booleanization: The Boolean variables are turned into integers (with a domain of $\{0, 1\}$) and Boolean constraints, along with non-linear integer constraints, all into integer linear constraints. In the experiments presented, the result of this linearization is saved in the input format of the CPLEX solver, and solved by CPLEX. Unfortunately, we have learned from the authors of [15] that their linearization program is currently unavailable due to recent changes in the language (Cadmium) in which it was written.

In conclusion, we have presented the first translation of a constraint modeling language to SAT, and using a large set of benchmarks have shown that it can outperform traditional constraint solvers. Our Booleanization uses a fixed, somewhat basic encoding largely based on a binary representation of integers. We expect our results to motivate the study of other encoding methods that are suitable for Booleanization of heterogeneous constraint models, as well as hybridizations of different techniques for constraint solving, part of which we shall undertake as our own future work.

Acknowledgements

This work has been carried out with the support of members of the G12 project at NICTA. NICTA is funded by the Australian Government's *Backing Australia's Ability* initiative, in part through the Australian Research Council.

³ Some of the solutions generated by Gecode for carseq contain all zeros, which appear to be possibly incorrect, but we have not been able to formally verify it.

References

1. 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)
2. Sinz, C.: Towards an optimal CNF encoding of Boolean cardinality constraints. In: [16], pp. 827–831
3. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation* 2, 1–26 (2006)
4. Bailleux, O., Boufkhad, Y., Roussel, O.: A translation of pseudo Boolean constraints to SAT. *Journal on Satisfiability, Boolean Modeling and Computation* 2, 191–200 (2006)
5. Silva, J.P.M., Lynce, I.: Towards robust CNF encodings of cardinality constraints. In: [17], pp. 483–497
6. Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 590–603. Springer, Heidelberg (2006)
7. Hawkins, P., Lagoon, V., Stuckey, P.J.: Solving set constraint satisfaction problems using ROBDDs. *Journal of Artificial Intelligence Research* 24, 109–156 (2005)
8. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard CP modelling language. In: [17], pp. 529–543
9. G12: MiniZinc distribution, version rotd-2008-03-03, <http://www.g12.csse.unimelb.edu.au/minizinc/>
10. Huang, J.: A case for simple SAT solvers. In: [17], pp. 839–846
11. Stuckey, P.J., de la Banda, M.J.G., Maher, M.J., Marriott, K., Slaney, J.K., Somogyi, Z., Wallace, M., Walsh, T.: The G12 project: Mapping solver independent models to efficient solutions. In: [16], pp. 13–16.
12. SAT: The Annual SAT Competitions, <http://www.satcompetition.org/>
13. Katsirelos, G., Bacchus, F.: Generalized nogoods in CSPs. In: Veloso, M.M., Kambhampati, S. (eds.) AAAI, pp. 390–396. AAAI Press / The MIT Press (2005)
14. Schulte, C., Lagerkvist, M., Tack, G.: Gecode, <http://www.gecode.org/>
15. Brand, S., Duck, G.J., Puchinger, J., Stuckey, P.J.: Flexible, rule-based constraint model linearisation. In: Hudak, P., Warren, D.S. (eds.) PADL 2008. LNCS, vol. 4902, pp. 68–83. Springer, Heidelberg (2008)
16. van Beek, P. (ed.): Principles and Practice of Constraint Programming - CP 2005. LNCS, vol. 3709. Springer, Heidelberg (2005)
17. Bessière, C. (ed.): Principles and Practice of Constraint Programming – CP 2007. LNCS, vol. 4741. Springer, Heidelberg (2007)

Flow-Based Propagators for the SEQUENCE and Related Global Constraints^{*}

Michael Maher¹, Nina Narodytska¹, Claude-Guy Quimper², and Toby Walsh¹

¹ NICTA and UNSW, Sydney, Australia

² Ecole Polytechnique de Montreal, Montreal, Canada

Abstract. We propose new filtering algorithms for the SEQUENCE constraint and some extensions of the SEQUENCE constraint based on network flows. We enforce domain consistency on the SEQUENCE constraint in $O(n^2)$ time down a branch of the search tree. This improves upon the best existing domain consistency algorithm by a factor of $O(\log n)$. The flows used in these algorithms are derived from a linear program. Some of them differ from the flows used to propagate global constraints like GCC since the domains of the variables are encoded as costs on the edges rather than capacities. Such flows are efficient for maintaining bounds consistency over large domains and may be useful for other global constraints.

1 Introduction

Graph based algorithms play a very important role in constraint programming, especially within propagators for global constraints. For example, Regin's propagator for the ALLDIFFERENT constraint is based on a perfect matching algorithm [1], whilst his propagator for the GCC constraint is based on a network flow algorithm [2]. Both these graph algorithms are derived from the bipartite value graph, in which nodes represent variables and values, and edges represent domains. For example, the GCC propagator finds a flow in such a graph in which each unit of flow represents the assignment of a particular value to a variable. In this paper, we identify a new way to build graph based propagators for global constraints: we convert the global constraint into a linear program and then convert this into a network flow. These encodings contain several novelties. For example, variables domain bounds can be encoded as costs along the edges. We apply this approach to the SEQUENCE family of constraints. Our results widen the class of global constraints which can be propagated using flow-based algorithms. We conjecture that these methods will be useful to propagate other global constraints.

2 Background

A constraint satisfaction problem (CSP) consists of a set of variables, each with a finite domain of values, and a set of constraints specifying allowed combinations of values for subsets of variables. We use capital letters for variables (e.g. X , Y and S), and lower

^{*} NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

case for values (e.g. d and d_i). A solution is an assignment of values to the variables satisfying the constraints. Constraint solvers typically explore partial assignments enforcing a local consistency property using either specialized or general purpose propagation algorithms. A *support* for a constraint C is a tuple that assigns a value to each variable from its domain which satisfies C . A *bounds support* is a tuple that assigns a value to each variable which is between the maximum and minimum in its domain which satisfies C . A constraint is *domain consistent* (DC) iff for each variable X_i , every value in the domain of X_i belongs to a support. A constraint is *bounds consistent* (BC) iff for each variable X_i , there is a bounds support for the maximum and minimum value in its domain. A CSP is DC/BC iff each constraint is DC/BC. A constraint is *monotone* iff there exists a total ordering $\prec$ of the domain values such that for any two values v, w if $v \prec w$ then v is substitutable for w in any support for C .

We also give some background on flows. A *flow network* is a weighted directed graph $G = (V, E)$ where each edge e has a capacity between non-negative integers $l(e)$ and $u(e)$, and an integer cost $w(e)$. A *feasible flow* in a flow network between a source (s) and a sink (t), (s, t) -flow, is a function $f : E \rightarrow \mathbb{Z}^+$ that satisfies two conditions: $f(e) \in [l(e), u(e)]$, $\forall e \in E$ and the *flow conservation* law that ensures that the amount of incoming flow should be equal to the amount of outgoing flow for all nodes except the source and the sink. The *value* of a (s, t) -flow is the amount of flow leaving the sink s . The *cost* of a flow f is $w(f) = \sum_{e \in E} w(e)f(e)$. A *minimum cost flow* is a feasible flow with the minimum cost. The Ford-Fulkerson algorithm can find a feasible flow in $O(\phi(f)|E|)$ time. If $w(e) \in \mathbb{Z}$, $\forall e \in E$, then a minimum cost feasible flow can be found using the successive shortest path algorithm in $O(\phi(f)SPP)$ time, where SPP is the complexity of finding a shortest path in the residual graph. Given a (s, t) -flow f in $G(V, E)$, the *residual graph* G_f is the directed graph (V, E_f) , where E_f is

$$\{e \text{ with cost } w(e) \text{ and capacity } 0..(u(e) - f(e)) \mid e = (u, v) \in E, f(e) < u(e)\} \cup \\ \{e \text{ with cost } -w(e) \text{ and capacity } 0..(f(e) - l(e)) \mid e = (u, v) \in E, l(e) < f(e)\}$$

There are other asymptotically faster but more complex algorithms for finding either feasible or minimum-cost flows [3].

In our flow-based encodings, a consistency check will correspond to finding a feasible or minimum cost flow. To enforce DC, we therefore need an algorithm that, given a minimum cost flow of cost $w(f)$ and an edge e checks if an extra unit flow can be pushed (or removed) through the edge e and the cost of the resulting flow is less than or equal to a given threshold T . We use the residual graph to construct such an algorithm. Suppose we need to check if an extra unit flow can be pushed through an edge $e = (u, v)$. Let $e' = (u, v)$ be the corresponding arc in the residual graph. If $w(e) = 0$, $\forall e \in E$, then it is sufficient to compute strongly connected components (SCC) in the residual graph. An extra unit flow can be pushed through an edge e iff both ends of the edge e' are in the same strongly connected component. If $w(e) \in \mathbb{Z}$, $\forall e \in E$, the shortest path p between v and u in the residual graph has to be computed. The minimal cost of pushing an extra unit flow through an edge e equals $w(f) + w(p) + w(e)$. If $w(f) + w(p) + w(e) > T$, then we cannot push an extra unit through e . Similarly, we can check if we can remove a unit flow through an edge.

3 The SEQUENCE Constraint

The SEQUENCE constraint was introduced by Beldiceanu and Contejean [4]. It constrains the number of values taken from a given set in any sequence of k variables. It is useful in staff rostering to specify, for example, that every employee has at least 2 days off in any 7 day period. Another application is sequencing cars along a production line (prob001 in CSPLib). It can specify, for example, that at most 1 in 3 cars along the production line has a sun-roof. The SEQUENCE constraint can be defined in terms of a conjunction of AMONG constraints. $\text{AMONG}(l, u, [X_1, \dots, X_k], v)$ holds iff $l \leq |\{i | X_i \in v\}| \leq u$. That is, between l and u of the k variables take values in v . The AMONG constraint can be encoded by channelling into 0/1 variables using $Y_i \leftrightarrow (X_i \in v)$ and $l \leq \sum_{i=1}^k Y_i \leq u$. Since the constraint graph of this encoding is Berge-acyclic, this does not hinder propagation. Consequently, we will simplify notation and consider AMONG (and SEQUENCE) on 0/1 variables and $v = \{1\}$. If $l = 0$, AMONG is an ATMOST constraint. ATMOST is *monotone* since, given a support, we also have support for any larger assignment [5]. The SEQUENCE constraint is a conjunction of overlapping AMONG constraints. More precisely, $\text{SEQUENCE}(l, u, k, [X_1, \dots, X_n], v)$ holds iff for $1 \leq i \leq n - k + 1$, $\text{AMONG}(l, u, [X_i, \dots, X_{i+k-1}], v)$ holds. A sequence like $X_i, \dots, X_{i+k-1}$ is a *window*. It is easy to see that this decomposition hinders propagation. If $l = 0$, SEQUENCE is an ATMOSTSEQ constraint. Decomposition in this case does not hinder propagation. Enforcing DC on the decomposition of an ATMOSTSEQ constraint is equivalent to enforcing DC on the ATMOSTSEQ constraint [5].

Several filtering algorithms exist for SEQUENCE and related constraints. Regin and Puget proposed a filtering algorithm for the Global Sequencing constraint (GSC) that combines a SEQUENCE and a global cardinality constraint (GCC) [6]. Beldiceanu and Carlsson suggested a greedy filtering algorithm for the CARDPATH constraint that can be used to propagate the SEQUENCE constraint, but this may hinder propagation [7]. Regin decomposed GSC into a set of variable disjoint AMONG and GCC constraints [8]. Again, this hinders propagation. Bessiere *et al.* [5] encoded SEQUENCE using a SLIDE constraint, and give a domain consistency propagator that runs in $O(nd^{k-1})$ time. van Hoeve *et al.* [9] proposed two filtering algorithms that establish domain consistency. The first is based on an encoding into a REGULAR constraint and runs in $O(n2^k)$ time, whilst the second is based on cumulative sums and runs in $O(n^3)$ time down a branch of the search tree. Finally, Brand *et al.* [10] studied a number of different encodings of the SEQUENCE constraint. Their asymptotically fastest encoding is based on separation theory and enforces domain consistency in $O(n^2 \log n)$ time down the whole branch of a search tree. One of our contributions is to improve on this bound.

4 Flow-Based Propagator for the SEQUENCE Constraint

We will convert the SEQUENCE constraint to a flow by means of a linear program (LP). We shall use $\text{SEQUENCE}(l, u, 3, [X_1, \dots, X_6], v)$ as a running example. We can formulate this constraint simply and directly as an integer linear program:

$$\begin{aligned}
l &\leq X_1 + X_2 + X_3 \leq u, \\
l &\leq X_2 + X_3 + X_4 \leq u, \\
l &\leq X_3 + X_4 + X_5 \leq u, \\
l &\leq X_4 + X_5 + X_6 \leq u
\end{aligned}$$

where $X_i \in \{0, 1\}$. By introducing surplus/slack variables, Y_i and Z_i , we convert this to a set of equalities:

$$\begin{aligned}
X_1 + X_2 + X_3 - Y_1 &= l, & X_1 + X_2 + X_3 + Z_1 &= u, \\
X_2 + X_3 + X_4 - Y_2 &= l, & X_2 + X_3 + X_4 + Z_2 &= u, \\
X_3 + X_4 + X_5 - Y_3 &= l, & X_3 + X_4 + X_5 + Z_3 &= u, \\
X_4 + X_5 + X_6 - Y_4 &= l, & X_4 + X_5 + X_6 + Z_4 &= u
\end{aligned}$$

where $Y_i, Z_i \geq 0$. In matrix form, this is:

$$\begin{pmatrix}
1 & 1 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix}
\begin{pmatrix}
X_1 \\
\vdots \\
X_6 \\
Y_1 \\
Z_1 \\
\vdots \\
Y_4 \\
Z_4
\end{pmatrix}
=
\begin{pmatrix}
l \\
u \\
l \\
u \\
l \\
u \\
l \\
u
\end{pmatrix}$$

This matrix has the *consecutive ones* property for columns: each column has a block of consecutive 1's or -1's and the remaining elements are 0's. Consequently, we can apply the method of Veinott and Wagner [11] (also described in Application 9.6 of [3]) to simplify the problem. We create a zero last row and subtract the i th row from $i + 1$ th row for $i = 1$ to $2n$. These operations do not change the set of solutions. This gives:

$$\begin{pmatrix}
1 & 1 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
-1 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\
0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1
\end{pmatrix}
\begin{pmatrix}
X_1 \\
\vdots \\
X_6 \\
Y_1 \\
Z_1 \\
\vdots \\
Y_4 \\
Z_4
\end{pmatrix}
=
\begin{pmatrix}
l \\
u-l \\
l-u \\
u-l \\
l-u \\
u-l \\
l-u \\
-u
\end{pmatrix}$$

This matrix has a single 1 and -1 in each column. Hence, it describes a network flow problem [3] on a graph $G = (V, E)$ (that is, it is a network matrix). Each row in the matrix corresponds to a node in V and each column corresponds to an edge in E . Down each column, there is a single row i equal to 1 and a single row j equal to -1 corresponding to an edge $(i, j) \in E$ in the graph. We include a source node s and a sink node t in V . Let b be the vector on the right hand side of the equation. If b_i is positive, then there is an edge $(s, i) \in E$ that carries exactly b_i amount of flow. If b_i is negative, there is an edge $(i, t) \in E$ that carries exactly $|b_i|$ amount of flow. The bounds

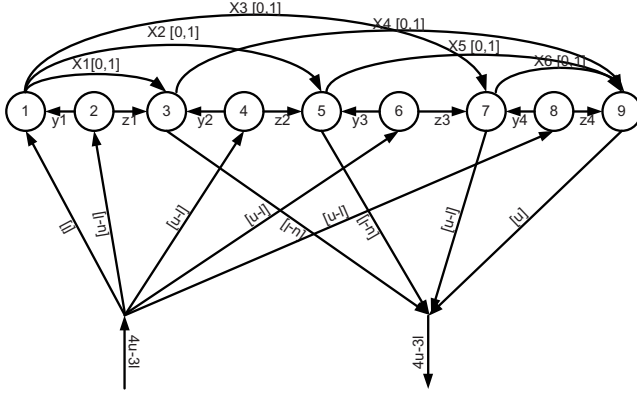


Fig. 1. A flow graph for $\text{SEQUENCE}(l, u, 3, [X_1, \dots, X_6], v)$

on the variables, which are not expressed in the matrix, are represented as bounds on the capacity of the corresponding edges.

The graph for the set of equations in the example is given in Figure 1. A flow of value $4u - 3l$ in the graph corresponds to a solution. If a feasible flow sends a unit flow through the edge labeled with X_i then $X_i = 1$ in the solution; otherwise $X_i = 0$. Each even numbered vertex $2i$ represents a window. The way the incoming flow is shared between y_j and z_j reflects how many variables X_i in the j 'th window are equal to 1. Odd numbered vertices represent transitions from one window to the next (except for the first and last vertices, which represent transitions between a window and nothing). An incoming X edge represents the variable omitted in the transition to the next window, while an outgoing X edge represents the added variable.

Theorem 1. *For any constraint $\text{SEQUENCE}(l, u, k, [X_1, \dots, X_n], v)$, there is an equivalent network flow graph $G = (V, E)$ with $5n - 4k + 5$ edges, $2n - 2k + 3 + 2$ vertices, a maximum edge capacity of u , and an amount of flow to send equal to $f = (n - k)(u - l) + u$. There is a one-to-one correspondence between solutions of the constraint and feasible flows in the network.*

The time complexity of finding a maximum flow of value f is $O(|E|f)$ using the Ford-Fulkerson algorithm [12]. Faster algorithms exist for this problem. For example, Goldberg and Rao's algorithm finds a maximum flow in $O(\min(|V|^{2/3}, |E|^{1/2})|E| \log(|V|^2/|E| + 2) \log C)$ time where C is the maximum capacity upper bound for an edge [13]. In our case, this gives $O(n^{3/2} \log n \log u)$ time complexity. We follow Régim [1,2] in the building of an incremental filtering algorithm from the network flow formulation. A feasible flow in the graph gives us a support for one value in each variable domain. Suppose $X_k = v$ is in the solution that corresponds to the feasible flow where v is either zero or one. To obtain a support for $X_k = 1 - v$, we find the SCC of the residual graph and check if both ends of the edge labeled with X_k are in the same strongly connected component. If so, $X_k = 1 - v$ has a support;

otherwise $1-v$ can be removed from the domain of X_k . Strongly connected components can be found in linear time, because the number of nodes and edges in the flow network for the SEQUENCE constraint is linear in n by Theorem 1. The total time complexity for initially enforcing DC is $O(n((n-k)(u-l)+u))$ if we use the Ford-Fulkerson algorithm or $O(n^{3/2} \log n \log u)$ if we use Goldberg and Rao's algorithm.

Still following Régim [1,2], one can make the algorithm incremental. Suppose during search X_i is fixed to value v . If the last computed flow was a support for $X_i = v$, then there is no need to recompute the flow. We simply need to recompute the SCC in the new residual graph and enforce DC in $O(n)$ time. If the last computed flow is not a support for $X_i = v$, we can find a cycle in the residual graph containing the edge associated to X_i in $O(n)$ time. By pushing a unit of flow over this cycle, we obtain a flow that is a support for $X_i = v$. Enforcing DC can be done in $O(n)$ after computing the SCC. Consequently, there is an incremental cost of $O(n)$ when a variable is fixed, and the cost of enforcing DC down a branch of the search tree is $O(n^2)$.

5 Soft SEQUENCE Constraint

Soft forms of the SEQUENCE constraint may be useful in practice. The ROADEF 2005 challenge [14], which was proposed and sponsored by Renault, puts forward a violation measure for the SEQUENCE constraint which takes into account by how much each AMONG constraint is violated. We therefore consider the soft global constraint, $\text{SOFTSEQUENCE}(l, u, k, T, [X_1, \dots, X_n], v)$. This holds iff:

$$T \geq \sum_{i=1}^{n-k+1} \max(l - \sum_{j=0}^{k-1} (X_{i+j} \in v), \sum_{j=0}^{k-1} (X_{i+j} \in v) - u, 0) \quad (1)$$

As before, we can simplify notation and consider SOFTSEQUENCE on 0/1 variables and $v = \{1\}$.

We again convert to a flow problem by means of a linear program, but this time with an objective function. Consider $\text{SOFTSEQUENCE}(l, u, 3, T, [X_1, \dots, X_6], v)$. We introduce variables, Q_i and P_i to represent the penalties that may arise from violating lower and upper bounds respectively. We can then express this SOFTSEQUENCE constraint as follows. The objective function gives a lower bound on T .

$$\begin{aligned} & \text{Minimize } \sum_{i=1}^4 (P_i + Q_i) && \text{subject to :} \\ & X_1 + X_2 + X_3 - Y_1 + Q_1 = l, && X_1 + X_2 + X_3 + Z_1 - P_1 = u, \\ & X_2 + X_3 + X_4 - Y_2 + Q_2 = l, && X_2 + X_3 + X_4 + Z_2 - P_2 = u, \\ & X_3 + X_4 + X_5 - Y_3 + Q_3 = l, && X_3 + X_4 + X_5 + Z_3 - P_3 = u, \\ & X_4 + X_5 + X_6 - Y_4 + Q_4 = l, && X_4 + X_5 + X_6 + Z_4 - P_4 = u \end{aligned}$$

where Y_i, Z_i, P_i and Q_i are non-negative. In matrix form, this is:

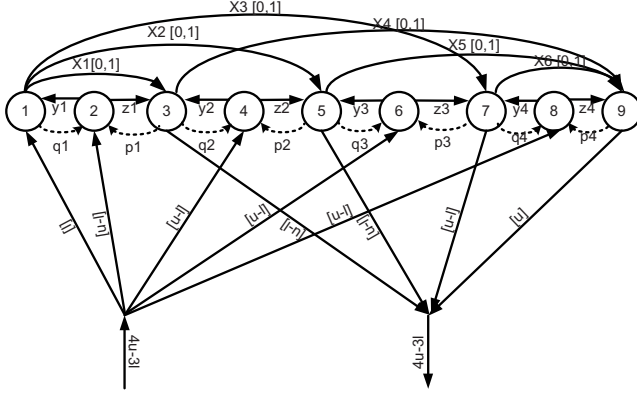


Fig. 2. A flow graph for $\text{SOFTSEQUENCE}(l, u, 3, T, [X_1, \dots, X_6])$

$X_i = 1$. Symmetrically, if the flow corresponds to an assignment with $X_i = 1$, pushing a unit of flow on (u, v) results in a solution with $X_i = 0$. If the shortest path in the residual graph between v and u is $k_{(v,u)}$, then the shortest cycle that contains (u, v) has length $k_{(u,v)} + k_{(v,u)}$. Pushing a unit of flow through this cycle results in a flow of cost $c + k_{(u,v)} + k_{(v,u)}$ which is the minimum-cost flow that contains the edge (u, v) . If $c + k_{(u,v)} + k_{(v,u)} > \max(\text{dom}(T))$, then no flows containing the edge (u, v) exist with a cost smaller or equal to $\max(\text{dom}(T))$. The variable X_i must therefore be fixed to the value taken in the current flow. Following Equation 1, the cost of the variable T must be no smaller than the cost of the solution. To enforce BC on the cost variable, we increase the lower bound of $\text{dom}(T)$ to the cost of the minimum flow in the graph G .

To enforce DC on the X variables efficiently we can use an all pairs shortest path algorithm on the residual graph [15]. This takes $O(n^2 \log n)$ time using Johnson's algorithm [12]. This gives an $O(n^2 \log n \log \log u)$ time complexity to enforce DC on SOFTSEQUENCE . The penalty variables used for SOFTSEQUENCE arise directly out of the problem description and occur naturally in the LP formulation. We could also view them as arising through the methodology of [16], where edges with costs are added to the network graph for the hard constraint to represent the softened constraint.

6 Generalized SEQUENCE Constraint

To model real world problems, we may want to have different size or positioned windows. For example, the window size in a rostering problem may depend on whether it includes a weekend or not. An extension of the SEQUENCE constraint proposed in [9] is that each AMONG constraint can have different parameters (start position, l , u , and k). More precisely, $\text{GEN-SEQUENCE}(\mathbf{p}_1, \dots, \mathbf{p}_m, [X_1, X_2, \dots, X_n], v)$ holds iff $\text{AMONG}(l_i, u_i, k_i, [X_{s_i}, \dots, X_{s_i+k_i-1}], v)$ for $1 \leq i \leq m$ where $\mathbf{p}_i = \langle l_i, u_i, k_i, s_i \rangle$. Whilst the methods in Section 4 easily extend to allow different bounds l and u for each window, dealing with different windows is more difficult. In general, the matrix now does not have the consecutive ones property. It may be possible to re-order the windows to achieve the consecutive ones property. If such a re-ordering exists, it can be

found and performed in $O(m + n + r)$ time, where r is the number of non-zero entries in the matrix [17]. Even when re-ordering cannot achieve the consecutive ones property there may, nevertheless, be an equivalent network matrix. Bixby and Cunningham [18] give a procedure to find an equivalent network matrix, when it exists, in $O(mr)$ time. Another procedure is given in [19]. In these cases, the method in Section 4 can be applied to propagate the GEN-SEQUENCE constraint in $O(n^2)$ time down the branch of a search tree.

Not all GEN-SEQUENCE constraints can be expressed as network flows. Consider the GEN-SEQUENCE constraint with $n = 5$, identical upper and lower bounds (l and u), and 4 windows: [1,5], [2,4], [3,5], and [1,3]. We can express it as an integer linear program:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & -1 & -1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & -1 & -1 & -1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & -1 & -1 & -1 \\ 1 & 1 & 1 & 0 & 0 \\ -1 & -1 & -1 & 0 & 0 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \\ X_5 \end{pmatrix} \geq \begin{pmatrix} l \\ -u \\ l \\ -u \\ l \\ -u \\ l \\ -u \end{pmatrix} \quad (2)$$

Applying the test described in Section 20.1 of [19] to Example 2, we find that the matrix of this problem is not equivalent to any network matrix.

However, all GEN-SEQUENCE constraint matrices satisfy the weaker property of total unimodularity. A matrix is *totally unimodular* iff every square non-singular sub-matrix has a determinant of $+1$ or -1 . The advantage of this property is that any totally unimodular system of inequalities with integral constants is solvable in $\mathbb{Z}$ iff it is solvable in $\mathbb{R}$.

Theorem 3. *The matrix of the inequalities associated with GEN-SEQUENCE constraint is totally unimodular.*

In practice, only integral values for the bounds l_i and u_i are used. Thus the consistency of a GEN-SEQUENCE constraint can be determined via interior point algorithms for LP in $O(n^{3.5} \log u)$ time. Using the failed literal test, we can enforce *DC* at a cost of $O(n^{5.5} \log u)$ down the branch of a search tree for any GEN-SEQUENCE constraint. This is too expensive to be practical. We can, instead, exploit the fact that the matrix for each GEN-SEQUENCE constraint has the consecutive ones property *for rows* (before the introduction of slack/surplus variables). Corresponding to the row transformation for matrices with consecutive ones for columns is a change-of-variables transformation into variable $S_j = \sum_{i=1}^j X_i$ for matrices with consecutive ones for rows. This gives the dual of a network matrix. This is the basis of an encoding of SEQUENCE in [10] (denoted there *CD*). Consequently that encoding extends to GEN-SEQUENCE. Adapting the analysis in [10] to GEN-SEQUENCE, we can enforce *DC* in $O(nm + n^2 \log n)$ time down the branch of a search tree.

In summary, for a compilation cost of $O(mr)$, we can enforce *DC* on a GEN-SEQUENCE constraint in $O(n^2)$ down the branch of a search tree, when it has a flow representation, and in $O(nm + n^2 \log n)$ when it does not.

7 A SLIDINGSUM Constraint

The SLIDINGSUM constraint [20] is a generalization of the SEQUENCE constraint from Boolean to integer variables, which we extend to allow arbitrary windows.

SLIDINGSUM $([X_1, \dots, X_n], [\mathbf{p}_1, \dots, \mathbf{p}_m])$ holds iff $l_i \leq \sum_{j=s_i}^{s_i+k_i-1} X_j \leq u_i$ holds where $\mathbf{p}_i = \langle l_i, u_i, k_i, s_i \rangle$ is, as with the generalized SEQUENCE, a window. The constraint can be expressed as a linear program $\mathcal{P}$ called the *primal* where W is a matrix encoding the inequalities. Since the constraint represents a satisfaction problem, we minimize the constant 0. The *dual* $\mathcal{D}$ is however an optimization problem.

$$\left. \begin{array}{l} \min 0 \\ \left[\begin{array}{c} W \\ -W \\ I \\ -I \end{array} \right] X \geq \left[\begin{array}{c} l \\ -u \\ a \\ -b \end{array} \right] \end{array} \right\} \mathcal{P} \quad \left. \begin{array}{l} \min [-l \ u \ -a \ b] Y \\ [W^T \ -W^T \ I \ -I] Y = 0 \\ Y \geq 0 \end{array} \right\} \mathcal{D} \quad (3)$$

Von Neumann's Strong Duality Theorem states that if the primal and the dual problems are feasible, then they have the same objective value. Moreover, if the primal is unsatisfiable, the dual is unbounded. The SLIDINGSUM constraint is thus satisfiable if the objective function of the dual problem is zero. It is unsatisfiable if it tends to negative infinity.

Note that the matrix W^T has the consecutive ones property on the columns. The dual problem can thus be converted to a network flow using the same transformation as with the SEQUENCE constraint. Consider the dual LP of our running example:

Minimize $-\sum_{i=1}^4 l_i Y_i + \sum_{i=1}^4 u_i Y_{4+i} - \sum_{i=1}^5 a_i Y_{8+i} + \sum_{i=1}^5 b_i Y_{13+i}$ subject to:

$$\left(\begin{array}{cccccccccccccccc} 1 & 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & -1 & -1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 1 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & -1 \end{array} \right) \left(\begin{array}{c} Y_1 \\ \vdots \\ Y_{18} \end{array} \right) = \left(\begin{array}{c} 0 \\ \vdots \\ 0 \end{array} \right)$$

Our usual transformation will turn this into a network flow problem:

Minimize $-\sum_{i=1}^4 l_i Y_i + \sum_{i=1}^4 u_i Y_{4+i} - \sum_{i=1}^5 a_i Y_{8+i} + \sum_{i=1}^5 b_i Y_{13+i}$ subject to:

$$\left(\begin{array}{cccccccccccccccccccc} 1 & 0 & 0 & 1 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 \\ -1 & 0 & -1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & 1 \end{array} \right) \left(\begin{array}{c} Y_1 \\ \vdots \\ Y_{18} \end{array} \right) = \left(\begin{array}{c} 0 \\ \vdots \\ 0 \end{array} \right)$$

The flow associated with this example is given in Figure 3. There are $n + 1$ nodes labelled from 1 to $n + 1$ where node i is connected to node $i + 1$ with an edge of cost $-a_i$ and node $i + 1$ is connected to node i with an edge of cost b_i . For each window $\mathbf{p}_i$, we have an edge from s_i to $s_i + k_i$ with cost $-l_i$ and an edge from $s_i + k_i$ to s_i with cost u_i . All nodes have a null supply and a null demand. A flow is therefore simply a circulation i.e., an amount of flow pushed on the cycles of the graph.

Theorem 4. *The SLIDINGSUM constraint is satisfiable if and only there are no negative cycles in the flow graph associated with the dual linear program.*

Proof. If there is a negative cycle in the graph, then we can push an infinite amount of flow resulting in a cost infinitely small. Hence the dual problem is unbounded, and the

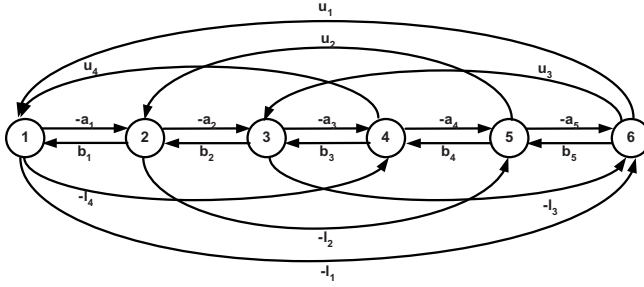


Fig. 3. Network flow associated to the SLIDINGSUM constraint posted on the running example

primal is unsatisfiable. Suppose that there are no negative cycles in the graph. Pushing any amount of flow over a cycle of positive cost results in a flow of cost greater than zero. Such a flow is not optimal since the null flow has a smaller objective value. Pushing any amount of flow over a null cycle does not change the objective value. Therefore the null flow is an optimal solution and since this solution is bounded, then the primal is satisfiable. Note that the objective value of the dual (zero) is in this case equal to the objective value of the primal. $\square$

Based on Theorem 4 we build a *BC* filtering algorithm for the SLIDINGSUM constraint. The SLIDINGSUM constraint is *BC* iff the following conditions hold:

- Value a_i is the lower bound of a variable X_i , $i = 1, \dots, n$ iff a_i is the smallest value in the domain of X_i such that there are no negative cycles through the edge weighted with $-a_i$ and labeled with the lower bound of X_i .
- Value b_i is the upper bound of a variable X_i , $i = 1, \dots, n$ iff b_i is the greatest value in the domain of X_i such that there are no negative cycles through the edge weighted with b_i and labeled with the upper bound of X_i .

The flow graph has $O(n)$ nodes and $O(n + m)$ edges. Testing whether there is a negative cycle takes $O(n^2 + nm)$ time using the Bellman-Ford algorithm. We find for each variable X_i the smallest (largest) value in its domain such that assigning this value to X_i does not create a negative cycle. We compute the shortest path between all pairs of nodes using Johnson's algorithm in $O(|V|^2 \log |V| + |V||E|)$ time which in our case gives $O(n^2 \log n + nm)$ time. Suppose that the shortest path between i and $i + 1$ has length $s(i, i + 1)$, then for the constraint to be satisfiable, we need $b_i + s(i, i + 1) \geq 0$. Since b_i is a value potentially taken by X_i , we need to have $X_i \geq -s(i, i + 1)$. We therefore assign $\min(\text{dom}(X_i)) \leftarrow \max(\min(\text{dom}(X_i)), -s(i, i + 1))$. Similarly, let the length of the shortest path between $i + 1$ and i be $s(i + 1, i)$. For the constraint to be satisfiable, we need $s(i + 1, i) - a_i \geq 0$. Since a_i is a value potentially taken by X_i , we have $X_i \leq s(i + 1, i)$. We assign $\max(X_i) \leftarrow \min(\max(X_i), s(i + 1, i))$. It is not hard to prove this is sound and complete, removing all values that cause negative cycles. Following [10], we can make the propagator incremental using the algorithm by Cotton and Maler [21] to maintain the shortest path between $|P|$ pairs of nodes in $O(|E| + |V| \log |V| + |P|)$ time upon edge reduction. Each time a lower bound a_i is

increased or an upper bound b_i is decreased, the shortest paths can be recomputed in $O(m + n \log n)$ time.

8 Experimental Results

To evaluate the performance of our filtering algorithms we carried out a series of experiments on random problems. The experimental setup is similar to that in [10]. The first set of experiments compares performance of the flow-based propagator FB on single instance of the SEQUENCE constraint against the $HPRS$ propagator¹ (the third propagator in [9]), the CS encoding of [10], and the AMONG decomposition (AD) of SEQUENCE. The second set of experiments compares the flow-based propagator FB_S for the SOFTSEQUENCE constraint and its decomposition into soft AMONG constraints. Experiments were run with ILOG 6.1 on an Intel Xeon 4 CPU, 2.0 Ghz, 4G RAM. Boost graph library version 1.34.1 was used to implement the flow-based algorithms.

8.1 The SEQUENCE Constraint

For each possible combination of $n \in \{500, 1000, 2000, 3000, 4000, 5000\}$, $k \in \{5, 15, 50\}$, $\Delta = u - l \in \{1, 5\}$, we generated twenty instances with random lower bounds in the interval $(0, k - \Delta)$. We used random value and variable ordering and a time out of 300 sec. We used the Ford-Fulkerson algorithm to find a maximum flow. Results for different values of Δ are presented in Tables 1, 2 and Figure 4. Table 1 shows results for tight problems with $\Delta = 1$ and Table 2 for easy problems with $\Delta = 5$. To investigate empirically the asymptotic growth of the different propagators, we plot average time to solve 20 instances against the instance size for each combination of parameters k and Δ in Figure 4. First of all, we notice that the CS encoding is the best on hard instances ($\Delta = 1$) and the AD decomposition is the fastest on easy instances

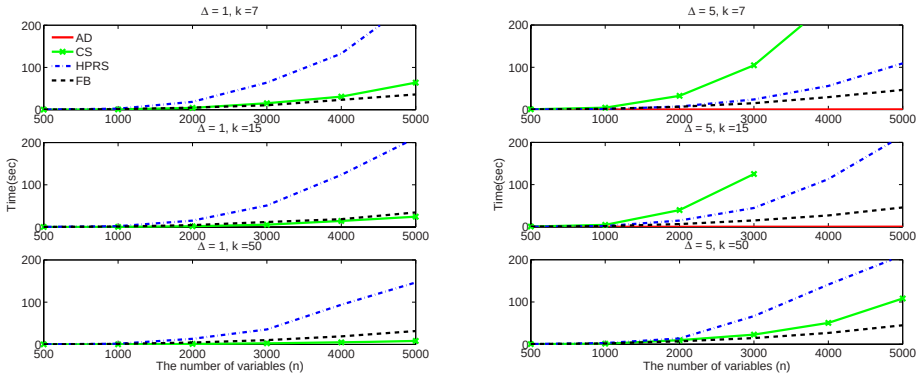


Fig. 4. Randomly generated instances with a single SEQUENCE constraints for different combinations of Δ and k

¹ We would like to thank Willem-Jan van Hoeve for providing us with the implementation of the $HPRS$ algorithm.

Table 1. Randomly generated instances with a single SEQUENCE constraint and $\Delta = 1$. Number of instances solved in 300 sec / average time to solve. We omit results for $n \in \{1000, 3000, 4000\}$ due to space limitation. The summary rows include all instances.

n	k	AD	CS	$HPRS$	FB
500	7	8 / 2.13	20 / 0.13	20 / 0.35	20 / 0.30
	15	6 / 0.01	20 / 0.09	20 / 0.30	20 / 0.29
	50	2 / 0.02	20 / 0.07	20 / 0.26	20 / 0.28
2000	7	4 / 0.04	20 / 4.25	20 / 18.52	20 / 4.76
	15	0 / 0	20 / 1.84	20 / 15.19	20 / 4.56
	50	1 / 0	20 / 1.16	20 / 13.24	20 / 4.42
5000	7	1 / 0	20 / 64.05	15 / 262.17	20 / 36.09
	15	0 / 0	20 / 24.46	17 / 211.17	20 / 34.59
	50	0 / 0	20 / 8.24	19 / 146.63	20 / 31.66
TOTALS					
solved/total		37 / 360	360 / 360	351 / 360	360 / 360
avg time for solved		0.517	9.943	60.973	11.874
avg bt for solved		17761	429	0	0

Table 2. Randomly generated instances with a single SEQUENCE constraint and $\Delta = 5$. Number of instances solved in 300 sec / average time to solve. We omit results for $n \in \{1000, 3000, 4000\}$ due to space limitation. The summary rows include all instances.

n	k	AD	CS	$HPRS$	FB
500	7	20 / 0.01	20 / 0.58	20 / 0.15	20 / 0.44
	15	20 / 0.01	20 / 0.69	20 / 0.25	20 / 0.44
	50	18 / 0.02	20 / 0.20	20 / 0.37	20 / 0.42
2000	7	20 / 0.07	20 / 32.41	20 / 7.19	20 / 6.62
	15	20 / 0.07	20 / 39.71	20 / 14.89	20 / 6.63
	50	5 / 5.19	20 / 9.52	20 / 13.71	20 / 6.94
5000	7	20 / 0.36	0 / 0	20 / 109.18	20 / 46.42
	15	20 / 0.36	6 / 160.99	17 / 215.97	20 / 45.97
	50	9 / 0.48	20 / 108.34	11 / 210.53	20 / 44.88
TOTALS					
solved/total		296 / 360	308 / 360	345 / 360	360 / 360
avg time for solved		0.236	52.708	50.698	16.200
avg bt for solved		888	1053	0	0

Table 3. Randomly generated instances with 4 soft SEQUENCES. Number of instances solved in 300 sec / average time to solve.

n	k	$\Delta = 1$		$\Delta = 5$	
		AD_S	FB_S	AD_S	FB_S
50	7	6 / 19.30	7 / 27.91	20 / 0.01	20 / 2.17
	15	8 / 36.07	13 / 20.41	11 / 49.49	10 / 30.51
	25	6 / 0.73	10 / 23.27	10 / 6.40	10 / 7.41
100	7	1 / 0	3 / 7.56	19 / 10.50	18 / 16.51
	15	0 / 0	5 / 6.90	3 / 0.01	3 / 7.20
	25	0 / 0	5 / 4.96	5 / 19.07	5 / 23.99
TOTALS					
solved/total		21 / 120	43 / 120	68 / 120	66 / 120
avg time for solved		19.463	18.034	13.286	13.051
avg bt for solved		245245	343	147434	128

($\Delta = 5$). This result was first observed in [10]. The *FB* propagator is not the fastest one but has the most robust performance. It is sensitive only to the value of n and not to other parameters, like the length of the window(k) or hardness of the problem(Δ). As can be seen from Figure 4, the *FB* propagator scales better than the other propagators with the size of the problem. It appears to grow linearly with the number of variables, while the *HPRS* propagator display quadratic growth.

8.2 The Soft SEQUENCE Constraint

We evaluated performance of the soft SEQUENCE constraint on random problems. For each possible combination of $n \in \{50, 100\}$, $k \in \{5, 15, 25\}$, $\Delta = \{1, 5\}$ and $m \in \{4\}$ (where m is the number of SEQUENCE constraints), we generated twenty random instances. All variables had domains of size 5. An instance was obtained by selecting random lower bounds in the interval $(0, k - \Delta)$. We excluded instances where $\sum_{i=1}^m l_i \geq k$ to avoid unsatisfiable instances. We used a random variable and value ordering, and a time-out of 300 sec. All SEQUENCE constraints were enforced on disjoint sets of cardinality one. Instances with $\Delta = 1$ are hard instances for SEQUENCE propagators [10], so that any *DC* propagator could solve only few instances. Instances with $\Delta = 5$ are much looser problems, but they are still hard to solve because each instance includes four overlapping SEQUENCE constraints. To relax these instances, we allow the SEQUENCE constraint to be violated with a cost that has to be less than or equal to 15% of the length of the sequence. Experimental results are presented in Table 3. As can be seen from the table, the *FB_S* algorithm is competitive with the decomposition into soft AMONG constraints on relatively easy problems and outperforms the decomposition on hard problems in terms of the number of solved problems.

We observed that the flow-based propagator for the SOFTSEQUENCE constraint (*FB_S*) is very slow. Note that the number of backtracks of *FB_S* is three order of magnitudes smaller compared to *AD_S*. We profiled the algorithm and found that it spends most of the time performing the all pairs shortest path algorithm. Unfortunately, this is difficult to compute incrementally because the residual graph can be different on every invocation of the propagator.

9 Conclusion

We have proposed new filtering algorithms for the SEQUENCE constraint and several extensions including the soft SEQUENCE and generalized SEQUENCE constraints which are based on network flows. Our propagator for the SEQUENCE constraint enforces domain consistency in $O(n^2)$ time down a branch of the search tree. This improves upon the best existing domain consistency algorithm by a factor of $O(\log n)$. We also introduced a soft version of the SEQUENCE constraint and propose an $O(n^2 \log n \log \log u)$ time domain consistency algorithm based on minimum cost network flows. These algorithms are derived from linear programs which represent a network flow. They differ from the flows used to propagate global constraints like GCC since the domains of the

variables are encoded as costs on the edges rather than capacities. Such flows are efficient for maintaining bounds consistency over large domains. Experimental results demonstrate that the *FB* filtering algorithm is more robust than existing propagators. We conjecture that similar flow based propagators derived from linear programs may be useful for other global constraints.

References

1. Régim, J.C.: A filtering algorithm for constraints of difference in csp. In: Proc. of the 12th National Conf. on AI (AAAI 1994), vol. 1, pp. 362–367 (1994)
2. Régim, J.C.: Generalized arc consistency for global cardinality constraint. In: Proc. of the 12th National Conf. on AI (AAAI 1996), pp. 209–215 (1996)
3. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms, and Applications. Prentice Hall, Englewood Cliffs (1993)
4. Beldiceanu, N., Contejean, E.: Introducing global constraints in CHIP. Mathematical and Computer Modelling 12, 97–123 (1994)
5. Bessiere, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: The slide meta-constraint. Technical report (2007)
6. Régim, J.C., Puget, J.F.: A filtering algorithm for global sequencing constraints. In: Proc. of the 3th Int. Conf. on Principles and Practice of Constraint Programming, pp. 32–46 (1997)
7. Beldiceanu, N., Carlsson, M.: Revisiting the cardinality operator and introducing cardinality-path constraint family. In: Proc. of the Int. Conf. on Logic Programming, pp. 59–73 (2001)
8. Régim, J.C.: Combination of among and cardinality constraints. In: Barták, R., Milano, M. (eds.) CPAIOR 2005. LNCS, vol. 3524, pp. 288–303. Springer, Heidelberg (2005)
9. Hoeve, W.J.v., Pesant, G., Rousseau, L.M., Sabharwal, A.: Revisiting the sequence constraint. In: Proc. of the 12th Int. Conf. on Principles and Practice of Constraint Programming, pp. 620–634 (2006)
10. Brand, S., Narodytska, N., Quimper, C.G., Stuckey, P., Walsh, T.: Encodings of the sequence constraint. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 210–224. Springer, Heidelberg (2007)
11. Veinott Jr., A.F., Wagner, H.: Optimal capacity scheduling I. Operations Research 10(4), 518–532 (1962)
12. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press, Cambridge (2001)
13. Goldberg, A.V., Rao, S.: Beyond the flow decomposition barrier. J. ACM 45, 753–782 (1998)
14. Solnon, C., Cung, V.D., Nguyen, A., Artigues, C.: The car sequencing problem: overview of state-of-the-art methods and industrial case-study of the ROADEF 2005 challenge problem. European Journal of Operational Research (EJOR) (in press, 2008)
15. Régim, 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)
16. van Hoeve, W.J., Pesant, G., Rousseau, L.M.: On global warming: Flow-based soft global constraints. J. Heuristics 12(4-5), 347–373 (2006)
17. Booth, K., Lueker, G.: Testing for the consecutive ones property, interval graphs and graph planarity using PQ-tree algorithms. Journal of Computer and Systems Sciences 13, 335–379 (1976)
18. Bixby, R., Cunningham, W.: Converting linear programs to network problems. Mathematics of Operations Research 5, 321–357 (1980)

19. Schrijver, A.: Theory of linear and integer programming. John Wiley & Sons, Inc., Chichester (1986)
20. Beldiceanu, N.: Global constraint catalog. T-2005-08, SICS Technical Report (2005)
21. Cotton, S., Maler, O.: Fast and flexible difference constraint propagation for DPLL(T). In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 170–183. Springer, Heidelberg (2006)

Guiding Search in QCSP⁺ with Back-Propagation^{*}

Guillaume Verger and Christian Bessiere

LIRMM, CNRS/University of Montpellier, France
`{verger,bessiere}@lirmm.fr`

Abstract. The Quantified Constraint Satisfaction Problem (QCSP) has been introduced to express situations in which we are not able to control the value of some of the variables (the universal ones). Despite the expressiveness of QCSP, many problems, such as two-players games or motion planning of robots, remain difficult to express. Two more modeler-friendly frameworks have been proposed to handle this difficulty, the Strategic CSP and the QCSP⁺. We define what we name back-propagation on QCSP⁺. We show how back-propagation can be used to define a goal-driven value ordering heuristic and we present experimental results on board games.

1 Introduction

The Constraint Satisfaction Problem (CSP) consists in finding values for variables such that a set of constraints involving these variables is satisfied. It is a decision problem, in which all variables are existentially quantified (*i.e.*, Is there a value for each variable such that all constraints are satisfied?). This framework is useful to express and solve many real applications.

Problems in which there is a part of uncertainty are hard to model in the CSP formalism, and/or require an exponential number of variables. The uncertainty part may come from weather or any external event, which is out of our control. The Quantified Constraint Satisfaction Problem (QCSP [4]) is a generalisation of the CSP in which variables can be either existentially (as in CSP) or universally quantified. We control the existential variables (we choose their value), but we have no control on universal variables (they can take any value in their domain). Solving such problems is finding values for existential variables according to the values taken by the preceding universal variables in the sequence of variables in order to respect constraints.

The structure of QCSP is such that the domains of universal variables do not depend on values of previous variables. But in many problems, values taken by variables depend on what has been done before. For example, if we want to express a board game like chess, some moves are forbidden, like stacking two pieces on the same cell.

In [2,3] this issue has been identified and new formalisms, SCSP and QCSP⁺, have been proposed to have symmetrical quantifier behaviors. Both in SCSP

^{*} Supported by the ANR project ANR-06-BLAN-0383-02.

and QCSP⁺, the meaning of the universal quantifier has been modified, and the domains of universal variables depend on the values of previous variables. In the field of Quantified Boolean Formulas, this problem has also been identified. Ansótegui *et al.* introduced new QBF formulations and solving strategies for adversarial scenarios [1].

In this paper, we propose a value ordering heuristic for the QCSP⁺. After preliminary definitions (Section 2) and clues for solving QCSP⁺ (Section 3), we analyse constraint propagation in QCSP⁺ in Section 4. Based on this analysis, we derive a value ordering heuristic for QCSP⁺ in Section 5. Finally, in Section 6, we present experimental results on board-games.

2 Definitions

In this section we focus on the two frameworks that have been proposed to tackle the QCSP modeling issue, Strategic CSPs [3] and QCSP⁺ [2]. But first of all, we give some background on CSP and QCSP.

2.1 CSP and QCSP

The constraint satisfaction problem. A *constraint network* $N = (X, D, C)$ consists of a finite set of variables $X = \{x_1, \dots, x_n\}$, a set of domains $D = \{D(x_1), \dots, D(x_n)\}$, where the domain $D(x_i)$ is the finite set of values that variable x_i can take, and a set of constraints $C = \{c_1, \dots, c_e\}$. Each constraint c_k is defined by the ordered set $var(c_k)$ of the variables it involves, and by the set $sol(c_k)$ of combinations of values on $var(c_k)$ satisfying it. A *solution* to a constraint network is an assignment of a value from its domain to each variable such that every constraint in the network is satisfied. A value v_i for a variable x_i is *consistent with* a constraint c_j involving x_i iff there exists an assignment I of all the variables in $var(c_j)$ with values from their domain such that x_i is assigned v_i and I satisfies c_j .

Given a constraint network $N = (X, D, C)$, the constraint satisfaction problem (CSP) is the problem of deciding whether there exists an assignment in D for the variables in X such that all constraints in C are satisfied. In a logical formulation, we write, “ $\exists x_1 \dots \exists x_n, C?$ ”.

In CSPs, the backtrack algorithm is inefficient when problems are big, and the most common way to solve CSPs is to combine depth-first search and constraint propagation. The aim is to use constraint propagation to reduce the size of the search tree by removing some inconsistent values in domains of variables. An inconsistent value is a value such that if it is assigned to its variable, the CSP is unsatisfiable. That is, removing inconsistent values does not change the set of solutions.

Most CSP solvers use Arc Consistency (AC) as the best compromise between tree pruning and time consumption. A constraint c_j is *arc consistent* iff for any $x_i \in var(c_j)$, for any $v_i \in D(x_i)$, v_i is consistent with c_j . To propagate AC during the backtrack, after each instantiation, we remove inconsistent values in domains of not yet instantiated variables x_j until all constraints are arc consistent.

The quantified constraint satisfaction problem. The quantified extension of the CSP [4] allows some of the variables to be universally quantified. A *quantified* constraint network consists of variables $X = \{x_1, \dots, x_n\}$, a set of domains $D = \{D(x_1), \dots, D(x_n)\}$, a quantifier sequence $\Phi = (\phi_1 x_1, \dots, \phi_n x_n)$, where $\phi_i \in \{\exists, \forall\}$, $\forall i \in 1..n$, and a set of constraints C . Given a quantified constraint network, the Quantified CSP (QCSP) is the question “ $\phi_1 x_1 \dots \phi_n x_n, C?$ ”.

Example 1. $\exists x_1 \forall y_1 \exists x_2, (x_1 \neq y_1) \wedge (x_2 < y_1)$ with $x_1, y_1, x_2 \in \{1, 2, 3\}$. This can be read as: is there a value for x_1 such that whatever the value chosen for y_1 , there will be a value for x_2 consistent with the constraints?

As in CSP, the backtrack search in QCSP is combined with constraint filtering. Propagation techniques are heavily depending on the quantifiers of variables. In Example 1 the QCSP is unsatisfiable because for each value of x_1 , there is a value of y_1 violating $x_1 \neq y_1$. As y_1 is a universal variable and x_1 is an existential variable earlier in the sequence, any value in the domain of x_1 that is not compatible with a value in the domain of y_1 is not part of a solution. Constraint propagation in QCSP has been studied in [4,8,7].

2.2 Restricted Quantification

One of the advantages that CSPs have on SAT problems (satisfaction of Boolean clauses) is that a CSP model is often close to the intuitive model of a problem, whereas a SAT instance is most of the time an automatic translation of a model to a clausal form, and is not human-readable. QCSP and QBF can be compared as CSP and SAT. To model a problem with a QBF, one needs to translate a model into a formula, and the QBF is not human-readable. QCSP, like CSP, should have the advantage of readability. But modeling a problem, even a simple one, with a QCSP, is a complex task. The prenex form of formulas is counter-intuitive. It would be more natural to have symmetrical behaviors for existential and universal variables. We describe here two frameworks that are more modeler-friendly: Strategic Constraint Satisfaction Problem (SCSP), and QCSP⁺.

The strategic CSP. In SCSP [3], the meaning of the universal quantifier is different from the universal quantifier in QCSP. It is noted $\forall^\circ$. Allowed values for universal variables are values consistent with previous assignments.

Let us change the universal quantifier of Example 1 into the universal quantifier of SCSPs. The problem is now $\exists x_1 \forall^\circ y_1 \exists x_2, (x_1 \neq y_1) \wedge (x_2 < y_1)$ with $x_1, y_1, x_2 \in \{1, 2, 3\}$. If x_1 takes the value 1, the domain of y_1 is reduced to $\{2, 3\}$ because of the constraint $(x_1 \neq y_1)$ that prevents y_1 from taking the same value as x_1 . This SCSP has a solution. If x_1 takes the value 1, y_1 can take either 2 or 3, and x_2 can always take the value 1 that satisfies the constraint $(x_2 < y_1)$.

Solving a SCSP is quite similar to solving a standard QCSP. The difference is that domains of universal variables are not static, they depend on variables already assigned in the left part of the sequence (before the universal variable we are ready to instantiate).

The quantified CSP⁺. QCSP⁺ [2] is based on the same idea as SCSP, which is to modify the meaning of the universal quantifier in order to make it more intuitive than in QCSP. It uses the notion of *restricted quantification*, which is more natural for the human mind than unrestricted quantification used in QCSP. Restricted quantification adds a “such that” right after the quantified variable. A QCSP⁺ can be written as follows:

$$P = \exists X_1[R_{X_1}], \forall Y_1[R_{Y_1}] \exists X_2[R_{X_2}] \dots \exists X_n[R_{X_n}], G$$

All constraints noted R_X are called *rules*. They are the restrictions of the quantifiers. Each existential (*resp.* universal) *scope* X_i (*resp.* Y_i) is a set of variables having the same quantifier. The order of variables inside a scope is not important, but two scopes cannot be swapped without changing the problem. The constraint G is called *goal*, it has to be satisfied when all variables are instantiated. The whole problem can be read as “Is there an instantiation of variables in X_1 such that the assignment respects the rule R_{X_1} and that for all tuples of values taken by the set of variables Y_1 respecting R_{Y_1} , there will be an assignment for variables in $X_2 \dots$ such that the goal G is reached?”.

A QCSP⁺ can be expressed as a QCSP. The difference between QCSP and QCSP⁺ is the prenex form of QCSP. The QCSP⁺ P is defined by the formula $\exists X_1(R_{X_1} \wedge (\forall Y_1(R_{Y_1} \rightarrow \exists X_2(R_{X_2} \wedge (\dots \exists X_n(R_{X_n} \wedge G))))))$. The prenex form of P is $\exists X_1 \forall Y_1 \exists X_2 \dots \exists X_n, (R_{X_1} \wedge (\neg R_{Y_1} \vee (R_{X_2} \wedge (\dots (R_{X_n} \wedge G))))$. We see that, in this formula, we lose all the structure of the problem because all information is merged in a big constraint. Furthermore, disjunctions of constraints do not propagate well in CSP solvers. Finally the poor readability of this formula makes it hard to deal with for a human user.

The example of SCSP derived from Example 1 (see above) is modelled as a QCSP⁺ as follows: $\exists x_1 \forall y_1 [y_1 \neq x_1] \exists x_2 [x_2 < y_1], \top$ with $x_1, y_1, x_2 \in \{1, 2, 3\}$, where $\top$ is the universal constraint.

The difference between SCSP and QCSP⁺ is mainly the place where constraints are put. A constraint of a SCSP containing a set X of variables should be placed in the rules of the rightmost variable of X in a QCSP⁺ modeling the same problem.

In the rest of the paper, we will only consider the QCSP⁺, but minor modifications of our contributions should be enough to adapt them to SCSPs.

3 Solving a QCSP⁺

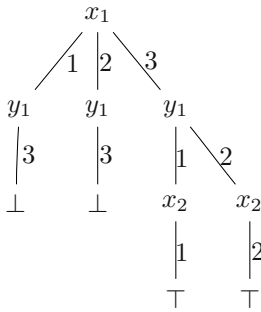
In this section, we focus on solving QCSP⁺. First, we show how we use a backtracking algorithm with universal and existential variables. Then we focus on constraint propagation in QCSP⁺.

As with classical CSPs or QCSPs, one can solve a QCSP⁺ by backtracking. For each variable, we choose a value consistent with the rules attached to the variable, and we go deeper in the search tree. At the very bottom of the tree, we need to check that the assignments are consistent with the goal.

For existential variables, we do as for classical CSPs. If it is possible to assign the current existential variable according to the rules, then we can go deeper in the tree. But if there is no value consistent with the rules, it means that the current branch failed. Then we jump back to the last existential variable we instantiated and make another choice. If there is no previous existential variable, the QCSP⁺ has no solution. Another case for which the branch can fail is when we instantiated all variables, but the assignments are not consistent with the goal.

On the other hand, if we instantiated all variables such that it is consistent with the goal, we just found a winning branch of the QCSP⁺. In this case we jump back to the last universal variable instantiated, we assign another value of its domain consistent with its rules and we try to find another winning branch. When all values of the universal variable have been checked and lead to winning branches, we can go back to the previous universal variable. When there is no previous universal variable, the QCSP⁺ has a solution. If at any moment, the current universal variable we want to instantiate has an empty domain, it is a winning branch. For example if it is a game, it means that the adversary cannot play because we blocked him, or because we just won before his move.

Example 2. Let $P = \exists x_1 \forall y_1 [y_1 \neq x_1] \exists x_2, x_2 = y_1, D(x_1) = D(y_1) = \{1, 2, 3\}, D(x_2) = \{1, 2\}$ be a QCSP⁺. In this QCSP⁺, x_1 can play any of the moves 1, 2, or 3. Then y_1 can play a move different from the move of x_1 , and x_2 can play 1 or 2. At the end, in order to win, x_2 and y_1 must have the same value.



If x_1 plays 1 or 2, y_1 will be able to play 3 and will win because x_2 cannot play 3 (this value is not in its domain). The only way for the $\exists$ -player to win is to play 3 at the first move to forbid y_1 to play 3. Then whatever the value taken by y_1 (1 or 2), x_2 will be able to play the same value. The QCSP⁺ has a solution.

Note that in Example 2, the goal could have been a rule for x_2 . When the last variable is an existential one, its rules and the goal have the same meaning.

Propagating the constraints in QCSP⁺

We try to identify how it is possible to use constraint propagation to reduce the domains of variables.

Example 3. Let $P = \exists x_1 \forall y_1 [y_1 < x_1], \perp. D(x_1) = D(y_1) = \{1, 2, 3\}$. x_1 has to choose a value, then y_1 has to take a lower value.

In Example 3, a standard CSP-like propagation of $y_1 < x_1$ would remove the value 1 for x_1 and the value 3 for y_1 . Like in CSPs, the value 3 in the domain of y_1 is inconsistent because whatever the value taken by x_1 , y_1 will never be able to take this value.¹ But contrary to the CSP-like propagation, the QCSP⁺

¹ Note the difference with the QCSP case for which a removal of value in a universal domain means a fail of the whole problem.

propagation should not remove the value 1 for x_1 : if x_1 takes the value 1 he will win because y_1 will have no possible move.

Let x_i be a variable in a QCSP⁺, with a rule R_{x_i} . If we only propagate the constraint R_{x_i} on the domain of x_i , we remove from its domain all values inconsistent with what is happening before. This propagation is allowed, because values removed this way cannot appear in the current search tree (when the solver will have to instantiate x_i , the allowed values are values consistent with R_{x_i}).

Let x_j be another variable in a scope before the scope of x_i . Now suppose that the rule R_{x_i} involves x_j too (*i.e.*, the values that x_i can take depend on the values taken by x_j). If the CSP-like propagation of the constraint R_{x_i} removes some values in the domain of x_j , it does not mean that x_j cannot take the values removed, but that, if x_j takes one of these values, then when it will be x_i 's turn to play, he will not be able to take any value. Hence a QCSP⁺ propagation should not remove these values.

Briefly speaking, it is allowed to propagate constraints from the left of the sequence to the right, but not to propagate from the right to the left. Benedetti *et al.* proposed the *cascade propagation*, a propagation following this principle.

Cascade propagation. In [2], cascade propagation is proposed as a propagation mechanism. The idea is that propagating a rule can modify the domains of variables of its scope, but not the domains of variables of previous scopes.

In [2], cascade propagation is implemented by creating a sequence of sub-problems. Each sub-problem P_i represents the restriction of problem P to its scopes from the first to the i th. That is, each P_i contains all rules belonging to scopes $1 \dots i$. If there are n scopes in P , P_n will be the problem excluding the goal, and $P_{n+1} = P$. In a sub-problem, propagation is used as in a classical CSP. Each P_i can be considered as representing the fact that we will be able to instantiate variables from the first one of the first scope to the last one of scope i according to the rules. Let P be the problem described in Example 2. Cascade propagation creates 4 sub-problems:

$$\begin{aligned} P_1 &= \exists x_1, \top \\ P_2 &= \exists x_1 \forall y_1 [y_1 \neq x_1], \top \\ P_3 &= \exists x_1 \forall y_1 [y_1 \neq x_1] \exists x_2, \top \\ P_4 &= \exists x_1 \forall y_1 [y_1 \neq x_1] \exists x_2, x_2 = y_1 \end{aligned}$$

For each sub-problem P_1 to P_3 , the goal is $\top$ because we say a sub-problem has a solution if we can instantiate all its variables, without thinking of the goal of P . Propagation in each sub-problem can be done independently, but to speed up the process, if a value is removed from the domain of a variable, it can safely be removed from the deeper sub-problems. Furthermore if the domain of any variable in a sub-problem P_k becomes empty with propagation, it means that it is impossible to do the k^{th} move. So, from there it is no longer necessary to check the problems $\{P_{k+1}, \dots, P_{n+1}\}$ since they are inconsistent too. If the scope k is universal and if P_{k-1} can be completely instantiated, then the current branch is a winning branch. If the scope k is an existential one and if P_{k-1} can be completely instantiated, the current branch is a losing branch.

4 Back-Propagation

In this section we present what we name *back-propagation*, a kind of constraint propagation which uses the information from the right part of the sequence. We also show that this propagation may not work properly in general.

4.1 Removing Values in Domains

The aim of constraint propagation in CSPs is to remove every value for which we know that, if we assign this value to the variable, it leads to a fail. In QCSP⁺, the aim of constraint propagation is to remove values in domains too. But values we can remove without loss of solution depend on the quantifiers. In the case of existential variables, the values we can remove are values that do not lead to a solution (as we do in CSPs). Intuitively, it means that the $\exists$ -player will not play this value because he knows that he will lose with this move. If all values are removed, it is impossible to win at this point. In the case of universal variables, the values we can remove are values that lead to a solution. Intuitively, it means that the $\forall$ -player will not play this value because he knows that it means a loss for him. If all values are removed, it means that the $\forall$ -player cannot win, so it is considered as a win for the $\exists$ -player.

4.2 An Illustrative Example

We will see how to propagate information from right to left. Back-propagation adds some redundant constraints inside the rules of variables. These constraints will help to prune domains.

Consider the Example 2 from Section 3. Let us propagate the goal $(x_2 = y_1)$. It removes the value 3 in the domain of y_1 . It means that if $y_1 = 3$, we cannot win. So x_1 has to prevent y_1 from taking the value 3. If y_1 is able to take this value, the $\exists$ -player will lose. The way to prevent this is to make sure that the rule belonging to y_1 , $y_1 \neq x_1$, will force the $\forall$ -player not to take the value 3. In other terms, the rule must be inconsistent for $y_1 = 3$. We can express it as $(\neg(y_1 \neq x_1) \wedge y_1 = 3)$, or $\neg(3 \neq x_1)$, that is $(x_1 = 3)$. This constraint can be posted as a rule for x_1 . The problem is now $P = \exists x_1[x_1 = 3]\forall y_1[y_1 \neq x_1]\exists x_2, x_2 = y_1, D(x_1) = D(y_1) = \{1, 2, 3\}, D(x_2) = \{1, 2\}$. The new rule we just added removes the inconsistent values 1 and 2 for x_1 .

4.3 General Behavior

First we know that if v is inconsistent with the rules of the scope of x , we can remove it from the domain of x . Then, we can look ahead in the sequence of variables. Consider a QCSP⁺ containing the sequence with $\phi = \forall$ and $\bar{\phi} = \exists$ or $\phi = \exists$ and $\bar{\phi} = \forall$: $\phi x_i[R_{x_i}], \bar{\phi} y_j[C_j(x_i, y_j)], \phi x_k[C_k(x_k, y_j)]$. Suppose that propagating $C_k(x_k, y_j)$ removes the value v_j in the domain of y_j . As we said before, it means that if we assign v_j to y_j , x_k will not be able to play. Then the ϕ -player will have to forbid the $\bar{\phi}$ -player to play v_j . If he is not able to forbid it,

then he will lose. He can reduce the domain of y_j with the rules belonging to y_j and involving x_i .

To ensure that y_j will not forbid any move for x_k , we have to make sure that the constraint $C_j(x_i, y_j)$ will remove the value v_j . It is equivalent to forcing $\neg C_j(x_i, y_j)$ to let the value v_j in the domain of y_j , or equivalently forcing $(\neg C_j(x_i, y_j) \wedge y_j = v_j)$. Hence, we can add to R_{x_i} the constraint $\neg C_j(x_i, v_j)$,² so that the ϕ -player is assured to have chosen a move that prevents the ϕ -player from doing a winning move.

4.4 Back-Propagation Does Not Work in General

In the case where there are variables between x_i and x_k , our previous treatment is not correct: imagine that the game always ends before x_k 's turn and we are unable to detect it with constraint propagation, we should not take into account the constraints on x_k . For example, consider the following problem:

Example 4. $P = \exists x_1 \forall z_1 z_2 [\neq (x_1, z_1, z_2)] \exists t_2 \forall y_1 [y_1 \neq x_1] \exists x_2, x_2 = y_1$. $D(x_1) = D(t_2) = D(y_1) = \{1, 2, 3\}$, $D(z_1) = D(z_2) = \{0, 1, 2\}$, $D(x_2) = \{1, 2\}$. The constraint $\neq (x_1, z_1, z_2)$ is a clique of binary inequalities between the different variables. The variable t_2 is here only to separate the two scopes of universal variables. This problem is the same as the problem in Example 2 in which we added the variables z_i and the variable t_2 .

From Example 4 we can add the same constraint $x_1 = 3$ as we did for Example 2 in Section 4.2. Doing this, we forbid x_1 to take either the value 1 or the value 2. But if x_1 would take any of these two values, we would win since it is not possible to assign values for the different z_i .

In the general case the back-propagation may remove values that are consistent, so it cannot be used as a proper propagation for QCSP⁺. But from the back-propagation, we can make a value ordering heuristic that will guide search towards a win, or at least prevent the adversary to win.

5 Goal-Driven Heuristic

In this section we present our value ordering heuristic for QCSP⁺. The behavior of the heuristic is based on the same idea as back-propagation. The difference is that, as it is a heuristic, it does not remove values in domains, but it orders them from the best to the worst in order to explore as few nodes as possible.

In the first part of the section, we discuss value ordering heuristics on QCSP⁺, and the difference with standard CSP. Afterwards, we present our contribution, a goal-driven value ordering heuristic based on back-propagation.

5.1 Value Heuristics

In CSPs, a value ordering heuristic is a function that helps the solver to go towards a solution. When the solver has to make a choice between the different

² The constraint C in which we replaced the occurrences of y_j with the value v_j .

values of a variable, the heuristic gives the value that seems the best for solving the problem. The best value is a value that leads to a solution. If we are able to find a perfect heuristic that always returns a value leading to a solution, it is possible to solve a CSP without backtracking. But, when there is no solution or when we want all the solutions of a CSP, the heuristic, even perfect, does not prevent from backtracking. In the case of QCSP⁺, value ordering heuristics can be defined too. But the search will not be backtrack-free, even with a good heuristic, because of the universal quantifiers.

In QCSP⁺, a *good* value for an existential variable (like for CSP) is a value that leads to a solution (*i.e.*, the $\exists$ -player wins). A *good* value for a universal variable is a value that leads to a fail (*i.e.*, we quickly prove that the $\forall$ -player wins). If the QCSP⁺ is satisfiable, the heuristic helps to choose values for existential variables, and if it is unsatisfiable, the heuristic helps to choose values for universal variables.

In the rest of the section, we describe the value ordering heuristic we propose for QCSP⁺.

5.2 The Aim of the Goal-Driven Heuristic

Our aim, with the proposal of our heuristic, is to explore the search tree looking ahead to win as fast as possible, to avoid traps from the adversary, and of course not to trap ourselves. For example, in a chess game, if you are able to put your opponent into checkmate this turn, you do not ask yourself if another move would make you win in five moves. Or if your adversary is about to put you into checkmate next turn unless you move your king, you will not move your knight!

In terms of QCSP⁺ checking that a move is considered as good or bad is a question of constraint satisfaction. We will use the same mechanisms as back-propagation, *i.e.*, checking classical arc consistency of rules.

Let see how to choose *good* values on different examples. In each of these example, the aim is to detect what would be a good value to try first for the first variable. In these examples, ϕ and $\bar{\phi}$ will be the quantifiers $\exists$ and $\forall$ or $\forall$ and $\exists$.

Self-preservation. In Example 5, a rule from a scope with the same quantifier removes some values in the domain of the current variable. The ϕ -player tries not to block himself.

Example 5. $P = \phi x_1 \dots \phi x_2 [x_2 < x_1] \dots$, $D(x_1) = D(x_2) = \{1, 2, 3\}$. AC on the rule of x_2 removes the value 1 in the domain of x_1 . As the ϕ -player wants to be able to play again, it could be a better choice to try the values $x_1 = 2$ and $x_1 = 3$ at first. If $x_1 = 1$ is played, the player knows that he will not be able to play for x_2 .

Blocking the adversary. In Example 6, a rule from a scope with the opposite quantifier removes some values in the domain of the current variable. The ϕ -player tries to block the $\bar{\phi}$ -player.

Example 6. $P = \phi x_1 \dots \bar{\phi} y_1 [y_1 < x_1] \dots, D(x_1) = D(y_1) = \{1, 2, 3\}$. AC on the rule of y_1 removes the value 1 in the domain of x_1 . As the ϕ -player wants to prevent the $\bar{\phi}$ -player from playing, it could be a better choice to try the value $x_1 = 1$ first. If $x_1 = 2$ or $x_1 = 3$ is played, the $\bar{\phi}$ -player could be able to keep playing.

If two rules are in contradiction, the heuristic will take into account the leftmost rule, because it is the rule which is the more likely to happen. We will implement this in our algorithm by checking the rules from left to right.

Annoying the adversary. Now imagine the other player finds a good value for his next turn with the same heuristic. Your aim is to prevent him from playing well, so the above process can be iterated.

This point is illustrated with the problem from Example 2:

$P = \exists x_1 \forall y_1 [y_1 \neq x_1] \exists x_2, x_2 = y_1, D(x_1) = D(y_1) = \{1, 2, 3\}, D(x_2) = \{1, 2\}$. The heuristic for finding values for y_1 detects that the value 3 is a good value (x_2 will not be able to win). x_1 is aware of that, and will try to avoid this case. His new problem can be expressed as $P' = \exists x_1 \forall y_1 [y_1 \neq x_1], \perp$ with $D(x_1) = \{1, 2, 3\}, D(y_1) = \{3\}$. (If the $\exists$ -player lets y_1 take the value 3, he thinks he will lose. There may be a value for x_1 such that the $\exists$ -player will prevent the $\forall$ -player from making him lose). The heuristic for finding values for x_1 in P' detects that x_1 should choose to play 3 first to prevent y_1 from playing well.

In the next section, we will discuss on the algorithm for choosing the best values for variables.

5.3 The Algorithm

In this section, we describe the algorithm **GDHeuristic** (Goal-Driven Heuristic) used to determine what values are good choices for the current variable. The algorithm takes as input, the current variable and the rightmost scope that we consider. Note that we consider the goal as a scope here. It returns a set of values which are considered better to try first as defined in the above part.

Note that the aim of an efficient heuristic is to make the exploration as short as possible. If we try to instantiate a variable of the $\exists$ -player, this means finding a value that leads to a winning branch. If we try to instantiate a variable of the $\forall$ -player, this means finding a value that proves the $\forall$ -player can win, that is, a losing branch. We see that in both cases, the best value to choose is a value that leads the current player to a win. Thus, in spite of the apparent asymmetry of the process, we can use the same heuristic for both players.

Algorithm 1 implements the goal driven value ordering heuristic. It is called when the solver is about to assign a value to the current variable. The aim is to give the solver the best value to assign to the variable. In fact, it is not more time consuming to return a set of equivalent values than a single value, so we return a set of values for which we cannot decide the best between them.

In this algorithm, we use different functions we explain here:

saveContext() saves the current state (domains of variables)

restoreContext() restores to the last state.

AC(P_i) runs the arc consistency algorithm on the problem P_i

quant(var) returns the quantifier of var ($\exists$ or $\forall$)

scope(var) returns the scope containing var

dom(var) returns the current domain of var

initDom(var) returns the domain of var before the first call to **GDHeuristic()**.

We now describe the algorithm's behavior. The first call to it is done with **GDHeuristic(current variable, goal)**. We try to find the best value for the current variable, for the whole problem (the rightmost scope to consider is the goal). Note that we could bound the depth of analysis by specifying another scope as the last one.

First of all, we save the context (line 1) because we do not want our heuristic to change the domains. The context will be restored each time we return a set of values (lines 5, 14, 17 and 19).

For each future sub-problem (*i.e.*, containing the variables at the right of the current variable), we will try to bring back information in order to select the best values for the current variable. This is the purpose of the loop (line 2). If no information can be used, it will return the whole domain of the current variable (line 20) since all its values seem equivalent.

For each sub-problem $P_{\#Scope}$ containing all variables from scope 1 to scope $\#Scope$, we enforce AC (line 3). If the domain of any variable in $P_{\#Scope-1}$ is reduced, we will decide the aim of our move: *self-preservation* (line 6), *blocking the adversary* (line 7) or *annoying the adversary* (line 13). The heuristic performs at most q^2 calls to AC, where q is the number of scopes. It appears when all recursive calls (line 13) are done with **scope(Var) = LastScope - 1**.

The rest of the main loop is made of two main parts. The first part, from line 4 to line 8, describes the case where the domain of the current variable is modified by the scope $\#Scope$. (Note that we know it is not modified due to an earlier scope because we have not exited from the main loop –line 2– at a previous turn.) If the current variable has the same quantifier as the scope $\#Scope$, the heuristic returns values consistent with the scope $\#Scope$ (*self-preservation* line 6). If the quantifiers are different, the heuristic returns values that block any move for the scope $\#Scope$, *i.e.*, values inconsistent with the rules of scope $\#Scope$ (*blocking the adversary* line 7). The second part of the main loop, from line 9 to line 18, represents the case where a variable (Var), between the current variable and the scope $\#Scope$ has a domain reduced (line 9). If Var and the scope $\#Scope$ have different quantifiers (*annoying the adversary* line 11), Var will try to block any move for the scope $\#Scope$ by playing one of the values not in its reduced domain (line 12). We then try to find a good value for the current variable according to this new information that we have for Var (line 13). If Var and the scope $\#Scope$ have the same quantifier (*self-preservation* line 16), Var will try to play values in its reduced domain, that is, those that do not block its future move at scope $\#Scope$. But we know that arc consistency has not removed any value in

Algorithm 1. *GDHeuristic*

```

input: CurrentVar, LastScope
Result: set of values
begin
1  saveContext()
2  for  $\#_{Scope} \leftarrow \text{scope}(CurrentVar) + 1$  to LastScope do
3      AC( $P_{\#_{Scope}}$ )
4      if  $\text{dom}(CurrentVar) \neq \text{initDom}(CurrentVar)$  then
5           $ReducedDomain \leftarrow \text{dom}(CurrentVar)$ 
6          restoreContext()
7          if  $\text{quant}(CurrentVar) = \text{quant}(\#_{Scope})$  then
8              return ReducedDomain
9          else
10             return  $\text{initDom}(CurrentVar) \setminus ReducedDomain$ 
11         else
12             if any domain has been reduced before scope  $\#_{Scope}$  then
13                 Var  $\leftarrow$  leftmost variable with reduced domain in  $P_{\#_{Scope}-1}$ 
14                 if  $\text{quant}(Var) \neq \text{quant}(\#_{Scope})$  then
15                      $\text{dom}(Var) \leftarrow \text{initDom}(Var) \setminus \text{dom}(Var)$ 
16                      $values \leftarrow GDHeuristic(CurrentVar, \text{scope}(Var))$ 
17                     restoreContext()
18                     return values
19                 else
20                     restoreContext()
21                     return  $\text{initDom}(CurrentVar)$ 
22             restoreContext()
23         return  $\text{initDom}(CurrentVar)$ 
24 end

```

the domain of the current variable. This means that whatever the value selected by the current variable, *Var* will be able to play in its reduced domain (*i.e.*, good values). As a result, the heuristic cannot discriminate in the initial domain of the current var (line 18).

If no domain is modified (except the domains in scope $\#_{Scope}$), we will have to check the next sub-problem (returning to the beginning of the loop).

Note that at lines 6 and 16, we immediately return a set of values for the current variable. We could imagine continuing deeper to try to break ties among equally good values for the current variable. We tested that variant and we observed that there were not much difference between the two strategies, both in terms of nodes exploration and cpu time. So, we chose the simpler one.

6 Experiments

We implemented a QCSP⁺ solver on top of the constraint library Choco [5]. Our solver accepts all constraints provided by Choco, and uses the classical

constraint propagators already present in the library. In this section, we show some experimental results on using either our goal-driven heuristic or a lexicographical value ordering heuristic. We compare the performance of this solver with QeCode [2], the state-of-the-art QCSP⁺ solver, built on top of GeCode, which uses a fail-first heuristic by default. This value ordering heuristic first tries the value inconsistent with the earliest future scope. We also tried our solver with the promise heuristic [6], but it was worse than Lexico. The effect of a heuristic trying to ensure that constraints will still be consistent is that a player does not want to win as early as possible (by making the other player's rules inconsistent). In all cases, the variable ordering chosen is the same: we instantiate variables in order of the sequence.

We implemented the same model of the generalized Connect-4 game for our solver and for QeCode.

Connect-4 is a two-player game that is played on a vertical board with 6 rows and 7 columns. The players have 21 pieces each, distinguished by color. The players take turns in dropping pieces in one of the non-full columns. The piece then occupies the lowest empty cell on that column. A player wins by placing 4 of his own pieces consecutively in a line (row, column or diagonal), which ends the game. The game ends in a draw if the board is filled completely without any player winning. The generalized Connect-4 is the same game with a board of m columns and n rows, where the aim is to place k pieces in a line.

At first we tried our solver on 4×4 grids, with alignments of 3 pieces. We ran QeCode and our solver with Lexicographical heuristic (Lexico) or with Goal-Driven heuristic on problems with different number of allowed moves, from 5 to 15. We compare the time taken to solve instances.

The results are presented in Figure 1 (note the log scale). There is a solution for 9 allowed moves and more. As we can see, our solver with lexicographical value ordering solves these instances faster than QeCode. It can be explained by different ways. First, it is possible that Choco works faster in propagating constraints defined as we did. The second reason is that QeCode uses cascade propagation (propagation on the whole problem for each instantiation), whereas our solver propagates only the rules of the current scope. Thus, our solver spends

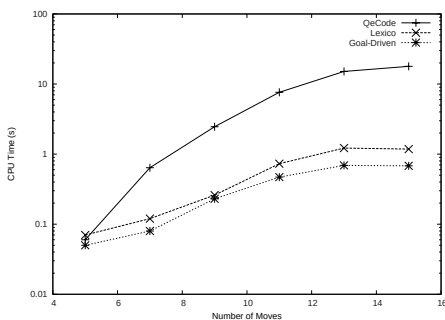


Fig. 1. Connect-3, on 4×4 grid

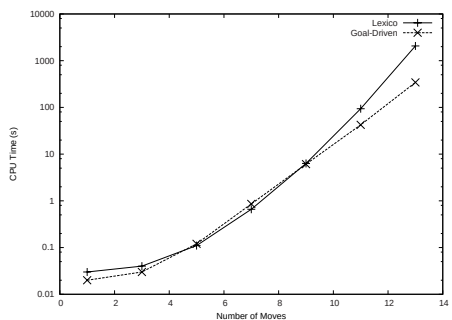


Fig. 2. Connect-4, on 7×6 grid

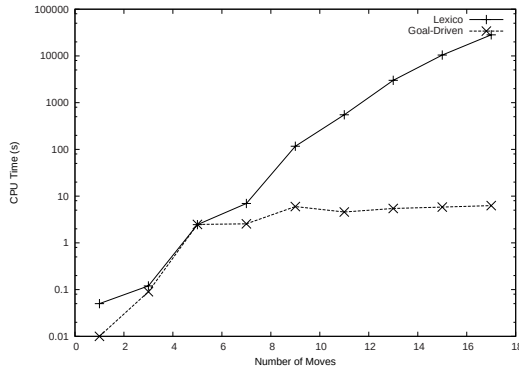


Fig. 3. Noughts and Crosses, on 5x5 grid

less time in propagation. We can see that the Goal-Driven heuristic speed up the resolution. It is about twice as fast as with Lexico.

In next experiments, we only compare our solver with Lexicographical value ordering and with Goal-Driven value ordering because QeCode was significantly slower and the aim is to test the accuracy of our heuristic.

In Figure 2, the real Connect-4 game is solved. We vary the number of moves from 1 to 13 and compare the performance in terms of running time. For a number of moves less than 9, Goal-Driven heuristic does not improve the performance, but from this point, the heuristic seems to be useful. In this problem, all instances we tested are unsatisfiable.

In Figure 3, we solve the game of Noughts and Crosses. This is the same problem except the gravity constraint which does not exist in this game. It is possible to put a piece on any free cell in the board. Instead of having n choices for a move, we have $n \times m$ choices. In the problem we tested, the aim is to align 3 pieces. This problem has a solution for 5 moves. We see that the Goal-Driven heuristic is very efficient here for solvable problems. Adding allowed moves (increasing the depth of analysis) has not a big influence on the running time of our solver with the Goal-Driven heuristic. The heuristic seems to be efficient when there are more allowed moves than necessary to finish the game.

Discussion. More generally, when can we expect our Goal-Driven heuristic to work well? As it is based on information computed by AC, it is expected to work well on constraints that propagate a lot, *i.e.*, tight constraints. Furthermore, as it actively uses quantifier alternation and tries to provoke wins/losses before the end of the sequence, it is expected to work well in problems where there exist winning/losing strategies that do not need to reach the end of the sequence.

7 Conclusion

In QCSP⁺, we cannot propagate constraints from the right of the sequence to the left. Thus, current QCSP⁺ solvers propagate only from left to right. In

this paper, we have analyzed the effect of propagation from right to left. We have derived a value ordering heuristic based on this analysis. We proposed an algorithm implementing this heuristic. Our experimental results on board games show the effectiveness of the approach.

References

1. Ansótegui, C., Gomes, C., Selman, B.: The Achille's heel of QBF. In: Proceedings AAAI 2005 (2005)
2. Benedetti, M., Lallouet, A., Vautard, J.: QCSP made practical by virtue of restricted quantification. In: Proceedings of IJCAI 2007, pp. 38–43 (2007)
3. Bessiere, C., Verger, G.: Strategic constraint satisfaction problems. In: Proceedings CP 2006 Workshop on Modelling and Reformulation, pp. 17–29 (2006)
4. Bordeaux, L., Montfroy, E.: Beyond NP: Arc-consistency for quantified constraints. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 371–386. Springer, Heidelberg (2002)
5. Choco. Java constraint library, <http://choco.sourceforge.net/>
6. Geelen, P.A.: Dual viewpoint heuristics for binary constraint satisfaction problems. In: Proceedings ECAI 1992, pp. 31–35 (1992)
7. Gent, I.P., Nightingale, P., Rowley, A., Stergiou, K.: Solving quantified constraint satisfaction problems. *Artif. Intell.* 172(6-7), 738–771 (2008)
8. Mamoulis, N., Stergiou, K.: Algorithms for quantified constraint satisfaction problems. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 752–756. Springer, Heidelberg (2004)

A New Framework for Sharp and Efficient Resolution of NCSP with Manifolds of Solutions

Alexandre Goldsztejn¹ and Laurent Granvilliers²

¹ CNRS, LINA, UMR 6241

`Alexandre.Goldsztejn@univ-nantes.fr`

² Université de Nantes, Nantes Atlantique Université, CNRS, LINA, UMR 6241

`Laurent.Granvilliers@univ-nantes.fr`

Abstract. When numerical CSPs are used to solve systems of n equations with n variables, the interval Newton operator plays a key role: It acts like a global constraint, hence achieving a powerful contraction, and proves rigorously the existence of solutions. However, both advantages cannot be used for under-constrained systems of equations, which have manifolds of solutions. A new framework is proposed in this paper to extend the advantages of the interval Newton to under-constrained systems of equations. This is done simply by permitting domains of CSPs to be parallelepipeds instead of the usual boxes.

1 Introduction

The paper presents a new framework for solving numerical CSPs formed of under-constrained systems of equations:

$$\langle \mathbf{x}, \mathbf{f}(\mathbf{x}) = \mathbf{0}, [\mathbf{x}] \rangle, \quad (1)$$

where vectorial notations¹ are used, i.e. $\mathbf{x} = (x_1, \dots, x_n)$ is a vector of variables, $\mathbf{f} = (f_1, \dots, f_m)$, with $m < n$, is a vector of functions and $[\mathbf{x}] = ([x_1], \dots, [x_n])$ is a vector of interval domains. These NCSPs arise naturally in a wide range of applications, among which are surface intersection characterization and plotting [1], a particular case of implicit equation solving [2], global optimization [3] and robots kinematics [4].

Algorithms designed for well constrained NCSP [5,6] are not efficient for solving (1). Indeed, these algorithms are variations of the branch and prune algorithm where the (preconditioned) interval Newton operator [7] plays a key role: On the one hand, it acts like a global constraint that performs powerful contraction when the domains become small enough. On the other hand, it can prove rigorously the existence of a solution of a well constrained system of equations. However, these two key contributions of the interval Newton operator are not operating when dealing with under-constrained systems of equations, mainly because no preconditioning of these systems of equations has been proposed yet. The main

¹ Vectors of reals and vectors of functions are represented with boldface symbols.

contribution of this paper is to present a framework that extends these two advantages of the interval Newton operator to NCSPs formed of under-constrained systems of equations.

In the usual definition of a CSP, each variable is given a domain. As a matter of fact, it is equivalent to consider the Cartesian product of these variable domains (which is a box when variable domains are intervals) as a search space for a vector made of the CSP variables. Then, more complicated sets, which are not anymore the Cartesian product of variable domains, can be used. The framework proposed in this paper shows that using parallelepiped domains instead of box domains can drastically improve the efficiency of branch and prune algorithms dedicated to (1). On the one hand, parallelepipeds offer a more flexible description of subsets of $\mathbb{R}^n$ than boxes, hence providing a more accurate enclosure of the NCSP solution set. On the other hand, using parallelepiped domains introduces an efficient preconditioning process for under-constrained systems of equations, hence allowing the interval Newton operator to both work as a global constraint and prove the existence of solutions.

Parallelepipeds have already been used to advantageously replace boxes, for example to rigorously enclose the solutions of initial values problems (cf. the survey paper [8] and references therein). The relationship between parallelepipeds and preconditioning has already been investigated in [9,10], where parallelepipeds are used to approximate the solution set of linear systems with interval uncertainties. However, the ways and means of the introduction of parallelepiped domains in the present work are totally different than in [9,10]. Finally, parallelepipeds have been used in conjunction with existence theorems to build inner approximations of function ranges in [11], but again in the restricted case of well-constrained systems of equations.

Outline. The framework proposed in this paper is presented on a motivating example in Section 2. Then, the concepts of interval analysis used in this paper are given in Section 3. The technical description of the proposed branch and prune algorithm is given in Section 4. Finally, promising experiments are presented in Section 5.

2 A Motivating Example

The usefulness of the usage of parallelepiped domains instead of box domains is now illustrated on a simple example.

2.1 Contractions and Bisections Using Box Domains

Let us consider the very simple under-constrained CSP

$$\langle \mathbf{x}, \{f(\mathbf{x}) = 0\}, [\mathbf{x}] \rangle, \quad (2)$$

with $\mathbf{x} = (x_1, x_2)$, $f(\mathbf{x}) = x_1^2 + x_2^2 - 1$ and $[\mathbf{x}] = ([0.3, 0.7], [0.6, 1.0])$. Its solution set is plotted in Figure 1-(a).

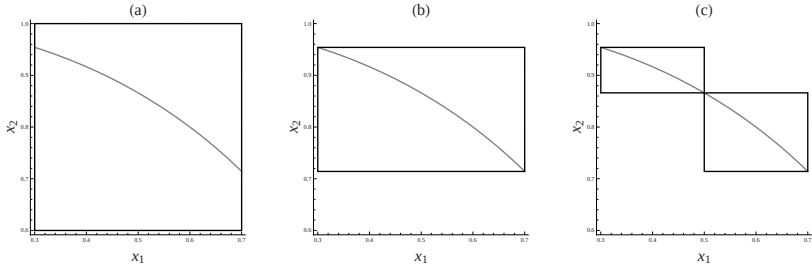


Fig. 1. (a) Solution set of the CSP (2). (b) Domain after one contraction. (c) Domains after one bisection and two more contractions.

The key point is to notice that the contraction and bisection processes of a branch and prune algorithm are not efficient in this situation: A contraction² is performed in Figure 1-(b), and two more contractions are performed after a bisection in Figure 1-(c). These plots clearly show that these contractions are not efficient because the solution set crosses the box domain in its diagonal. The contraction/bisection would be much more efficient if the solution set crossed the box domain along one of the axes. Reaching this situation is the goal of using parallelepiped domains instead of box domains.

2.2 From a Box Domain to a Parallelepiped Domain

To this end, a parallelepiped³ is built whose two axes are respectively approximately parallel and perpendicular to the solution set. These directions are related to the gradient of the function f evaluated at the midpoint of the box domain. Once the parallelepiped axes are computed, the parallelepiped is chosen as small as possible under the constraint that it contains the original box domain (cf. Figure 2-(a) where the former box domain is represented using dashed lines).

Note that changing the box domain to an enclosing parallelepiped domain introduces new solutions on the border of the parallelepiped domain (the solutions that are inside the parallelepiped domain but outside the former box domain). In order to reject these additional solutions (which otherwise would be redundant with the neighbor domains), the former box domain is reintroduced as four inequality constraints which are added to the constraint store.

To see how a parallelepiped domain can improve the contraction/bisection process, we have to formalize its definition: This parallelepiped is the image of a box through an affine map $\mathbf{u} \mapsto C \cdot \mathbf{u} + \tilde{\mathbf{x}}$, i.e.

$$\{C \cdot \mathbf{u} + \tilde{\mathbf{x}} : \mathbf{u} \in [\mathbf{u}]\}, \quad (3)$$

² In this introducing example, the best contractions are performed. In practice, contractions are not that efficient. Nevertheless, this illustrates the best that can be obtained from both methods.

³ Note that in this motivating example parallelepipeds have perpendicular axes, but this is not the case in general for higher dimensions.

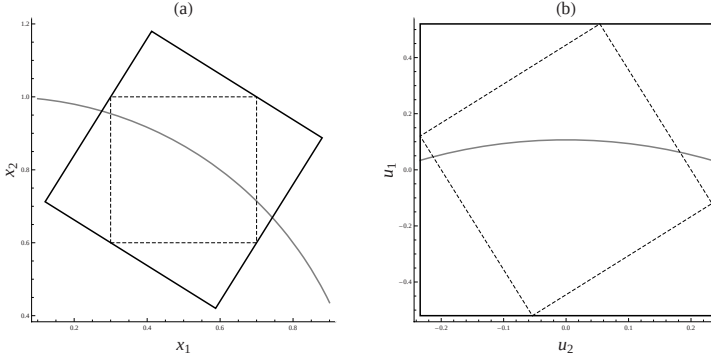


Fig. 2. (a) Enclosing parallelepiped domain for the CSP (2). (b) The same CSP expressed in the auxiliary basis formed of the parallelepiped axes.

where

- The matrix $C \in \mathbb{R}^{2 \times 2}$ is a square matrix. This matrix gives its shape to the parallelepiped and is chosen so that the solution set is approximately parallel to one of its sides. This is done considering the gradient of the function evaluated at the midpoint of the box (cf. Section 4 for more details).
- The vector $\tilde{\mathbf{x}}$ is the midpoint of the box $[\mathbf{x}]$.
- The box $[\mathbf{u}]$ is computed in such a way that the parallelepiped encloses the box domain $[\mathbf{x}]$, i.e. $[\mathbf{u}] = C^{-1} \cdot ([\mathbf{x}] - \tilde{\mathbf{x}})$ where interval arithmetic is used (cf. Section 3).

Then, the CSP (2) can be expressed in the auxiliary basis formed of the characteristic axes of the parallelepiped, giving rise to the auxiliary CSP

$$\langle \mathbf{u}, \{g(\mathbf{u}) = 0\}, [\mathbf{u}] \rangle, \quad (4)$$

with $g(\mathbf{u}) = f(C \cdot \mathbf{u} + \tilde{\mathbf{x}})$, that is explicitly

$$g(u_1, u_2) = f(C_{11}u_1 + C_{12}u_2 + \tilde{x}_1, C_{21}u_1 + C_{22}u_2 + \tilde{x}_2). \quad (5)$$

The solution sets of (4) is represented by Figure 2-(b). As mentioned previously, four linear inequalities are added to the constraints of (4) which represent the belonging to the original box domain (they are not given explicitly in (4) for clarity). These inequalities are represented using dashed lines in Figure 2-(b). Note that the solution sets of (2) and (4) are closely related: The former is exactly the image of the latter through the affine transformation $\mathbf{u} \mapsto C \cdot \mathbf{u} + \tilde{\mathbf{x}}$.

The next two subsections show how to contract and bisect this parallelepiped domain.

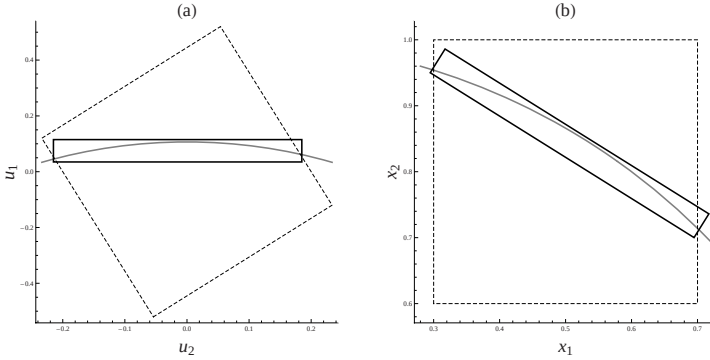


Fig. 3. Contracting parallelepiped domains

2.3 Contracting Parallelepiped Domains

Contracting the parallelepiped $\{C \cdot \mathbf{u} + \tilde{\mathbf{x}} : \mathbf{u} \in [\mathbf{u}]\}$ consists in contracting its characteristic domain $[\mathbf{u}]$. The aim is to keep all the solutions of the original CSP (2) within the contracted parallelepiped. This is obviously equivalent to contracting $[\mathbf{u}]$ without losing any solution of the auxiliary CSP (4). As this later CSP has a box domain, one can use the usual techniques dedicated to NCSPs. Note that since the solution set crosses the parallelepiped in the direction u_2 , the constraint $g(\mathbf{u}) = 0$ will contract efficiently the domain $[u_1]$ but will certainly be useless for the domain $[u_2]$. On the other hand, the inequality constraints coming from the box domain will help contracting the domain $[u_2]$. The contraction of $[\mathbf{u}]$ obtained for this introducing example is shown in Figure 3. Figure 3-(a) shows how $[\mathbf{u}]$ is contracted using the auxiliary CSP (4) while Figure 3-(b) shows the corresponding contraction for the parallelepiped domain of the original CSP (2). Comparing Figure 3-(b) to Figure 1-(b) shows how much more efficient the contraction of the parallelepiped domain is compared to the contraction of the original box domain.

The auxiliary CSP (4) is actually more complicated than the original CSP (2): The function $g(\mathbf{u}) = f(C \cdot \mathbf{u} + \tilde{\mathbf{x}})$ contains more occurrences of each variables (cf. Equation (5)), which is well known to decrease the efficiency of interval based methods. However, this situation is very similar to the preconditioning of the interval Newton operator. And indeed, this acts like a *right-preconditioning*⁴ where the interval Newton operator should be very efficient. Actually, Section 4 shows that this right-preconditioning process allows the interval Newton to both act like a global constraint, and rigorously prove the existence of the manifold of solutions. In the context of this introducing example, the interval Newton operator proves that for all $u_2 \in [u_2]$ there exists $u_1 \in [u_1]$ such that $g(\mathbf{u}) = 0$, hence proving that the auxiliary solution set (4) crosses $[\mathbf{u}]$ along u_2 . Therefore, the original (2) is proved to cross the parallelepiped in this direction.

⁴ I.e. a preconditioning where the change of basis is applied before the function, see e.g. [9,10,11].

2.4 Bisecting Parallelepiped Domains

Bisecting a parallelepiped $\{C \cdot \mathbf{u} + \tilde{\mathbf{x}} : \mathbf{u} \in [\mathbf{u}]\}$ into two smaller parallelepipeds is naturally done by bisecting its characteristic domain $[\mathbf{u}]$ (cf. Figure 4-(a)), thus obtaining two new parallelepipeds $\{C \cdot \mathbf{u} + \tilde{\mathbf{x}} : \mathbf{u} \in [\mathbf{u}']\}$ and $\{C \cdot \mathbf{u} + \tilde{\mathbf{x}} : \mathbf{u} \in [\mathbf{u}'']\}$ (cf. Figure 4-(b)). Note that it obviously makes no sense to bisect $[\mathbf{u}]$ to $([u'_1], [u_2])$ and $([u''_1], [u_2])$, which would not preserve the solution set transverse crossing. Instead, bisecting $[\mathbf{u}]$ to $([u_1], [u'_2])$ and $([u_1], [u''_2])$ does preserve this transversality, and is therefore a very efficient bisection heuristic. This bisection heuristic will be trivially extended to the general case of arbitrary dimensions.

Once bisected, their characteristic matrices are updated using the same process as described previously, but based on the gradient vector of f evaluated at the center of each new parallelepiped. This allows the new parallelepipeds adapting their shape more accurately to the shape of the solution set (cf. Figure 4-(c)). Furthermore, as done with the original box domain, each former parallelepiped domain is expressed as four additional inequality constraints in its CSP (represented in dashed lines in Figure 4-(c)), which will be used to reduce the overlapping introduced when updating the characteristic matrices of the new parallelepipeds. Finally, a contraction is performed on these two new parallelepipeds, leading to Figure 4-(d). Comparing this figure to Figure 1-(c)

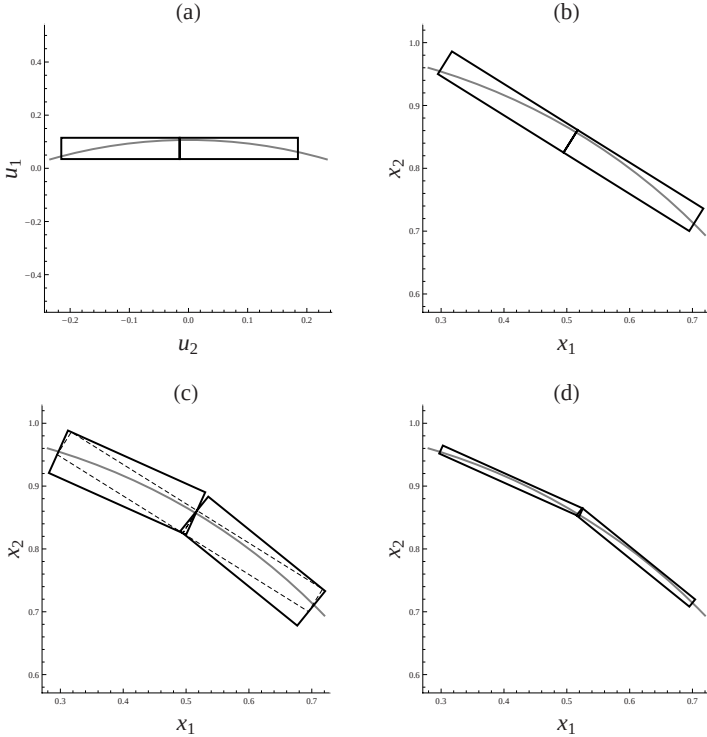


Fig. 4. Bisecting parallelepiped domains

shows how important is the improvement obtained using parallelepipeds domains instead of box domains.

The gains in contracting efficiency and existence proving illustrated on this motivating example are even more important when dealing with more complicated NCSPs of higher dimension (c.f. subsections 5.3 and 5.4).

3 Interval Analysis for NCSP Resolution

The modern interval analysis was born in the 60's with [12]. Since, it has been widely developed and is today one central tool in the resolution of constraints acting over continuous domains (see [13] and extensive references).

3.1 Interval Arithmetic

Intervals, interval vectors and interval matrices are denoted using brackets. Their set are denoted respectively by $\mathbb{IR}$, $\mathbb{IR}^n$ and $\mathbb{IR}^{n \times m}$. The elementary functions are extended to intervals in the following way: let $\circ \in \{+, -, \times, /\}$ then $[x] \circ [y] = \{x \circ y : x \in [x], y \in [y]\}$ (division is defined only for non zero containing interval denominators). E.g. $[a, b] + [c, d] = [a + c, b + d]$. Also, continuous one variable functions $f(x)$ are extended to intervals using the same definition: $f([x]) = \{f(x) : x \in [x]\}$, which is an interval because f is continuous. When one represents numbers using a finite precision, the previous operations cannot be computed in general. The outer rounding is then used so as to keep valid the interpretations. For example, $[1, 2] + [2, 3]$ would be equal to $[2.999, 5.001]$ if rounded with a three decimal accuracy.

Then, an expression which contains intervals can be evaluated using this interval arithmetic. The main property of interval analysis is that such an interval evaluation gives rise to a superset of the image through the function of the interval arguments: For example, $[x] \times ([y] - [x]) \supseteq \{x(y - x) : x \in [x], y \in [y]\}$. In some cases (e.g. when the expression contains only one occurrence of each variable), this enclosure is optimal. In particular, the computation $C \cdot [\mathbf{u}] + \tilde{\mathbf{x}}$ is the smallest box that contains the parallelepiped $\{C \cdot \mathbf{u} + \tilde{\mathbf{x}} : \mathbf{u} \in [\mathbf{u}]\}$.

3.2 Interval Contractors

Given an n -ary constraint c and a box $[\mathbf{x}] \in \mathbb{IR}^n$, a contractor for c will contract the box $[\mathbf{x}]$ without losing any solution of c . Some widely used contractors are based on the 2B-consistency (also called hull-consistency) or the box consistency [14,15], which are adaptations of the arc-consistency to continuous domains. They are both applied to one constraint at a time, hence suffering of the usual drawbacks of the locality of their application. On the other hand, the preconditioned interval Newton [7] can be applied to a set of n equations and n variables. Under some hypothesis on the Jacobian matrix of the system evaluated on the domain of the CSP, it will be able to treat this set of constraint as a global constraint and hence achieve a powerful contraction. Furthermore,

the interval Newton can rigorously prove the existence of a solution in a CSP domain. These two characteristics of the interval Newton makes it a key tool for the resolution of NCSPs, however previously restricted to well-constrained systems of equations.

4 Description of the Algorithm

The branch and prune Algorithm 1 used here is classical except for line 1 which will be explained in the rest of the paper: The input is a CSP $\mathcal{P}$ and the output a set of CSPs $\mathcal{L} = \{\mathcal{P}_1, \dots, \mathcal{P}_s\}$ whose disjunction is equivalent to the original CSP. Formally,

$$\text{Sol}(\mathcal{P}) = \bigcup_{Q \in \mathcal{L}} \text{Sol}(\mathcal{Q}). \quad (6)$$

Normally, the set of CSPs $\mathcal{L}$ is a more accurate description of the solution set than the original CSP: First, the union of their domains is much smaller than the original domain, hence providing a sharper enclosure. Second, it is often possible to prove that these CSPs actually contains some solutions, which is a crucial information. The solution existence proof is not explicitly described in Algorithm 1 for clarity. Informally, some existence proof can result of the contraction performed at Line 1 when it is computed using the interval Newton operator. When the existence of solutions is proved, this information is attached to the CSP.

When the algorithm starts, the domain of the CSP is a box. The function `UpdateDomainShape` at Line 1 allows changing the shape of the domain (changing a box to a parallelepiped when first successfully applied, or a parallelepiped to another parallelepiped more suited to the shape of the solution set). The first change from a box to a parallelepiped is performed only when the shape of the solution set can be foreseen from the evaluation of the constraints derivatives (cf. Subsection 4.1), which implies that the box domain is small enough. As a consequence, the algorithm will use box domains in a first phase, and parallelepiped domains as soon as box domains are small enough to identify the directions of the solution manifold. While the CSP domain is a box, usual contractors are used to prune its domain (2B-consistency based contractors and non-preconditioned interval Newton operator in our implementation of Algorithm 1, whose collaboration is known to be efficient). When the domain is changed to a parallelepiped, the interval Newton operator is adapted and allows powerful contractions and existence proof of solutions (cf. Subsection 4.2). Finally, parallelepiped domains are bisected similarly to box domains (cf. Subsection 4.3).

4.1 Changing the Shape of a Parallelepiped

The function `UpdateDomainShape`($\mathcal{P}$) attempts to find a new parallelepiped domain $\{C \cdot \mathbf{u} + \tilde{\mathbf{x}} : \mathbf{u} \in [\mathbf{u}]\}$ that will be more suited to the solution set of $\mathcal{P}$. The former domain of $\mathcal{P}$ is the parallelepiped $\{D \cdot \mathbf{v} + \tilde{\mathbf{x}} : \mathbf{v} \in [\mathbf{v}]\}$, which is possibly a box in which case $D = \text{id}$ (note that the former and the new

Algorithm 1. Branch and prune algorithm using parallelepiped domains

```

Input:  $\mathcal{P} = \langle \mathbf{x}, C, [\mathbf{x}] \rangle, \epsilon$ 
Output:  $\mathcal{L} = \{\mathcal{P}_1, \dots, \mathcal{P}_s\}$ 
1  $\mathcal{T} \leftarrow \{\mathcal{P}\}; \mathcal{L} \leftarrow \emptyset;$ 
2 while ( not  $\mathcal{T} = \emptyset$  ) do
3    $\mathcal{P} \leftarrow \text{Extract}(\mathcal{T});$ 
4   if (  $\text{Measure}(\text{Domain}(\mathcal{P})) > \epsilon$  ) then
5      $\mathcal{P}' \leftarrow \text{UpdateDomainShape}(\mathcal{P});$ 
6      $\mathcal{P}'' \leftarrow \text{Contract}(\mathcal{P}');$ 
7     if (  $\text{Domain}(\mathcal{P}'') \neq \emptyset$  ) then
8        $(\mathcal{Q}, \mathcal{Q}') \leftarrow \text{Bisect}(\mathcal{P}'');$ 
9        $\mathcal{T} \leftarrow \mathcal{T} \cup \{\mathcal{Q}, \mathcal{Q}'\};$ 
10    end
11  else
12     $\mathcal{L} \leftarrow \mathcal{L} \cup \{\mathcal{P}\};$ 
13  end
14 end
15 return ( $\mathcal{L}$ );

```

parallelepiped domains share the same characteristic vector $\tilde{\mathbf{x}}$). This process is done in two steps: First a candidate new parallelepiped domain $\{C \cdot \mathbf{u} + \tilde{\mathbf{x}} : \mathbf{u} \in [\mathbf{u}]\}$ is computed. Then, the efficiency of this candidate parallelepiped domain is verified a posteriori. If not useful, the former parallelepiped is kept. If the candidate parallelepiped domain is chosen, then some inequality constraints are added to the CSP in order to prevent some parasite solutions to appear due to the enclosure of the former parallelepiped domain inside a new parallelepiped (cf. Section 2).

Computation of the Candidate Parallelepiped Domain. As illustrated in Section 2, the aim of using a new parallelepiped domain $\{C \cdot \mathbf{u} + \tilde{\mathbf{x}} : \mathbf{u} \in [\mathbf{u}]\}$ is to apply the interval Newton operator in the auxiliary basis of the parallelepiped to reduce directly the box domain $[\mathbf{u}]$. Hence, the parallelepiped characteristic matrix C is chosen aiming an efficient application of the interval Newton operator. The box domain $[\mathbf{u}]$ will be reduced using the constraint $\mathbf{g}(\mathbf{u}) = \mathbf{0}$ where $\mathbf{g}(\mathbf{u}) = \mathbf{f}(C \cdot \mathbf{u} + \tilde{\mathbf{x}})$ (cf. Section 2). To obtain an efficient application of the interval Newton operator, the under-constrained system $\mathbf{g}(\mathbf{u}) = \mathbf{0}$ needs to be interpreted as a system of equations m equations and m variables, where the remain $n - m$ variables are considered as parameters. Then, to be efficient the interval Newton operator requires the Jacobian $J\mathbf{g}$ of $\mathbf{g}$ to be close to the matrix $(I_{m \times m} \mid 0_{m \times (n-m)})$, where $I_{m \times m}$ is the identity matrix of size $m \times m$ and $0_{m \times (n-m)}$ is the null matrix of size $m \times (n - m)$. As $J\mathbf{g} = J\mathbf{f} \cdot C$, it is natural to choose C such that

$$J\mathbf{f}([\mathbf{x}]) \cdot C \approx (I_{m \times m} \mid 0_{m \times (n-m)}), \quad (7)$$

where $[\mathbf{x}]$ is the interval hull of the parallelepiped domain, i.e. $[\mathbf{x}] = C \cdot [\mathbf{u}] + \tilde{\mathbf{x}}$.

To this end, C is constructed based on the evaluation of the Jacobian of $\mathbf{f}$ at $\tilde{\mathbf{x}}$. The matrix C^{-1} is first constructed as follows, and C will be obtained inverting C^{-1} . The i^{th} line $(C^{-1})_i$ of C^{-1} is defined by:

- The gradient of f_i evaluated at $\tilde{\mathbf{x}}$ if $i \leq m$.
- A vector orthogonal to all previously computed $(C^{-1})_k$ for $1 \leq k < i$ (these vectors are not uniquely determined, but a set of such vectors is easily obtain using a Gram-Schmidt orthogonalization).

Then, this matrix is inverted⁵ to obtain C . The matrix C thus satisfies $\mathbf{Jf}(\tilde{\mathbf{x}}) \cdot C = (I_{m \times m} \mid 0_{m \times (n-m)})$, up to rounding errors (indeed by construction, $\mathbf{Jf}(\tilde{\mathbf{x}}) \cdot C$ is made of the m first rows of the $n \times n$ identity matrix).

Finally, the characteristic domain $[\mathbf{u}]$ is computed so that the new parallelepiped domain encloses the former: $[\mathbf{u}] = (C^{-1}D)[\mathbf{v}]$. This time, interval arithmetic is used to ensure a rigorous enclosure.

Verification of the Efficiency of the Candidate New Parallelepiped Domain. Formally, the interval Newton will be efficient if the square matrix formed of the first m columns of $\mathbf{Jf}([\mathbf{x}]) \cdot C$ is diagonally dominant (cf. Theorem 5.2.5 in [7]). Therefore, the interval matrix $\mathbf{Jf}([\mathbf{x}]) \cdot C$ is computed explicitly, and the diagonal dominance of its first m columns is checked. If it is diagonally dominant, the parallelepiped domain is updated, and some inequality constraints are added to the CSP (cf. the next section). Otherwise, the former parallelepiped domain is kept, i.e. $C = D$ and $[\mathbf{u}] = [\mathbf{v}]$.

Remark 1. If $\mathbf{x}$ is a solution (i.e. $\mathbf{f}(\mathbf{x}) = \mathbf{0}$) and is singular (i.e. $D\mathbf{f}(\mathbf{x})$ is not full rank, e.g. has two proportional lines) then $\mathbf{Jf}([\mathbf{x}]) \cdot C$ will never be diagonally dominant. In this case, the algorithm keeps working with box domains around this point. This situation is untypical as some arbitrary small perturbation of the problem can change the singular solutions to regular solutions.

Adding Linear Inequalities to the new CSP. As illustrated in Section 2, the enclosure of the former parallelepiped domain by a new parallelepiped domain is not perfect. Hence, some solutions that are inside the new domain may not be in the former, which would lead to redundant solutions if not properly treated. To this end, the former parallelepiped domain is reintroduced in the new CSP as $2n$ linear constraints:

$$\underline{v}_i \leq \sum_{1 \leq j \leq n} D_{ij}x_j \leq \bar{v}_i, \quad (8)$$

for $1 \leq i \leq n$. As a consequence, the former and the new CSPs have the same solution set. These linear inequalities will be denoted using the scalar product notation $\mathbf{a} \cdot \mathbf{x} \leq b$ where $\mathbf{a} = (D_{i1}, \dots, D_{in})$ and $b = \bar{v}_i$, or $\mathbf{a} = -(D_{i1}, \dots, D_{in})$ and $b = -\underline{v}_i$.

⁵ Note that C needs only to be computed approximately and thus standard double precision computations can be used at this step.

4.2 Contracting a Parallelepiped Domain

When the domain of the CSP is a box, the usual techniques are used to contract it. When it is a parallelepiped, two kind of constraints are in the store: the linear inequalities $\mathbf{a} \cdot \mathbf{x} \leq b$ and the nonlinear equalities $\mathbf{f}(\mathbf{x}) = \mathbf{0}$.

Linear Inequalities. In order to contract the characteristic domain $[\mathbf{u}]$ of the parallelepiped $\{C \cdot \mathbf{u} + \tilde{\mathbf{x}} : \mathbf{u} \in [\mathbf{u}]\}$ under the linear inequality $\mathbf{a} \cdot \mathbf{x} \leq b$, this constraint is simply expressed in the basis of the parallelepiped:

$$\mathbf{a} \cdot \mathbf{x} \leq b \iff (\mathbf{a} \cdot C) \cdot \mathbf{u} \leq b - \mathbf{a} \cdot \tilde{\mathbf{x}}. \quad (9)$$

Then, $[\mathbf{u}]$ is contracted under the new linear inequality using the 2B-consistency.

Nonlinear Equalities. The nonlinear equalities are also expressed in the basis of the parallelepiped: $\mathbf{f}(\mathbf{x}) = \mathbf{0} \iff \mathbf{g}(\mathbf{u}) = \mathbf{0}$, with $\mathbf{g}(\mathbf{u}) = \mathbf{f}(C \cdot \mathbf{u} + \tilde{\mathbf{x}})$. Then the m first components of $[\mathbf{u}]$ are contracted considering this under-constrained system of equations as a well constrained parametric systems of equations. Let $\mathbf{u}' = (u_1, \dots, u_m)$ be the vector of the first m variables, and $\mathbf{u}'' = (u_{m+1}, \dots, u_n)$ be the vector of the remaining variables, which are considered as parameters. The interval Newton is then applied to the parametric system of equation (see [16] for details) to contract $[\mathbf{u}']$ to a smaller box $[\tilde{\mathbf{u}}']$

$$[\tilde{\mathbf{u}}'] = \hat{\mathbf{u}}' + \Gamma([J_{\mathbf{u}'}], [\mathbf{u}'] - \hat{\mathbf{u}}', \mathbf{b}) \quad (10)$$

with $\mathbf{b} = -\mathbf{f}(C \cdot \mathbf{u} + \tilde{\mathbf{x}}) - [J_{\mathbf{u}''}] \cdot ([\mathbf{u}''] - \hat{\mathbf{u}}'')$ where $[J_{\mathbf{u}'}]$ and $[J_{\mathbf{u}''}]$ are respectively the square interval matrix formed of the first m columns of $J\mathbf{f}(C \cdot [\mathbf{u}] + \tilde{\mathbf{x}}) \cdot C$ and the rectangular matrix formed of the remaining columns. The operator Γ is the interval Gauss-Seidel method [7]. The vectors $\hat{\mathbf{u}}'$ and $\hat{\mathbf{u}}''$ are the midpoint of the respective boxes.

As the interval matrix $[J_{\mathbf{u}'}]$ is centered on the identity matrix and diagonally dominant (cf. Section 4.1), the interval Newton operator may be strictly contracting (i.e. $[\tilde{\mathbf{u}}']$ is included inside the interior of $[\mathbf{u}']$) and hence is able to prove the existence of solutions. In this case, the following existence statement holds [16]: $\forall \mathbf{u}'' \in [\mathbf{u}''], \exists \mathbf{u}' \in [\mathbf{u}'], \mathbf{g}(\mathbf{u}) = \mathbf{0}$, hence proving that the solution set crosses the whole parallelepiped domain transversally. When the existence is proved, this information is recorded together with the CSP.

4.3 Bisecting a Parallelepiped Domain

A parallelepiped domain $\{C \cdot \mathbf{u} + \tilde{\mathbf{x}} : \mathbf{u} \in [\mathbf{u}]\}$ is bisected splitting $[\mathbf{u}]$. However, as the contraction of $[\mathbf{u}']$ performed using the interval Newton is convergent, it is useless to bisect these components of the box. Therefore, $[\mathbf{u}]$ is bisected to $[\tilde{\mathbf{u}}_1]$ and $[\tilde{\mathbf{u}}_2]$ where the largest component of $\mathbf{u}''$ has been bisected. This is sufficient to ensure the convergence of the algorithm by Theorem 5.2.5 in [7]. Finally, the two bisected parallelepipeds $\{C \cdot \mathbf{u} + \tilde{\mathbf{x}}_1 : \mathbf{u} \in [\mathbf{u}_1]\}$ and $\{C \cdot \mathbf{u} + \tilde{\mathbf{x}}_2 : \mathbf{u} \in [\mathbf{u}_2]\}$ are defined as follows: for both $k = 1$ and $k = 2$, $\tilde{\mathbf{x}}_k = C \cdot \text{mid}([\tilde{\mathbf{u}}_k]) + \tilde{\mathbf{x}}$ so that this vector is at the center of the parallelepiped, and hence is representative of the domain. The domains $[\mathbf{u}_1]$ and $[\mathbf{u}_2]$ are then updated accordingly translating $[\tilde{\mathbf{u}}_1]$ and $[\tilde{\mathbf{u}}_2]$, i.e. $[\mathbf{u}_k] = [\tilde{\mathbf{u}}_k] + C^{-1}(\tilde{\mathbf{x}} - \tilde{\mathbf{x}}_k)$.

5 Experiments

Experiments presented in this section cover a wide range of situations, from 2D implicit functions to higher dimensional systems. For comparing the usage of parallelepiped domains and box domains, the volume of each enclosure will be compared. In order to have a dimension free measure, the *reduced-volume*, defined as the n^{th} root of the volume where n is the dimension of the problem, will be used. Note that the volume of a parallelepiped is simply the volume of its characteristic domain multiplied by the absolute value of the determinant of its characteristic matrix. The algorithms have been run on a Intel(R) Core(TM)2 Duo CPU at 2.20 GHz, with 4Gb of memory, under Windows XP.

5.1 Intersection of Surfaces

The validated intersection between a sphere and a cylinder is modeled by a CSP with 3 variables and 2 constraints. Both parallelepiped domains and box domains have been used to compute the enclosure of this geometrical object for comparison purpose. Figure 5 shows that, for the same number of bisections, the parallelepiped domains provide a much more accurate enclosure of the solution set. Furthermore, the solution set has been proved rigorously to cross each parallelepiped domain. That information is unavailable when using box domains.

5.2 Two Dimensional Implicit Plot

The problem consists in determining the implicit graph of a complicated function proposed in [1]:

$$f(\mathbf{x}) = x_1 \cos(x_2) \cos(x_1 x_2) + x_2 \cos(x_1) \cos(x_1 x_2) + x_1 x_2 \cos(x_1) \cos(x_2). \quad (11)$$

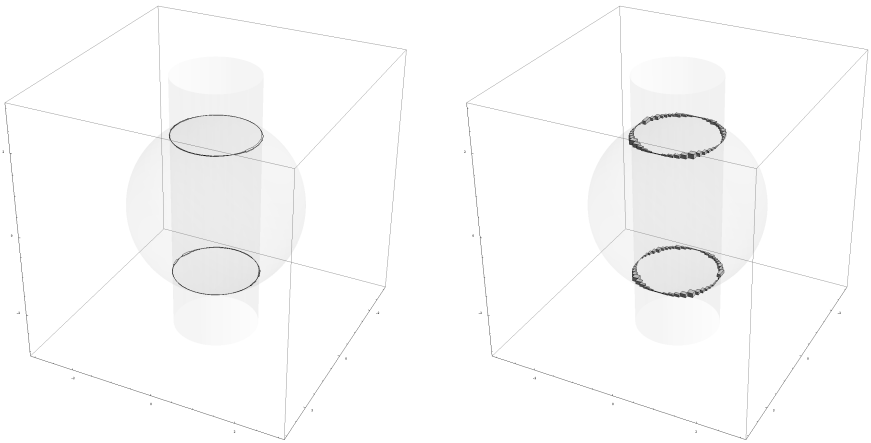


Fig. 5. Intersection of a sphere and a cylinder (both plotted in transparent for information). Enclosures obtained using parallelepiped domains (left) or with box domains (right) after 100 bisections.

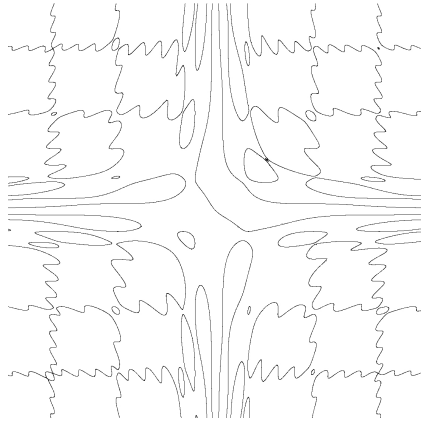


Fig. 6. Verified implicit plot of Tupper's function

Note that in [1] each plus sign is actually consider as a plus/minus sign, hence leading to four different functions, while here we treat (11). After 72 seconds, the enclosure shown on Figure 6 has been computed for $f(\mathbf{x}) = 0$. All the parallelepiped domains are rigorously proved to be crossed by the solution set, this latter being thus completely determined. Timings cannot be compared wrt. the algorithm proposed in [1] because the present approach provides much more information than [1], where a simple outer approximation is computed, exactly as accurate as the pixel size of the resolution.

5.3 The Layne-Watson Exponential Cosine Curve

This system of 3 variables and 2 equations is taken from [2]. This NCSP is an intersection of surfaces whose equations involve compositions of cosine and exponential functions. Once more, the parallelepiped domains provide a more accurate enclosure than box domains, and allow proving the existence of the

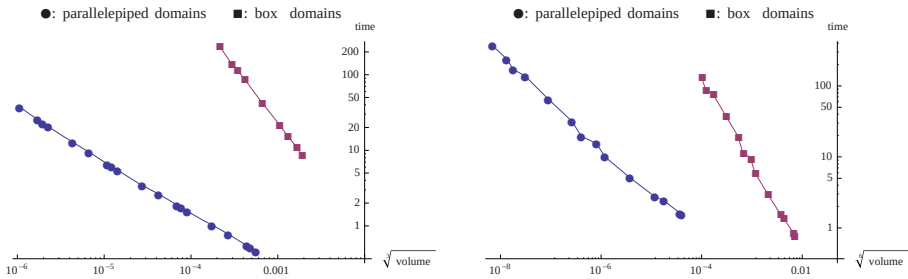


Fig. 7. Log/log plots of the reduced-volume of the enclosure against the time needed to compute this enclosure. Left: The Layne-Watson Exponential Cosine Curve. Right: The Parametrized Broyden Tridiagonal.

whole manifold of solutions. For a more accurate comparison of performances, the left hand side graphic of Figure 7 displays the time needed to obtain a given reduced-volume for both methods. At a first glance, the parallelepiped domains are much more efficient than box domains. Interpreting more precisely these log/log plots, both curves are almost lines. Therefore, both algorithms display a time increasing polynomially with the inverse of the reduced-volume. What is noteworthy is that the slopes of the two lines are different (corresponding to polynomials of different degree), which means that the parallelepiped domains improve not only the timings but also the complexity of the branch and prune algorithm.

5.4 The Parametrized Broyden Tridiagonal

The Broyden tridiagonal problem is a well-known system of n equations and n unknowns [17]. The problem is changed to a parametric problem adding a variable α :

$$(\alpha - 2x_i)x_i - x_{i-1} - 2x_{i+1} + 1 = 0 \quad (12)$$

for $1 \leq i \leq n$, with $x_0 = x_{n+1} = 0$ for compact notations. The original value of α is 3, so we choose the domain $[2, 4]$ for this additional variable. As in the original problem, the domains of the other variables are $[-100, 100]$. Introducing such a parameter allows studying how the solutions of the original Broyden tridiagonal problem change with the variations of this parameter. Once more, the usage of parallelepiped domains drastically reduces the time needed to obtain an enclosure of a given reduced-volume, cf. Figure 7. And again, Figure 7 shows that the time needed is approximately a polynomial of the inverse of the reduced-volume, while the usage of parallelepiped domains has reduced the degree of this polynomial.

6 Discussion

These first experiments show that indeed the global constraint contraction performed on parallelepiped domains drastically improves the resolution process (the degree of the polynomial time complexity to reach a given accuracy seems to have been reduced, leading to timings divided by more than 100). Furthermore, the proof of existence of solutions works well for non singular solutions, and allows giving a complete description of the solution set under the form of a sharp enclosure which is proved to contain solutions.

Under-constrained systems of equations appear in many contexts. An important part of the forthcoming work will be to include this framework in an efficient solver to be able to tackle real life problems, like robot workspace computation. Furthermore, parallelepiped domains may improve the efficiency of global optimization algorithms, which often have to solve under-constrained systems of equations. On a theoretical point of view, the introduction of universally quantified parameters in this framework will allow tackling interesting problems, like the robust intersection of surfaces with uncertain parameters.

Acknowledgments. This work was partially funded by the ANR grant number PSIROB06_174445. The authors are grateful to Arnold Neumaier for the useful discussions they had about the relationship between the preconditioned interval Newton operator and global constraints.

References

1. Tupper, J.: Reliable two-dimensional graphing methods for mathematical formulae with two free variables. In: SIGGRAPH 2001, pp. 77–86. ACM, New York (2001)
2. Kearfott, R.B., Xing, Z.: An Interval Step Control for Continuation Methods. *SIAM J. Numer. Anal.* 31(3), 892–914 (1994)
3. Hansen, E.: *Global Optimization Using Interval Analysis*, 2nd edn. Marcel Dekker, NY (1992)
4. Merlet, J.: *Parallel robots*. Kluwer, Dordrecht (2000)
5. Van Hentenryck, P., McAllester, D., Kapur, D.: Solving polynomial systems using a branch and prune approach. *SIAM J. Numer. Anal.* 34(2), 797–827 (1997)
6. Granvilliers, L., Benhamou, F.: RealPaver: An Interval Solver using Constraint Satisfaction Techniques. *ACM Trans. Math. Soft.* 32(1), 138–156 (2006)
7. Neumaier, A.: *Interval Methods for Systems of Equations*. Cambridge U. P, Cambridge (1990)
8. Nedialkov, N.S., Jackson, K.R., Corliss, G.F.: Validated Solutions of Initial Value Problems for Ordinary Differential Equations. *Applied Mathematics and Computation* 105(1), 21–68 (1999)
9. Neumaier, A.: Overestimation in linear interval equations. *SIAM J. Numer. Anal.* 24(1), 207–214 (1987)
10. Goldsztejn, A.: A Right-Preconditioning Process for the Formal-Algebraic Approach to Inner and Outer Estimation of AE-solution Sets. *Reliab. Comp.* 11(6), 443–478 (2005)
11. Goldsztejn, A., Hayes, W.: Reliable Inner Approximation of the Solution Set to Initial Value Problems with Uncertain Initial Value. In: *Proc. of SCAN 2006* (2006)
12. Moore, R.: *Interval Analysis*. Prentice-Hall, Englewood Cliffs (1966)
13. Benhamou, F., Older, W.: Applying Interval Arithmetic to Real, Integer and Boolean Constraints. *Journal of Logic Programming* 32(1), 1–24 (1997)
14. Benhamou, F., McAllister, D., Hentenryck, P.V.: CLP(Intervals) Revisited. In: *International Symposium on Logic Programming*, pp. 124–138 (1994)
15. Collavizza, H., Delobel, F., Rueher, M.: Comparing Partial Consistencies. *Reliab. Comp.* 1, 1–16 (1999)
16. Goldsztejn, A.: A Branch and Prune Algorithm for the Approximation of Non-Linear AE-Solution Sets. In: *Proc. of ACM SAC 2006*, pp. 1650–1654 (2006)
17. Moré, J., Garbow, B., Hillstom, K.: Testing unconstrained optimization software. *ACM Trans. Math. Software* 7(1), 136–140 (1981)

A Branch and Bound Algorithm for Numerical MAX-CSP

Jean-Marie Normand¹, Alexandre Goldsztejn^{1,2}, Marc Christie^{1,3},
and Frédéric Benhamou¹

¹ University of Nantes, LINA UMR CNRS 6241

² CNRS, France

³ INRIA Rennes Bretagne-Atlantique

firstname.lastname@univ-nantes.fr

Abstract. The Constraint Satisfaction Problem (CSP) framework allows users to define problems in a declarative way, quite independently from the solving process. However, when the problem is over-constrained, the answer “no solution” is generally unsatisfactory. A Max-CSP $\mathcal{P}_m = \langle V, \mathbf{D}, C \rangle$ is a triple defining a constraint problem whose solutions maximise constraint satisfaction. In this paper, we focus on numerical CSPs, which are defined on real variables represented as floating point intervals and which constraints are numerical relations defined in extension. Solving such a problem (*i.e.*, exactly characterizing its solution set) is generally undecidable and thus consists in providing approximations. We propose a branch and bound algorithm that computes under and over approximations of its solution set and determines the maximum number $m_{\mathcal{P}}$ of satisfied constraints. The technique is applied on three numeric applications and provides promising results.

1 Introduction

CSP provides a powerful and efficient framework for modeling and solving problems that are well defined. However, in the modeling of real-life applications, under or over-constrained systems often occur. In embodiment design, for example, when searching for different concepts engineers often need to tune both the constraints and the domains. In such cases, the classical CSP framework is not well adapted (a *no solution* answer is certainly not satisfactory) and there is a need for specific frameworks such as Max-CSP. Yet in a large number of cases, the nature of the model (over-, well or under-constrained) is *a priori* not known and thus the choice of the solving technique is critical. Furthermore, when tackling over-constrained problems in the Max-CSP framework, most techniques focus on discrete domains.

In this paper, we propose a branch and bound algorithm that approximate the solutions of a numerical Max-CSP. The algorithm computes both an under (often called inner in continuous domains) and an over (often called outer) approximation of the solution set of a Max-CSP along with the sets of constraints that maximise constraint satisfaction. These approximations ensure that the

inner approximation contains only solutions of the Max-CSP while the outer approximation guarantees that no solution is lost during the solving process.

This paper is organized as follows: Section 2 motivates the research and presents a numerical Max-CSP application. Section 3 presents the definitions associated to the Max-CSP framework. A brief introduction to Interval analysis and its related concepts is drawn in section 4, while our branch and bound algorithm is described in Section 5. Section 6 presents the work related to Max-CSP in both continuous and discrete domains while Section 7 is dedicated to experimental results showing the relevance of our approach.

2 A Motivating Example

We motivate the usefulness of the numerical Max-CSP approach on a problem of virtual camera placement (VCP), which consists in positioning a camera in a 2D or a 3D environment w.r.t. constraints defined on the objects of a virtual scene.

Within a classical CSP framework, these cinematographic properties are transformed into numerical constraints in order to find an appropriate camera position and orientation. One of the counterparts of the expressiveness offered by the constraint programming framework lies in the fact that it is easy for the user to specify an over-constrained problem. In such cases, classical solvers will output a *no solution* statement and does not provide any information neither on why the problem was over-constrained nor on how it could be softened. The Max-CSP framework will, on the other hand, explore the search space in order to compute the set of configurations that satisfy as many constraints as possible. This provides a classification of the solutions according to satisfied constraints and enables the user to select one, or to remodel the problem.

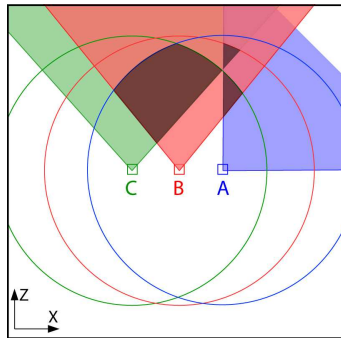


Fig. 1. Top-view of a 3D scene representing camera's position search space for a VCP problem. Orientations of objects are represented by colored areas, *shot* distances are represented as circles. Dark areas represent the max-solution regions of the over-constrained problem.

For the purpose of illustration, we present a small over-constrained problem. The scene consists of three characters defined by their positions, and by an orientation vector (that denotes the front of the character). VCP frameworks generally provide the *angle* property that specifies the orientation the character should have on the screen (*i.e.*, profile, *etc.*), and the *shot* property that specifies the way a character is shot on the screen.

In Figure 1, a sample scene is displayed with three characters, each of which must be viewed from the front and with a long shot. The regions corresponding to each property are represented (wedges for the *angle* property and circles for the *shot distance*). No region of the space fully satisfies all the constraints. The solution of the Max-CSP are represented as dark areas. The solving process associated to this simple example is provided in section 7.1.

3 The Max-CSP Framework

This section is dedicated to the Max-CSP framework and its related concepts. According to [1,2], a Max-CSP $\mathcal{P}$ is a triplet $\langle V, \mathbf{D}, C \rangle$ where $V = \{x_1, \dots, x_n\}$, $\mathbf{D} = \{D_1, \dots, D_n\}$ and $C = \{c_1, \dots, c_m\}$ are respectively a set of variables, a set of domains assigned to each variable and a set of constraints. For clarity sake, we use vectorial notations where vectors are denoted by boldface characters: the vector $\mathbf{x} = (x_1, \dots, x_n)$ represents the variables and $\mathbf{D}$ is interpreted as the Cartesian product of the variables domains. To define solutions of a Max-CSP, a function s_i is first defined for each constraint c_i as follows: $\forall \mathbf{x} \in \mathbf{D}, s_i(\mathbf{x}) = 1$ if $c_i(\mathbf{x})$ is true and $s_i(\mathbf{x}) = 0$ otherwise. Then, the maximum number of satisfied constraint is denoted by $m_{\mathcal{P}}$ and formally defined as follows:

$$m_{\mathcal{P}} := \max_{\mathbf{x} \in \mathbf{D}} \sum_i s_i(\mathbf{x}). \quad (1)$$

We denote $\text{MaxSol}(\mathcal{P})$ the *solution set* of the numerical Max-CSP. Formally, the following set of solution vectors:

$$\text{MaxSol}(\mathcal{P}) = \{ \mathbf{x} \in \mathbf{D} : \sum_i s_i(\mathbf{x}) = m_{\mathcal{P}} \}. \quad (2)$$

4 Interval Analysis for Numerical CSPs

Numerical CSPs present the additional difficulty of dealing with continuous domains, not exactly representable in practice. As a consequence, it is impossible to enumerate all the values that can take variables in their domains. Interval analysis (*cf.* [3,4]) is a key framework in this context: soundness and completeness properties of real values belonging to some given intervals can be ensured using correctly rounded computations over floating point numbers. We now present basic concepts allowing us to ensure that an interval vector contains no solution (*resp.* only solutions) of an inequality constraint.

4.1 Intervals, Interval Vectors and Interval Arithmetic

An interval $[x]$ with representable bounds is called a *floating-point interval*. It is of the form: $[x] = [\underline{x}, \overline{x}] = \{x \in \mathbb{R} : \underline{x} \leq x \leq \overline{x}\}$ where $\underline{x}, \overline{x} \in \mathbb{F}$. The set of all floating-point intervals is denoted by $\mathbb{IF}$. An interval vector (also called a box) $[\mathbf{x}] = ([x_1], \dots, [x_n])$ represents the Cartesian product of the n intervals $[x_i]$. The set of interval vectors with representable bounds is denoted by $\mathbb{IF}^n$.

Operations and functions over reals are also replaced by an *interval extension* having the *containment property* (see [5]).

Definition 1 (Interval extension [3]). *Given a real function $f: \mathbb{R}^n \rightarrow \mathbb{R}$, an interval extension $[f]: \mathbb{IF}^n \rightarrow \mathbb{IF}$ of f is an interval function verifying*

$$[f]([\mathbf{x}]) \supseteq \{f(\mathbf{x}) : \mathbf{x} \in [\mathbf{x}]\}. \quad (3)$$

The interval extensions of elementary functions can be computed formally thanks to their simplicity, *e.g.*, for $f(x, y) = x + y$ we have $[f]([x], [y]) = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$. To obtain an efficient framework, these elementary interval extensions are used to define an interval arithmetic. For example, the interval extension of $f(x, y) = x + y$ is used to define the interval addition: $[x] + [y] = [\underline{x} + \underline{y}, \overline{x} + \overline{y}]$. Therefrom, an arithmetical expression can be evaluated for interval arguments. The fundamental theorem of interval analysis [5] states that the interval evaluation of an expression gives rise to an interval extension of the corresponding real function: For example

$$[x] + \exp([x] + [y]) \supseteq \{x + \exp(x + y) : x \in [x], y \in [y]\}. \quad (4)$$

4.2 Interval Contractors

Discarding all inconsistent values from a box of variables domains is intractable when the constraints are real ones. There exists many techniques to obtain contractors, like 2B consistency (also called hull consistency) [6,7], box consistency [8] or interval Newton operator [3]. Those methods rely on outer contracting operators that discard values according to a given consistency. These methods are instances of the local consistency framework in artificial intelligence [9].

Definition 2 (Interval contracting operator). *Let c be an n -ary constraint, a contractor for c is a function $\text{Contract}_c: \mathbb{IF}^n \rightarrow \mathbb{IF}^n$ which satisfies*

$$\mathbf{x} \in [\mathbf{x}] \wedge c(\mathbf{x}) \implies \mathbf{x} \in \text{Contract}_c([\mathbf{x}]). \quad (5)$$

Experiments of Section 7 are performed with two different contractors dedicated to inequality constraints $f(\mathbf{x}) \leq 0$. The first is called the *evaluation contractor* and is defined as follows: Given an interval extension $[f]$ of f

$$\text{Contract}_{f(\mathbf{x}) \leq 0}(\mathbf{x}) = \begin{cases} \emptyset & \text{if } [f]([\mathbf{x}]) > 0 \\ [\mathbf{x}] & \text{otherwise.} \end{cases} \quad (6)$$

The second, called the *hull contractor*, is based on the hull consistency, cf. [6,7] for details¹.

Interval contractors can be used to partition a box into smaller boxes where the constraint can be decided. To this end, we introduce the *inflated set difference* as follows: $\text{idiff}([\mathbf{x}], [\mathbf{y}])$ is a set of boxes $\{[\mathbf{d}_1], \dots, [\mathbf{d}_k]\}$ included in $[\mathbf{x}]$ such that $[\mathbf{d}_i] \cap [\mathbf{y}] = \emptyset$ and $[\mathbf{d}_1] \cup \dots \cup [\mathbf{d}_k] \cup [\mathbf{y}]^\oplus = [\mathbf{x}]$, where $[\mathbf{y}]^\oplus$ is the smallest box that strictly contains $[\mathbf{y}]$. The inflated set difference is illustrated in two dimensions in Figure 2. In this way, $\text{idiff}([\mathbf{x}], \text{Contract}_{f(\mathbf{x}) \leq 0}([\mathbf{x}]))$ constructs a set of boxes which are all proved to contain no solution. Following [10,11] this technique can also be used to compute boxes that contain only solutions, applying the contractor to the negation of the constraint: $\text{idiff}([\mathbf{x}], \text{Contract}_{f(\mathbf{x}) > 0}([\mathbf{x}]))$ is a set of boxes that contain only solutions. Let us illustrate this process on a simple example. Consider the CSP $\langle x, [-1, 1], x \leq 0 \rangle$. Then $\text{Contract}_{x \leq 0}([-1, 1]) = [-1, 0]$ and $\text{idiff}([-1, 1], [-1, 0]) = [0^\oplus, 1]$, where $0^\oplus$ is the smallest representable number bigger than 0. The constraint $x \leq 0$ is therefore proved to be false inside $[0^\oplus, 1]$. The domain $[-1, 0^\oplus]$ remains to be processed, and we look for solutions therein by computing $\text{Contract}_{x > 0}([-1, 0^\oplus]) = [0, 0^\oplus]$. We use again $\text{idiff}([-1, 0^\oplus], [0, 0^\oplus]) = [-1, 0^\ominus]$ (where $0^\ominus$ is the largest representable number lower than 0) to infer that $[-1, 0^\ominus]$ contains only solutions. Finally, the constraint cannot be decided for the values of the interval $[0^\ominus, 0^\oplus]$.

5 The Algorithm

The main idea of the algorithm is to attach to each box the set of constraints that were proved to be satisfied for all vectors of this box ($\mathcal{S}$) and the set of constraints that have not been decided yet ($\mathcal{U}$). Then, using inner or outer contractions and bisections, we can split the box into smaller boxes where more constraints are decided. We thus propose the definition of SU-boxes representing a triplet consisting of a box $[\mathbf{x}]$ and the two sets of constraints described above. The formal definition of a SU-box is given in the first subsection and the usage of outer and inner contractions is described in the next three subsections. Finally the branch and bound algorithm for numerical Max-CSP is given in the last subsection.

5.1 SU-Boxes

The main idea of our algorithm is to associate to a box $[\mathbf{x}]$ the sets of constraints $\mathcal{S}$ and $\mathcal{U}$ which respectively contain the constraints proved to be satisfied for all vectors in $[\mathbf{x}]$ and the constraints which remain to be treated. Such a box, altogether with the satisfaction information of each constraints is called a *SU-box*² and is denoted by $\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle$. The following definition naturally links a SU-box with a CSP:

¹ A contractor based on box consistency could also be used but their optimality between hull and box consistency is problem dependent.

² *SU-boxes* have been introduced in [12] in a slightly but equivalent way.

Algorithm 1. Branch and Bound Algorithm for MAX-CSP

```

Input:  $\mathcal{P} = \langle \mathbf{x}, \mathcal{C}, [\mathbf{x}] \rangle, \epsilon$ 
Output:  $([\underline{m}, \overline{m}], \mathcal{M}, \mathcal{B})$ 
1  $\mathcal{L} \leftarrow \{ \langle [\mathbf{x}], \emptyset, \mathcal{U} \rangle \};$  /* SU-boxes to be processed */
2  $\mathcal{E} \leftarrow \{ \};$  /* Epsilon SU-boxes */
3  $\mathcal{D} \leftarrow \{ \};$  /* SU-boxes where every constraint is decided */
4  $\underline{m} \leftarrow \text{MultiStartLocalSearch}(\mathcal{P}, [\mathbf{x}])^3;$  /* Lower bound on max */
5 while (  $\neg \text{empty}(\mathcal{L})$  ) do
6    $\langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \leftarrow \text{extract}(\mathcal{L})^4;$ 
7   if (  $\text{wid}([\mathbf{x}]) < \epsilon$  ) then
8      $\mathcal{E} \leftarrow \mathcal{E} \cup \{ \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \};$ 
9   else
10     $\mathcal{T} \leftarrow \{ \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \};$ 
11    foreach (  $c \in \mathcal{U}$  ) do
12       $[\mathbf{y}] \leftarrow \text{Contract}_c([\mathbf{x}]);$ 
13       $\mathcal{T} \leftarrow \text{InferLSU}_{\text{outer}}(\mathcal{T}, [\mathbf{y}], c);$ 
14       $\mathcal{T} \leftarrow \{ \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{T} : (\# \mathcal{S}) + (\# \mathcal{U}) \geq \underline{m} \};$ 
15       $[\mathbf{y}] \leftarrow \text{Contract}_{\neg c}([\mathbf{x}]);$ 
16       $\mathcal{T} \leftarrow \text{InferLSU}_{\text{inner}}(\mathcal{T}, [\mathbf{y}], c);$ 
17    end
18    foreach (  $\langle [\mathbf{x}'], \mathcal{S}', \mathcal{U}' \rangle \in \mathcal{T}$  ) do
19      if (  $(\# \mathcal{S}') > \underline{m}$  ) then  $\underline{m} \leftarrow (\# \mathcal{S}')$ ;
20      if (  $(\# \mathcal{U}') = 0$  ) then
21         $\mathcal{D} \leftarrow \mathcal{D} \cup \{ \langle [\mathbf{x}'], \mathcal{S}', \emptyset \rangle \};$ 
22      else
23         $\{ [\mathbf{x}^1], [\mathbf{x}^2] \} \leftarrow \text{Bisect}([\mathbf{x}'])^5;$ 
24         $\mathcal{L} \leftarrow \mathcal{L} \cup \{ \langle [\mathbf{x}^1], \mathcal{S}', \mathcal{U}' \rangle, \langle [\mathbf{x}^2], \mathcal{S}', \mathcal{U}' \rangle \};$ 
25      end
26    end
27  end
28 end
29  $\overline{m} \leftarrow \max\{ (\# \mathcal{S}) + (\# \mathcal{U}) : \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{D} \cup \mathcal{E} \};$ 
30 if (  $\underline{m} = \overline{m}$  ) then
31    $\mathcal{M} \leftarrow \{ \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{D} : (\# \mathcal{S}) \geq \underline{m} \};$ 
32    $\mathcal{B} \leftarrow \{ \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{E} : (\# \mathcal{S}) + (\# \mathcal{U}) \geq \underline{m} \};$ 
33 else
34    $\mathcal{M} \leftarrow \emptyset;$ 
35    $\mathcal{B} \leftarrow \{ \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{D} \cup \mathcal{E} : (\# \mathcal{S}) + (\# \mathcal{U}) \geq \underline{m} \};$ 
36 return  $([\underline{m}, \overline{m}], \mathcal{M}, \mathcal{B});$ 

```

⁶ Constraints satisfaction is considered on uniformly distributed random points in $[\mathbf{x}]$, the lower bound $\underline{m}$ being updated accordingly.

⁷ SU-boxes are stored in $\mathcal{L}$ according to $(\# \mathcal{S}) + (\# \mathcal{U})$, most interesting (*i.e.*, highest $(\# \mathcal{S}) + (\# \mathcal{U})$) being extracted first.

⁸ Bisect splits the box $[\mathbf{x}']$ into two equally sized smaller boxes $[\mathbf{x}^1]$ and $[\mathbf{x}^2]$.

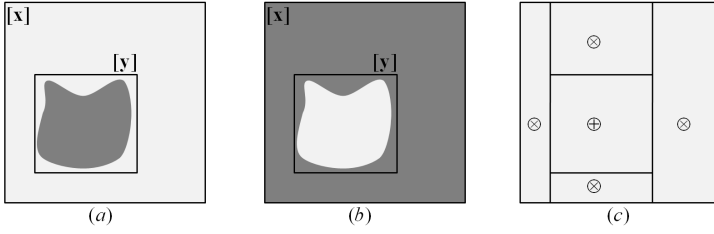


Fig. 2. Diagrams (a) and (b) show the constraint c (satisfied in dark gray regions and unsatisfied in light gray regions), the initial SU-box $\langle [x], \mathcal{S}, \mathcal{U} \rangle$ and the contracted box $[y]$ for two different situations: (a) outer contraction and (b) inner contraction for constraint c . Diagram (c): Five new SU-boxes inferred from the contraction of $[x]$ to $[y]$. The contraction does not provide any information on the SU-box $\oplus = \langle [x] \cap [y], \mathcal{S}, \mathcal{U} \rangle$, hence its sets of constraints remain unchanged. The SU-boxes $\otimes = \langle [x]', \mathcal{S}', \mathcal{U}' \rangle$ are proved to contain no solution of c (if outer pruning was applied) or only solutions (if inner pruning was applied), their sets of constraint can thus be updated: $\mathcal{U}' = \mathcal{U} \setminus \{c\}$, and, $\mathcal{S}' = \mathcal{S}$ for outer pruning, while $\mathcal{S}' = \mathcal{S} \cup \{c\}$ and $\mathcal{U}' = \mathcal{U}$ for inner pruning.

Definition 3. Let $\mathcal{P}$ be a CSP whose set of constraints is $\mathcal{C}$. Then, a SU-box $\langle [x], \mathcal{S}, \mathcal{U} \rangle$ is consistent with $\mathcal{P}$ if and only if the three following conditions hold:

1. $(\mathcal{S} \cup \mathcal{U}) \subseteq \mathcal{C}$ and $(\mathcal{S} \cap \mathcal{U}) = \emptyset$
2. $\forall \mathbf{x} \in [x], \forall c \in \mathcal{S}, c(\mathbf{x})$ (the constraints of $\mathcal{S}$ are true everywhere in $[x]$)
3. $\forall \mathbf{x} \in [x], \forall c \in (\mathcal{C} \setminus (\mathcal{S} \cup \mathcal{U})), \neg c(\mathbf{x})$ (the constraints of $\mathcal{C}$ which are neither in $\mathcal{S}$ nor in $\mathcal{U}$ are false everywhere in $[x]$).

A SU-box $\langle [x], \mathcal{S}, \mathcal{U} \rangle$ will be denoted by $\langle \mathbf{x} \rangle$, without any explicit definition when there is no confusion. Given a SU-box $\langle \mathbf{x} \rangle := \langle [x], \mathcal{S}, \mathcal{U} \rangle$, $[x]$ is denoted by $\text{box}(\langle \mathbf{x} \rangle)$ and called the *domain* of the SU-box.

5.2 The Function $\text{InferSU}_{\text{outer}}$

Let us consider a SU-box $\langle [x], \mathcal{S}, \mathcal{U} \rangle$, a box $[y] \subseteq [x]$ and a constraint c , where the box $[y]$ is obtained computing $[y] = \text{Contract}_c([x])$. Therefore, every box of $\text{idiff}([x], [y])$ does not contain any solution of c . This is illustrated by Figure 2. The SU-boxes formed using these boxes can thus be improved discarding c from their set of constraint $\mathcal{U}$. This is formalized by the following definition:

Definition 4. The function $\text{InferSU}_{\text{outer}}(\langle [x], \mathcal{S}, \mathcal{U} \rangle, [y], c)$ returns the following set of SU-boxes:

$$\{ \langle [x'], \mathcal{S}, \mathcal{U}' \rangle : [x'] \in \text{idiff}([x], [y]) \} \cup \{ \langle [y]^\oplus \cap [x], \mathcal{S}, \mathcal{U} \rangle \}, \quad (7)$$

where $\mathcal{U}' = \mathcal{U} \setminus \{c\}$.

Here, the box $[x]$ is partitioned to $\text{idiff}([x], [y])$ and $[y]^\oplus \cap [x]$. Each of these boxes is used to form new SU-boxes, with a reduced set of constraints $\mathcal{U}'$ for the

SU-boxes coming from $\text{idiff}([x], [y])$. This is formalized by the following proposition, which obviously holds:

Proposition 1. *Let $\langle x \rangle := \langle [x], \mathcal{S}, \mathcal{U} \rangle$ be a SU-box consistent with a CSP $\mathcal{P}$, $c \in \mathcal{U}$ and $[y]$ satisfying $x \in ([x] \setminus [y]) \Rightarrow \neg c(x)$. Define $\mathcal{T} := \text{InferSU}_{\text{outer}}(\langle x \rangle, [y], c)$. Then the SU-boxes of $\mathcal{T}$ are all consistent with $\mathcal{P}$. Furthermore, $\cup \{\text{box}(\langle x' \rangle) : \langle x' \rangle \in \mathcal{T}\}$ is equal to $[x]$.*

5.3 The Function $\text{InferSU}_{\text{inner}}$

This function is the same as $\text{InferSU}_{\text{outer}}$, but dealing with inner contraction. As a consequence, the only difference is that, instead of just discarding the constraint c from $\mathcal{U}$ for the SU-boxes outside $[y]$, this constraint is stored in $\mathcal{S}$. Figure 2 also illustrates these computations. The following definition and proposition mirror Definition 4 and Proposition 1.

Definition 5. *The function $\text{InferSU}_{\text{inner}}(\langle [x], \mathcal{S}, \mathcal{U} \rangle, [y], c)$ returns the following set of SU-boxes:*

$$\{\langle [x'], \mathcal{S}', \mathcal{U}' \rangle : [x'] \in \text{idiff}([x], [y])\} \cup \{\langle [y]^\oplus \cap [x], \mathcal{S}, \mathcal{U} \rangle\}, \quad (8)$$

where $\mathcal{S}' = \mathcal{S} \cup \{c\}$ and $\mathcal{U}' = \mathcal{U} \setminus \{c\}$.

Proposition 2. *Let $\langle x \rangle := \langle [x], \mathcal{S}, \mathcal{U} \rangle$ be a SU-box consistent with a CSP $\mathcal{P}$, $c \in \mathcal{U}$ and $[y]$ satisfying $x \in ([x] \setminus [y]) \Rightarrow c(x)$. Define $\mathcal{T} := \text{InferSU}_{\text{inner}}(\langle x \rangle, [y], c)$. Then the SU-boxes of $\mathcal{T}$ are all consistent with $\mathcal{P}$. Furthermore, $\cup \{\text{box}(\langle x' \rangle) : \langle x' \rangle \in \mathcal{T}\}$ is equal to $[x]$.*

5.4 The Function $\text{InferLSU}_{\text{type}}$

Finally, the functions $\text{InferSU}_{\text{outer}}$ and $\text{InferSU}_{\text{inner}}$ are naturally applied to a set of SU-boxes $\mathcal{T}$ in the following way:

$$\text{InferLSU}_{\text{type}}(\mathcal{T}, [y], c) := \bigcup_{\langle x \rangle \in \mathcal{T}} \text{InferSU}_{\text{type}}(\langle x \rangle, [y], c), \quad (9)$$

where *type* is either *outer* or *inner*. This process is illustrated by Figure 3. As a direct consequence of propositions 1 and 2, the following proposition holds:

Proposition 3. *Let $\mathcal{T}$ be a set of SU-boxes, each of them being consistent with a CSP $\mathcal{P}$, and $[y]$ satisfying $x \in ([x] \setminus [y]) \Rightarrow \neg c(x)$ (respectively $x \in ([x] \setminus [y]) \Rightarrow c(x)$). Define $\mathcal{T}' := \text{InferLSU}_{\text{outer}}(\langle x \rangle, [y], c)$ (respectively $\mathcal{T}' := \text{InferLSU}_{\text{inner}}(\langle x \rangle, [y], c)$). Then the SU-boxes of $\mathcal{T}'$ are all consistent with $\mathcal{P}$. Furthermore,*

$$\cup \{\text{box}(\langle x' \rangle) : \langle x' \rangle \in \mathcal{T}'\} = \cup \{\text{box}(\langle x' \rangle) : \langle x' \rangle \in \mathcal{T}\}. \quad (10)$$

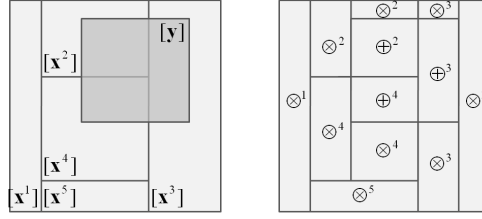


Fig. 3. Left hand side: Boxes obtained from Figure 2. The box $[y]$ represents the contraction (inner or outer) of a box. Right hand side: The process illustrated by Figure 2 is repeated for each box $[x^k]$, leading to twelve new SU-boxes, among which $\otimes^k$ have their sets of constraints updated.

5.5 The Branch and Bound Algorithm

Algorithm 1 uses three sets of SU-boxes $\mathcal{L}$ (SU-boxes to be proceeded), $\mathcal{D}$ (SU-boxes where all constraints are decided, *i.e.*, where $(\#\mathcal{U}) = 0$) and $\mathcal{E}$ (SU-boxes that will not be processed anymore because considered as too small). The mainline of the while-loop is to maintain the following property:

$$\text{MaxSol}(\mathcal{P}) \subseteq \bigcup \{ \text{box}(\langle \mathbf{x} \rangle) : \langle \mathbf{x} \rangle \in \mathcal{L} \cup \mathcal{D} \cup \mathcal{E} \}, \quad (11)$$

while using inner and outer contractions and the function $\text{InferLSU}_{\text{type}}$ to decide more and more constraints (*cf.* lines 11–17), and hence moving SU-boxes from $\mathcal{L}$ to $\mathcal{D}$ (*cf.* Line 21). Note that a lower bound $\underline{m}$ for $m_{\mathcal{P}}$ is updated (*cf.* Line 19) and used to drop SU-boxes that are proved to satisfy less constraints than $\underline{m}$ (*cf.* Line 14). When $\mathcal{L}$ is finally empty, (11) still holds and hence:

$$\text{MaxSol}(\mathcal{P}) \subseteq \bigcup \{ \text{box}(\langle \mathbf{x} \rangle) : \langle \mathbf{x} \rangle \in \mathcal{D} \cup \mathcal{E} \}. \quad (12)$$

Eventually, the sets of SU-boxes $\mathcal{D}$ and $\mathcal{E}$ are used to obtain new sets of SU-boxes $\mathcal{M}$ and $\mathcal{B}$, which describe the max-solution set (*i.e.*, both the inner and outer approximation of the solution set of the NCSP), and an interval $[\underline{m}, \overline{m}]$ for $m_{\mathcal{P}}$. We already have a lower bound $\underline{m}$ on $m_{\mathcal{P}}$. Now, as (11) holds, an upper bound can be computed in the following way:

$$\overline{m} = \max \{ (\#\mathcal{S}) + (\#\mathcal{U}) : \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{D} \cup \mathcal{E} \}. \quad (13)$$

Two cases can then happen, which are handled at lines 29–36:

1. If $\underline{m} = \overline{m}$ then the algorithm has succeeded in computing $m_{\mathcal{P}}$, which is equal to $\underline{m} = \overline{m}$. In this case the domains of the SU-box of $\mathcal{D}$ are proved to contain only max-solutions of $\mathcal{P}$:

$$\forall \langle \mathbf{x} \rangle \in \mathcal{D}, \text{box}(\langle \mathbf{x} \rangle) \subseteq \text{MaxSol}(\mathcal{P}). \quad (14)$$

Furthermore, the max-solutions which are not in $\mathcal{D}$ obviously have to be in the SU-box of:

$$\mathcal{B} = \{ \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{E} : (\#\mathcal{S}) + (\#\mathcal{U}) \geq \underline{m} \}. \quad (15)$$

As a consequence, the algorithm outputs both an inner and an outer approximation of $\text{MaxSol}(\mathcal{P})$.

2. If $\underline{m} < \overline{m}$, $m_{\mathcal{P}}$ is still proved to belong to the interval $[\underline{m}, \overline{m}]$. However, the algorithm has not been able to compute any solution of the Max-CSP $\mathcal{P}$, hence $\mathcal{M} = \emptyset$. In this case, all the max-solutions are obviously in the domains of the SU-boxes of:

$$\mathcal{B} = \{ \langle [\mathbf{x}], \mathcal{S}, \mathcal{U} \rangle \in \mathcal{D} \cup \mathcal{E} : (\#\mathcal{S}) + (\#\mathcal{U}) \geq \underline{m} \}. \quad (16)$$

As a consequence of Proposition 3, constraints are correctly removed $\mathcal{U}$ and possibly added to $\mathcal{S}$ while no solution is lost. This ensures the correctness of the algorithm (a formal proof is not presented here due to lack of space).

Remark 1. The worst case complexity of the foreach-loop at lines 11–17 is exponential with respect to the number of constraints in $\mathcal{U}$. Indeed, the worst case complexity of the loop running through lines 11–17, is $(2n)^q$, where n is the dimension and q is the number of constraints. This upper bound is very pessimistic, and not met in any typical situation. Nonetheless, we expect a possibly high number of boxes generated during the inflated set difference process when the number of constraints is important. In order to bypass this complexity issue, that could lead the algorithm to generate too many boxes within the outer and inner contractions steps, we have chosen to switch between different contracting operators. As soon as the number of constraints becomes greater than a threshold (t_c), contractions are performed with the *evaluation contractor* instead of the *hull contractor*. This will reduce creation of boxes within the inner and outer contracting steps, thus preventing the algorithm from falling into a complexity pit. Experiments on relatively high number of constraints (cf. Section 7.2) have shown that t_c should be set between 5 and 10 in order to speed up the solving process.

6 Related Work

A large literature is related to Max-CSP frameworks and solving techniques. Most formulate the problem as a combinatorial optimisation one by minimizing the number of violated constraints. Approaches are then shared between complete and incomplete techniques depending on the optimality criterion and the size of the problem. For large problems, heuristic-based techniques such as Min-conflict [13] and Random-walk [14] have proved to be efficient as well as meta-heuristic approaches (Tabu, GRASP, ACO). However all are restricted to the study of binary or discrete domains, and their extension to continuous values requires to redefine the primitive operations (*e.g.*, move, evaluate, compare, *etc.*). Though some contributions have explored such extensions for local optimisation problems (cf. *e.g.*, Continuous-GRASP[15]), these are not dedicated to manage numerical Max-CSPs. More general techniques built upon classical optimization schemes such as Genetic Algorithms, Simulated Annealing or Generalized Gradient can be employed to express and solve Max-CSPs, but do not guarantee the optimality of the solution, nor compute a paving of the search space.

Regardless the fact that applications of CSP techniques in continuous domains are less numerous than in discrete domains, both present the same need for

Max-CSP models. Interestingly very few contributions have explored this class of problems when optimality is required. Here, we report the contribution of Jaulin *et al.* [16] that looks at a problem of parameter estimation with bounded error measurements. The paper considers the number of error measurements as known (*i.e.*, $m_{\mathcal{P}}$ is given beforehand) and the technique computes a Max-CSP approximation over a continuous search space by following an incremental process that runs the solving for each possible value of $m_{\mathcal{P}}$ and decides which is the correct one. At each step the technique relies on an evaluation-bisection.

7 Experiments

7.1 Camera Control Problem

This subsection is dedicated to the virtual camera placement example presented in section 2. Let us recall briefly that this motivating example involves 2 variables and 6 constraints expressed as dot products and Euclidian distances in 2D. The corresponding paving is shown in Figure 4 while results are presented in Table 1 within the search space $\{[-30; 30], [1; 30]\}$ with $\epsilon = 0.01$.

Although Christie *et al.* [18] do not compute an outer and an inner approximation of the solution set of the Max-CSP (they only aim at computing an inner approximation), their approach is similar enough to compare timings obtained on the VCP example.

An advantage of our algorithm is that each SU-box contains the set of constraints that are fulfilled within this box. We are thus able to give the user

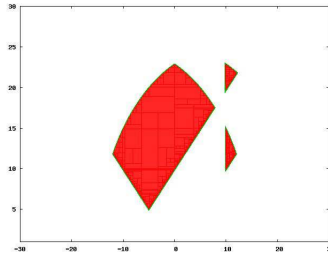


Fig. 4. Pavings generated by our algorithm on the VCP problem. Red boxes represent the max solution set (*cf.* Figure 1), while green boxes illustrate boundary boxes encompassing the solution set.

Table 1. Approximated timings thanks to [17] for the VCP example between our approach (PentiumM750 laptop, 1Go RAM) and Christie *et al.* [18](Pentium T7600 2.33Ghz 2Go RAM)

Method	ϵ	T_{original} (seconds)	T_{approx} (seconds)
Max-NCSP	0.01	3.2	3.2
Christie <i>et al.</i>	0.01	4.8	≈ 15

information on which constraint is fulfilled in each part of the search space, allowing him to choose between the different solutions of the over-constrained problem. Indeed Figure 4 shows that the search space is divided into three connected components each encompassing a different set of equivalent solutions. An automatic connected component identification algorithm (*cf.* [19]) could be processed after solving a problem with our algorithm in order to present the user the sets of equivalent SU-boxes, allowing him to choose which constraint he wants to be relaxed into the original problem.

7.2 The Facility Location Problem

This example is based on a well-known problem in Operational Research: the facility location problem [20]. The problem is defined by a number of customer locations that must be delivered by some facilities. We consider that a customer c can be served by a facility f if the distance between f and c is smaller than a delivery distance d . The problem consists in finding the smallest number of facilities such that each customer can be served by at least one facility.

Although the facility location problem is not strictly speaking a Max-CSP, we can use the Max-CSP framework to build a greedy algorithm to find an estimation of the number of facilities. The idea consists in computing the Max-CSP areas of the problem which corresponds to the areas of the search space that maximises customers delivery. Instead of computing the whole MaxSol paving, we stop the algorithm as soon as we have found a MaxSol SU-box $\langle [\mathbf{x}_m], \mathcal{S}_m, \mathcal{U}_m \rangle$. We then remove from the original problem all the constraints of $\mathcal{S}_m$ and re-run the algorithm again. We stop when the problem is empty, *i.e.*, when all the constraints have been solved. The numbers of runs of the algorithm then correspond to a number of facilities that solves the problem.

In order to test our greedy algorithm, we have implemented a basic OR technique to solve the same problem (*i.e.*, discretisation of the search space and resolution of a linear integer programming problem) in Wolfram Mathematica⁶.

In 3D this problem boils down to modeling each customer by a random 3D position in the search space $\{[-5; 5], [-5; 5], [-5; 5]\}$ and trying to position some facilities such that the distance between the facility and the customer is inferior to $d = 2$. Results are shown in Table 2. The solution found by our algorithm involves 3×39 variables (3 coordinates per computed facility). Each iteration of our algorithm removes on average 13 constraints (ranging from 33 to 1 constraint removed at each iteration) of the 500 original ones.

Although, this basic OR technique seems less efficient than our greedy algorithm, this is at least encouraging and thorough experiments need to be carried out to compare our greedy approach with more specific OR techniques.

7.3 Parameter Estimation with Bounded Error Measurements

This experiment is based on a parameter estimation problem presented in [16]. The problem consists in determining unknown parameters $\mathbf{p} = (\mathbf{p}_1, \mathbf{p}_2) \in$

⁶ The corresponding Mathematica notebook is available for download at <http://www.goldztejn.com/src/FacilityLocation3D.nb>.

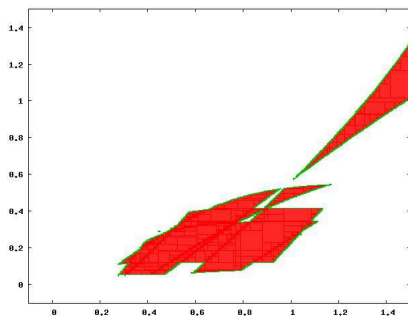


Fig. 5. Paving generated for 3 possible outliers. Red boxes represent inner approximation (*i.e.*, satisfying at least 7 of the 10 constraints) while green boxes illustrate boundary boxes encompassing the solution set.

Table 2. Timings for the facility location problem for 500 customers randomly generated in $\{[-5; 5], [-5; 5], [-5; 5]\}$ on a PentiumM750 laptop, 1Go RAM

Method	Time (seconds)	Number of facilities
Greedy Num. Max-CSP	69.25	39
Basic OR technique	≈ 140	45

Table 3. Original and approximate timings obtained from a Pentium M750 laptop processor (Max-NCSP) and from a 486 DX4-100 processor (Jaulin *et al.*) thanks to [17]

Method	ϵ bisection	T_{original} (seconds)	T_{approx} (seconds)
Max-NCSP	0.005	0.46	0.46
Jaulin <i>et al.</i>	0.005	≈ 180	≈ 1.9

$([-0.1, 1.5], [-0.1, 1.5])$ of a physical model thanks to a set of experimental data obtained from a system.

Here the model used is a two-parameter estimation problem taken from Jaulin and Walter [21] which is a 2D extension of a problem presented by Milanese and Vicino [22]. In fact, the solution set of the Max-CSP does not contain the true parameters values. Indeed, the maximum number of satisfied constraints is 9, while there are 2 outliers. This means that an outlier is compatible with the other error measurements. Hence, the Max-CSP cannot directly help solving this problem. From an application point of view, it is possible to provide an upper bound on the number of outliers (this assumption is realistic since manufacturers are able to ensure a percentage of maximum failures of their probes). Algorithm 1 is easily modified to output the set of vectors which satisfy at least a fixed number of constraints instead of the solution set of the Max-CSP.

In order to compare our approach with [16], we set a maximum of 3 outliers, Figure 5 shows the corresponding paving. Although Jaulin *et al.* do not compute an inner and outer approximation of the solution set, Table 3 compares our results. Approximate timings have been computed using [17] in order to compare

the results that were obtained on totally different computers. Time comparisons are presented to give an idea of the gain offered by our algorithm, which is approximately 4 times more efficient (although no exact comparison of the pavings computed by the two algorithms is possible, the pavings presented in [16] are clearly less accurate than the pavings computed by our approach). Moreover, we want to emphasize on the fact that our algorithm is much more scalable since Jaulin *et al.* have to re-run their algorithm for each possible number of outliers whereas we only need one run to compute the solution set of the Max-CSP. Moreover our approach fully characterizes this solution set w.r.t. the satisfied constraints of the algorithm.

8 Conclusion

In this paper we have presented a branch and bound algorithm for numerical Max-CSP that computes both an inner and an outer approximation of the solution set of a numerical Max-CSP. The algorithm is based on the notion of SU-box that stores the sets of constraints *Satisfied* or *Unknown* for each box of the search space. Our method can address any kind of CSP: well-constrained, under-constrained and over-constrained. The algorithm outputs a set of boxes that encompass the inner and outer approximation of the solution set of a numerical Max-CSP. For each box, it computes the sets of constraints that are satisfied, thus giving the user additional information on the classes of solutions of the problem. An algorithm of connected component identification could then be applied on the solution SU-boxes in order to create regions of the search space that share similar characteristics (*i.e.*, that satisfies the same constraints). Users could thus benefit from this feature when modeling conceptual design problems for example.

Our algorithm could be extended to manage hierarchical constraints in a predicate *locally-better* way [23]. Hierarchical constraints could be modeled as different “layers” of numerical Max-CSP problems that could be solved in sequence, starting from the top-priority constraints down to the lowest ones and maximizing constraints satisfaction of each “layer” of hierarchical constraints.

Acknowledgments

The authors are grateful to Frédéric Saubion and to the anonymous referees whose comments significantly improved the paper.

References

1. Petit, T., Régim, J.C., Bessière, C.: Range-based algorithm for max-csp. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 280–294. Springer, Heidelberg (2002)
2. de Givry, S., Larrosa, J., Meseguer, P., Schiex, T.: Solving max-sat as weighted csp. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 363–376. Springer, Heidelberg (2003)

3. Neumaier, A.: *Interval Methods for Systems of Equations* (1990)
4. Hayes, B.: A Lucid Interval. *American Scientist* 91(6), 484–488 (2003)
5. Moore, R.E.: *Interval Analysis*. Prentice-Hall, Englewood Cliffs (1966)
6. Lhomme, O.: Consistency techniques for numeric cps. In: *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 232–238 (1993)
7. Benhamou, F., Older, W.J.: Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming* 32(1), 1–24 (1997)
8. Benhamou, F., McAllester, D., van Hentenryck, P.: Clp(intervals) revisited. In: *ILPS 1994*, pp. 124–138. MIT Press, Cambridge (1994)
9. Mackworth, A.K.: Consistency in networks of relations. *AI* 8(1), 99–118 (1977)
10. Collavizza, H., Delobel, F., Rueher, M.: Extending Consistent Domains of Numeric CSP. In: *Proceedings of IJCAI 1999* (1999)
11. Benhamou, F., Goualard, F., Languenou, E., Christie, M.: Interval Constraint Solving for Camera Control and Motion Planning. *ACM Trans. Comput. Logic* 5(4), 732–767 (2004)
12. Normand, J.M.: *Placement de caméra en environnements virtuels*. PhD thesis, Université de Nantes (2008)
13. Minton, S., Johnston, M.D., Philips, A.B., Laird, P.: Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *AI* 58(1-3), 161–205 (1992)
14. Wallace, R.J.: Analysis of heuristic methods for partial constraint satisfaction problems. In: *Principles and Practice of Constraint Programming*, pp. 482–496 (1996)
15. Hirsch, M.J., Meneses, C.N., Pardalos, P.M., Resende, M.G.C.: Global optimization by continuous grasp. *Optimization Letters* 1, 201–212 (2007)
16. Jaulin, L., Walter, E.: Guaranteed tuning, with application to robust control and motion planning. *Automatica* 32(8), 1217–1221 (1996)
17. Dongarra, J.: Performance of various computers using standard linear equations software. Technical Report CS-89-85, University of Tennessee (2007)
18. Christie, M., Normand, J.M., Truchet, C.: Computing inner approximations of numerical maxcsp. In: *IntCP 2006* (2006)
19. Delanoue, N., Jaulin, L., Cottenceau, B.: Counting the number of connected components of a set and its application to robotics. In: Dongarra, J., Madsen, K., Waśniewski, J. (eds.) *PARA 2004*. LNCS, vol. 3732, pp. 93–101. Springer, Heidelberg (2006)
20. Drezner, Z., Hamacher, H. (eds.): *Facility Location. Applications and Theory*. Springer, New-York (2002)
21. Jaulin, L., Walter, E.: Guaranteed nonlinear parameter estimation from bounded-error data via interval analysis. *Math. Comput. Simul.* 35(2), 123–137 (1993)
22. Milanese, M., Vicino, A.: Estimation theory for nonlinear models and set membership uncertainty. *Automatica* 27(2), 403–408 (1991)
23. Borning, A., Freeman-Benson, B., Wilson, M.: Constraint hierarchies. *Lisp Symb. Comput.* 5(3), 223–270 (1992)

A Geometric Constraint over k -Dimensional Objects and Shapes Subject to Business Rules

Mats Carlsson¹, Nicolas Beldiceanu², and Julien Martin³

¹ SICS, P.O. Box 1263, SE-164 29 Kista, Sweden
Mats.Carlsson@sics.se

² École des Mines de Nantes, LINA UMR CNRS 6241, FR-44307 Nantes, France
Nicolas.Beldiceanu@emn.fr

³ INRIA Rocquencourt, BP 105, FR-78153 Le Chesnay Cedex, France
Julien.Martin@inria.fr

Abstract. This paper presents a global constraint that enforces rules written in a language based on arithmetic and first-order logic to hold among a set of objects. In a first step, the rules are rewritten to Quantifier-Free Presburger Arithmetic (QFPA) formulas. Secondly, such formulas are compiled to generators of k -dimensional forbidden sets. Such generators are a generalization of the indexicals of cc(FD). Finally, the forbidden sets generated by such indexicals are aggregated by a sweep-based algorithm and used for filtering.

The business rules allow to express a great variety of packing and placement constraints, while admitting effective filtering of the domain variables of the k -dimensional object, without the need to use spatial data structures.

1 Introduction

This paper extends a global constraint $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$ for handling the location in space of k -dimensional objects $\mathcal{O}$ ($k \in \mathbb{N}^+$), each of which taking a shape among a set of shapes $\mathcal{S}$, subject to rules $\mathcal{R}$ in a language based on arithmetic and first-order logic. This language can also be seen as a natural target constraint of the Rules2CP modeling language [1].

In order to model directly a lot of side constraints, which always show up in the context of real-life applications, many global constraints have traditionally been extended with extra options or arguments. This is why, in a closely related area, the *diffn* constraint [2] of CHIP provides, beside non-overlapping, a variety of other geometrical constraints (in fact more than 10 side constraints). Even if this makes sense when one wants to efficiently solve specific real-life applications, this proliferation of arguments and options has two major drawbacks:

- Having a lot of ad-hoc side constraints is too specific and can sometimes be quite frustrating since it does not allow to express a variant of an existing side constraint.
- Designing a filtering algorithm for each side constraint independently is not enough and managing the interaction of several side constraints becomes more and more challenging as the number and variety of side constraints increase.

The approach presented in this paper addresses these two issues in the following way:

- Firstly, having a rule language for expressing side constraints is obviously more flexible than having a large set of predefined side constraints.
- Secondly, as we will see later on, our filtering algorithms allow to directly take into account the interaction between all rules.

In $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$, each shape from $\mathcal{S}$ is defined as a finite set of shifted boxes, where each shifted box is described by a box in a k -dimensional space at the given offset with the given sizes. More precisely a *shifted box* $s \in \mathcal{S}$ is an entity defined by its shape id $s.sid$, shift offset $s.t[d]$, $1 \leq d \leq k$, and sizes $s.l[d]$ (where $s.l[d] > 0$ and $1 \leq d \leq k$). All attributes of a shifted box are integer values. A *shape* is a collection of shifted boxes all sharing the same shape id.

Each object $o \in \mathcal{O}$ is an entity defined by its unique object id $o.oid$ (an integer), shape id $o.sid$ (an integer if the object has a fixed shape, or a domain variable for *polymorphic* objects, which have alternative shapes), and origin $o.x[d]$, $1 \leq d \leq k$ (integers, or domain variables that do not occur anywhere else in the constraint).¹ Objects and shifted boxes may also have additional, integer (but see also Section 6) attributes, such as weight, customer, or fragility, used by the rules.

Each rule in $\mathcal{R}$ is a first-order logical formula over the attributes of objects and shifted boxes. From the point of view of domain filtering, the main contribution of this paper is that multi-dimensional forbidden sets can be automatically derived from such formulas and used by the sweep-based algorithm of *geost* [3].² This contrasts with the previous version of *geost*, where an ad-hoc algorithm computing the multi-dimensional forbidden sets had to be worked out for each side constraint. $\mathcal{R}$ may also contain macros, providing abbreviations for expressions occurring in formulas or in other macros.

The rule language. The language that makes up the rules to be enforced by the *geost* constraint is based on first-order logic with arithmetic, as well as several features including macros, bounded quantifiers, folding and aggregation operators. We will show how all but a core fragment of the language can be eliminated by equivalence-preserving rewriting. The remaining fragment is a subset of Quantifier-Free Presburger Arithmetic (QFPA), which has a very simple semantics and, as we also will show, is amenable to efficient compilation.

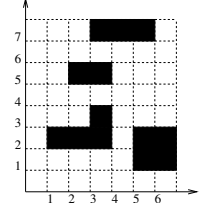
Constraint satisfaction problems using quantified formulas (QCSP) have for instance been studied by Benedetti et al. [4], mostly in the context of modeling games. QCSP does not provide disjunction but actively uses quantifiers in the evaluation, whereas we eliminate all quantifiers in the process of rewriting to QFPA.

Example 1. This running example will be used to illustrate the way we compile rules to code used by the sweep-based algorithm [3] for filtering the nonground attributes of each object. Suppose that we have five objects o_1, o_2, o_3, o_4 and o_5 such that:

¹ A *domain variable* v is a variable ranging over a finite set of integers denoted by $\text{dom}(v)$; $\underline{v}$ and $\overline{v}$ denote respectively the minimum and maximum possible values for v .

² The sweep-based algorithm performs recursive traversals of the placement space for each coordinate increasing as well as decreasing lexicographic order and skips unfeasible points that are located in a multi-dimensional forbidden set.

- o_1, o_2 and o_4 are rectangles fixed at $(1, 2)$, $(3, 3)$ and $(3, 7)$ of respective size 3×1 , 1×1 and 3×1 .
- The rectangle o_3 is fixed at $(2, 5)$ but not its shape variable s_3 , which can take values corresponding to size 1×2 or 2×1 . We will denote by ℓ_{31} resp. ℓ_{32} the length resp. height of o_3 .
- The coordinates of the non-fixed square o_5 of size 2×2 correspond to the two variables $x_{51} \in [1, 9]$ and $x_{52} \in [1, 6]$.
- o_2, o_4 and o_5 have the additional attribute *type* with value 1 whereas o_1 and o_3 have *type* with value 2.
- Two rules must be obeyed; see Fig. 1:
 - All objects should be mutually non-overlapping.
 - If the *type* attribute of two objects both equal 1, the two objects should not touch, not even their corners.
- The figure on the right shows one solution.



```

overlap( $D, o_i, s_i, o_j, s_j$ )  $\rightarrow$ 
   $\forall d \in D : \text{end}(o_i, s_i, d) > \text{ori}(o_j, s_j, d) \wedge \text{end}(o_j, s_j, d) > \text{ori}(o_i, s_i, d)$ 
meet( $D, o_i, s_i, o_j, s_j$ )  $\rightarrow$ 
  ( $\forall d \in D : \text{end}(o_i, s_i, d) \geq \text{ori}(o_j, s_j, d) \wedge \text{end}(o_j, s_j, d) \geq \text{ori}(o_i, s_i, d)$ )  $\wedge$ 
  ( $\exists d \in D : \text{end}(o_i, s_i, d) = \text{ori}(o_j, s_j, d) \vee \text{end}(o_j, s_j, d) = \text{ori}(o_i, s_i, d)$ )
all_not_overlap( $D, OIDs$ )  $\rightarrow$ 
   $\forall o_i \in OIDs, \forall s_i \in o_i.\text{sid}, \forall o_j \in OIDs :$ 
     $o_i.\text{oid} < o_j.\text{oid} \Rightarrow (\forall s_j \in o_j.\text{sid} : \neg \text{overlap}(D, o_i, s_i, o_j, s_j))$ 
all_type1_not_meet( $D, OIDs$ )  $\rightarrow$ 
   $\forall o_i \in OIDs, \forall s_i \in o_i.\text{sid}, \forall o_j \in OIDs :$ 
     $o_i.\text{oid} < o_j.\text{oid} \wedge o_i.\text{type} = 1 \wedge o_j.\text{type} = 1 \Rightarrow$ 
     $\forall s_j \in o_j.\text{sid} : \neg \text{meet}(D, o_i, s_i, o_j, s_j)$ 
all_not_overlap_sboxes( $[1, 2], [1, 2, 3, 4, 5]$ )
all_type1_not_meet_sboxes( $[1, 2], [1, 2, 3, 4, 5]$ )

```

Fig. 1. Macros and rules of the running example. $\text{ori}(o, s, d)$ (resp. $\text{end}(o, s, d)$) stands for the origin (resp. end) in dimension d of object o with shape s . $\square$

Declarative semantics. As usual, the semantics is given in terms of ground objects. The constraint $\text{geost}(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$ holds if and only if the conjunction of the logical formulas in $\mathcal{R}$ is true.

Implementation overview. Fig. 2 provides the overall architecture of the implementation. When the geost constraint is posted, the given business rules are translated, first into QFPA, then into generators of k -dimensional forbidden sets. Such generators, k -indexicals, are a generalization of the indexicals of $\text{cc}(\text{FD})$ [5]. Each time the constraint wakes up, the sweep-based algorithm [3] generates forbidden sets for a specific object o by invoking the relevant k -indexicals, then looks for points that are not contained in any forbidden set in order to prune the nonground attributes of o .

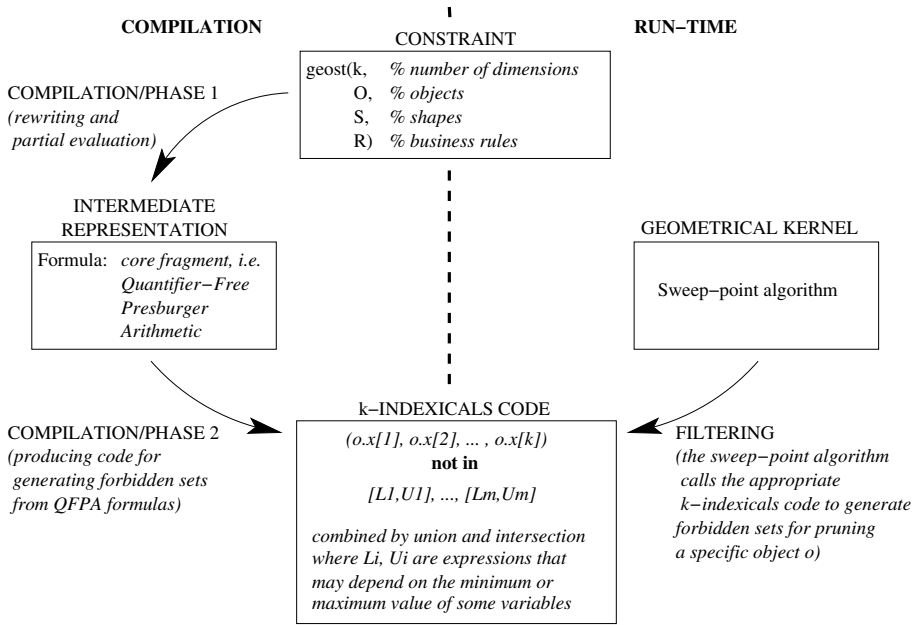


Fig. 2. Overall architecture of the implementation

Paper outline. In Section 2, we present the rule language, its abstract syntax and its features. In Section 3, we present the QFPA core fragment of the language, its declarative semantics, and how the rule language is rewritten into QFPA. In Section 4, we describe (1) how a QFPA formula is compiled to generators of k -dimensional forbidden sets, and (2) how the forbidden sets generated by such generators are aggregated by a sweep-based algorithm and used for filtering. In Section 5, we provide experimental evidence for search space reduction due to the global treatment of side constraints. Before concluding, in Section 6, we mention a number of issues that we are currently working on. An expanded version of this paper is available as a technical report [6]. In particular, see [6, Section 5] for an extension of the filtering to accommodate polymorphic objects.

2 The Rule Language: Syntax and Features

A sentence is either a *macro* or a *fol*. A *macro* is simply a shorthand device: during a rewriting phase, whenever an expression matching the left-hand side of a macro is encountered, it is replaced by the corresponding right-hand side. A *fol* is a first-order logic formula that must hold for the constraint to be true, and is one of: a comparison between two arithmetic expressions, the constant **true** or **false**, a cardinality formula $\#(var, collection, integer, integer, fol)$, a quantified formula $\forall(var, collection, fol)$

or $\exists(\text{var}, \text{collection}, \text{fol})$, or formulas combined with logical connectives: $\neg \text{fol}$, $\text{fol} \wedge \text{fol}$, $\text{fol} \vee \text{fol}$, $\text{fol} \Rightarrow \text{fol}$, or $\text{fol} \Leftrightarrow \text{fol}$.

An *expr* (arithmetic expression) is an integer, an *attref* (a reference to an attribute of an entity, where an *entity* is an object or a shifted box), a fold expression $@(\text{var}, \text{collection}, \circ, \text{expr}, \text{expr})$ where $\circ \in \{+, \min, \max\}$, or an expression $\text{expr} \circ \text{expr}$ where $\circ \in \{+, -, \times, /, \min, \max\}$. Arithmetic expressions must be *linear*: in a product, at most one factor can be nonground; in a quotient, the divisor must be ground.

A *collection* is the shorthand $\text{objects}(S)$, denoting the collection of objects with object id in S , or the shorthand $\text{sboxes}(S)$ denoting the collection of shifted boxes with shape id in S , or a list of terms, where a *term* is a *variable*, an *integer*, an *identifier*, or a *compound term*. A *compound term* consists of a *functor* (an identifier) and one or more arguments (terms). A term is *ground* if it is free of variables.

Quantified formulas are meaningful if the quantified variable occurs in the quantified *fol*. They are treated by expansion to a disjunction resp. a conjunction of instances of that *fol* where each element of the *collection* is substituted for the quantified variable. In the context of our application, quantified variables typically vary over a collection of dimensions, objects, or shifted boxes.

A cardinality formula specifies a variable quantified over a list of terms, a lower and an upper bound, and a *fol* template mentioning the quantified variable. The formula is true if and only if the number of true instances of the *fol* template is within the given bounds. Cardinality formulas [7] are treated by expansion to $\neg$, $\wedge$ and $\vee$ connectives [8].

Arithmetic expressions and comparisons are over the rational numbers. The rationale for this is that business rules often involve fractions of measures like weight or volume. However, such fractions are converted to integers during rewriting.

Fold expressions allow to express e.g. the sum of some attribute over a set of objects. The operator specifies a variable quantified over a list of terms, a binary operator, an identity element, and a template mentioning the quantified variable. The identity element is needed for the empty list case.

3 QFPA Core Fragment

In this section, we show how a formula p in the rule language is rewritten by a series of equivalence-preserving transformations into a *qfpa*, i.e. a QFPA formula, which here either is of the form $\sum_i \text{integer}_i \cdot \text{attref}_i \geq \text{integer}$ or is a conjunction or a disjunction of *qfpas*.

QFPA is widely used in symbolic verification, and there has been much work on deciding whether a given QFPA formula is satisfiable [9]. Many methods based on integer programming techniques [10] rely on having the formula on disjunctive normal form. However, for constraint programming purposes, we are interested in necessary conditions that can be used for filtering domain variables, and we are not aware of any such work on QFPA. In [11], filtering algorithms for logical combinations of adhoc constraints³ are proposed, but it is not clear whether that approach can be extended to QFPA. For that, we would need to provide supports of *qfpas*.

³ Also known as constraints given in extension.

3.1 Rewriting into QFPA

We now show the details of rewriting the formula given as the *geost* parameter $\mathcal{R}$ in the following eight steps into a *qfpa* $\hat{\mathcal{R}}$. We will later show how $\hat{\mathcal{R}}$ is translated to generators of forbidden sets.

Macro expansion and constant folding. The implication and equivalence connectives, bounded quantifiers, and cardinality and folding operators are eliminated. Ground integer expressions are replaced by their values. Object and shifted box collections are expanded.

Elimination of negation. Using DeMorgan's laws and negating relevant *relops*.

Normalization of arithmetic. Arithmetic relations are normalized to one of the forms $expr \geq 0$ or $expr > 0$.

Elimination of $\times$, $/$ and $-$. Any occurrence of these operators in arithmetic expressions is eliminated. At the same time, all operands are associated with a rational coefficient (c in the table). The elimination is made possible by the fact that in multiplication, at least one factor must be ground and is simply multiplied into the coefficient. Similarly, in division, the coefficient is simply divided by the divisor, which must be ground.

Moving $+$ inside $\min$ and $\max$. Any expression with $\min$ or $\max$ occurring inside $+$ are rewritten by using the commutative and distributive laws (1) so that the $+$ is moved inside the other operator.

$$\begin{aligned} a + b &= b + a \\ a + \min(b, c) &= \min(a + b, a + c) \\ a + \max(b, c) &= \max(a + b, a + c) \end{aligned} \tag{1}$$

Elimination of $\min$ and $\max$. Any $\min$ or $\max$ operators occurring in arithmetic relations are eliminated, replacing such relations by new relations combined by $\wedge$ or $\vee$. After this step, an arithmetic expression is a linear combination of *attrefs* with rational coefficients, plus an optional constant.

Elimination of rational numbers. Any arithmetic relation r , which can now only be of the form $e > 0$ or $e \geq 0$, is normalized into the form $e'' \geq c''$ where e' and c' are intermediate expressions in:

- Let e' be the linear combination obtained by multiplying e by the least common multiplier of the denominators of the coefficients of e . Recall that those coefficients are rational numbers. Thus, the coefficients of e' are integers.
- Let c' be 1 if r is of the form $e > 0$, or 0 if r is of the form $e \geq 0$.
- If e' contains a constant term c , then $e'' = e' - c$ and $c'' = c' - c$. Otherwise, $e'' = e'$ and $c'' = c'$.

Simplification. Any entailed or disentailed arithmetic comparison is replaced by the appropriate logical constant (`true` or `false`). Any $\wedge$ or $\vee$ expression containing one of these constants is simplified using partial evaluation.

Example 2. Returning to our running example the resulting *gfpa* $\hat{\mathcal{R}}$ shown on the right is a conjunction of six subformulas corresponding respectively to:

- From the business rule `all_not_overlap_sboxes`, conditions to prevent o_5 from overlapping o_1, o_2, o_3 and o_4 .
- From the business rule `all_type1_not_meet_sboxes`, conditions to prevent o_5 from meeting o_2 and o_4 . $\square$

$$\left(\begin{array}{c} \vee \left(\begin{array}{c} x_{51} \geq 4 \\ x_{52} \geq 3 \end{array} \right) \\ \vee \left(\begin{array}{c} x_{51} \geq 4 \\ -1 \cdot x_{51} \geq -1 \\ x_{52} \geq 4 \\ -1 \cdot x_{52} \geq -1 \end{array} \right) \\ \vee \left(\begin{array}{c} -1 \cdot \ell_{31} + x_{51} \geq 2 \\ -1 \cdot \ell_{32} + x_{52} \geq 5 \\ -1 \cdot x_{52} \geq -3 \end{array} \right) \\ \vee \left(\begin{array}{c} x_{51} \geq 6 \\ -1 \cdot x_{51} \geq -1 \\ -1 \cdot x_{52} \geq -5 \end{array} \right) \end{array} \right) \wedge \left(\begin{array}{c} \vee \left(\begin{array}{c} x_{51} \geq 5 \\ x_{52} \geq 5 \end{array} \right) \\ \vee \left(\begin{array}{c} \vee \left(\begin{array}{c} -1 \cdot x_{51} \geq -3 \\ x_{51} \geq 5 \end{array} \right) \\ x_{51} \geq 2 \\ \vee \left(\begin{array}{c} -1 \cdot x_{52} \geq -3 \\ x_{52} \geq 5 \end{array} \right) \\ x_{52} \geq 2 \end{array} \right) \\ \vee \left(\begin{array}{c} x_{51} \geq 7 \\ -1 \cdot x_{52} \geq -4 \\ \vee \left(\begin{array}{c} -1 \cdot x_{51} \geq -5 \\ x_{51} \geq 7 \end{array} \right) \\ x_{51} \geq 2 \\ \vee \left(\begin{array}{c} x_{52} \geq 6 \\ -1 \cdot x_{52} \geq -4 \end{array} \right) \end{array} \right) \end{array} \right)$$

4 Compiling to an Efficient Run-Time Representation

It is straightforward to obtain necessary conditions for *gfpa*s as well as pruning rules operating on one variable at a time. Based on such conditions and pruning rules, we will show how to construct generators of k -dimensional forbidden sets. We call such generators *k-indexicals*, for they are generalization of the indexicals of `cc(FD)` [5]. Finally, we show how the forbidden sets generated by such indexicals are aggregated by the sweep-based algorithm [3] and used for filtering.

Indexicals were first introduced for the language `cc(FD)` [5] and later used in the context of `CLP(FD)` [12,13], `AKL` [14], finite set constraints [15] and adhoc constraints [16]. They have proven a powerful and efficient way of implementing constraint propagation. A key feature of an indexical is that it is a function of the current domains of the variables on which it depends. Thus, indexicals also capture the propagation from variables to variables that occurs as variables are pruned. In the cited implementations, an indexical is a procedure that computes the feasible set of values for a variable. We generalize this notion to generating a forbidden set of k -dimensional points, for an

object, and so k -indexicals captures the propagation from objects to objects that occurs as object attributes are pruned.

4.1 Necessary Conditions

For a formula R denoting a linear combination of variables, let $MAX(R)$ denote the expression that replaces every *attref* x in R by $\bar{x}$ if x occurs with a positive coefficient, and by $\underline{x}$ otherwise. Thus, $MAX(R)$ is a formula that computes an upper bound of R wrt. the current domains.

We will ignore the degenerate cases where $\hat{\mathcal{R}}$ is *true* resp. *false*, in which case *geost* merely succeeds resp. fails. For the normal *qfpa* cases, we obtain the necessary conditions shown in Table 1.

Table 1. Necessary condition $N(t)$ for *qfpa* t

<i>qfpa</i> t	necessary condition $N(t)$
$\sum_i c_i \cdot x_i \geq r$	$MAX(\sum_i c_i \cdot x_i) \geq r$
$p \vee q$	$N(p) \vee N(q)$
$p \wedge q$	$N(p) \wedge N(q)$

4.2 Pruning Rules

For the base case $\sum_i c_i \cdot x_i \geq r$, we have the well-known pruning rules (2), which provide sharp bounds; see e.g. [17] for details.

$$\forall j \begin{cases} x_j \geq \lceil \frac{r - MAX(\sum_{i \neq j} c_i \cdot x_i)}{c_j} \rceil, & \text{if } c_j > 0 \\ x_j \leq \lfloor \frac{-r + MAX(\sum_{i \neq j} c_i \cdot x_i)}{-c_j} \rfloor, & \text{otherwise} \end{cases} \quad (2)$$

Consider now a disjunction $p \vee q$ of two base cases and a variable x_j occurring in at least one disjunct.

- If x_j occurs in p but not in q , rule (2) is only valid for p if the necessary condition for q does not hold.
- Similarly if x_j occurs in q but not in p .
- If x_j occurs in both p and q , we can use rule (2) for both p and q and conclude that x_j must be in the union of the two feasible intervals.

Finally, consider a conjunction $p \wedge q$, i.e. both p and q must hold. If x_j occurs in both p and q , we can use rule (2) for both p and q and conclude that x_j must be in the intersection of the two feasible intervals.

Example 3. Returning to our running example, consider the fragment $x_{51} \geq 4 \vee x_{52} \geq 3$ of the *qfpa*, which comes from a rule preventing o_5 from overlapping o_1 . Suppose that we want to prune x_{52} . Then we can combine the necessary condition for $x_{51} \geq 4$ with rule (2) for $x_{52} \geq 3$ into the conditional pruning rule:

$$\max(x_{51}) < 4 \Rightarrow x_{52} \geq 3$$

However, as we will show in the next section, instead of using such conditional pruning rules, we unify necessary conditions and pruning rules into multi-dimensional forbidden sets and aggregate them per object. For the above fragment, the two-dimensional forbidden set for o_5 is $([1, 3], [1, 2])$, denoting the fact that (x_{51}, x_{52}) should be distinct from all the pairs $(1, 1)$, $(1, 2)$, $(2, 1)$, $(2, 2)$, $(3, 1)$, $(3, 2)$. $\square$

4.3 k -Indexicals

Recall that the set of rules given in $\mathcal{R}$ has been rewritten into a *qffa* $\hat{\mathcal{R}}$. Consider this formula, or some subformula $\hat{\mathcal{R}}_i$ of it if $\hat{\mathcal{R}}$ is a conjunction (see Section 4.4). The idea is to compile this subformula, for each object o mentioned by it, into a k -indexical for $\hat{\mathcal{R}}_i$ and o . The forbidden sets that it generates can then be aggregated and used by the sweep-point kernel [3] to prune the nonground attributes of o . Let us introduce some notation to make this idea clear.

Definition 1. A forbidden set for a *qffa* r and object o is a set⁴ of k -dimensional points such that, if o is placed at any of these points, r is disentailed.

Such a forbidden set can also be seen as the multi-dimensional generalization of a set of inconsistent assignments [18].

Definition 2. A k -indexical for a *qffa* r and an object o is a procedure that functions as a generator of forbidden sets for r and o . It is of the form $o.x \notin ibody$ where *ibody* is defined in Fig. 3. The k -indexical depends on object o' if *ibody* mentions o' .

$k\text{-indexical}$	$::= object.x \notin ibody$	
$ibody$	$::= ibody \cap ibody$ $\mid ibody \cup ibody$ $\mid \{p \in \mathbb{Z}^k \mid p[d] < \lceil \frac{integer - \sum_{usi} ubterm}{usi} \rceil\}$ $\mid \{p \in \mathbb{Z}^k \mid p[d] > \lfloor \frac{integer + \sum_{usi} ubterm}{usi} \rfloor\}$ $\mid \text{if } \sum ubterm < r \text{ then } \mathbb{Z}^k \text{ else } \emptyset$	
$ubterm$	$::= usi \cdot \overline{attref}$ $\mid -usi \cdot \underline{attref}$ $\mid integer$	
d	$::= integer$	{ denoting a dimension }
usi	$::= integer$	{ > 0 }

Fig. 3. k -indexicals

⁴ A forbidden set is not explicitly represented as a set of points, but rather by a set of boxes, as is the case in the earlier implementation [3].

k -indexicals are described by the inductive definition shown in Fig. 3. They are built up from generators of k -dimensional half-planes, combined by union and intersection operations.

4.4 Compilation

The *qfpa* $\hat{\mathcal{R}}$, normally⁵ a conjunction $\hat{r}_1 \wedge \dots \wedge \hat{r}_n$, is compiled to k -indexicals by the following steps:

1. Partition the conjuncts of $\hat{\mathcal{R}}$ into equivalence classes $\hat{\mathcal{R}}_1, \dots, \hat{\mathcal{R}}_m$ such that for all $1 \leq i < j \leq n$, $\hat{r}_i$ and $\hat{r}_j$ are in the same equivalence class if and only if they mention⁶ the same set of objects of $\mathcal{O}$.
2. For each equivalence class $\hat{\mathcal{R}}_i$ and object $o \in \mathcal{O}$ mentioned by $\hat{\mathcal{R}}_i$, map $\hat{\mathcal{R}}_i$ (as a conjunction) into a k -indexical for o , of the form $o.x \notin F_o(\hat{\mathcal{R}}_i)$, according to Table 2.

The mapping closely follows the pruning rules (2), except now we want to obtain a forbidden set instead of a feasible interval. Rows 1-2 of Table 2 are analogous to the recursive computation of inconsistent assignments in [18, Table 1]. Row 5 corresponds to the case where r does not mention o , in which case all points are forbidden for o if r is disentailed, and no points are forbidden for o otherwise.

Table 2. Mapping a *qfpa* r to a generator of forbidden sets, $F_o(r)$, for the object o . We assume here that o is not polymorphic.

r	$F_o(r)$	condition
$p \vee q$	$F_o(p) \cap F_o(q)$	
$p \wedge q$	$F_o(p) \cup F_o(q)$	
$\sum_i c_i \cdot x_i \geq r$	$\{p \in \mathbb{Z}^k \mid p[d] < \lceil \frac{r - \text{MAX}(\sum_{i \neq j} c_i \cdot x_i)}{c_j} \rceil\}$	$x_j = o.x[d], c_j > 0$
$\sum_i c_i \cdot x_i \geq r$	$\{p \in \mathbb{Z}^k \mid p[d] > \lfloor \frac{-r + \text{MAX}(\sum_{i \neq j} c_i \cdot x_i)}{-c_j} \rfloor\}$	$x_j = o.x[d], c_j < 0$
$\sum_i c_i \cdot x_i \geq r$	if $\text{MAX}(\sum_i c_i \cdot x_i) < r$ then $\mathbb{Z}^k$ else $\emptyset$	$o.x[d] \notin \{x_i\}$

The rationale for aggregating the conjuncts into equivalence classes, as opposed to mapping one conjunct at a time, is the opportunity to increase the granularity of the indexicals and to merge subformulas coming from different business rules. This opens the scope for future work on global simplification of formulas, and increases the amount of subexpressions that can be shared within a k -indexical.

It is well known that indexicals can be efficiently compiled and executed by a virtual machine [12,13]. In our context, we predict that there will be a large amount of common subterms in the k -indexicals, and so common subexpression elimination will be quite important. Therefore, a register-based virtual machine would seem an appropriate choice.

⁵ Since it comes from the conjunction of business rules stated in the last argument of *geost*.

⁶ A formula *mentions* an object o if it refers to a nonground attribute of o .

Example 4. Returning to our running example, we obtained a *qffa* which was a conjunction of six subformulas. They are partitioned into two equivalence classes: one for the single conjunct that mentions both o_3 and o_5 , mapped to k -indexicals (3) and (4) below; and one for the five conjuncts that only mention o_5 (because o_1 , o_2 and o_4 are ground), mapped to k -indexical (5) below. The three k -indexicals reflect the following business rules:

1. o_3 must not take a shape that will cause it to overlap o_5 . Note that this k -indexical propagates from o_5 to the shape id of o_3 . Pruning of shape ids of polymorphic objects is discussed in [6, Section 5]. Initially, no forbidden boxes are generated.

$$s_3 \notin \bigcap \left(\begin{array}{l} \{i \in \text{dom}(s_3) \mid s_3 = i \Rightarrow \ell_{31} > \overline{x_{51}} - 2\} \\ \{i \in \text{dom}(s_3) \mid s_3 = i \Rightarrow \ell_{32} > \overline{x_{52}} - 5\} \\ \text{if } \underline{x_{52}} > 3 \text{ then } \mathbb{Z} \text{ else } \emptyset \end{array} \right) \quad (3)$$

2. o_5 must not overlap o_3 . Note that this k -indexical propagates from o_3 to o_5 .

$$o_5.x \notin ([1, (\underline{\ell}_{31} + 1)], [4, (\underline{\ell}_{32} + 4)]) \quad (4)$$

3. o_5 must not overlap o_1 , o_2 nor o_4 , nor meet o_2 nor o_4 .

$$o_5.x \notin \bigcup \left(\begin{array}{l} ([1, 3], [1, 2]) \\ ([2, 3], [2, 3]) \\ ([2, 5], [6, 6]) \\ ([1, 4], [1, 4]) \\ \left(\bigcup \left(\begin{array}{l} ([4, 4], [1, 6]) \\ ([1, 1], [1, 6]) \\ ([1, 9], [4, 4]) \\ ([1, 9], [1, 1]) \end{array} \right) \right) \\ \left(\bigcap \left(\begin{array}{l} ([1, 6], [5, 6]) \\ ([1, 9], [5, 5]) \\ ([6, 6], [1, 6]) \\ ([1, 1], [1, 6]) \end{array} \right) \right) \end{array} \right) \quad (5)$$

□

4.5 Filtering Algorithm

We now give a sketch of a filtering algorithm for $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$. Let $I(o)$ denote the set of k -indexicals for object $o \in \mathcal{O}$ wrt. the given rules $\mathcal{R}$, let $\text{eval}(i)$ denote the evaluation of k -indexical i wrt. the current domains, let $\text{sweep}(o, F)$ denote the application of the sweep-based algorithm to the object o wrt. the forbidden set F , which prunes the minimum and maximum values of the origin coordinates of o . Our proposed Algorithm 1 is a straightforward propagation loop.

PROCEDURE Filter($\mathcal{O}, I$)

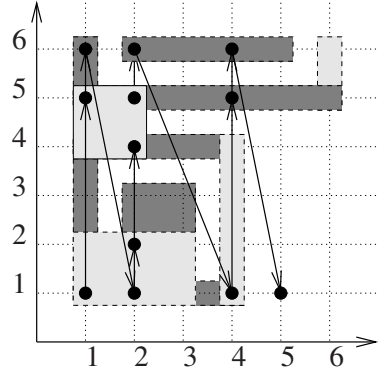
```

1:  $Q \leftarrow \mathcal{O}$ 
2: while  $Q \neq \emptyset$  do
3:    $o \leftarrow$  some element from  $Q$ 
4:    $Q \leftarrow Q \setminus \{o\}$ 
5:    $F \leftarrow \bigcup \{\text{eval}(i) \mid i \in I(o)\}$ 
6:   if  $\neg \text{sweep}(o, F)$  then
7:     return fail
8:   else if a coordinate of  $o$  was pruned then
9:      $Q \leftarrow Q \cup \{o' \mid I(o') \text{ depends on } o\}$ 
10:  end if
11: end while
12: if all objects in  $\mathcal{O}$  are ground then
13:   return succeed
14: else
15:   return suspend
16: end if

```

Algorithm 1. Sketch of a filtering algorithm for $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{R})$

Example 5. Returning to our running example, suppose now that the sweep-point kernel wants to adjust the lower bound of x_{51} . The figure on the right traces the steps performed by the algorithm when it walks from a lexicographically smallest position to the first feasible position of o_5 . The result is that the lower bound of x_{51} is adjusted to 5. $\square$



5 Experimental Results

The *geost* constraint, including the rewriting, compilation, and sweep-based algorithms, has been implemented in SICStus Prolog 4 [19] using its global constraint programming API. A direct performance comparison of this proof-of-concept implementation with the earlier implementation [3], coded in C, is not meaningful. Therefore, we focus on showing the potential for search space reduction and stronger filtering due to the global treatment of side constraints.

We studied the placement problem given in [6, Appendix C], provided by Peugeot Citroën, which involves a $1203 \times 235 \times 239$ container and 9 objects with an extra *weight* attribute, subject to the rules:

- (a) Each object is placed inside the container.
- (b) Each object is either on the floor or resting on some other object.

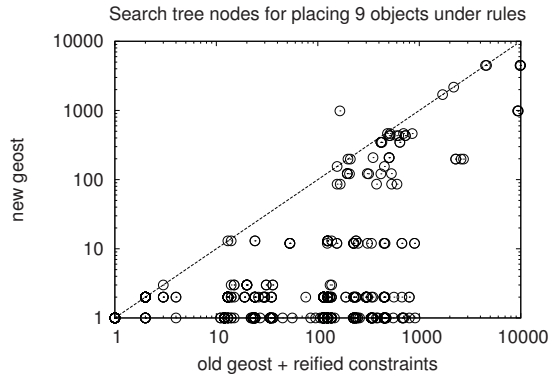
- (c) The objects do not pairwise overlap.
- (d) A heavier object cannot be piled on top of a lighter one.
- (e) For any two objects in a pile, the overhang can be at most 10 units.

In [6] we provide encodings of these rules as well as other rules encoding a packing-unpacking problem with visibility constraints⁷ and the time dimension.

We generated 600 problem instances by randomly permuting the objects. The search was performed by labeling the coordinates, grouped by object, in the permuted order, under a time limit of one CPU minute. For each instance, we posted the constraint, and measured the number of search space nodes visited during search for the first solution. Each instance was run twice:

1. with the new *geost* constraint, and
2. without it, but expressing constraint (c) with the earlier implementation [3] and constraints (a, b, d, e) with logical combinations of arithmetic constraints.

The scatter plot shown on the right summarizes the results. Each dot represents an instance, its X (resp. Y) coordinate corresponding to the old (resp. new) implementation. The search effort was decreased by 100 times or more in 26% of the cases and by 10 times or more in another 33% of the cases.



6 Discussion

Generality. Our restriction that object attributes (except shape id and origin) must be ground is somewhat artificial, and we plan to lift it. The rewritten QFPA formulas would simply have more variables per object, and the sweep-based algorithm would deal not with a k - or $k + 1$ -dimensional *placement* space, but with an m -dimensional *solution* space, where m is the number of possibly nonground attributes per object. In particular, in order to deal with objects whose length in some dimension is a domain variable that occurs in some other constraint, the length and possibly the end-point would have to be expressed as nonground object attributes. Similarly, to treat the time dimension, we would add three nonground object attributes *start*, *duration*, and *completion*, as in [3], to be included in the solution space.

Theoretical properties. It has been shown [1, Proposition 1-2] that the PKML/Rules2CP rewriting system is confluent and Noetherian (i.e., terminating). Since our rule language is essentially a subset of Rules2CP, the results apply to *geost* rules as well. A size

⁷ See the *visible* constraint in <http://www.emn.fr/x-info/sdemasse/gccat/>.

bound on programs generated from Rules2CP is also known [1, Proposition 3] and applies to *geost* provided that min, max and cardinality is not used in the rules, since these operators can cause an exponential (for min and max) resp. quadratic (for cardinality) [8] blow-up. Consequently, one can certainly construct pathological cases where the rewrite phases and/or runtime representation require huge amounts of memory. Even if, at this time, this has not really been a problem for the instances and rules we have experimented with⁸, one way to manage the complexity of the rewrite phases is to apply simplifying rewrites, e.g. Phase 8, as eagerly as possibly. Another way could be to memoize patterns that have already been rewritten. Finally, common subexpression elimination will mitigate this problem.

7 Conclusion

We have presented a global constraint that enforces rules written in a language based on arithmetic and first-order logic to hold among a set of objects. By rewriting the rules to QFPA formulas, we have shown how to compile them to k -indexicals and how the forbidden sets generated by such indexicals can be aggregated by a sweep-based algorithm and used for filtering. Initial experiments support the feasibility of the approach. The approach combines an expressive logic-based rule modeling language for stating business rules with a generic geometrical algorithm for effective filtering.

Finally, QFPA is a language in which also many other problems, unrelated to packing and placement, can be stated. In this paper, we have begun to explore a way to compile and run it efficiently.

Acknowledgements

This research was conducted under European Union Sixth Framework Programme Contract FP6-034691 “Net-WMS”. The second author was also partly supported by ANR (CANAR/06-BLAN-0383-02).

References

1. Fages, F., Martin, J.: From rules to constraint programs with the Rules2CP modelling language. Research Report RR-6495, INRIA Rocquencourt (2008)
2. Beldiceanu, N., Contejean, E.: Introducing global constraints in CHIP. *Mathl. Comput. Modelling* 20(12), 97–123 (1994)
3. Beldiceanu, N., Carlsson, M., Poder, E., Sadek, R., Truchet, C.: A generic geometrical constraint kernel in space and time for handling polymorphic k -dimensional objects. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 180–194. Springer, Heidelberg (2007)
4. Benedetti, M., Lallouet, A., Vautard, J.: QCSP made practical by virtue of restricted quantification. In: IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, pp. 38–43 (2007)
5. Van Hentenryck, P., Saraswat, V., Deville, Y.: Constraint processing in cc(FD). Computer Science Department, Brown University (unpublished manuscript) (1991)

⁸ They involved at most 100 objects.

6. Beldiceanu, N., Carlsson, M., Martin, J.: A geometric constraint over k -dimensional objects and shapes subject to business rules. SICS Technical Report T2008:04, Swedish Institute of Computer Science (2008)
7. Van Hentenryck, P., Deville, Y.: The cardinality operator: a new logical connective in constraint logic programming. In: *Int. Conf. on Logic Programming (ICLP 1991)*. MIT Press, Cambridge (1991)
8. Eén, N., Sörensson, N.: Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation* 2, 1–25 (2006)
9. Ganesh, V., Berezin, S., Hill, D.L.: Deciding presburger arithmetic by model checking and comparisons with other methods. In: Aagaard, M.D., O’Leary, J.W. (eds.) *FMCAD 2002*. LNCS, vol. 2517, pp. 171–186. Springer, Heidelberg (2002)
10. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. In: *Supercomputing*, pp. 4–13 (1991)
11. Lhomme, O.: Arc-consistency filtering algorithms for logical combinations of constraints. In: Régin, J.-C., Rueher, M. (eds.) *CPAIOR 2004*. LNCS, vol. 3011, pp. 209–224. Springer, Heidelberg (2004)
12. Codognet, P., Diaz, D.: Compiling constraints in $\text{clp}(\text{FD})$. *Journal of Logic Programming* 27(3), 185–226 (1996)
13. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Glaser, H., Hartel, P., Kuchen, H. (eds.) *PLILP 1997*. LNCS, vol. 1292, pp. 191–206. Springer, Heidelberg (1997)
14. Carlson, B.: *Compiling and Executing Finite Domain Constraints*. PhD thesis, Uppsala University (1995)
15. Tack, G., Schulte, C., Smolka, G.: Generating propagators for finite set constraints. In: Benhamou, F. (ed.) *CP 2006*. LNCS, vol. 4204, pp. 575–589. Springer, Heidelberg (2006)
16. Cheng, K.C.K., Lee, J.H.M., Stuckey, P.J.: Box constraint collections for adhoc constraints. In: Rossi, F. (ed.) *CP 2003*. LNCS, vol. 2833, pp. 214–228. Springer, Heidelberg (2003)
17. Harvey, W., Stuckey, P.J.: Constraint representation for propagation. In: Maher, M.J., Puget, J.-F. (eds.) *CP 1998*. LNCS, vol. 1520, pp. 235–249. Springer, Heidelberg (1998)
18. Bacchus, F., Walsh, T.: Propagating logical combinations of constraints. In: *IJCAI 2005, Proceedings of the 19th International Joint Conference on Artificial Intelligence*, pp. 35–40 (2005)
19. Carlsson, M., et al.: *SICStus Prolog User’s Manual*, 4th edn. Swedish Institute of Computer Science, pp. 91–630 (2007) ISBN 91-630-3648-7

Cost-Based Domain Filtering for Stochastic Constraint Programming^{*}

Roberto Rossi¹, S. Armagan Tarim², Brahim Hnich³, and Steven Prestwich¹

Cork Constraint Computation Centre - CTVR, University College, Cork, Ireland
`{r.rossi,s.prestwich}@4c.ucc.ie`

Department of Management, Hacettepe University, Ankara, Turkey
`armagan.tarim@hacettepe.edu.tr`

Faculty of Computer Science, Izmir University of Economics, Turkey
`brahim.hnich@ieu.edu.tr`

Abstract. Cost-based filtering is a novel approach that combines techniques from Operations Research and Constraint Programming to filter from decision variable domains values that do not lead to better solutions [7]. Stochastic Constraint Programming is a framework for modeling combinatorial optimization problems that involve uncertainty [19]. In this work, we show how to perform cost-based filtering for certain classes of stochastic constraint programs. Our approach is based on a set of known inequalities borrowed from Stochastic Programming — a branch of OR concerned with modeling and solving problems involving uncertainty. We discuss bound generation and cost-based domain filtering procedures for a well-known problem in the Stochastic Programming literature, the static stochastic knapsack problem. We also apply our technique to a stochastic sequencing problem. Our results clearly show the value of the proposed approach over a pure scenario-based Stochastic Constraint Programming formulation both in terms of explored nodes and run times.

1 Introduction

Constraint Programming (CP) [1] has been recognized as a powerful tool for modeling and solving combinatorial optimization problems. CP provides global constraints offering concise and declarative modeling capabilities and efficient domain filtering algorithms. These algorithms remove combinations of values which cannot appear in any consistent solution. Cost-based filtering is an elegant way of combining techniques from CP and Operations Research (OR) [7]. OR-based optimization techniques are used to remove from variable domains values that cannot lead to better solutions. This type of domain filtering can be

^{*} S. Armagan Tarim and Brahim Hnich are supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under Grant No. SOBAG-108K027. Roberto Rossi is supported by Science Foundation Ireland under Grant No. 03/CE3/I405 as part of the Centre for Telecommunications Value-Chain-Driven Research (CTVR) and Grant No. 05/IN/I886.

combined with the usual CP-based filtering methods and branching heuristics, yielding powerful hybrid search algorithms. Cost-based filtering is a novel technique that has been the subject of significant recent research.

Stochastic Constraint Programming (SCP) [19] is an extension of CP, in which there is a distinction between decision variables, which we are free to set, and stochastic (or observed) variables, which follow some probability distribution. SCP is designed to handle problems in which uncertainty comes into play. Uncertainty may take different forms: data about events in the past may not be known exactly due to measuring or difficulties in sampling, and data about events in the future may simply not be known with certainty.

In this work we propose a novel approach to performing cost-based filtering for certain classes of stochastic constraint programs. Our approach is based on a well-known inequality borrowed from Stochastic Programming [4], a branch of OR that is concerned with modeling constraint satisfaction/optimization problems under uncertainty. We implemented this approach for two problems in which uncertainty plays a role. In both cases we obtained significant improvements with respect to a pure SCP formulation both in terms of explored nodes and run times.

The rest of the paper is structured as follows. In Section 2 we give the necessary formal background. In Section 3 we review relevant inequalities from Stochastic Programming. In Section 4, we introduce global optimization chance constraints. We describe our empirical results in Section 5 and review related works in Section 6. Finally, we conclude and outline our future work in Section 7.

2 Formal Background

A *Constraint Satisfaction Problem* (CSP) [1] is a triple $\langle V, C, D \rangle$, where $V = \{V_1, \dots, V_n\}$ is a set of decision variables, D is a function mapping each element of V to a domain of potential values, and C is a set of constraints stating allowed combinations of values for subsets of variables in V . A *solution* to a CSP is an assignment to every variable of a value in its domain, such that all of the constraints are satisfied. We may also be interested in finding a feasible solution that maximizes (minimizes) the value of a given objective function over a subset of the variables. With no loss of generality, we restrict our discussion to maximization problems.

Optimization-oriented global constraints embed an optimization component, representing a proper relaxation of the constraint itself, into a global constraint [7]. This component provides three pieces of information: (a) the optimal solution of the relaxed problem; (b) the optimal value of this solution representing an upper bound on the original problem objective function; (c) a *gradient function* $\text{grad}(V, v)$, which returns for each variable-value pair (V, v) an optimistic evaluation of the profit obtained if v is assigned to V . These pieces of information are exploited both for propagation purposes and for guiding the search.

In [19], a *stochastic CSP* is defined as a 6-tuple $\langle V, S, D, P, C, \theta \rangle$, where V is a set of decision variables and S is a set of stochastic variables, D is a function

mapping each element of V and each element of S to a domain of potential values. A decision variable in V is *assigned* a value from its domain. P is a function mapping each element of S to a probability distribution for its associated domain. C is a set of constraints. A constraint $h \in C$ that constrains at least one variable in S is a *chance-constraint*. θ_h is a threshold value in the interval $[0, 1]$, indicating the minimum satisfaction probability for chance-constraint h . Note that a chance-constraint with a threshold of 1 (or without any explicit threshold specified) is equivalent to a hard constraint. A stochastic CSP consists of a number of *decision stages*. A decision stage is a pair $\langle V_i, S_i \rangle$, where V_i is a set of decision variables and S_i is a set of stochastic variables. In an m -stage stochastic CSP, V and S are partitioned into disjoint sets, $V_1, \dots, V_m$ and $S_1, \dots, S_m$, and we consider multiple stages, $\langle V_1, S_1 \rangle, \langle V_2, S_2 \rangle, \dots, \langle V_m, S_m \rangle$. To solve an m -stage stochastic CSP an assignment to the variables in V_1 must be found such that, given random values for S_1 , assignments can be found for V_2 such that, given random values for $S_2, \dots$, assignments can be found for V_m so that, given random values for S_m , the hard constraints are satisfied and the chance constraints are satisfied in the specified fraction of all possible scenarios. The solution of an m -stage stochastic CSP is represented by means of a *policy tree* [18]. A policy tree is a set of decisions where each path represents a different possible scenario and the values assigned to decision variables in this scenario. Let $\mathcal{S}$ denote the space of policy trees representing all the solutions of a stochastic CSP. We may be interested in finding a feasible solution, i.e. a policy tree $s \in \mathcal{S}$, that maximizes the value of a given objective function $f(\cdot)$ over the stochastic variables S (edges of the policy tree) and over a subset $\hat{V} \subseteq V$ of the decision variables (nodes in the policy tree). A *Stochastic COP* is then defined in general as $\max_{s \in \mathcal{S}} f(s)$. In [19] a policy-based view of stochastic constraint programs is proposed. Such an approach has been further investigated in [3]. An alternative semantics for stochastic constraint programs comes from a scenario-based view [4,18]: this solution method consists in generating a scenario-tree that incorporates all possible realizations of discrete stochastic variables into the model explicitly.

3 Value of Stochastic Solutions

Let Ξ be a discrete stochastic (vector) variable whose realizations correspond to the various scenarios. Recall that in the policy-based view of stochastic CP a scenario is a set of edges in the policy tree connecting the root to a leaf. Define

$$P = \max_{x \in S} z(x, \xi)$$

as the optimization problem associated with one particular scenario $\xi \in \Xi$, where S is a *finite* set, and $z(x, \xi)$ is a real valued function of two (vector) variables x and ξ . Note that in what follows the discussion is dual for minimization problems. In order to simplify the notation used, we will here use the same notation for referring to a problem and to the value of its optimal solution. The meaning will be made clear by the context.

The function $z(x, \xi)$ can be seen as a payoff table that for a given decision x provides the profit with respect to a given scenario ξ having probability $\Pr\{\xi\}$. We may be then interested in computing the optimal solution value to the *recourse problem* [4] $RP(P) = \max_{x \in S} \sum_{\Xi} \Pr\{\xi\} z(x, \xi)$. This can be expressed, by using the expectation operator $\mathbb{E}$, as

$$RP(P) = \max_{x \in S} \mathbb{E}z(x, \Xi),$$

with an optimal solution x^* .

The *expected value problem*, the deterministic problem obtained by replacing all the stochastic (vector) variables by their expected values, is defined as

$$EV(P) = \max_{x \in S} z(x, \mathbb{E}[\Xi]).$$

Let us denote by $\hat{x}$ an optimal solution of the expected value problem, called the *expected value solution*. Anyone familiar with Stochastic Programming or realizing that uncertainty is a fact of life would feel a little insecure about taking decision $\hat{x}$. Indeed, unless such a decision is independent of Ξ , there is no reason to believe that this decision is even close to the optimal solution of the recourse problem.

For any stochastic maximization (minimization) program, under the assumptions that (i) $z(x, \Xi)$, the profit function, is a concave¹ (convex) function of Ξ and (ii) $\max_{x \in S} z(x, \Xi)$ ($\min_{x \in S} z(x, \Xi)$) exists for all Ξ ,

Proposition 1. $EV(P) - RP(P) \geq 0$ ($EV(P) - RP(P) \leq 0$).

Proof. A proof is given in [2].

It directly follows that $EV(P) \geq RP(P)$ ($EV(P) \leq RP(P)$). We will base our cost-based filtering strategies on this inequality.² Assumption (i) restricts the form of the cost function. As witnessed by much of the Stochastic Programming literature [4,11], many real life applications exhibit such a behavior in the profit (cost) function. Nevertheless, it is often possible to encounter stochastic constraint programs whose objective exhibits a generalized non-convex dependence on the stochastic variables. Note that, although the classical Jensen (Proposition 1) and Edmundson-Madansky type bounds [4], which we will employ in the following sections, or their extensions are generally not available for such problems, tight bounds may still be constructed under mild regularity conditions as discussed in [13]. Assumption (ii) states that Proposition 1 provides a valid bound only when a feasible solution exists and its existence is not affected by the distribution of the stochastic variables. Intuitively, this means that nothing can be inferred by using Proposition 1 if $EV(P)$ is infeasible or, clearly, if $RP(P)$ is infeasible. Assumption (ii) may be violated in problems where

¹ A real-valued function f is *convex* if for any x_1, x_2 in the domain and any $\lambda \in [0, 1]$, $\lambda f(x_1) + (1 - \lambda)f(x_2) \geq f(\lambda x_1 + (1 - \lambda)x_2)$ [5]. f is *concave* if $-f$ is convex.

² Other inequalities are discussed in [4], pp. 140–141. Effective relaxations can be also built on these other inequalities.

chance-constraints appear. We will not discuss how to handle generic chance-constraints and how to produce deterministic equivalent reformulations for them in EV(P): the reader may refer to [6]. In this work we will consider only examples of stochastic COPs that always satisfy assumptions (i) and (ii). In particular, to comply with assumption (ii), we will consider problems for which a feasible solution always exists and for which the chance-constraints are “hard” ($\theta = 1$). Note that “hard” chance-constraints in RP(P) become deterministic in EV(P).

4 Global Optimization Chance-Constraints

Solving stochastic constraint programs is computationally a challenging task. In [19], the computational complexity — membership in PSPACE — of these models is discussed. In [18], the authors proposed a standard way of compiling down these models into conventional (non-stochastic) CP models that can be solved by any available commercial software. This approach employs a scenario-based [4] modelling strategy for representing stochastic variables. Of course this approach has a price since the number of scenarios that need to be considered in order to fully represent the problem grows exponentially with the number of decision stages in the problem. A possible way to overcome this difficulty is to reduce the number of scenarios considered by sampling them, but this obviously affects the completeness of the model. Another possibility consists instead in developing specialized and efficient filtering strategies. For this purpose *global chance-constraints* have been proposed in [16]. These constraints differ from conventional global constraints in the fact that they represent relations among a non-fixed number of decision variables and stochastic variables.

In this work, by creating a parallel with [7], we present *optimization-oriented global chance-constraints* as a way of enhancing the solving process of stochastic constraint programs. Conventional optimization-oriented global constraints perform cost-based filtering by encapsulating in global constraints optimization components representing suitable relaxations of the constraint itself. Similarly optimization-oriented global chance-constraints also encapsulate suitable relaxations of the constraint considered, but in contrast to conventional optimization-oriented global constraints this relaxation may involve stochastic variables.

A *global optimization chance-constraint* provides the same three pieces of information provided by optimization-oriented global constraints. The difference is the fact that in a global optimization chance-constraint we find two stages of relaxations. At the first stage of relaxation, we are mainly involved with the stochastic variables and we exploit well-known inequalities such as the one in Proposition 1 to replace stochastic variables in our stochastic programs with deterministic quantities and to yield a valid relaxation that is a deterministic problem. This deterministic problem, however, may still be computationally very challenging (NP-hard in general). Therefore, a second stage of relaxation may be needed to produce a further relaxation that is computationally more tractable. Finally, as we will see, a global optimization chance-constraint may also provide a valid, and possibly good, solution at each node of the search tree.

Objective: $\max \left\{ \sum_{i=1}^k r_i X_i - c \mathbb{E} \left[\sum_{i=1}^k \mathcal{W}_i X_i - q \right]^+ \right\}$ Decision variables: (1) $X_i \in \{0, 1\} \quad \forall i \in 1, \dots, k$ Stochastic variables: $\mathcal{W}_i \rightarrow$ item i weight

Fig. 1. RP(SSKP). Note that $[y]^+ = \max\{y, 0\}$.

In this section and in the following ones we will refer to a running example and we will employ the following problem to better understand the concepts explained. Consider the *Static Stochastic Knapsack Problem* (SSKP) [12]: a subset of k items has to be chosen, given a knapsack of size q into which to fit the items. Each item i has an expected reward of r_i . The size $\mathcal{W}_i$ of each item is not known at the time the decision has to be made, but we assume that the decision maker has an estimate of the probability distribution of $\overline{\mathcal{W}} = (\mathcal{W}_1, \dots, \mathcal{W}_k)$. A per unit penalty of c has to be paid for exceeding the capacity of the knapsack. By modeling this problem as a one-stage Stochastic COP, the recourse problem RP(SSKP) can be formulated as shown in Fig. 1. The objective function maximizes the trade-off between the reward brought by the objects selected in the knapsack (those for which the binary decision variable X_i is set to 1) and the expected penalty paid for buying additional capacity units in those scenarios in which the low cost capacity q is not sufficient.

Example 1. Consider 5 items, item rewards r_i are $\{10, 15, 20, 5, 25\}$. The discrete probability distribution functions $f(i)$ for the weight of item $i = 1, \dots, 5$ are respectively, $f(1) = \{10(0.5), 8(0.5)\}$, $f(2) = \{10(0.5), 12(0.5)\}$, $f(3) = \{9(0.5), 13(0.5)\}$, $f(4) = \{4(0.5), 6(0.5)\}$, $f(5) = \{12(0.5), 15(0.5)\}$. The figures in parenthesis represent the probability that an item takes a certain weight. The other problem parameters are $c = 2$, $q = 30$. The optimal solution of the recourse problem selects items $\{2, 3, 5\}$ and has a value of RP(SSKP)=49.

This solution can be obtained by solving a deterministic equivalent conventional constraint program obtained by employing a scenario-based representation [18]. Let $\mathcal{W}_i^j$ be the realized weight of object i in scenario j . We hand-crafted a deterministic equivalent model DetEquiv(RP(SSKP)) for RP(SSKP) following the guidelines in [18]. This model is shown in Fig. 2. Constraint (1) states that Z_j , total excess weight in scenario j , must be greater than the sum of the weights of the objects selected in this scenario minus the low cost capacity q . Constraint (2) restricts the decision variables X_i to be binary. X_i is equal to 1 iff item i is selected in the knapsack. Constraint (3) fixes an upper bound for Z_j ; this upper bound is the sum of the weights of all the k objects in scenario j . The objective function maximizes the trade-off between the total reward brought by the objects selected and the sum of penalty costs — weighted by the respective scenario probability — paid for those scenarios where the low cost capacity q is not sufficient.

Objective:		
$\max \left\{ \sum_{i=0}^k r_i X_i - c \left[\sum_{j=1}^n Z_j \Pr\{j\} \right] \right\}$		
Constraints:		
(1)	$Z_j \geq \sum_{i=1}^k \mathcal{W}_i^j X_i - q$	$\forall j \in 1, \dots, n$
Decision variables:		
(2)	$X_i \in \{0, 1\}$	$\forall i \in 1, \dots, k$
(3)	$Z_j \in [0, \sum_{i=1}^k \mathcal{W}_i^j]$	$\forall j \in 1, \dots, n$

Fig. 2. DetEquiv(RP(SSKP)). $\Pr\{j\}$ is the probability of scenario $j \in \{1, \dots, n\}$. Note that $\sum_{j=1}^n \Pr\{j\} = 1$.

4.1 Expectation-Based Relaxation for Stochastic Variables

The first step in our cost-based filtering strategy consists in applying a relaxation involving the stochastic variables. By applying Proposition 1, if the profit (respectively cost for minimization problems) function satisfies the two assumptions discussed, an upper (lower) bound for the cost of an optimal solution to RP(P) can be obtained by solving EV(P), that is the deterministic problem in which all the stochastic variables are replaced by their respective expected values.

Lemma 1. *The profit function for RP(SSKP) is concave in $\overline{\mathcal{W}}$.*

Proof. When proving concavity w.r.t. $\overline{\mathcal{W}}$ we can ignore the constant term $\sum_{i=1}^k r_i X_i$. What remains is $f(\overline{\mathcal{W}}) = -c\mathbb{E} \left[\overline{\mathcal{W}}^T \cdot \overline{\mathcal{X}} - q \right]^+$, where “.” is the inner product and $\overline{\mathcal{W}}^T$ is vector $\overline{\mathcal{W}}$ transposed. We now prove that $-f(\overline{\mathcal{W}}) = c\mathbb{E} \left[\overline{\mathcal{W}}^T \cdot \overline{\mathcal{X}} - q \right]^+$ is convex in $\overline{\mathcal{W}}$. By recalling that a maximum of convex functions is convex [5], this function is clearly convex w.r.t. each element of vector $\overline{\mathcal{W}}$ and it is therefore convex in $\overline{\mathcal{W}}$. This implies that $-f$ is concave in $\overline{\mathcal{W}}$.

Obviously, in RP(SSKP), it is always possible to find a feasible assignment for decision variables, therefore both the assumptions are satisfied for this problem. The expected value problem EV(SSKP) can be obtained by replacing every random variable $\mathcal{W}_i$ in RP(SSKP) with the respective expected value $\mathbb{E}[\mathcal{W}_i]$, thus obtaining a fully deterministic model.

Example 2. Here we solve the problem where the weights of the objects are deterministic and equal to the respective expected weights³: $\lfloor \mathbb{E}[f(1)] \rfloor = 9$, $\lfloor \mathbb{E}[f(2)] \rfloor = 11$, $\lfloor \mathbb{E}[f(3)] \rfloor = 11$, $\lfloor \mathbb{E}[f(4)] \rfloor = 5$, $\lfloor \mathbb{E}[f(5)] \rfloor = 13$. This problem provides the first two pieces of information needed by our cost-based filtering method, that is (a) the optimal solution of the relaxed problem and (b) the optimal value of this solution, which represents, according to Proposition 1, an upper bound for the original problem objective function. In our running example this solution selects items 3, 4, 5 and has a value of $\text{EV}(\text{SSKP}) = 50$.

³ As this is a maximization problem, the expected weight of each object is rounded down to the nearest integer ($\lfloor \cdot \rfloor$) in order to keep the bound provided by the relaxation optimistic.

4.2 Relaxing the Expected Value Problem

It should be noted that, although the expected value problem is easier than the recourse problem, it may still be difficult to solve (NP-hard). For this reason we can further relax the expected value problem in order to obtain a valid bound by solving an easier problem. Let $R(EV(P))$ be a generic relaxation of $EV(P)$. Then for a maximization problem $EV(P) \leq R(EV(P))$ holds, therefore $R(EV(P))$ provides a valid bound for the recourse problem.

In SSKP, for instance, instead of solving to optimality the deterministic (NP-Complete) knapsack problem obtained for the expected value scenario, we may instead solve in linear time its continuous relaxation, thus obtaining Dantzig's upper bound, $DUB(EV(SSKP))$ [15]. $DUB(EV(SSKP)) \geq EV(SSKP)$ therefore $DUB(EV(SSKP)) \geq RP(SSKP)$. $DUB(EV(SSKP))$ is a valid upper bound for our recourse problem.

Example 3. To obtain $DUB(EV(SSKP))$ we order items for profit over expected weight: $\{25/13, 20/11, 15/11, 10/9, 5/5\}$, and we insert items until the first that does not fit completely into the remaining knapsack capacity. Of this last item we take a fraction of the profit proportional to the capacity available. Therefore $DUB(EV(SSKP)) = 25 + 20 + (6 * 15/11) = 53.18$.

Obviously at any node of the search tree it is possible to solve the expected value problem taking into account decision variables already assigned. The bound obtained can be used to exclude part of the tree that cannot lead to a better solution.

In [7] the authors discuss filtering strategies based on reduced costs (RC). As we shall see in the next section a similar technique can be adopted for SSKP, provided that an efficient way of obtaining bounds is available for the expected value problem.

4.3 Cost-Based Filtering

In order to perform cost-based filtering, as in RC-based filtering, we need a *gradient function* $\mathbf{grad}(V, v)$, which returns for each variable-value pair (V, v) an optimistic evaluation of the profit obtained if v is assigned to V . This function is obviously problem dependent, but regardless of the strategy adopted in the former section — i.e. whether we are using a relaxation for the expected value problem or solving this problem to optimality — it is possible to specify it and use it to filter provably suboptimal values. In what follows we present a gradient function for SSKP. At each node of the search tree, in order to compute this function, we use a continuous relaxation of the expected value problem similar to the one proposed by Dantzig for the well-known 0-1 Knapsack Problem [15]. We will now define the gradient function for SSKP by reasoning on the expected value problem. Assume that a partial assignment for decision variables is given. Let K be the set of all the items in the problem, $|K| = k$. Let S be the set of items for which a decision has been fixed, with $|S| < k$. Let q^* be the sum of the expected weights of the elements in S that are part of the knapsack. The profit

$\bar{r}$ associated with this assignment is equal to the sum of the profits of the items in the knapsack minus the eventual expected penalty cost $c(q^* - q)$, if $q - q^*$ is negative. Now we consider an element $i \in K/S$. There are two possible options: taking it into the knapsack or not. If we take it, we increase the profit by r_i minus any eventual expected penalty cost we pay if the expected residual capacity is or becomes negative. Finally for every other element in K/S we check if the balance between its profit and the eventual expected penalty gives an overall positive profit and, if so, we add it to the knapsack. This procedure requires at most $O(k)$ steps for each element for which a decision has not yet been taken, therefore it can be applied at each node of the search tree to compute a valid upper bound associated with a certain decision on an item, which therefore may be filtered if suboptimal.

Example 4. We now consider the case in which items 2 and 3 have been selected in the knapsack and item 4 is not selected. We still have to decide on items 1 and 5. The total capacity used is $c^* = 11 + 11 = 22$. The profit $\bar{r}$ brought by items 2 and 3 is 35. We consider the set of the remaining items for which a decision must be taken, $K/S \equiv \{1, 5\}$. Let us reason on item 1: this is a critical item, in fact if taken in the knapsack it will use more capacity than the residual $30 - 22 = 8$ units. If we consider the option of taking this item, then the expected profit is $\bar{r}_1 = 10 - 2 * (30 - 22 - 9) = 8$, there is no more residual capacity and item 5 is therefore excluded in the bound computation since $25 - 4 * 13 \leq 0$. The computed bound is $35 + 8 = 43$. The reasoning is similar for item 5. If we consider the option of taking this item, then the expected profit is $\bar{r}_5 = 25 - 2 * (30 - 22 - 13) = 15$, there is no more residual capacity and item 1 is therefore excluded in the bound computation since $10 - 4 * 9 \leq 0$. The computed bound is $35 + 15 = 50$. Assume now that the current best solution has a value of 46, corresponding to a knapsack that contains elements 3, 4 and 5: then element 1 can be excluded from the knapsack.

Obviously, as discussed in [7] the information provided by the relaxed model (EV(P)), i.e. expected weights, gradient function etc., can be also used to define search strategies. For instance in SSKP we may branch on variables according to a decreasing profit over expected weight heuristic, or selecting the one for which the chosen gradient function gives the most promising value.

4.4 Finding Good Feasible Solutions

In CP, it is critical, in order to achieve efficiency, to quickly obtain a good feasible solution so that cost-based filtering can prune provably suboptimal nodes as early as possible. In Stochastic COPs the EV(P) solution can be often used as a good starting solution in the search process. If such a solution is feasible with respect to RP(P) — in our examples assumption (ii) guarantees this — we can easily compute EEV(P), that is *the expected result of using the EV(P) solution in the recourse problem RP(P)*. Furthermore, at every node of the search tree it is possible to adopt a variable fixing strategy and compute the EV(P) solution with respect to such a node, that is the best possible EV(P) solution incorporating the

partial decisions represented by the given node of the search tree. This provides a full assignment for decision variables in $RP(P)$ at each point of the search. By using this assignment, we can again easily compute $EEV(P)$. In this case $EEV(P)$ is the cost of a feasible, and possibly good, solution for $RP(P)$ incorporating the partial assignment identified by the current node explored in the search tree.

Example 5. In our SSKP example the solution of the expected value problem, $EV(SSKP)$, selects items 3, 4 and 5 in the optimum knapsack. This solution is clearly feasible for $RP(SSKP)$. We can therefore compute $EEV(SSKP) = 46$. This is, of course, a good lower bound for the objective function value.

5 Experimental Results

In this section we report our computational experience on two one-stage stochastic COPs, the SSKP and the Stochastic Sequencing with Release Times and Deadlines (SSEQ). In our experiments we used Choco 1.2, an open source solver written in Java [14]. We ran our experiments on an Intel(R) Centrino(TM) CPU 1.50GHz with 2Gb of RAM.

5.1 Static Stochastic Knapsack Problem

We created a Choco CP model for $DetEquiv(RP(SSKP))$, and we implemented for it a global optimization chance-constraint incorporating the filtering discussed in the former sections. To recall, within this constraint at each node of the search tree the stochastic variables are replaced by their respective expected values. Then, after fixing decision variables according to the partial solution associated with the given search tree node, $EV(SSKP)$ is solved and the bound obtained is used to prune suboptimal parts of the search tree. Furthermore cost-based filtering is performed as explained in Section 4.3. Finally $EEV(P)$, the *expected result of using the $EV(P)$ solution in the recourse problem*, is computed at each node of the search tree and used as a valid lower bound (profit of a feasible solution). In fact $RP(SSKP)$ satisfies assumption (ii) for Proposition 1, therefore the solution of $EV(SSKP)$ is feasible for $RP(SSKP)$.

In our experiments we adopted a randomly generated test bed similar to the one proposed in [12]. There are three sets of instances considered: the first set has $k = 10$, the second set has $k = 15$ and the third has $k = 20$ items. For all the instances, item random weights, W_i , from which scenarios are generated, are independent and normally distributed with probability distribution function $N(\mu_i, \sigma_i)$. The expected weights, μ_i , are generated from the uniform (20,30) distribution, and the weight standard deviations, σ_i , are generated from the uniform (5,10) distribution. Rewards r_i are generated from the uniform (10,20) distribution. The per unit penalty is $c = 4$, while the available low cost capacity is $q = 250$ for 20 items, $q = 187$ for 15 items, and $q = 125$ for 10 items. We randomly generated, using simple random sampling, sets of scenarios having different sizes: {100, 300, 500, 1000}. Scenarios are equally likely. The variable selection heuristic branches first on items with lower profit over expected weight

Table 1. Experimental results for SSKP. Comparison between a pure SCP approach (SCP) and an SCP model enhanced with optimization-oriented global-chance constraints (SCP-OO), times are in seconds. In each line we indicated in bold the best performance in terms of run time and explored nodes.

Instance		Time		Nodes	
k	Scenarios	SCP	SCP-OO	SCP	SCP-OO
10	100	0.4	0.5	916	100
10	300	1.3	0.5	2630	59
10	500	2.4	0.2	4237	8
10	1000	7.2	2.4	6227	120
15	100	2.5	0.3	4577	11
15	300	15	2.3	10408	252
15	500	33	1.1	9982	75
15	1000	150	6.3	16957	222
20	100	70	10	102878	1024
20	300	250	13	85073	953
20	500	860	9.5	129715	225
20	1000	3200	240	134230	7962

ratio. The value selection tries first not to insert an item into the knapsack. In Table 1 we report our computational results. In all the instances considered our approach outperforms a pure SCP model in terms of explored nodes: the maximum improvement reaches a factor of 576.5. Run times are also shorter in our approach for almost all the instances. An exception is observed for the smallest instance, where the cost of filtering domains is not compensated by the payoff in terms of reduction of the search space. The maximum speed-up observed for run times reaches a factor of 90.5.

5.2 Stochastic Sequencing with Release Times and Deadlines

We consider a specific sequencing problem similar to the one considered by Hooker et. al [9]. Garey and Johnson [8] also mention this problem in their list of NP-hard problems and they refer to it as “Sequencing with Release Times and Deadlines” (SSEQ). An optimization version of this scheduling problem was also described in [10]. The problem consists in finding a feasible schedule to process a set I of k orders (or jobs) using a set M of n parallel machines. Processing an order $i \in I$ can only begin after the release date r_i and must be completed at the latest by the due date d_i . Order i can be processed on any of the machines. The processing time of order $i \in I$ on machine $m \in M$ is P_{im} . The model just described is fully deterministic, but we will now consider a generalization of this problem to the case where some inputs are uncertain. For convenience we will just consider uncertain processing times $\mathcal{P}_{im}$ for order $i \in I$ on machine $m \in M$. Instead of simply finding a feasible plan we now aim to minimize the expected total tardiness of the plan (the deterministic version of this problem is known as “Sequencing to minimize weighted tardiness” [8] and it is NP-hard). A solution for our SSEQ problem consists in an assignment for the jobs on the machines and in a total order between jobs on the same machine. In such a plan, a job will be processed on its release date if no other previous job is still processing,

Objective:	
$\min \left\{ \sum_{i=1}^k \mathbb{E} [C_i - d_i]^+ \right\}$	
Constraints:	
(1) $S_{ab} + S_{ba} \leq 1$	$\forall a, b \in 1, \dots, k, a \neq b$
(2) $X_{am} + X_{bm} \leq S_{ab} + S_{ba} + 1$	$\forall a, b \in 1, \dots, k, a \neq b, \forall m \in 1, \dots, n$
(3) $\sum_{m=1}^n X_{im} = 1$	$\forall i \in 1, \dots, k$
(4) $C_i - \sum_{m=1}^n \mathcal{P}_{im} X_{im} \geq r_i$	$\forall i \in 1, \dots, k$
(5) $S_{ab} = 1 \rightarrow C_b \geq C_a + \sum_{m=1}^n \mathcal{P}_{bm} X_{bm}$	$\forall a, b \in 1, \dots, k, a \neq b$
Decision variables:	
(6) $X_{im} \in \{0, 1\}$	$\forall i \in 1, \dots, k, \forall m \in 1, \dots, n$
(7) $S_{ab} \in \{0, 1\}$	$\forall a, b \in 1, \dots, k, a \neq b$
Stochastic variables:	
$\mathcal{P}_{im}$: processing time of job i on machine m	
Auxiliary variables:	
C_i : stochastic completion time of job i .	

Fig. 3. RP(SSEQ). Note that $[y]^+ = \max\{y, 0\}$. $\mathbb{E}$ denotes the expectation operator.

or as soon as the previous job terminates. The recourse problem RP(SSEQ) can be formulated as a one-stage Stochastic COP. This is shown in Fig. 3.

Decision variable X_{im} takes value 1 iff job i is processed on machine m , decision variable S_{ab} takes value 1 iff job a is processed before job b . Constraints (1) and (2) enforce a total order among jobs on the same machine. Constraint (3) enforces that each job must be processed on one and only one machine. Constraint (4) states that the (stochastic) completion time, C_i , of a job i minus its (stochastic) duration $\mathcal{P}_{im}$ on the machine on which it is processed must be greater than or equal to its release date r_i , where C_i is an auxiliary variable used for simplifying notation. Let $I_m \equiv \{\mathcal{J}_{1m}, \mathcal{J}_{2m}, \dots, \mathcal{J}_{qm}\} \subseteq I$ be the ordered set of jobs assigned to machine m . $\mathcal{C}_{\mathcal{J}_{qm}}$ is defined recursively as $\mathcal{C}_{\mathcal{J}_{qm}} = \max\{r_{\mathcal{J}_{qm}}, \mathcal{C}_{\mathcal{J}_{(q-1)m}}\} + \mathcal{P}_{\mathcal{J}_{qm}m}$, and $\mathcal{C}_{\mathcal{J}_{0m}} = 0$. Constraint (5) states that if two jobs a and b are processed on the same machine and if a is processed before b , that is $S_{ab} = 1$, then the (stochastic) completion time of job a plus the (stochastic) duration of job b on the machine on which it is processed must be less than or equal to the (stochastic) completion time of job b . Finally, the objective function minimizes the sum of the expected tardiness of each job. The tardiness is defined as $\max\{0, C_i - d_i\}$. The cost function to be minimized can easily be proved convex in the random job durations. The expected total tardiness is in fact minimized for n machines. Job completion times on different machines are independent, therefore if we prove convexity for machine $m \in M$, then it directly follows that the cost function of the problem is also convex⁴. The cost function for machine m can be expressed as $\mathbb{E} [\sum_{i \in I_m} (C_i - d_i)^+]$.

Lemma 2. *The expected total tardiness for machine m is convex in the uncertain processing times $\mathcal{P}_{im}$.*

Proof. Maximum of convex functions is convex. $\mathcal{C}_{\mathcal{J}_{1m}} = r_{\mathcal{J}_{1m}} + \mathcal{P}_{\mathcal{J}_{1m}m}$ is convex: it follows that C_i for any $i \in I_m$ is convex, since function “max” is a convex function. Therefore the objective function is convex.

⁴ Note that the sum of convex functions is convex [5].

Objective:	
$\min \left\{ \sum_{i=1}^k \sum_{v=1}^w \Pr\{w\} [C_i^v - d_i]^+ \right\}$	
Constraints:	
(1) $S_{ab} + S_{ba} \leq 1$	$\forall a, b \in 1, \dots, k, a \neq b$
(2) $X_{am} + X_{bm} \leq S_{ab} + S_{ba} + 1$	$\forall a, b \in 1, \dots, k, a \neq b, \forall m \in 1, \dots, n$
(3) $\sum_{m=1}^n X_{im} = 1$	$\forall i \in 1, \dots, k$
and $\forall v \in 1, \dots, w$	
(4) $C_i^v - \sum_{m=1}^n \mathcal{P}_{im}^v x_{im} \geq r_i$	$\forall i \in 1, \dots, k$
(5) $S_{ab} = 1 \rightarrow C_b^v \geq C_a^v + \sum_{m=1}^n \mathcal{P}_{bm}^v X_{bm}$	$\forall a, b \in 1, \dots, k, a \neq b$
Decision variables:	
(6) $X_{im} \in \{0, 1\}$	$\forall i \in 1, \dots, k, \forall m \in 1, \dots, n$
(7) $S_{ab} \in \{0, 1\}$	$\forall a, b \in 1, \dots, k, a \neq b$
(8) $C_i^v \in \{0, \max_{i=1, \dots, k} r_i + \sum_{t=1}^k (\max_{m=1, \dots, n} \pi_{tm}^v)\}$	$\forall i \in 1, \dots, k, \forall v \in 1, \dots, w$

Fig. 4. DetEquiv(RP(SSEQ)). Note that $[y]^+ = \max\{y, 0\}$. $\Pr\{v\}$ is the probability of scenario $v \in \{1, \dots, w\}$. Note that $\sum_{v=1}^w \Pr\{v\} = 1$.

In RP(SSEQ) a feasible solution can be found for any given set of stochastic job lengths, therefore both the assumptions are satisfied for this problem. We hand-crafted a deterministic equivalent model DetEquiv(RP(SSEQ)) shown in Fig. 4 for the RP(SSEQ) following the guidelines of scenario-based approach described in [18]. In this model, $\mathcal{P}_{im}^v$ is the deterministic length of job i on machine m in scenario v and C_i^v is the deterministic completion time of job i in scenario v .

Finally, as discussed for SSKP, we can obtain the expected value problem EV(SSEQ) by replacing every stochastic variable $\mathcal{P}_{im}$ in RP(SSEQ) with the respective expected value $\mathbb{E}[\mathcal{P}_{im}]$. Since all the chance-constraints in RP(SSEQ) are “hard”, they are retained in EV(SSEQ) and they become deterministic.

We implemented DetEquiv(RP(SSEQ)) in Choco and we coded an optimization-oriented global chance-constraint which exploits the expected value problem both in order to generate valid bounds at each node of the search tree and to filter provably suboptimal values from decision variable domains. At each node of the search tree, we consider the associated partial assignment for decision variables X_{im} and S_{ab} and we fix decision variables in EV(SSEQ) according to it. Then we solve EV(SSEQ) with respect to the remaining decision variables that have not been assigned. This provides a lower bound for the cost of a locally optimal solution associated with the node considered. This bound can be used for pruning suboptimal nodes. Furthermore at any given node, after performing variable fixing in EV(SSEQ) for every variable X_{im} and S_{ab} already assigned, all the remaining binary variables X_{im} that have not been assigned yet can be forward checked by fixing the respective value to 1, by solving EV(SSEQ) with this new decision fixed, and by employing the new bound provided.

In order to generate instances for our experiments, we adopted release times, deadlines and deterministic processing times from the first two “hard” instances proposed in [9], the one with 3 jobs and 2 machines and the one with 7 jobs and 3 machines. In each scenario, we generated processing times uniformly distributed in $[1, 2 * J_{im}]$, where J_{im} is the deterministic processing time required for job i on machine m for the instance considered. We considered different number of

Table 2. Experimental Results for SSEQ. Comparison between a pure SCP approach (SCP) and an SCP model enhanced with optimization-oriented global-chance constraints (SCP-OO), times are in seconds. In each line we indicated in bold the best performance in terms of run time and explored nodes.

Instance			Time		Nodes	
Jobs	Machines	Scenarios	SCP	SCP-OO	SCP	SCP-OO
3	2	10	0.3	0.3	203	48
3	2	30	1.3	0.6	701	133
3	2	50	3.2	1.1	927	418
3	2	100	12	3.5	1809	838
7	3	10	180	866	57688	1723
7	3	30	1800	880	186257	5293
7	3	50	3300	1100	212887	6586
7	3	100	14000	1200	277804	8862

scenarios in $\{10, 30, 50, 100\}$. Scenarios are equally likely in terms of probability. The variable selection heuristic branches first on binary decision variables. The value selection tries increasing values in the domain. In Table 2 we report the results observed with and without the improvement brought by our cost-based filtering approach.

It should be noted that in this case, in contrast to the approach employed for SSKP, we only relax stochastic variables and we do not employ a relaxation for the deterministic equivalent problem, which therefore remains NP-hard. Recall that in SSKP we adopted Dantzig’s relaxation to efficiently obtain a bound for the deterministic equivalent problem. A direct consequence of this is that, while in the SSKP example the improvement is significant both in terms of explored nodes and run times for all the instances, in this example the run time improvement starts to be significant (a factor of 11.6) only for the largest instance (7 jobs and 3 machines) and for a high number of scenarios (100 scenarios). This is due to the fact that at every node of the search tree we solve a difficult problem (though far easier than the original stochastic constraint program) to obtain bounds and perform cost-based filtering. In terms of explored nodes, however, we obtain a significant improvement for every instance — the maximum improvement factor is of 32.3 — since the bounds generated are tight.

6 Related Work

This paper extends the original work by Focacci et al. [7] on optimization-oriented global constraints. It also extends the original idea of global chance-constraints [16] to optimization problems. It should be noted that dedicated cost-based filtering techniques for stochastic combinatorial optimization problems have been presented in [17], but these techniques are specialized for inventory control problems, while those here presented can be applied to a wider class of stochastic constraint programs. On the other hand this work also builds on known inequalities borrowed from Stochastic Programming [2,4] usually exploited for relaxing specific classes of stochastic programs and obtaining good bounds or approximate solutions. Nevertheless Stochastic Programming models

are typically formulated as dynamic programs or MIP models. In both cases these bounds are not exploited for filtering decision variable domains as in our approach and they cannot be used for guiding the search.

7 Conclusions

We proposed a novel strategy to performing cost-based filtering for certain classes of stochastic constraint programs, under the assumptions that (i) the objective function is concave or convex in the stochastic variables, and (ii) the existence of a feasible solution is not affected by the distribution of the stochastic variables. This strategy is based on a known inequality borrowed from Stochastic Programming. We applied this technique to two combinatorial optimization problem involving uncertainty from the literature. Our results confirm that orders-of-magnitude improvements in terms of explored nodes and run times can be achieved. In the future, we aim to apply cost-based filtering to multi-stage Stochastic COPs, define strategies to handle generic chance-constraints, which are currently ruled out by our assumptions, and to extend the approach to other valid inequalities such as Edmundson-Madansky [4] or to suitable inequalities for non-convex problems [13]. Finally, we plan to exploit the information provided by optimization-oriented global chance-constraints to define search strategies.

References

1. Apt, K.: Principles of Constraint Programming. Cambridge University Press, Cambridge (2003)
2. Avriel, M., Williams, A.C.: The value of information and stochastic programming. *Operations Research* 18(5), 947–954 (1970)
3. Balafoutis, T., Stergiou, K.: Algorithms for stochastic csp. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 44–58. Springer, Heidelberg (2006)
4. Birge, J.R., Louveaux, F.: Introduction to Stochastic Programming. Springer, New York (1997)
5. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge University Press, Cambridge (2004)
6. Charnes, A., Cooper, W.W.: Deterministic equivalents for optimizing and satisficing under chance constraints. *Operations Research* 11(1), 18–39 (1963)
7. Focacci, F., Lodi, A., Milano, M.: Optimization-oriented global constraints. *Constraints* 7, 351–365 (2002)
8. Garey, M.R., Johnson, D.S.: Computer and Intractability. A guide to the theory of NP-Completeness. Bell Laboratories, Murray Hill, New Jersey (1979)
9. Hooker, J.N., Ottosson, G., Thorsteinsson, E.S., Kim, H.J.: On integrating constraint propagation and linear programming for combinatorial optimization. In: Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI 1999), pp. 136–141. The AAAI Press/MIT Press, Cambridge (1999)
10. Jain, V., Grossmann, I.E.: Algorithms for hybrid milp/cp models for a class of optimization problems. *INFORMS Journal on computing* 13, 258–276 (2001)

11. Kall, P., Wallace, S.W.: Stochastic Programming. John Wiley & Sons, Chichester (1994)
12. Kleywegt, A.J., Shapiro, A., Homem-De-Mello, T.: The sample average approximation method for stochastic discrete optimization. *SIAM Journal of Optimization* 12(2), 479–502 (2001)
13. Kuhn, D.: Generalized bounds for convex multistage stochastic programs. *Lecture Notes in Economics and Mathematical Systems*, vol. 584
14. Laburthe, F.: The OCRE project team. Choco: Implementing a cp kernel. Technical report, Bouygues e-Lab, France (1994)
15. Martello, S., Toth, P.: Knapsack Problems. John Wiley & Sons, NY (1990)
16. Rossi, R., Tarim, S.A., Hnich, B., Prestwich, S.: A global chance-constraint for stochastic inventory systems under service level constraints. *Constraints* 13(4) (2008)
17. Tarim, S.A., Hnich, B., Rossi, R., Prestwich, S.: Cost-based filtering techniques for stochastic inventory control under service level constraints. *Constraints* (forthcoming) (2008)
18. Tarim, S.A., Manandhar, S., Walsh, T.: Stochastic constraint programming: A scenario-based approach. *Constraints* 11(1), 53–80 (2006)
19. Walsh, T.: Stochastic constraint programming. In: *Proceedings of the 15th ECAI. European Conference on Artificial Intelligence*. IOS Press, Amsterdam (2002)

Dichotomic Search Protocols for Constrained Optimization*

Meinolf Sellmann and Serdar Kadioglu

Brown University, Department of Computer Science
115 Waterman Street, P.O. Box 1910, Providence, RI 02912
{serdark,sello}@cs.brown.edu

Abstract. We devise a theoretical model for dichotomic search algorithms for constrained optimization. We show that, within our model, a certain way of choosing the breaking point minimizes both expected as well as worst case performance in a skewed binary search. Furthermore, we show that our protocol is optimal in the expected and in the worst case. Experimental results illustrate performance gains when our protocols are used within the search strategy by Streeter and Smith.

1 Introduction

In Constrained Optimization, there are two fundamental strategies being used to find and prove optimal feasible solutions. By far the most common strategy is branch-and-bound. By recursively partitioning the problem into sub-problems (“branching”), we systematically cover all parts of the search space. When our objective is to minimize costs, we use a relaxation of the problem to compute an under-estimate of the best solution for a given sub-problem (“bounding”). By comparing this bound with the best previously found solution, we may find that a given sub-problem cannot contain improving solutions, which allows us to discard (or “prune”) the sub-problem from further consideration. There exist a variety of relaxations which can be computed efficiently, the most commonly used is linear relaxation.

Obviously, the efficiency of a branch-and-bound approach depends heavily on the quality of the bounds. For many problems, standard relaxation techniques are reasonably accurate or they can be improved to be reasonably accurate, for example by automatically adding valid inequalities to a linear programming formulation. However, for some problems we have grave difficulty in providing lower bounds that can effectively prune the search. In particular, by exploiting constraint filtering techniques, in Constraint Programming (with few exception such as optimization constraints [5]) the primary focus is on feasibility and not on optimality considerations.

* This work was supported by the National Science Foundation through the Career: Cornflower Project (award number 0644113). We would like to express our thanks to three anonymous reviewers for their helpful comments as well as John Hughes, Anna Lysyanskaya, Claire Mathieu, Steven Smith, and Mathew Streeter for supporting this work and some very insightful discussions.

In order to augment a black-box feasibility solver to handle discrete objective functions, there exists a second strategy known as “dichotomic” or “binary search.” Given an initial interval $[l, u]$ in which the optimal objective value must lie, we can compute the optimum by testing whether a cost lower or equal $l + \lfloor (u - l)/2 \rfloor$ can still be achieved. If so, we continue searching recursively in $[l, l + \lfloor (u - l)/2 \rfloor - 1]$. If not, we know the optimum must lie in $[l + \lfloor (u - l)/2 \rfloor + 1, u]$. When a query to the feasibility solver incurs a cost of T , using classic binary search we can compute the optimum in time $O(T \log(u - l))$.

An implicit assumption in dichotomic search is that positive trials incur the same costs as negative trials. However, based on our empirical knowledge from phase transition experiments [2,9,11,14] we expect that negative trials, where we prove that no better solution exists, are generally more costly than positive trials, where we only need to find one improving solution.

Assume that we are trying to minimize costs within the interval $[0, 100]$, and the true minimum is (seemingly conveniently) 50. A classic binary search hits the optimum immediately, and then attempts to find solutions with objective lower or equal 24, 37, 43, 46, 48, and 49. While we need to consider the bound 49 in any case to prove optimality of 50, given that a proof of unsatisfiability may be costly, it is unfortunate that binary search considers a rather large number of almost satisfiable instances before 49.

To avoid this situation, we could of course start with an upper bound of 100, and whenever we find a solution with value v only require that from then on we are only interested in solutions with objective value $v - 1$ or lower (see for example the minimization goal in Ilog CP Solver). The downside of this strategy is that we may end up making very slow progress in finding improving solutions.

Our objective is therefore to devise a strategy that allows fast upper bound improvement while avoiding as much as possible costly proofs of unsatisfiability. In particular, we consider *skewed binary searches* [1] where we do not split the remaining objective interval in half but according to a given ratio a . In our example above, assume we use $a = 0.6$ to organize our dichotomic search. Then, we consider 60, 35, 49, 55, 52, 50. Compared to classic binary search, we see that this skewed search considers a number of almost infeasible problems instead of almost feasible problems. In Figure 1 we illustrate the costs of classic binary search and the skewed binary search in a model where a negative trial costs of a factor $c \geq 1$ more than a positive trial.

Based on the community’s empirical experience on typical runtime over constrainedness, we expect that finding near-optimal solutions is often significantly easier than proving optimality/infeasibility. In Figure 2 we sketch the two dichotomic searches when assuming a typical curve describing the cost of finding a feasible solution or proving infeasibility for a given upper bound on the objective with the typical easy–hard–less-hard regions (when considering subproblems with increasing constrainedness in the sketch from right to left).

In this paper, we provide dichotomic search protocols for such skewed search problems. In particular, we consider the theoretical model where failures incur costs a factor $c \geq 1$ more than positive trials. For this model, we devise a

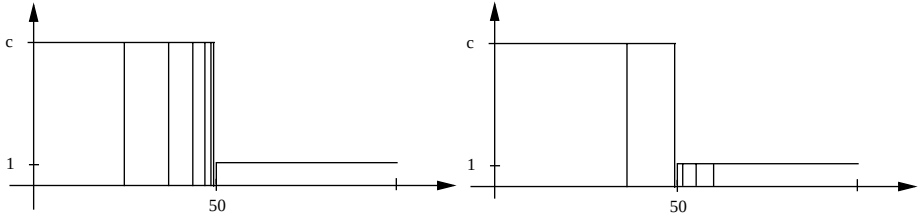


Fig. 1. Dichotomic search for the optimum 50 in the interval $[0,100]$ when the cost of a negative trial is c and the cost for a positive trial is 1. The left picture illustrates the costs of a classic binary search, the right the costs of a skewed search.

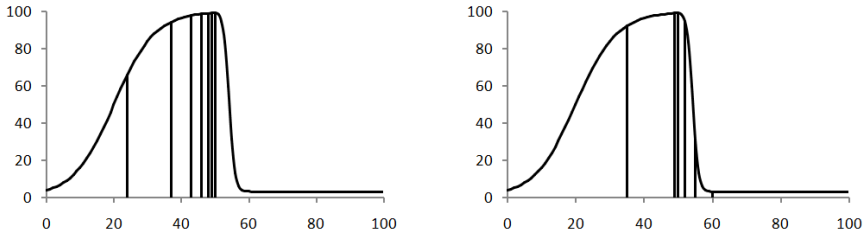


Fig. 2. Dichotomic search for the optimum 50 in the interval $[0,100]$ when the cost of trials follows a typical easy-hard-less-hard pattern. The left picture illustrates the costs of a classic binary search, the right the costs of a skewed search.

provably optimal dichotomic search protocol. We then exploit this protocol in a heuristic algorithm which integrates dichotomic search and restarted branch-and-bound. Experimental results on weighted quasi-group and weighted magic square problems illustrate the performance improvements achieved by the new algorithm.

2 Skewed Binary Search

We consider the following theoretical model.

Definition 1. Given a search-interval $\{l, \dots, u\}$ and a function $f : \{l, \dots, u\} \rightarrow \{0, 1\}$ such that $f(x) = 1 \Rightarrow f(x+1) = 1 \forall l \leq x < u$, we call the problem of finding $y = \min\{x \in \{l, \dots, u\} \mid f(x) = 1\}$ a dichotomic or binary search problem. We call the test whether $f(x) = 1$ for some $x \in \{l, \dots, u\}$ a trial at x . A trial at x is called positive when $f(x) = 1$, otherwise it is called negative or a failure. If the cost of a negative trial is c times the cost of a positive trial for some $c \geq 1$, we call c the bias. A binary search problem is called skewed when $c > 1$. An algorithm that makes trials at x to continue its search in $\{x+1, \dots, u\}$ in case of a failure and in $\{l, \dots, x-1\}$ in case of a positive trial is called a (skewed) dichotomic search or a (skewed) binary search. In the case that the search considers trials $x = l + \lfloor a(u-l) \rfloor$ for some constant $a \in [0, 1]$, we call a the balance of the search.

Theorem 1. *When we assume a uniform distribution of optima in the given interval, the expected effort for a skewed binary search with bias $c \geq 1$ is minimized when setting the balance $a \in [0.5, 1)$ such that*

$$a^c + a = 1.$$

Proof. Let us assume our search interval has length $n \in \mathbb{N}$. According to [1,15,19], the expected search cost in a skewed binary tree with balance a is in $\Theta(f(a))$ with¹

$$f(a) := \frac{a + (1-a)c}{-a \log a - (1-a) \log(1-a)} \log(n) + c.$$

Let us denote with $H(a) := -a \log(a) - (1-a) \log(1-a) \in (0, 1]$ the entropy of $a \in (0, 1)$. Then, for the first derivative of f , it holds

$$f'(a) = \left(\frac{(a + (1-a)c)(\log a - \log(1-a))}{H^2(a)} - \frac{(c-1)}{H(a)} \right) \log(n) \quad (1)$$

$$= (a(\log(a) - \log(1-a)) + c \log(a) + cH(a) - cH(a) + H(a)) \frac{\log(n)}{H^2(a)} \quad (2)$$

$$= \frac{(c \log(a) - \log(1-a))}{H^2(a)} \log(n) \quad (3)$$

We note that the sign of the first derivative depends solely on the sign of $c \log(a) - \log(1-a)$. When a satisfies $a^c + a = 1$ then $f'(a) = 0$. For all lower values for $a \in [0.5, 1)$ the derivative is negative, for all larger values it is positive. Consequently, a with $a^c + a = 1$ marks a global minimum of f in the interval $[0.5, 1)$. $\square$

When our objective is to minimize expected costs under the uniform distribution, the previous theorem tells us how to choose the balance a . The question arises how we should choose a when our goal is to minimize the worst-case performance. Interestingly, we find:

Theorem 2. *The worst-case effort for a skewed binary search with bias $c \geq 1$ is minimized when setting the balance $a \in [0.5, 1)$ such that $a^c + a = 1$.*

Proof. When searching an interval of length $n \in \mathbb{N}$, the worst-case effort of a skewed binary search with balance a is given by the value of the following optimization problem: Maximize $x + cy + c$ such that $a^x(1-a)^y \geq 1/n$, $x, y \geq 0$. We linearize this optimization problem and get

$$\begin{array}{ll} \max & x + cy + c \\ \text{such that} & \log\left(\frac{1}{a}\right)x + \log\left(\frac{1}{1-a}\right)y \leq \log(n) \\ & x, y \geq 0 \end{array}$$

¹ The additional summand c is caused by the fact that we incur costs at nodes and not on branches.

From linear programming theory we know that the maximum is achieved in a corner of this 2-dimensional polytope. The maximum value is thus in

$$\Theta \left(\max \left\{ \frac{1}{\log\left(\frac{1}{a}\right)}, \frac{c}{\log\left(\frac{1}{1-a}\right)} \right\} \log(n) + c \right).$$

Since $-\log(a)$ is strictly monotonically decreasing and $-\log(1-a)$ is strictly monotonically increasing over $[0.5, 1)$, this cost is minimized when choosing the balance $a \in [0.5, 1)$ such that $\frac{1}{\log\left(\frac{1}{a}\right)} = \frac{c}{\log\left(\frac{1}{1-a}\right)}$, which is the same as $\log(1-a) = c \log(a)$, or $1 = a^c + a$. $\square$

Consequently, we conveniently minimize both expected and worst-case time when setting $a \in [0.5, 1)$ such that $a^c + a = 1$. Then, for the runtime it holds:

Lemma 1. *The expected and worst-case costs of a skewed binary search with bias $c \geq 1$ and balance $a \in [0.5, 1)$ such that $a^c + a = 1$ are in $\Theta \left(c \left(\frac{\log(n)}{\log\left(\frac{1}{1-a}\right)} + 1 \right) \right)$.*

Proof. First, note that $a^c + a = 1$ iff $c = \frac{\log(1-a)}{\log(a)}$. Recall from the proof of Theorem 1 that the expected runtime is in $\Theta(f(a))$ with

$$f(a) = \frac{a + (1-a)c}{-a \log a - (1-a) \log(1-a)} \log(n) + c.$$

Then,

$$f(a) = \frac{a \log(a) + (1-a) \log(1-a)}{(-a \log a - (1-a) \log(1-a)) \log(a)} \log(n) + c \quad (4)$$

$$= \frac{\log(n)}{-\log(a)} + c \quad (5)$$

$$= \frac{c}{\log\left(\frac{1}{1-a}\right)} \log(n) + c \quad (6)$$

Regarding the worst-case runtime, recall from the proof of Theorem 2 that $a^c + a = 1$ implies $\frac{1}{\log\left(\frac{1}{a}\right)} = \frac{c}{\log\left(\frac{1}{1-a}\right)}$. Then,

$$\Theta \left(\max \left\{ \frac{1}{\log\left(\frac{1}{a}\right)}, \frac{c}{\log\left(\frac{1}{1-a}\right)} \right\} \log(n) + c \right) = \Theta \left(\frac{c}{\log\left(\frac{1}{1-a}\right)} \log(n) + c \right). \quad \square$$

So we essentially gain a factor of $\log\left(\frac{1}{1-a}\right)$ by skewing our search. The question arises how big this factor is in terms of the given bias c .

Lemma 2. *Given $c \geq 1$ and $a \in [0.5, 1)$ such that $a^c + a = 1$, we have that*

$$\log \left(\frac{1}{1-a} \right) \geq \frac{\log(c)}{2}.$$

Proof. Since $a^c + a = 1$ is equivalent with $c = \frac{\log(1-a)}{\log(a)}$, it is sufficient to show that

$$\left(\frac{1}{1-a}\right)^2 \geq \frac{\log(1-a)}{\log(a)},$$

or equivalently that $(1-a)^{(1-a)^2} - a \geq 0$. Let us define $b := 1-a \in (0, 0.5]$, $x := 1/b \geq 2$, and $g(b) := b^{b^2} + b - 1$. Our claim is then equivalent to showing that

$$b^{b^2} + b - 1 = g(b) \geq 0$$

for all $b \in (0, 0.5]$. Consider the first derivative of g :

$$g'(b) = b^{b^2+1}(1 + 2 \ln(b)) + 1.$$

To show that g is monotonically increasing over $(0, 0.5]$, we show that $g'(b) \geq 0$ for all $b \in (0, 0.5]$. Since $1 + 2 \ln(b) < 0$ for all $b \in (0, 0.5]$, it is sufficient to show that $b(1 + 2 \ln(b)) + 1 \geq 0$, or equivalently, that

$$h(x) := 1 + x - 2 \ln(x) \geq 0 \quad \forall x \geq 2.$$

A simple extremum analysis based on the first and second derivative of h shows that h is convex and takes its unique minimum for $x = 2$. Since $f(2) > 0$, we have shown that $g'(b) \geq 0$ over $(0, 0.5]$, and therefore that g is monotonically increasing over $(0, 0.5]$. However, as g approaches 0 from above, we have

$$\lim_{b \rightarrow 0^+} g(b) = \lim_{b \rightarrow 0^+} b^{b^2} + b - 1 \tag{7}$$

$$= \lim_{b \rightarrow 0^+} e^{b^2 \ln(b)} - 1 \tag{8}$$

$$= e^{\lim_{b \rightarrow 0^+} b^2 \ln(b)} - 1 \tag{9}$$

$$= e^0 - 1 = 0. \tag{10}$$

Consequently, $g(b) \geq 0$ for all $b \in (0, 0.5]$. □

With the help of Lemmas 1 and 2, we get immediately:

Theorem 3. *The expected and worst-case costs of a skewed binary search with bias $c \geq 1$ and balance $a \in [0.5, 1)$ such that $a^c + a = 1$ are in $O\left(c \left(\frac{\log(n)}{\log(c)} + 1\right)\right)$.*

To summarize our findings so far: Given a minimization problem where negative trials cost a factor $c \geq 1$ more than positive ones, we minimize the (expected and worst-case) costs of a skewed binary search by choosing the balance $a \in [0.5, 1)$ such that $a^c + a = 1$. With this setting, we essentially gain an asymptotic factor in $\Omega(\log(c))$.

The question arises whether there are other protocols that could minimize the costs further. For example, one may consider a protocol where the balance is not chosen as a constant for the entire search, but that $a \in [0, 1]$ is set in each iteration according to some function over c and also n , the remaining interval length. The following theorem proves that all other dichotomic search protocols cannot perform asymptotically better.

Theorem 4. *Given an interval with length n , considering the breaking point $a \cdot n$ with $a^c + a = 1$ in a skewed binary search with bias $c \geq 1$ is expected optimal in the O -calculus when we assume a uniform distribution of optima in the given interval.*

Proof. Consider a dichotomic search protocol that selects the next trial according to some function $s(c, n)$. For any given interval length $n \in \mathbb{N}$ and bias $c \geq 1$ we show that the expected time that a skewed search using function s takes is greater or equal $\frac{c}{\log(\frac{1}{1-a})} \log(n) + \frac{c}{2}$, where $a^c + a = 1$. We induce over n . For $n = 1$ the claim is trivially true. Now assume $n > 1$ and that the claim holds for all $m < n$. Denote with $p = s(c, n)$ the current trial point. Given that the chance for a positive trial at p is p/n (and $(n-p)/n$ for a negative trial), and by induction hypothesis, for the expected costs it holds that

$$\text{cost}(s, c, n) \geq \frac{n-p}{n} \left(c + \frac{c \log(n-p)}{\log(\frac{1}{1-a})} + \frac{c}{2} \right) + \frac{p}{n} \left(1 + \frac{c \log(p)}{\log(\frac{1}{1-a})} + \frac{c}{2} \right).$$

With $c = \frac{\log(\frac{1}{1-a})}{\log(\frac{1}{a})}$, it follows

$$\text{cost}(s, c, n) \geq c + \frac{\log(n-p)}{\log(\frac{1}{a})} + \frac{p}{n} \left(1 + \frac{\log(p) - \log(n-p)}{\log(\frac{1}{a})} - c \right) + \frac{c}{2} \quad (11)$$

$$= \frac{1}{\log(\frac{1}{a})} \left(\log\left(\frac{n-p}{1-a}\right) + \frac{p}{n} \log\left(\frac{(1-a)p}{a(n-p)}\right) \right) + \frac{c}{2}. \quad (12)$$

Since we wish to show $\text{cost}(s, c, n) \geq \frac{c \log(n)}{\log(\frac{1}{1-a})} + \frac{c}{2} = \frac{\log(n)}{\log(\frac{1}{a})} + \frac{c}{2}$, it is therefore sufficient to show that

$$\log\left(\frac{n-p}{n(1-a)}\right) + \frac{p}{n} \log\left(\frac{(1-a)p}{a(n-p)}\right) \geq 0,$$

or equivalently that

$$\left(\frac{(1-a)p}{a(n-p)}\right)^{\frac{p}{n}} \geq \frac{n(1-a)}{n-p},$$

or that

$$t(a) := \left(\frac{p}{a}\right)^{\frac{p}{n}} - n \left(\frac{1-a}{n-p}\right)^{\frac{n-p}{n}} \geq 0.$$

For the first and second derivation of t we have

$$t'(a) = \left(\frac{1-a}{n-p}\right)^{\left(\frac{n-p}{n}\right)-1} - \frac{1}{n} \left(\frac{p}{a}\right)^{\frac{p}{n}+1} \quad \text{and}$$

$$t''(a) = \frac{p}{a^2 n} \left(\frac{p}{a}\right)^{\frac{p}{n}} \left(\frac{p}{n} + 1\right) + \frac{1}{n-p} \left(1 - \frac{n-p}{n}\right) \left(\frac{1-a}{n-p}\right)^{\frac{n-p}{n}-2}.$$

Clearly, $t''(a) > 0$ for all $a \in [0.5, 1)$, and therefore t is convex on this interval. Furthermore, $t'(\frac{p}{n}) = 0$ and $t(\frac{p}{n}) = 0$, and therefore $t(a) \geq 0$ over $[0.5, 1)$. $\square$

As a consequence of the previous theorem and Lemma 1, which states that our skewed binary search protocol does not work worse in the worst-case than it does in the expected case, we finally get:

Corollary 1. *Given an interval with length n , considering the breaking point $a \cdot n$ with $a^c + a = 1$ in a skewed binary search with bias $c \geq 1$ is asymptotically optimal in the worst-case.*

3 Skewed Dichotomic Search for Constrained Optimization

The previous theoretical study, while interesting in its own right and applicable in realistic scenarios like the one considered in [1], cannot be exploited directly when considering constrained optimization. This is for various reasons. First of all, as we discussed earlier and illustrated in Figure 2, in optimization practice, failures do not generally incur costs that are a constant factor higher than those of positive trials. Consequently, there is a disconnect between the theoretical model and reality.

The second reason why our protocol is not directly applicable is because, in practice, we do not actually know the factor by which a negative trial is – say, on average – more expensive than a positive trial. We could of course try to estimate such a ratio based on our experience with past trials. However, when the skewed search actually works well we hope to avoid negative trials as best as we can, so the sampling is skewed and there will be very little statistical data to work with. Furthermore, in some cases the lower bounds on the objective that we can compute may be so bad that we may not even strive to find and prove an optimal solution. Instead, our objective may be to compute high quality solutions as quickly as possible.

Finally, in real applications, we may expect that, when a backtracking algorithm finds a new upper bound, there may be other solutions that further improve the objective and can be found quickly when investing only a little more search. Classic branch-and-bound algorithms (to which we will refer as $B+B$), where the current upper bound on the objective is based on the best solution found so far, benefit from such a clustering of good solutions. Note that branch-and-bound can also be parameterized to improve on upper bounds more aggressively. For example, when only an approximately optimal solution is sought, we can set the new upper bound to $(1 - \varepsilon)u$ where $\varepsilon > 0$ and u is the value of the best solution found. Or, following an idea presented in [18], one could set the upper bound for pruning more aggressively based on empirical evidence where the optimal objective may be expected.

3.1 The Streeter-Smith Strategy

To address some of these issues, we follow the work from Streeter and Smith [17] who propose a dichotomic search strategy which considers (potentially incomplete)

Query strategy. $S_3(\beta, \gamma, \rho)$:

1. Initialize $T \leftarrow \frac{1}{\gamma}$, $l \leftarrow 1$, $u \leftarrow U$, $t_l \leftarrow \infty$, and $t_u \leftarrow -\infty$.
2. While $l < u$:
 - (a) If $[l, u - 1] \subseteq [t_l, t_u]$ then set $T \leftarrow \frac{T}{\gamma}$, set $t_l \leftarrow \infty$, and set $t_u \leftarrow -\infty$.
 - (b) Let $u' = u - 1$. If $[l, u']$ and $[t_l, t_u]$ are disjoint (or $t_l = \infty$) then define

$$k = \begin{cases} \lfloor (1 - \beta)l + \beta u' \rfloor & \text{if } (1 - \rho)l > \rho(U - u') \\ \lfloor \beta l + (1 - \beta)u' \rfloor & \text{otherwise;} \end{cases}$$

else define

$$k = \begin{cases} \lfloor (1 - \beta)l + \beta(t_l - 1) \rfloor & \text{if } (1 - \rho)(t_l - l) > \rho(u' - t_u) \\ \lfloor (1 - \beta)u' + \beta(t_u + 1) \rfloor & \text{otherwise.} \end{cases}$$

- (c) Execute the query $\langle k, T \rangle$. If the result is “yes” set $u \leftarrow k$; if the result is “no” set $l \leftarrow k + 1$; and if the result is “timeout” set $t_l \leftarrow \min\{t_l, k\}$ and set $t_u \leftarrow \max\{t_u, k\}$.

Algorithm 1. The Streeter-Smith Strategy

trials with a given fail-limit. They show that their parameterized strategy given in Algorithm 1 achieves an optimal competitive ratio for any fixed set of parameters $0 < \beta \leq 0.5$, $0 < \gamma < 1$, and $0 < \rho < 1$.

Assume we set $\beta = \rho = \gamma = 0.5$. Strategy S_3 then proceeds as follows: It first tries the midpoint of the given interval under some fail-limit. When the trial is inconclusive, the next trial is at $\frac{3}{4}$ of the interval and $\frac{1}{4}$ if the first is also inconclusive. This way, the search points are driven to the borders of the search interval where we expect cheaper trials. If no improved upper and lower bounds are found even for trials at the very border of the interval, the fail-limit is multiplied with $\frac{1}{\gamma}$, and the entire process is repeated. As soon as an improved upper or lower bound is found, the search interval is shrunk accordingly. Note how parameter β shifts the trial point towards the upper bound for lower values of β . Parameter ρ determines the balance how much effort we put on upper-bound rather than lower-bound improvement. In our experiments, negative trials were so costly that the best performance was always achieved by setting $\rho \leftarrow 1$. The parameter γ finally determines how quickly the fail-limit grows. In our experiments, we chose the initial fail-limit as 1000 and $\gamma \leftarrow \frac{2}{3}$. We will refer to this algorithm with the acronym *SS*.

The way how the algorithm proceeds is illustrated in Figure 3. The algorithm sets a fail-limit T and then maintains the current upper and lower bound as well as a time-out interval. The algorithm then performs two interleaved dichotomic searches with bias β , one in the interval $[t, t_l]$, the other in $[t_u, u]$, until the best upper and lower bounds for the given time-limit are achieved. Then, the fail-limit is increased geometrically, and two new dichotomic searchers are initiated.

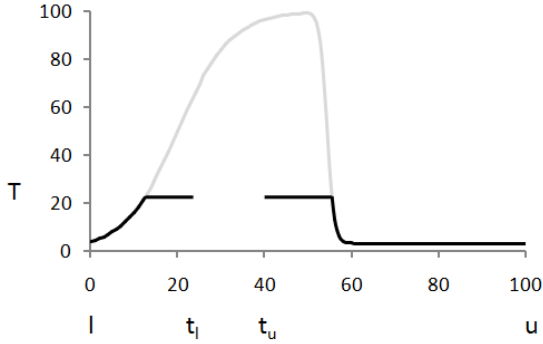


Fig. 3. The Streeter-Smith strategy for constrained optimization on the interval $[1, 100]$

3.2 Parameter Tuning Based on Skewed Binary Search Protocols

While the Streeter-Smith strategy exploits a black-box feasibility solver, the specific solvers that we use for constraint satisfaction are known to benefit from randomization and restarts. Therefore, in a variant of algorithm SS we choose to set the fail-limits in a more continuous fashion than in the Streeter-Smith strategy: After each inconclusive trial, we update the fail-limit linearly to $1000(t+1)$, where t is the number of the last trial that was inconclusive.

With respect to the fact that a backtrack-search may actually yield feasible and potentially improving solutions near a new solution that has been found, we also propose not to stop the search in case of a positive trial. Instead, we choose the next trial point and use this upper bound to prune the search from then on. When we find a new improving solution, we again set the new upper bound aggressively. If we prove unsatisfiability of the new trial or end the search at the initially given fail-limit, we continue in accordance to S_3 .

We observe that the interleaved searches for the best achievable upper and lower bound under some fail-limit depicted in Figure 3 resemble our cost-model from Figure 1. Based on our theoretical study of this cost-model, we are now in a good position to exploit our dichotomic search protocol to tune the parameter β which we propose to choose dynamically for every trial rather than treating it as fixed. Our modified Streeter-Smith strategy works as follows: Whenever we find an improving upper bound, we record how many fails it took within the current restart to produce the new upper bound. Based on these numbers, we keep track of the current average number of failures that it takes to compute a new upper bound. Then, we set the bias c to the ratio of the current fail-limit and this running average, as we expect the search for an improved upper bound to take the running average while a negative trial incurs at most the costs of the current fail-limit.

Of course, for our bias $c \geq 1$, we could approximate $a \in [0.5, 1)$ online. The algorithm will be faster, however, when we pre-compute the corresponding a -values for realistic values of c , say for all natural numbers lower than 1000. In our implementation, we pre-computed values for a corresponding to c which

Query strategy. $SS-lc-skewed(l, u, \delta)$

1. Initialize $f \leftarrow 0$, $s \leftarrow 0$, $avg \leftarrow 1$, $T \leftarrow \delta$, $T' \leftarrow T$, $timeout \leftarrow false$, $l' \leftarrow l$.
2. While $l < u$:
 - (a) Let $u' \leftarrow u - 1$
 - (b) If $timeout = true$ and $l' \geq u'$, then set $timeout \leftarrow false$ and $l' \leftarrow l$, $T \leftarrow T + \delta$, $T' \leftarrow T$
 - (c) Let $\beta \leftarrow 1 - a[T/avg]$
If $timeout = true$, then set $k \leftarrow l' + (u' - l') * \beta$ else set $k \leftarrow l + (u' - l) * \beta$
 - (d) Execute a limited randomized backtrack search with parameters $\langle in : k, in : T', out : failures, out : bestSol \rangle$.
 - (e)
 - i. If the result is “yes”
Set $u \leftarrow bestSol$, $s \leftarrow s + 1$, $f \leftarrow f + failures$, $T' \leftarrow T - failures$, $avg \leftarrow f/s$, $\beta \leftarrow 1 - a[T/avg]$, $k \leftarrow l + (u - l) * \beta$, $timeout \leftarrow false$ and $l' \leftarrow l$. Continue the latest backtrack search with parameters $\langle in : k, in : T', out : failures, out : bestSol \rangle$. Go back to (e)
 - ii. If the result is “no”
Set $l \leftarrow k + 1$, $T \leftarrow T + \delta$, $T' \leftarrow T$, $timeout \leftarrow false$ and $l' \leftarrow l$.
 - iii. If the result is “timeout”
Set $l' \leftarrow k$, $T \leftarrow T + \delta$ and $T' \leftarrow T$, $timeout \leftarrow true$.

Algorithm 2. Skewed Restarted Search

grows exponentially starting at 1 by setting $c_{t+1} := c_t(1 + \varepsilon)$ for some small $\varepsilon > 0$. For a concrete c , we then interpolate the value for a . The parameter β is then dynamically set to $\beta \leftarrow 1 - a$.

Depending on whether we use a specific β or our skewed protocol $\beta = 1 - a$ we refer to this variation of the Streeter-Smith strategy with the acronym $SS-lc$ or $SS-lc-skewed$, respectively. The latter is outlined in Algorithm 2: Given a search interval $[l, u]$, as well as an increment unit δ to update the successive fail-limits T , the average number of failures to compute a new upper bound, avg , is initialized to 1 and the trial point k , is determined by the skewing parameter, $\beta \leftarrow 1 - a[T/avg]$, where $a[T/avg]$ gives the skewing parameter a for bias T/avg . The algorithm performs a search with fail-limit T' , and returns the number of failures along with a new upper bound, $bestSol$, if a solution is ever found. If the search for an improving solution is successful, we decrease the upper bound u , increase the number of successful trials s as well as the total number of failures f , and reset the timeout flag. Then, the backtrack search is continued with the updated values of β , k and T' . If the search proves that no solution with costs lower or equal k exists, we increase the lower bound l , and the fail-limit and reset the time-out flag. If the query result is “timeout”, the timeout flag is set to $true$, a temporary lower bound l' is set to k , and the fail-limit is increased. During the search, if the temporary lower bound meets the upper bound, we reset the timeout flag and restart the search from the lower bound l with a linearly increased fail-limit T . This entire process is repeated until the search interval is consumed. To facilitate the presentation, we only show the modified upper bound improvement here. Just as in the Streeter-Smith strategy, we can

of course interleave the while-loop in step (2) with another skewed search that aims at increasing the lower bound quickly.

4 Numerical Results

In [1], skewed dichotomic search has been thoroughly investigated in the context of sorting. Here, branch-prediction and cache-misses can cause a skewed search to work more efficiently than classic binary search. Experimental results show that skewing the search leads to gains in the order of around 15%. To assess the effect of skewing dichotomic search for constrained optimization, in this section we compare the three algorithms outlined above on two benchmark problems, the weighted quasigroup-completion problem and the weighted magic square problem.

Definition 2. [Weighted Quasigroup Completion] *Given a natural number $n \in \mathbb{N}$, a quasigroup Q on symbols $1, \dots, n$ is an $n \times n$ matrix in which each of the numbers from 1 to n occurs exactly once in each row and in each column. We denote each element of Q by q_{ij} , $i, j \in \{1, 2, \dots, n\}$. n is called the order of the quasigroup. Given profit values $p_{ij} \in \mathbb{N}$, $i, j \in \{1, 2, \dots, n\}$, and a set of tuples $F = \{(k, i, j) \mid 1 \leq i, j, k \leq n\}$, the Weighted Quasigroup Completion problem consists in computing a quasigroup Q such that $q_{ij} = k$ for all $(k, i, j) \in F$ and the value $\min_i \sum_j p_{ij} q_{ij}$ is minimized.*

Definition 3. [Weighted Magic Square Problem] *Given a natural number $n \in \mathbb{N}$, a magic square M of order n is an $n \times n$ matrix in which each of the numbers from 1 to n^2 occurs exactly once and such that the sum of all values in each row, column, and main diagonal are identical. We denote each element of M by m_{ij} , $i, j \in \{1, 2, \dots, n\}$. Given profit values $p_{ij} \in \mathbb{N}$, $i, j \in \{1, 2, \dots, n\}$, the Weighted Magic Square Problem consists in computing a magic square M such that the value $\min_i \sum_j p_{ij} m_{ij}$ is minimized.*

From the perspective of the Constraint Programming (CP), Artificial Intelligence (AI), and Operations Research (OR) communities, combinatorial design problems as the ones given above are interesting as they are easy to state but possess rich structural properties that are also observed in real-world applications such as scheduling, timetabling, and error correcting codes. Thus, the area of combinatorial designs has been a good source of challenge problems for these research communities. In fact, the study of combinatorial design problem instances has pushed the development of new search methods both in terms of systematic and stochastic procedures. For example, the question of the existence and non-existence of certain quasigroups with intricate mathematical properties gives rise to some of the most challenging search problems in the context of automated theorem proving [20]. So-called general purpose model generation programs, used to prove theorems in finite domains, or to produce counterexamples to false conjectures, have been used to solve numerous previously open problems about the existence of quasigroups with specific mathematical properties. Considerable

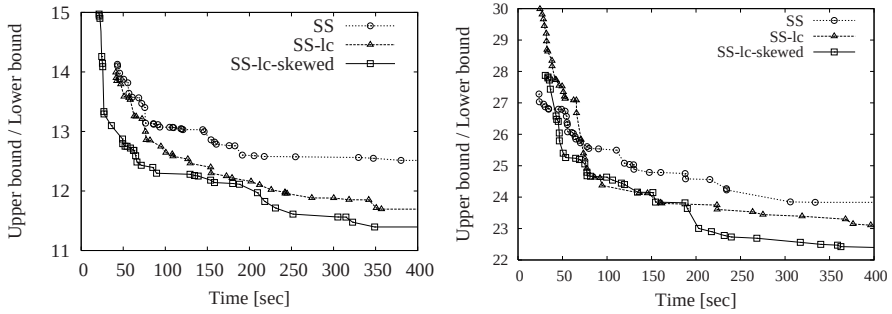


Fig. 4. Comparison of SS, SS-lc, and SS-lc-skewed on weighted magic square problems. We show the average ratio of upper to lower bound for 20 instances with 36 (left) and 64 cells (right) and objective weights p_{ij} for each cell (i, j) chosen uniformly in $[1, 36]$ and $[1, 64]$, respectively.

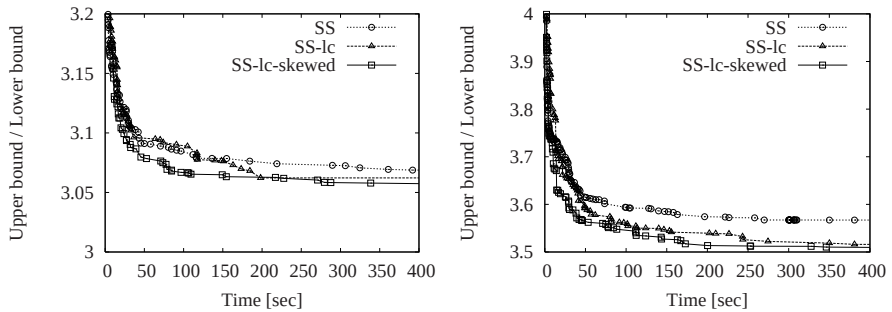


Fig. 5. Comparison of SS, SS-lc, and SS-lc-skewed on weighted quasigroup-completion problems. We show the average ratio of upper to lower bound for 20 instances with 100 (left) and 144 cells (right) and objective weights p_{ij} for each cell (i, j) chosen uniformly in $[1, 100]$ and $[1, 144]$, respectively.

progress has also been made in the understanding of symmetry breaking procedures using benchmark problems based on combinatorial designs [3, 6, 8, 16]. The study of search procedures on benchmarks based on quasigroups has led to the discovery of the non-standard probability distributions that characterize complete (randomized) backtrack search methods, so-called heavy-tailed distributions [7].

For the purpose of testing dichotomic search protocols, the chosen benchmarks are interesting since even finding feasible solutions only is already hard. Moreover, it is a challenge to provide tight bounds on the objective, which is exactly when experts usually revert to a dichotomic search to solve a problem.

Our results are illustrated in Figures 4 and 5. Experiments were run on an AMD Athlon 64 X2 Dual Core Processor 3800+ using Ilog Solver 6.5. For both problems, the CP-models used to solve particular queries are based on the obvious AllDifferent constraints. We fill the squares row by row, whereby the row

to be filled next is determined by the row that currently marks the lower bound on the objective. Within a row, we pick a random variable with minimal domain and assign the lowest value in its domain first. All dichotomic algorithms perform an initial improvement phase where we try to quickly tighten the initial search interval as best as possible. Because of the difficulty to find even feasible solutions only, we did not use local search for this purpose, but a number of short, restarted tree-searches with a tight fail-limit.

The pure B+B approach without restarts often fails to provide feasible solutions within the given time-frame. Consequently, we do not show the results for this method in the figures. We believe that the inferior performance of B+B is due to the fact that it conducts one continuous search that is not restarted. Thus, it gets easily stuck in an area of the search space which does not contain feasible and improving solutions. This trap is particularly big as the CP domain-based lower bounds available to our algorithms are not of very high quality. All other algorithms avoid this problem by exploiting the benefits of a somewhat randomized branching variable selection with frequent restarts.

With respect to the remaining algorithms, we observe that SS-lc works better than the pure Streeter-Smith strategy SS. That is, we find that continuously updating the fail-limit and continuing the search with an improved upper bound after a new solution has been found is beneficial for constrained optimization. In the B+B approach, when it does find a solution, we often find that more improving solutions are found shortly afterwards. We believe that this clustering of solutions in some small subtree is caused by the algorithm having found a desirable partial assignments. Such a clustering is exploited by SS-lc by continuing the search rather than restarting directly.

Finally, we see that SS-lc-skewed leads to an additional improvement. In this method, the fact that the Streeter-Smith strategy considers strict fail-limits allows us to get a good estimate on the search-bias c . As we had hoped, using an optimistic but not overly aggressive way to set new upper bounds based on this estimate of the bias and our theoretically optimal setting allows us to find improving solutions faster and thereby close the gap between upper and lower bound more rapidly.

5 Conclusions

We studied a theoretical model for dichotomic search algorithms and devised a protocol which minimizes both expected as well as worst case performance in a skewed binary search. Furthermore, we showed that our protocol is optimal in the expected and in the worst case. Earlier experiments in the sorting domain by Brodal and Moruz had already shown practical gains from skewing binary search algorithms. In the context of constrained optimization, by exploiting the strategy proposed by Streeter and Smith, dichotomic search can be exploited in practice while skewing the search leads to faster improvements of the upper bound in constrained minimization.

References

1. Brodal, G.S., Moruz, G.: Skewed Binary Search Trees. In: Azar, Y., Erlebach, T. (eds.) ESA 2006. LNCS, vol. 4168, pp. 708–719. Springer, Heidelberg (2006)
2. Crawford, J.A., Auton, L.D.: Experimental Results on the Cross-Over Point in Random 3-SAT. *Artificial Intelligence* 81, 31–57 (1996)
3. Fahle, T., Schamberger, S., Sellmann, M.: Symmetry Breaking. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 93–107. Springer, Heidelberg (2001)
4. Ferré, S., King, R.D.: A dichotomic search algorithm for mining and learning in domain-specific logics. *Fundamenta Informaticae* 66(1–2), 1–32 (2005)
5. 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)
6. Focacci, F., Milano, M.: Global Cut Framework for Removing Symmetries. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 77–92. Springer, Heidelberg (2001)
7. Gomes, C.P., Selman, B., Crato, N., Kautz, H.: Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Automated Reasoning* 24(1–2), 67–100 (2000)
8. Harvey, W.: Symmetry Breaking and the Social Golfer Problem. In: SymCon. (2001)
9. Hogg, T., Huberman, B.A., Williams, C.P.: Phase Transitions and Complexity. *Artificial Intelligence* 81 (1996)
10. Jussien, N., Lhomme, O.: Dynamic Domain Splitting for Numeric CSPs. In: ECAI, pp. 224–228 (1998)
11. Kirkpatrick, S., Selman, B.: Critical Behavior in the Satisfiability of Random Boolean Expressions. *Science* 264, 1297–1301 (1994)
12. Leong, G.T.: Constraint Programming for the Traveling Tournament Problem. Project Thesis, National University of Singapore (2003)
13. Lustig, I., Puget, J.-F.: Constraint Programming. *Encyclopedia of Operations Research and Management Science*, 136–140 (2001)
14. Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Troyansky, L.: Determining computational complexity from characteristic ‘phase transitions’. *Nature* 400, 133–137 (1999)
15. Nievergelt, J., Reingold, E.M.: Binary search trees of bounded balance. In: Annual ACM Symposium on Theory of Computing, pp. 137–142 (1972)
16. Smith, B.: Reducing Symmetry in a Combinatorial Design Problem. In: CPAIOR, pp. 351–360 (2001)
17. Streeter, M., Smith, S.F.: Using Decision Procedures Efficiently for Optimization. In: ICAPS, pp. 312–319 (2007)
18. Wojtaszek, D., Chinneck, J.W.: Faster MIP Solutions via New Node Selection Rules. In: INFORMS (2007)
19. Wong, C.K., Nievergelt, J.: Upper Bounds for the Total Path Length of Binary Trees. *Journal of the ACM* 20(1), 1–6 (1973)
20. Zhang, H.: Specifying Latin Square Problems in Propositional Logic. In: Automated Reasoning and Its Applications. MIT Press, Cambridge (1997)

Length-Lex Bounds Consistency for Knapsack Constraints^{*}

Yuri Malitsky¹, Meinolf Sellmann¹, and Willem-Jan van Hoeve²

¹ Brown University, Department of Computer Science
115 Waterman Street, P.O. Box 1910, Providence, RI 02912

² Carnegie Mellon University, Tepper School of Business
5000 Forbes Avenue, Pittsburgh, PA 15213

Abstract. Recently, a new domain store for set-variables has been proposed which totally orders all values in the domain of a set-variable based on cardinality and lexicography. Traditionally, knapsack constraints have been studied with respect to the required and possible set domain representation. For this domain-store efficient filtering algorithms achieving relaxed and approximated consistency are known. In this work, we study the complexity of achieving length-lex and approximated length-lex bounds consistency. We show that these strengthened levels of consistency can still be achieved in (pseudo-)polynomial time. In addition, we devise heuristic algorithms that work efficiently in practice.

1 Introduction

The constraint programming paradigm is inherently associated with the decomposition of a given problem into its constituting parts as given by the constraints describing it. Based on this decomposition, the standard solution scheme of constraint programming interleaves search and constraint propagation. The latter consists in constraint-specific filtering algorithms that remove inconsistent values from the variable domains. Thus, information from one constraint is propagated to the other constraints solely through variable domains.

While the decomposition allows the solver to apply filtering algorithms that are custom tailored for specific constraints, the main drawback of this scheme is that the information exchange via variable domains inherently weakens the ability to reason about the given problem. Consider for example the well-known example of three binary not-equal constraints working on three variables that all must be assigned either value A or value B. Clearly, no solution exists, but no constraint is able to convey information to the other constraints through the domain store that would make it possible to infer that the problem is infeasible.

Consequently, constraint programming research has looked for ways to strengthen inference by softening the effects of problem decomposition. The concept of global constraints has been an area of very productive research. In our example above, the All-Different constraint represents the conjunction of all three binary not-equal constraints.

^{*} This work was supported by the National Science Foundation through the Career: Cornflower Project (award number 0644113).

Other schemes such as CP-based Lagrangian relaxation [17] or CP-based Bender's decomposition [5,9] have been proposed where constraints exchange dual or no-good information on top of the traditional domain information. Recently, [1] proposed to use a multi-dimensional decision diagram as domain store which allows much more information to be represented and exchanged.

An alternative route is to consider more elaborate domains. For set variables, the traditional domain representation has been to specify a lower bound of *mandatory elements* and an upper bound of *possible elements*. In other words, the domain of a set-variable would be (partially) ordered based on the subset relation. This representation is equivalent to a representation through a number of binary variables.

As an easy way to strengthen the information captured, a variable representing the cardinality of the set could be added to this representation. In [13], Sadler and Gervet apply a *lexicographic* order to the domain of a set-variable. That is, the lower bound and upper bound become the lexicographically smallest and largest set that the variable can be assigned. In [8], Gervet and Van Hentenryck propose a *length-lexicographic* order for the domains of set variables. Domain elements (i.e., the sets that can potentially be assigned to the set-variable) are first ordered by increasing cardinality, while sets with equal cardinality are ordered lexicographically. The benefits of the length-lexicographic representation with respect to the traditional subset bounds representation are that it has less space requirements, unary constraints can be efficiently filtered, and it automatically breaks some symmetry.

In order to fully benefit from new domain representations such as the length-lexicographic set-variable domains, ways must be devised on how to exploit and strengthen the additional information efficiently. For the length-lexicographic representation, a first step in this direction was made by Dooms et al. [4], who presented domain filtering algorithms for *open constraints* where the set-variable representing the scope of an open constraint has a length-lexicographically ordered domain. For many important constraints involving set-variables, however, it remains an open question whether or not it is possible to efficiently filter the domains to some specified level of consistency when they are length-lexicographically ordered. Addressing this question with respect to knapsack constraints is the central topic of this paper.

Knapsack constraints have typically been defined over a number of binary variables which model whether an item is included in or excluded from the knapsack. An alternative way of modeling the constraint is by associating it with one set-variable. For the latter case, filtering algorithms for knapsack constraints achieving bounds-consistency for the subset-domain representation (which is equivalent to achieving generalized arc-consistency (GAC) for the binary variable representation) have been studied in [19,14]. Since determining whether an item must be included or excluded in all feasible improving solutions is naturally NP-hard, the filtering algorithms either run in pseudo-polynomial time [14,19], or they only achieve relaxed or approximated consistency [6,11,14,16].

The length-lex domain representation provides information beyond the classical sets of required and possible items. A priori, it is not clear that the knapsack problem does not become strongly NP-hard when we require that the set of items must be assigned a value within given length-lex bounds. If this was the case, we would no

longer be able to approximate the problem efficiently. In this paper, we show that there still exists a pseudo-polynomial filtering algorithm that establishes length-lex bounds-consistency for knapsacks. We also show how to transform this algorithm into a fully-polynomial approximation scheme and explain how this algorithm can be used to obtain a polynomial-time algorithm which achieves approximated length-lex bounds-consistency for knapsack constraints.

While a complexity analysis of knapsack problems under length-lex constraints is interesting in its own right, in practice we often find that even polynomial-time filtering algorithms can be too heavy to pay-off within a search. For that reason, we also propose efficient heuristic filtering algorithms that communicate and exploit only the cardinality information embedded in the length-lex domain representation. We evaluate those heuristics empirically in the context of multi-knapsack problems where we find that exploiting cardinality information can effectively reduce the size of the search-tree.

2 Domain Representations for Set-Variables

We assume basic familiarity with constraint programming. Here we recall the basic definitions concerning constraint programming with set-variables [7].

A *set-variable* is a variable whose domain values are sets. We assume that the elements originate from a finite universe of elements. Because the number of possible values of a set-variable can be enormous (the size of a power set, in the worst case), one usually represents the domain of a set-variable S by a ‘lower bound’ $L(S)$ and an ‘upper bound’ $U(S)$ on the values that S can take.

A natural representation for the domain of a set-variable is based on the *subset ordering* of the universe. That is, the lower bound $L(S)$ represents all *mandatory* elements, while the upper bound $U(S)$ represents all *possible* elements, i.e., $D(S) = \{s \mid L(S) \subseteq s \subseteq U(S)\}$. We refer to this representation as the *subset* representation. In addition, at times also a lower bound $l(S)$ and upper bound $u(S)$ on the *cardinality* of S are maintained. We can add these two bounds to the subset representation for the domain of S , i.e., $D(S) = [L(S), U(S), l(S), u(S)] = \{s \mid L(S) \subseteq s \subseteq U(S), l(S) \leq |s| \leq u(S)\}$. We refer to this representation as the *subset-cardinality* representation.

An alternate representation is based on the *length-lexicographic* ordering of the universe [8] where the lower bound $L(S)$ represents the smallest set that can be assigned to S , while the upper bound $U(S)$ represents the largest set, i.e., $D(S) = \{s \mid L(S) \leq_{LL} s \leq_{LL} U(S)\}$. Here $\leq_{LL}$ denotes the length-lexicographic order. We refer to this representation as the *length-lex* representation.

Example 1. Let S be a set-variable representing a set of cardinality 2 or 3, in which element 4 is required, while any element from the set $\{1, 2, 3, 4, 5\}$ may appear. Using the subset-cardinality representation, $D(S) = [\{4\}, \{1, 2, 3, 4, 5\}, 2, 3]$. Note that the bounds of this representation do not correspond to feasible assignments for this variable.

Using the length-lex representation, we have $D(S) = [\{1, 4\}, \{3, 4, 5\}]$. Note that in this case, the bounds do correspond to feasible assignments for this variable. However, this representation also allows to assign sets that do not include element 4, for example, sets $\{1, 5\}$ and $\{2, 3\}$ are within the two specified bounds.

As pointed out in the example, a drawback of the length-lex representation is that it does not allow to represent (and exploit) mandatory elements directly. Therefore, we introduce an adapted representation that does allow to capture that information, combining the subset-cardinality and length-lex representations. We propose to maintain a set $R(S)$ of required elements, a set $P(S)$ of possible elements other than $R(S)$, and two sets $L_{lex}(S)$ and $U_{lex}(S)$ that denote the length-lexicographically smallest and largest set that we can add to $R(S)$. In other words, the domain of a set-variable S is represented as $D(S) = [R(S), L_{lex}(S), U_{lex}(S), P(S)] = \{s \mid s = R(S) \cup t, L_{lex}(S) \leq_{LL} t \leq_{LL} U_{lex}(S), t \subseteq P(S)\}$. We finally define the shorthands $L(S) = R(S) \cup L_{lex}(S)$ and $U(S) = R(S) \cup U_{lex}(S)$. We refer to this representation as the length-lex* representation.¹

Example 2. Continuing Example 1, the length-lex* representation gives $R(S) = \{4\}$, $L_{lex}(S) = \{1\}$, $U_{lex}(S) = \{3, 5\}$, and $P(S) = \{1, 2, 3, 5\}$. This defines the following domain (in length-lex order) for S : $\{1, 4\}$, $\{2, 4\}$, $\{3, 4\}$, $\{4, 5\}$, $\{1, 2, 4\}$, $\{1, 3, 4\}$, $\{1, 4, 5\}$, $\{2, 3, 4\}$, $\{2, 4, 5\}$, $\{3, 4, 5\}$.

For constraints involving set-variables, the filtering task is to increase the lower bounds and decrease the upper bounds of the domains. Ideally, we would like to achieve a specified level of consistency, for example bounds consistency or approximated bounds consistency. The respective definitions of these consistencies depend on the applied domain representation. For the length-lex* representation, we have:

Definition 1. Let S denote a set-variable with length-lex* domain $D(S) = [R(S), L_{lex}(S), U_{lex}(S), P(S)]$. A constraint $C(S)$ is called length-lex* bounds consistent iff

- $R(S) = \inf_{\subseteq} \{s \mid s \in D(S) \wedge s \in C(S)\}$,
- $P(S) = \sup_{\subseteq} \{s \mid s \in D(S) \wedge s \in C(S)\} \setminus R(S)$,
- $L_{lex}(S) = \min_{\leq_{LL}} \{s \mid s \in D(S) \wedge s \in C(S)\} \setminus R(S)$,
- $U_{lex}(S) = \max_{\leq_{LL}} \{s \mid s \in D(S) \wedge s \in C(S)\} \setminus R(S)$.

The notion of approximated length-lex* bounds consistency will be presented in Section 5.

3 The Knapsack Problem

In this section we fix notation that we use throughout the paper. First, we define the knapsack constraint $KP(S, p, B, w, C)$ representing the knapsack problem on a set of items denoted by a set-variable S defined on the universe of n items I , a profit vector p such that $p_i > 0$ is the profit of item $i \in I$, a weight vector w such that $w_i > 0$ is the weight of item $i \in I$, a lower bound B on the total profit, and a capacity C on the total weight of the knapsack. More formally, we have

¹ Note that our length-lex* representation is closely related to the ‘hybrid’ domain representation of Sadler and Gervet [13, Definition 3]. The difference is that the hybrid representation separates the lexicographic ordering and the bounds on the cardinality, while we treat them simultaneously using length-lex bounds. Furthermore, the hybrid representation allows the lexicographic bounds and the required elements to share elements, which we forbid.

$$KP(S, p, B, w, C) = \{s \mid s \in D(S), \sum_{i \in s} p_i \geq B, \sum_{i \in s} w_i \leq C\}.$$

As an alternative to the set-variable S , we can represent the set of items to include by a vector of binary variables x indexed by I , i.e., $(i \in S) \Leftrightarrow (x_i = 1)$. It is well-known that achieving generalized arc consistency with respect to the variables x corresponds to achieving subset bounds consistency with respect to the set-variable S [7].

4 Exact Pseudo-polynomial Algorithms for Knapsack

Let us begin by ignoring the lexicographic bounds and assume that we are only given bounds on the number of items to include. The traditional way to achieve GAC for the binary variable representation of a knapsack constraint is by exploiting the dynamic programming (DP) principle. Since the maximum profit is achieved by either excluding or including any particular item, we have the following recursion equation: For all $k \in \mathbb{N}$ and $0 \leq q \leq \sum_i p_i$, the minimum weight $D_{k,q}$ needed to achieve profit q using only items in $\{1, \dots, k\}$ is

$$D_{k,q} \leftarrow \min\{D_{k-1,q}, D_{k-1,q-p_k} + w_k\}.$$

The maximum profit is then easily determined by finding the maximal q such that $D_{n,q} \leq C$. In [14], a filtering algorithm for knapsacks was developed that exploits Trick's well-known filtering technique for dynamic programs [19]. The main idea is to consider a dynamic program as a graph where each cell is a node and each node has exactly those predecessors as given by the dynamic programming recursion equation. For example, the predecessors of $D_{k,q}$ are $D_{k-1,q}$ and $D_{k-1,q-p_k}$. Edges are weighted by associating the edge from predecessor $D_{k-1,q}$ with weight 0 (as it does not cost anything to not include item k), and the edge from predecessor $D_{k-1,q-p_k}$ with weight w_k . This way, the values computed by the DP correspond directly to the shortest path distances from root-node $D_{0,0}$.

Moreover, every path from the root to some node $D_{n,q}$ corresponds directly to a knapsack solution and vice versa. We call such paths "admissible" if and only if $q \geq B$ and their length is lower or equal C . Now, by exploiting shorter path constraint filtering [15], we can shrink the graph by eliminating all edges in the graph which can not be visited by any admissible path. To this end, we introduce an artificial sink-node t that has predecessors $D_{n,q}$ for all $q \geq B$. Then, shortest path distances from the root to all nodes and the corresponding shortest-path distances from all nodes to the sink-node t can be used to determine which edges can still be used on some admissible path [15]. Finally, we infer that item k must (cannot) be included in any feasible and improving knapsack iff for all q the only predecessor of $D_{k,q}$ in the shrunken graph is $D_{k-1,q-p_k}$ ($D_{k-1,q}$) [19].

In the presence of constraints limiting the cardinality to fall into a given interval $[l, u]$, the following analogous dynamic programming recursion solves the knapsack problem with cardinality bounds:

$$W_{k,q,c} \leftarrow \min\{W_{k-1,q,c}, W_{k-1,q-p_k,c-1} + w_k\}.$$

Again, since item k is either included or excluded in the optimal solution, $W_{k,q,c}$ gives the minimum weight needed to achieve a profit of exactly $q \in \mathbb{N}$ when using exactly c of the first k items. And again, based on this recursion the optimum is easily determined by finding the maximum q and $c \in [l, u]$ such that $W_{n,q,c} \leq C$. By introducing an artificial sink t with predecessors $W_{n,q,c}$ for all $q \geq B$ and $c \in [l, u]$, we exploit once more shorter path filtering to determine all items that must or cannot be included by a knapsack constraint augmented by a constraint on the cardinality. Furthermore, we can infer new bounds on the cardinalities by finding the minimum and maximum values for c for which there exists a predecessor $W_{n,q,c}$ of t in the shrunk graph. The total runtime of this algorithm is in $O(n^2 u \|p\|_\infty) = O(n^3 \|p\|_\infty)$, where $\|p\|_\infty = \max_i p_i$.

Now, when we want to achieve length-lex* bounds consistency for knapsack constraints in the set-variable representation, we can exploit the above algorithm by decomposing the constraint as follows. Let S be a set variable with length-lex* domain $D(S) = [R(S), L_{lex}(S), U_{lex}(S), P(S)]$, based on a universe of items $\{1, \dots, n\}$ (recall we use shorthands $L(S) = R(S) \cup L_{lex}(S)$ and $U(S) = R(S) \cup U_{lex}(S)$). Then

$$KP(S, p, B, w, C) \Leftrightarrow KP(S^1, p, B, w, C) \vee KP(S^2, p, B, w, C) \vee KP(S^3, p, B, w, C), \quad (1)$$

where S^1 , S^2 , and S^3 are set-variables with respective length-lex* domains

$$\begin{aligned} D(S^1) &= D(S) \cap [L(S), \min_{\leq_{LL}}(\{n - |L(S)| + 1, \dots, n\}, U(S))], \\ D(S^2) &= D(S) \cap [\{1, \dots, |L(S)| + 1\}, \{n - |U(S)| + 2, \dots, n\}], \\ D(S^3) &= D(S) \cap [\max_{\leq_{LL}}(L(S), \{1, \dots, |U(S)|\}), U(S)]. \end{aligned}$$

Note that S^1 and S^3 have real lexicographic bounds but are fixed in cardinality with $|S^1| = |L(S)|$ and $|S^3| = |U(S)|$. On the other hand, S^2 has only trivial lexicographic bounds, and it holds that $|L(S)| < |S^2| < |U(S)|$ (provided that $|U(S)| - |L(S)| \geq 2$). Therefore, for S^2 we can exploit the algorithm that we sketched above. Thus, for a complete pseudo-polynomial length-lex bounds consistency algorithm, we only lack a pseudo-polynomial filtering algorithm for knapsacks with fixed cardinality and arbitrary lexicographic bounds.

So let us consider the following problem: Given a natural number n , a profit vector $p \in \mathbb{Q}^n$, a profit threshold $B \in \mathbb{N}$, a weight vector $w \in \mathbb{Q}^n$, a capacity $C \in \mathbb{N}$, a fixed cardinality $\kappa \in \mathbb{N}$, and lexicographic bounds $L, U \subseteq \{1, \dots, n\}$, find a solution $x \in \{0, 1\}^n$ such that

$$p^T x \geq B \quad \quad \quad w^T x \leq C \quad (2)$$

$$1^T x = \kappa \quad \quad \quad L \leq_{lex} \{i \mid x_i = 1\} \leq_{lex} U \quad (3)$$

$$x \in \{0, 1\}^n. \quad (4)$$

By using a three-dimensional DP like before, we can directly enforce the capacity and profit restrictions (simply by only allowing nodes $W_{n,q,c}$ to connect to the sink-node t for which $c = \kappa$ and $q \geq B$). The filtering problem can then be addressed by identifying edges in the DP-induced graph for which there exists no admissible path from the root to the sink, whereby admissibility now enforces both a path-length lower

or equal C and that the corresponding knapsack solution is a set of items S for which $L \leq_{lex} S \leq_{lex} U$. To this end, we intend to reuse the idea of shorter-path constraint filtering. However, a simple forward and backward shortest path computation is no longer sufficient because the concatenation of a path from the source to a node in the graph and a path from that node to the sink may violate the lexicographic bounds.

Note that both L and U define (potentially non-admissible) paths in the DP-induced graph that we denote with π_L and π_U , respectively. Conversely, for any path π from the root to any node in the DP, we can define a corresponding set S of items that the path includes in the knapsack: $S_\pi \leftarrow \{k \mid (W_{k-1,q-p_k,c-1}, W_{k,q,c}) \in \pi\}$.

In order to identify exactly those nodes in the graph that have no admissible paths running through them, it will be important to know the shortest path distance from the root to a given node when the choices implied by that path π already ensure that the resulting set S_π must obey the lexicographic bounds L, U . Formally:

Definition 2. For a path π from the root to $W_{k,q,c}$, we write $L <_{lex} S_\pi$ (or $S_\pi <_{lex} U$) if and only if for all $S \subseteq \{1, \dots, n\}$ such that $S \cap (S_\pi \cup \{k+1, \dots, n\}) = S$ and $|S| = \kappa$ it holds that $L <_{lex} S$ ($S <_{lex} U$).

Conversely, we will also need to argue about paths from nodes in the DP-induced graph to the sink-node t :

Definition 3. For a path π from $W_{k,q,c}$ to t , we write $L \leq_{lex} S_\pi$ (or $S_\pi \leq_{lex} U$) if and only if for $T \leftarrow S_\pi \cup (L \cap \{1, \dots, k\})$ ($T \leftarrow S_\pi \cup (U \cap \{1, \dots, k\})$) it holds that $|T| = \kappa$ and $L \leq_{lex} T$ ($T \leq_{lex} U$).

To make our task easier, we may assume that the first item is a member of L (otherwise the item is disallowed and can be removed from consideration), and that the first item is not in U (as otherwise the item must be taken and could also be removed from the problem). Then, for all nodes but the root, we distinguish three situations.

Remark 1

1. For all nodes $W_{k,q,c}$ that are neither on π_L nor on π_U , a shortest admissible path from the root to t that visits this node obviously decomposes into a part from the root to the given node π_1 , and from the node to the sink π_2 . Since the current node is neither on π_L nor on π_U , we know that $L <_{lex} S_{\pi_1} <_{lex} U$.
2. For a node $W_{k,q,c}$ on π_L , on top of option 1, the shortest admissible path from root to t through this node may follow π_L from the root to the node, and some path π_2 from the node to t with $L \leq_{lex} S_{\pi_2}$.
3. For a node $W_{k,q,c}$ on π_U , on top of option 1, the shortest admissible path from root to t through this node may follow π_U from the root to the node, and some path π_2 from the node to t with $S_{\pi_2} \leq_{lex} U$.

Note that options 2 and 3 may occur at the same time.

Consequently, we can compute the length of the shortest admissible path through a given node $W_{k,q,c}$, if we know the following six quantities:

- For $W_{k,q,c} \in \pi_L$, $M_{k,q,c}^1$ gives the distance from the root to $W_{k,q,c}$ when following π_L , that is $M_{k,q,c}^1 \leftarrow \sum_{i \in L, i \leq k} w_i$. For $W_{k,q,c} \notin \pi_L$, we set $M_{k,q,c}^1 \leftarrow \infty$.
- For $W_{k,q,c} \in \pi_U$, $M_{k,q,c}^2$ gives the distance from the root to $W_{k,q,c}$ when following π_U , that is $M_{k,q,c}^2 \leftarrow \sum_{i \in U, i \leq k} w_i$. For $W_{k,q,c} \notin \pi_U$, we set $M_{k,q,c}^2 \leftarrow \infty$.
- For arbitrary nodes $W_{k,q,c}$, $\bar{M}_{k,q,c}^3$ gives the length of the shortest path π from the root to $W_{k,q,c}$ with $L <_{lex} S_\pi <_{lex} U$.
- For arbitrary nodes $W_{k,q,c}$, $M_{k,q,c}^4$ gives the length of the shortest path π from $W_{k,q,c}$ to t .
- For arbitrary nodes $W_{k,q,c}$, $M_{k,q,c}^5$ gives the length of the shortest path π from $W_{k,q,c}$ to t with $L \leq_{lex} S_\pi$.
- For arbitrary nodes $W_{k,q,c}$, $M_{k,q,c}^6$ gives the length of the shortest path π from $W_{k,q,c}$ to t with $S_\pi \leq_{lex} U$.

Lemma 1

- The length of a shortest admissible path through an edge $(W_{k,q,c}, W_{k+1,q,c})$ is

$$\min\{M_{k,q,c}^3 + M_{k+1,q,c}^4, M_{k,q,c}^1 + D_{k,q,c}^1, M_{k,q,c}^2 + D_{k,q,c}^2\},$$

where $D_{k,q,c}^1 = M_{k+1,q,c}^4$ if $k+1 \in L$ and $D_{k,q,c}^1 = M_{k+1,q,c}^5$ otherwise, and $D_{k,q,c}^2 = M_{k+1,q,c}^6$ if $k+1 \notin U$ and $D_{k,q,c}^2 = \infty$ otherwise.

- The length of a shortest admissible path through an edge $(W_{k,q,c}, W_{k+1,q+p_{k+1},c+1})$ is

$$\min\{M_{k,q,c}^3 + M_{k+1,q+p_{k+1},c+1}^4 + w_{k+1}, M_{k,q,c}^1 + E_{k,q,c}^1 + w_{k+1}, M_{k,q,c}^2 + E_{k,q,c}^2 + w_{k+1}\},$$

where $E_{k,q,c}^1 = M_{k+1,q+p_{k+1},c+1}^5$ if $k+1 \in L$ and $E_{k,q,c}^1 = \infty$ otherwise, and $E_{k,q,c}^2 = M_{k+1,q+p_{k+1},c+1}^6$ if $k+1 \notin U$ and $E_{k,q,c}^2 = \infty$ otherwise.

Proof. Assume $W_{k,q,c} \in \pi_L$. Denote with π_1 the path from the root to $W_{k,q,c}$ by following π_L . Since $1 \in L$ and $1 \notin U$, we know that $S_{\pi_1} <_{lex} U$. Consequently, it is sufficient for quantity M^5 to consider the lower lexicographic bound only when combined with M^1 . The analogue holds for the combination of M^2 and M^6 . With this observation, the lemma follows from Remark 1. $\square$

Consequently, our task is to compute quantities $M^1, \dots, M^6$. For M^1 and M^2 , this is straightforward. For $M^3, \dots, M^6$, in the following we devise recursion equations which allow us to compute them by means of dynamic programming.

Recall that M^3 measures the shortest path distance from the root when this path already ensures that the final path will strictly obey both lexicographic bounds (let us call such a path an M^3 -path). When considering the inclusion or exclusion of item k , we can either use an M^3 -path to reach the predecessor of a node, in which case we know that any continuation results in an M^3 -path. Alternatively, we can consider a predecessor node on π_L when the exclusion of k results in an M^3 path. Analogously, we can consider a predecessor node on π_U when the inclusion of k results in an M^3 path. Consequently, we have the following recursion equation:

$$M_{k,q,c}^3 = \min\{M_{k-1,q,c}^3, M_{k-1,q-p_k,c-1}^3 + w_k, A_{k,q,c}^1, A_{k,q,c}^2\},$$

whereby $A_{k,q,c}^1 = M_{k-1,q,c}^1$ if $k \in L$ and $A_{k,q,c}^1 = \infty$ otherwise, and $A_{k,q,c}^2 = M_{k-1,q-p_k,c-1}^2 + w_k$ if $k \notin U$ and $A_{k,q,c}^2 = \infty$ otherwise.

Quantity M^4 plainly computes the shortest path distance to the sink t , so the common recursion equation works without modification:

$$M_{k,q,c}^4 = \min\{M_{k+1,q,c}^4, M_{k+1,q+p_{k+1},c+1}^4 + w_{k+1}\}.$$

Quantity M^5 measures the distance to the sink-node t when only paths are allowed that obey the lexicographic lower bound when we prepend the lower-bound path to the current node. When considering the exclusion of an item $k+1$ with $k+1 \in L$, we are sure to strictly obey the lexicographic lower bound and can therefore use the unrestricted shortest path distance to the sink of the corresponding successor node. Consequently, we have the following recursion equation:

$$M_{k,q,c}^5 = \min\{B_{k,q,c}, M_{k+1,q,c}^5\},$$

whereby $B_{k,q,c} = \min\{M_{k+1,q,c}^4, M_{k+1,q+p_{k+1},c+1}^5 + w_{k+1}\}$ if $k+1 \in L$ and $B_{k,q,c} = \infty$ otherwise. The analogue argument for M^6 gives:

$$M_{k,q,c}^6 = \min\{C_{k,q,c}, M_{k+1,q+p_{k+1},c+1}^6 + w_{k+1}\},$$

whereby $C_{k,q,c} = \min\{M_{k+1,q+p_{k+1},c+1}^4 + w_{k+1}, M_{k+1,q,c}^6\}$ if $k+1 \notin U$ and $C_{k,q,c} = \infty$ otherwise.

With these results, we are now able to prove the following theorem:

Theorem 1. *Let S be a set-variable with length-lex* domain based on a universe of elements $\{1, \dots, n\}$. For a Knapsack constraint $KP(S, p, B, w, C)$, length-lex* bounds consistency can be achieved in time $O(n^3||p||\infty)$.*

Proof. We first decompose the constraint according to Equation 1. As discussed earlier, we can identify all possible and required items for $KP(S^2, p, B, w, C)$ in pseudo-polynomial time. Next we consider $KP(S^1, p, B, w, C)$ and $KP(S^3, p, B, w, C)$ and filter edges according to the algorithm sketched above (whereby the lexicographic upper or lower bound are set to the trivial bound for the given cardinality $\kappa \leftarrow |L(S)|$ or $\kappa \leftarrow |U(S)|$ when $|L(S)| < |U(S)|$). We set up the DP-induced graph and compute quantities $M^1, \dots, M^6$ for all nodes. Then, we filter all nodes and edges from the graph which cannot be visited by any admissible path. Using Trick's DP-filtering technique, this allows us to identify all items which must or cannot be part of any feasible improving solution for $KP(S^1, p, B, w, C)$ and $KP(S^3, p, B, w, C)$.

In this way we also determine whether there exist admissible paths at the cardinality bounds at all, i.e., whether the constraints can still be satisfied or not. If this is the case, in order to compute a new lexicographic lower bound at the lower cardinality bound, we simply include the first item if that is still possible after filtering edges from the graph. Then we filter again and try to include the next item and so forth. The correctness of the edge-filtering algorithm guarantees that we compute an admissible path π such that S_π is the lexicographically smallest feasible and improving solution with $|S_\pi| = |L(S)|$. For the new lexicographic upper bound we proceed analogously.

If we find that one of the two constraints is not satisfiable anymore, then we use $KP(S^2, p, B, w, C)$ again to determine a new lower and/or upper bound on the cardinality. If one of the cardinality bounds are updated, we repeat the computation of a new lexicographical lower and/or upper bound as before.

The total runtime of this algorithm is dominated by the computation of new lexicographical upper and lower bounds which require up to $|U(S)|$ calls to the edge-filtering algorithm whose runtime is determined by the size of the DP-induced graph which is in $O(n|U(S)||p||_\infty)$. The total runtime is therefore in $O(n|U(S)|^2||p||_\infty) = O(n^3||p||_\infty)$. $\square$

5 Approximated Length-Lex Bounds Consistency for Knapsack Constraints

The results of the previous section show that the fully polynomial-time approximability of knapsack problems is not affected by additional length-lex bounds constraints. We can utilize our approximation scheme to achieve approximated length-lex* bounds consistency for knapsack constraints in the spirit of [14]:

Definition 4. Let S denote a set-variable with length-lex* domain $D(S) = [R(S), L_{lex}(S), U_{lex}(S), P(S)]$. The knapsack constraint $KP(S, p, B, w, C)$ is called ε -length-lex* bounds consistent when it holds:

- $P^*[i \in S] \geq B - \varepsilon P^*$, for all $i \in P(S)$,
- $P^*[i \notin S] < B - \varepsilon P^*$, for all $i \in R(S)$,
- $P^*[S = R(S) \cup L_{lex}(S)] \geq B - \varepsilon P^*$ and $P^*[S = R(S) \cup U_{lex}(S)] \geq B - \varepsilon P^*$,

where P^* gives the optimal knapsack solution (potentially under the additional constraints given in brackets).

Theorem 2. Approximated ε -length-lex* bounds consistency for knapsack constraints can be achieved in time $O(\frac{n^4}{\varepsilon})$.

Proof. In this proof we again use the shorthands $L(S) = R(S) \cup L_{lex}(S)$ and $U(S) = R(S) \cup U_{lex}(S)$.

We apply the standard approach from [10] for transforming a dynamic program into a fully polynomial-time approximation scheme (FPTAS): We scale the profits by setting $\tilde{p}_i \leftarrow \lfloor \frac{p_i}{K} \rfloor$ for $K \leftarrow \frac{\varepsilon||p||_\infty}{|U(S)|}$. Then, we invoke our pseudo-polynomial filtering algorithm on $KP(S, \tilde{p}, B - \varepsilon||p||_\infty, w, C)$. Note that $||\tilde{p}||_\infty \leq \frac{|U(S)|}{\varepsilon}$. Therefore, the algorithm runs in time $O(n^2|U(S)||\tilde{p}||_\infty) = O(\frac{n^2|U(S)|^2}{\varepsilon})$.

We show that the algorithm is sound and achieves ε -length-lex* bounds consistency.

Soundness: Assume our algorithm excludes an item s from $P(S)$. It does so only when there exists no admissible path that includes the item, which is the same as to say that there exists no admissible solution to $KP(S, \tilde{p}, \frac{B - \varepsilon||p||_\infty}{K}, w, C)$ that includes the item. Denote with $\tilde{S} \in D(S)$ the solution with $s \in \tilde{S}$ that maximizes $\tilde{P}^*[s \in \tilde{S}] = \sum_{i \in \tilde{S}} \tilde{p}_i$ while $\sum_{i \in \tilde{S}} w_i \leq C$. Furthermore, denote with $S^* \in D(S)$ the solution with $s \in S^*$ that maximizes $P^*[s \in S] = \sum_{i \in S^*} p_i$ while $\sum_{i \in S^*} w_i \leq C$. It holds:

$$B - \varepsilon||p||_\infty > K\tilde{P}^*[i \in S] \quad (5)$$

$$\geq \sum_{i \in \tilde{S}} p_i - K|U(S)| \geq \sum_{i \in \tilde{S}} p_i - \varepsilon||p||_\infty \quad (6)$$

Therefore, $P^*[s \in S] = \sum_{i \in \tilde{S}} p_i < B$, which means it is sound to remove item s from consideration. The analogous argument holds for items that are included by our algorithm. Next, our algorithm computes length-lex lower and upper bounds $L_{lex}(S)$, $U_{Lex}(S)$ on the undecided items such that no set lower than $L(S) = R(S) \cup L_{lex}(S)$ and no set larger than $U(S) = R(S) \cup U_{Lex}(S)$ is admissible for $KP(S, \tilde{p}, \frac{B - \varepsilon \|p\|_\infty}{K}, w, C)$. The same argument as before shows that no set lower than $L(S)$ or larger than $U(S)$ can then be admissible for $KP(S, p, B, w, C)$.

Completeness: Assume for some item s it holds that $P^*[s \in S] < B - \varepsilon P^*$. Therefore, for all $\tilde{S} \in D(S)$ with $s \in \tilde{S}$ and $\sum_{i \in \tilde{S}} w_i \leq C$ it holds that $\sum_{i \in \tilde{S}} p_i < B - \varepsilon P^* \leq B - \varepsilon \|p\|_\infty$. Thus:

$$B - \varepsilon \|p\|_\infty > \sum_{i \in \tilde{S}} \tilde{p}_i \geq K \sum_{i \in \tilde{S}} \tilde{p}_i$$

for all $\tilde{S}$, and therefore s is removed from $P(S)$. The analogous results follows for items that must be included. For the length-lex bounds, finally, it holds that they define admissible solutions for $KP(S, \tilde{p}, \frac{B - \varepsilon \|p\|_\infty}{K}, w, C)$. It holds:

$$B - \varepsilon \|p\|_\infty \leq K \sum_{i \in L(S)} \tilde{p}_i \leq \sum_{i \in L(S)} p_i = P^*[S = L(S)].$$

And the analogue is true for the set $U(S)$. □

6 Fast Heuristic Filtering Algorithms for Knapsacks with Bounded Cardinalities

We have seen that cardinality and lexicographic information can be inferred and taken into account for knapsack constraints without compromising the fully polynomial-time approximability of the problem. However, although polynomial, a runtime in $O(n^4)$ is not practically appealing in light of the delicate trade-off between the time to perform this type of inference and the value of the additional information gained by it. In order to make inference faster, we may decide that we only want to reason about the cardinality of the final set of items included in the knapsack. We presented an exact pseudo-polynomial time algorithm for this task in Section 4. In an effort to reduce the filtering-time, in this section we devise a heuristic algorithm which runs in linear time.

6.1 Lagrangian Relaxation-Based Cardinality Bounds

To derive cardinality bounds, as we did earlier in [18], we consider the Lagrangian relaxation of the knapsack problem. In linear programming, it is well-known that the optimal dual value for the capacity constraint is the efficiency (the profit over weight) of the critical item s , which is defined as the first item in the efficiency ordering whose inclusion overloads the knapsack: $s \leftarrow \min\{s' \mid \sum_{i=1}^{s'} w_i > C\}$, whereby $i < j$ implies $p_i/w_i \geq p_j/w_j$. Using this value as Lagrangian multiplier, we are left with the following relaxed problem: maximize $p^T x - (w^T x - C)p_s/w_s = (p - wp_s/w_s)^T x + Cp_s/w_s$ such that $x_i \in \{0, 1\}$. Obviously, the maximum is obtained by setting $x_i \leftarrow 1$

Input: set S with associated profit vector $\tilde{p}$ and a bound $\tilde{B}$.
 Pick a random element r in S .
 Set $L \leftarrow \{i \in S \mid \tilde{p}_i \geq \tilde{p}_r\}$ and $R \leftarrow \{i \in S \mid \tilde{p}_i < \tilde{p}_r\}$.
 $\tilde{p}(L) \leftarrow \sum_{i \in L} \tilde{p}_i$
if $\tilde{p}(L) \geq \tilde{B}$ **then**
 return lowerBound($L, \tilde{p}, \tilde{B}$)
else
 return $|L| + \text{lowerBound}(R, \tilde{p}, \tilde{B} - \tilde{p}(L))$

Algorithm 1. Linear-time algorithm to determine a lower bound on the cardinality

if $\tilde{p}_i \leftarrow p_i - w_i p_s / w_s > 0$, and $x_i \leftarrow 0$ if $\tilde{p}_i < 0$ (for $\tilde{p}_i = 0$ we can set x_i arbitrarily). By this setting, we obtain a valid upper bound U on the profit that can be achieved. If $U < B$, we can backtrack right away as the current subproblem cannot have any improving feasible solutions. Otherwise, we would like to infer which items must be included/excluded as they must/cannot be part of any improving feasible solution. Moreover, we would like to tighten the bounds on the number of items that must/can be included in the knapsack.

When sorting items according to decreasing Lagrangian profits $\tilde{p}_i$, we can easily deduce lower and upper bounds on the number of items that must/can be included:

$$l \leftarrow \max\{l, \min\{l' \mid \sum_{i=1}^{l'} \tilde{p}_i \geq B - C \frac{p_s}{w_s}\}\},$$

whereby we assume that $i < j$ implies $\tilde{p}_i \geq \tilde{p}_j$. Analogously, we set

$$u \leftarrow \min\{u, \max\{u' \mid \sum_{i=1}^{u'} \tilde{p}_i \geq B - C \frac{p_s}{w_s}\}\}.$$

The effort for the above update is obviously dominated by sorting the items, which can be done in time $O(n \log n)$. However, a complete sorting is actually not necessary. Just like the critical item s of a knapsack instance can be computed in linear time [3], so can the new cardinality bounds l and u . In Algorithm 1, we show how a lower bound on the cardinality can be computed in expected linear time. The algorithm works like a quick-sort algorithm, whereby only one part of the items needs to be investigated recursively. According to the master theorem for recursive algorithms [2, Section 4.3, 4.4], this lowers the time from $O(n \log n)$ to $O(n)$ when we assume that, on average, the set of items is cut in half in each recursion iteration. While this works well in practice, in theory we can even guarantee a linear runtime by replacing the random choice of the splitting item by the median item, where the median can be computed in linear time [2].

Of course, the Lagrangian relaxation also allows us to filter items by exploiting the idea of CP-based Lagrangian relaxation [17]. In our case, filtering is particularly easy as items that must be included have $\tilde{p}_i > B - U$. Those that cannot be included have $\tilde{p}_i < U - B$. In case that the external cardinality bounds are tight (this happens when $\tilde{p}_l < 0$ or $\tilde{p}_u > 0$), we can even decide that an item must be included when

$\tilde{p}_i - \tilde{p}_{l+1} > B - U$ when the lower cardinality bound is tight, and that an item must be excluded when $\tilde{p}_i - \tilde{p}_u < B - U$ when the upper cardinality bound is tight.

6.2 Redundant Knapsack Constraints

In terms of running time, the above linear time algorithm that heuristically filters knapsack constraints and exploits and provides upper and lower bounds on the cardinality is already much more appealing than the exact or approximate algorithms devised earlier. However, we can do even more: In [11], an algorithm for the propagation of knapsack constraints was devised which runs in amortized expected sublinear time. The question arises how this algorithm can be exploited to reason about knapsack cardinalities at the same time as it infers which items must or cannot be included in any feasible improving solution. A simple option is to post redundant knapsack constraints: If we are given the conjunction $KP(S, p, B, w, C) \wedge (l \leq |S| \leq u)$, we can post the following three traditional knapsack constraints (whereby $x_1, \dots, x_n$ are binary variables): $KP(x_1, \dots, x_n, p, B, w, C)$, $KP(x_1, \dots, x_n, (1, \dots, 1)^T, l, w, C)$, and $KP(x_1, \dots, x_n, p, B, (1, \dots, 1)^T, u)$. Note that the different constraints do not only allow us to perform filtering in the item variables, they also allow us to infer strengthened bound on the cardinalities. For example, the floor of the linear upper bound computed for the propagation of $KP(x_1, \dots, x_n, (1, \dots, 1)^T, l, w, C)$ gives a valid upper bound on the number of items that can be included in any feasible and improving solution.

7 Experimental Results

Despite the fact that the algorithms developed in Sections 4 and 5 are polynomial, their comparably large computation costs render them impractical within backtrack search where we face a delicate trade-off between inference efficiency and effectiveness. To assess whether communicating information beyond the traditional inclusion or exclusion of items, we therefore implemented the heuristic algorithms for reasoning about knapsacks with cardinality constraints.

As our benchmark, we use multi-knapsack problems where we have to distribute a set of items over multiple knapsacks while the capacity restrictions on the individual knapsacks must be respected. We aim at maximizing the profit of the knapsack that is assigned the least profit. Problems are generated using the code from Pisinger [12]. We distinguish three different problem classes:

- P1:** Multi-Knapsack problems where all knapsacks use the same profit and weight vector. Constraints differ in the available capacity for each knapsack.
- P2:** Multi-Knapsack problems where all knapsacks use the same profit vector. Constraints differ in the weight vector and the available capacity for each knapsack.
- P3:** Multi-Knapsack problems where all knapsacks use different profit and weight vectors.

In each algorithm, we branch on the item that has the least knapsacks left to be assigned to, and we assign it to that knapsack that has been assigned the least profit yet. We compare three different models:

- **KP**: The plain knapsack model where each knapsack constraint is propagated by the expected sublinear-time algorithm introduced in [11]. The partitioning of items is enforced by introducing item variables, whereby each of those variables has the indices of the knapsacks plus a dummy index for left-over items as its domain.
- **KP + Card**: The model where redundant knapsack constraints exploit cardinality information. Each knapsack is modelled by three constraints, one for the actual knapsack, one for the combination of cardinality upper bound and the knapsack's profit, and one for the combination of the original weights in combination with the cardinality lower bound. All knapsacks are propagated by the algorithm from [11]. The partitioning of items is enforced by a global cardinality constraint which exploits and strengthens the cardinality bounds on the knapsacks.
- **Lagrangian**: The model where we use a Lagrangian bound to propagate knapsack constraints and infer knapsack cardinalities. The partitioning of items is again enforced by a global cardinality constraint.

All experiments were run on an AMD Athlon 64 X2 Dual Core Processor 3800+ using Ilog Solver 6.5. In Tables 1 and 2 we show the average runtime and choice points

Table 1. Average running time (seconds) and the average number of choice points over 20 instances with five knapsack constraints and 15 or 20 items using three different models

Class	#Items	Lagrangian		KP		KP+Card	
		Time	#CPs	Time	#CPs	Time	#CPs
P1	15	1.89	26.7k	1.51	24.1k	2.19	22.4k
	20	2.40k	31.4M	888.51	12.6M	978.79	8.62M
P2	15	5.98	85.4k	1.51	22.1k	2.34	21.6k
	20	4.26k	57.8M	414.25	5.48M	468.65	3.74M
P3	15	2.41	32.2k	0.33	4.88k	0.54	4.84k
	20	264.33	3.32M	12.85	0.19M	22.37	0.19M

Table 2. Average (avg) percent difference in running times and choice points as well as their variance (var) when comparing models on a collection of multi-knapsack problems. A positive value states the strategy listed first is the given percentage larger.

Class	#Items	KP vs KP+Card				Lagrangian vs KP+Card			
		Time		#CPs		Time		#CPs	
		avg	var	avg	var	avg	var	avg	var
P1	15	-43.4	0.8	7.0	0.3	-4.6	16.4	24.9	10.7
	20	-32.4	9.6	18.1	3.0	51.0	12.1	67.2	6.1
P2	15	-54.2	0.6	1.2	0.0	64.1	5.4	77.5	3.1
	20	-50.6	4.5	8.9	1.5	92.3	0.6	95.8	0.2
P3	15	-55.3	1.1	1.2	0.2	81.5	5.2	90.2	2.1
	20	-67.3	0.3	0.6	0.0	-353	3.9k	-284	2.8k

on collections of 20 instances in the different banchmark classes P1, P2, and P3. We see how exploiting cardinality information effectively reduces the number of choice points. This is generally highly desirable as a more effective inference mechanism leaves less room for mistakes when organizing the search. For multi-knapsack problems, the trade-off between inference time and effectiveness is not in favor of even slightly more costly inference, and we observe that the plain KP model works fastest in all cases. Note that, in this model, inference works in expected sublinear time, while in the two other models global cardinality constraints need to be propagated to infer new cardinality bounds on the knapsacks. The Lagrangian model also suffers from a linear-time filtering routine for knapsack, and we see that it cannot compete with with KP and KP+Card.

We were curious to see whether the reductions in choice points become more important as problem instances become even harder. In Table 3 we compare KP and KP+Card on ten 22 item knapsack problems. We observe that the exchange of cardinality information is becoming more and more competitive, and for even harder problem instances we expect that the more costly yet more effective inference will eventually become beneficial. The results show that, with increasing difficulty of the problem, the cardinality constraints significantly boost the performance of the algorithm, providing an average decrease of more than 30% in the number of choice points. For more general problems, where knapsack constraints are mixed with other constraints, this reduction may be very beneficial.

Table 3. Running times (seconds) and the number of choice points for 10 instances with five knapsack constraints and 22 items using the KP and KP+Card models

ID	KP		KP+Card	
	Time	#CPs	Time	#CPs
1	4.5K	52.2M	5.4K	39.5M
2	1.4K	17.2M	2.6K	16.7M
3	2.4K	31.7M	2.4K	19.6M
4	1.8K	20.8M	2.9K	20.7M
5	15.3K	196.2M	17.6K	136.3M
6	0.5K	6.7M	0.5K	4.0M
7	4.3K	55.8M	3.1K	23.6M
8	2.4K	33.2M	2.6K	22.1M
9	3.2K	41.9M	4.5K	35.9M
10	4.3K	56.5M	3.1K	24.9M
avg	4.0K	51.2M	4.5K	34.3M

8 Conclusions

We studied the complexity of knapsack constraints with length-lex domains and showed that the problem remains fully polynomial-time approximeable. Based on this result, we showed how ε -approximate length-lex bounds consistency for knapsacks can be achieved in time $O(n^4/\varepsilon)$. Compromising inference effectiveness for efficiency, we provided heuristic filtering algorithms for knapsack constraints that incorporate cardinality information only. Experiments on multi-knapsack problems showed that these algorithms effectively reduce the number of choice points. Whether or not this reduction is worthwhile will depend on the concrete problem that needs to be solved.

References

1. Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemann, P.: A Constraint Store Based on Multivalued Decision Diagrams. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 118–132. Springer, Heidelberg (2007)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. MIT Press / McGraw-Hill (2001)
3. Dantzig, G.: Discrete variable extremum problems. *Operations Research* 5, 226–277 (1957)
4. Dooms, G., Mercier, L., Van Hentenryck, P., van Hoeve, W.-J., Michel, L.: Length-Lex Open Constraints. Technical Report CS-07-09, Brown University (2007)
5. Eremin, A., Wallace, M.: Hybrid Benders Decomposition Algorithms in Constraint Logic Programming. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 1–15. Springer, Heidelberg (2001)
6. Fahle, T., Sellmann, M.: Cost Based Filtering for the Constrained Knapsack Problem. *Annals of Operations Research* 115(1), 73–93 (2002)
7. Gervet, C.: Constraints over structured domains. In: Rossi, F., van Beek, P., Walsh, T. (eds.) *Handbook of Constraint Programming*, ch. 17. Elsevier, Amsterdam (2006)
8. Gervet, C., Van Hentenryck, P.: Length-lex ordering for set CSPs. In: *Proceedings of AAAI* (2006)
9. Hooker, J.N., Ottosson, G.: Logic-based Benders decomposition. *Mathematical Programming* 96(33–60), 22 (2003)
10. Ibarra, O.H., Kim, C.E.: Fast Approximation Algorithms for the Knapsack and Sum of Subset Problems. *Journal of the ACM* 22(4), 463–468 (1975)
11. Katriel, I., Sellmann, M., Upfal, E., Van Hentenryck, P.: Propagating Knapsack Constraints in Sublinear Time. In: *Proceedings of AAAI*. AAAI Press, Menlo Park (2007)
12. Pisinger, D.: Where are the hard knapsack problems? *Computers and Operations Research* 32, 2271–2282 (2005)
13. Sadler, A., Gervet, C.: Enhancing set constraint solvers with lexicographic bounds. *Journal of Heuristics* 14(1), 23–67 (2008)
14. Sellmann, M.: Approximated Consistency for Knapsack Constraints. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 679–693. Springer, Heidelberg (2003)
15. Sellmann, M.: Cost-Based Filtering for Shorter Path Constraints. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 694–708. Springer, Heidelberg (2003)
16. Sellmann, M.: The Practice of Approximated Consistency for Knapsack Constraints. In: *Proceedings of AAAI*, pp. 179–184. AAAI Press, Menlo Park (2004)
17. Sellmann, M., Fahle, T.: Constraint Programming Based Lagrangian Relaxation for the Automatic Recording Problem. *Annals of Operations Research* 118(1), 17–33 (2003)
18. Sellmann, M., Kliewer, G., Koberstein, A.: Lagrangian Cardinality Cuts and Variable Fixing for Capacitated Network Design. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 845–858. Springer, Heidelberg (2002)
19. Trick, M.A.: A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *Annals of Operations Research* 118(1), 73–84 (2003)

A Framework for Hybrid Tractability Results in Boolean Weighted Constraint Satisfaction Problems

T. K. Satish Kumar*

Institute for Human and Machine Cognition
Pensacola, Florida, U.S.A.
skumar@ihmc.us

Abstract. Many tasks in automated reasoning can be modeled as *weighted constraint satisfaction problems* over Boolean variables (Boolean WCSPs). Tractable classes of such problems have traditionally been identified by exploiting either: (a) the topology of the associated constraint network, or (b) the structure of the weighted constraints. In this paper, we introduce the notion of a *constraint composite graph* (CCG) associated with a given (Boolean) WCSP. The CCG provides a unifying framework for characterizing/exploiting both the graphical structure of the constraint network as well as the structure of the weighted constraints. We show that a given (Boolean) WCSP can be reduced to the problem of computing the *minimum weighted vertex cover* for its associated CCG; and we establish the following two important results: (1) “*the CCG of a given Boolean WCSP has the same treewidth as its associated constraint network,*” and (2) “*many classes of Boolean WCSPs that are tractable by virtue of the structure of their weighted constraints have associated CCGs that are bipartite in nature.*”

1 Introduction

In many application domains, we are required to efficiently represent and reason about natural factors like fuzziness, probabilities, preferences and/or costs. Automated reasoning tasks in such domains involve combinatorial problems that typically exhibit both a *satisfaction* component and an *optimization* component. While the satisfaction component of a problem can be captured using *hard* constraints, *soft* constraints are used to capture the optimization component. Many extensions to the basic CSP model have been introduced to incorporate and reason about soft constraints. These include the many variants like *fuzzy CSPs*, *probabilistic CSPs* and *weighted CSPs*.¹

A WCSP is an optimization version of a CSP in which the constraints are no longer “hard,” but are extended by associating non-negative *costs* to the tuples. The goal is to find an assignment of values to all the variables (from their respective domains) such that the *total cost* is *minimized*. More formally, a WCSP is defined by a triplet $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X} = \{X_1, X_2 \dots X_N\}$ is a set of *variables*, and $\mathcal{C} = \{C_1, C_2 \dots C_M\}$ is a set of *weighted constraints* on subsets of the variables. Each variable X_i is associated with

* Several parts of this research were done by the author at the University of California, Berkeley.

¹ These in turn can be viewed as particular instances of certain meta-frameworks like *valued CSPs* and/or *semiring-based CSPs* [2].

a discrete-valued *domain* $D_i \in \mathcal{D}$, and each constraint C_i is defined on a certain subset $S_i \subseteq \mathcal{X}$ of the variables. S_i is referred to as the *scope* of C_i ; and C_i specifies a non-negative *cost* for every possible combination of values to the variables in S_i . An *optimal* solution is an assignment of values to all the variables (from their respective domains) so that the *sum* of the costs (as specified locally by each weighted constraint) is *minimized*. In a Boolean WCSP, the size of any variable's domain is equal to 2 — i.e., $|D_i| = 2$ for all $i \in \{1, 2 \dots N\}$. Boolean WCSPs are representationally as powerful as WCSPs; and it is well known that optimally solving (Boolean) WCSPs is NP-hard in general.

(Boolean) WCSPs can be used to model important combinatorial problems arising in many different application domains. Examples include (but are not limited to) representing and reasoning about user preferences [4], over-subscription planning with goal preferences [8], resource allocation, combinatorial auctions, and Bioinformatics. Quite importantly, they also arise as *energy minimization problems* (EMPs) in probabilistic settings. In computer vision applications, for example, tasks such as image restoration, total variation minimization and panoramic image stitching can be formulated as EMPs derived in the context of Markov Random Fields (MRFs) [10].²

Although (Boolean) WCSPs are fundamental combinatorial problems, a lot of work still remains to be done in characterizing/exploiting the combinatorial structure in specific instances/subclasses of them. One traditional way in which this has been done is by studying the underlying *variable-interaction graphs* (also referred to as *constraint networks*) [7]. The variable-interaction graph incorporates basic information about which variables are constrained with which other variables in the problem instance, and this “locality” information can be exploited in solution procedures that employ dynamic programming. Despite its apparent usefulness, the constraint network does not represent/capture information about the costs in the weighted constraints, and therefore cannot be used to characterize/exploit any important combinatorial structure that might be present in them. In fact, there are many fundamental combinatorial problems — like the *hypergraph min-st-cut* problem — that can be formulated as Boolean WCSPs, and that are tractable not by virtue of the graphical structure in their associated constraint networks, but by virtue of the numerical structure in their associated weighted constraints.

In this paper, we will introduce the notion of a *constraint composite graph* (CCG) associated with a given (Boolean) WCSP. The CCG provides a unifying framework for characterizing/exploiting both the graphical structure of the constraint network as well as the structure of the weighted constraints. We will show that a given (Boolean) WCSP can be reduced to the problem of computing the minimum weighted vertex cover for its associated CCG; and in turn, we will prove two important computational properties of CCGs. First, we will constructively show that the treewidth of the CCG is equal to that of the corresponding constraint network. Because the minimum weighted vertex cover for a graph can be computed in time exponential only in its treewidth, this property of the CCG captures any *topological structure* in the variable-interactions. Second, we will show that the CCG is always *tripartite*, and that many interesting classes of Boolean WCSPs that are tractable by virtue of the structure of their weighted constraints have associated CCGs that are *bipartite* in nature. Now because the minimum weighted vertex cover can be efficiently computed in a bipartite graph also, this property of the

² Here the minimum energy corresponds to a *maximum a-posteriori* labeling of the variables.

CCG captures any *numerical structure* of the weighted constraints. Put together, the CCG provides the necessary unifying framework as claimed above. Further, in the process of proving/elucidating the many computationally attractive properties of CCGs, we will provide simple arguments for establishing the tractability of the language $\mathcal{L}_{bipartite}^{Boolean}$ (that in turn captures many useful subclasses of Boolean WCSPs).

2 Background Results in Graph Theory

We will now review some basic results in graph theory, and set up the groundwork for the rest of the paper. We will also briefly review the treewidth-based characterization of the complexity of solving Boolean WCSPs. In later sections, we will study the relevance of these graph-theoretic results to important computational properties of the CCG.

Given an undirected graph $G = \langle V, E \rangle$, a *matching* is a subset of edges $M \subseteq E$ such that no two edges in M share a common end-point. A *maximum matching* is a matching of maximum cardinality. A *vertex cover* is a subset of nodes $U \subseteq V$ such that every edge in E has at least one of its end-points included in U . A *minimum vertex cover* is a vertex cover of minimum cardinality. While the problem of computing the maximum matching can be solved using very efficient polynomial-time algorithms [14], the problem of computing the minimum vertex cover is NP-hard in general. Nonetheless, for *bipartite* graphs, the minimum vertex cover problem can be solved very efficiently in $O(|V|^{2.5})$ time using a staged *maxflow* computation [5]. Moreover, even in the general case, the minimum vertex cover can be approximated within a factor of $2 - \frac{\log \log |V|}{2 \cdot \log |V|}$ in polynomial time; and this approximation factor can further be improved to $2 - \frac{2}{k}$ for *k-partite* graphs [9]. It is also well known that the size of a maximum matching serves as a *lower bound* for the size of the minimum vertex cover [5]. Finally, the above results can be extended to the “weighted” case in which the nodes/edges of the graph G are associated with non-negative weights. The *maximum weighted matching* is then defined to be a matching of maximum total weight on its edges, and the *minimum weighted vertex cover* is defined to be a vertex cover of minimum total weight on its nodes.

It is well known that WCSPs — among many other combinatorial problems like in probabilistic inference, constraint satisfaction and query optimization — can be solved in time exponential only in the treewidth of their associated variable-interaction graphs. A *tree-decomposition*³ of a graph is a tree whose nodes represent some appropriately chosen subsets of variables (nodes) from the original graph. A combinatorial problem defined on the original graph can typically be solved by using a dynamic programming-based algorithm that proceeds by solving subproblems defined over the variables included in each tree-node. The *treewidth* of the graph measures the largest number of graph-nodes within any tree-node in an optimal tree-decomposition of the graph — i.e., it measures the size of the largest subproblem that needs to be solved.

In the above context, it is often very valuable to conceptually equate the treewidth of a graph to its *minimum induced width* [7] (well illustrated in Figure 1). Consider solving a given combinatorial problem — say, a CSP — over N variables. One strategy is to eliminate the variables one at a time and recursively solve the smaller

³ Also referred to as a *clique-tree* or a *join-tree* in different research communities.

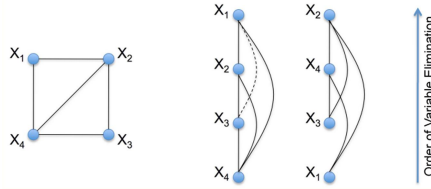


Fig. 1. Shows a simple variable-interaction graph over 4 variables and two different variable-elimination orderings for it. The second ordering produces an induced width less than that of the first. The solid edges indicate the original variable-interactions/constraints and the dotted edge(s) indicate newly induced ones between the parents (X_1 and X_3) of an eliminated node (X_4).

subproblems. However, each time a variable X_i is eliminated from the current subproblem, a new constraint has to be introduced among all the variables that X_i is currently constrained with. This new constraint is required to ensure that a solution to the new subproblem will indeed have a consistent extension to X_i . The size of the largest subproblem (clique) that we encounter using a proposed variable-elimination ordering is referred to as the *induced width* of that ordering. The problem of constructing an optimal tree-decomposition of a given graph is therefore equivalent to the problem of finding an optimal variable-elimination ordering that minimizes the induced width — referred to as the *minimum induced width* of the graph. Finally, although finding the minimum induced width (treewidth) of a graph is NP-hard in general, heuristically chosen variable-elimination orderings yield tree-decompositions that are of much practical value [7].

It is important to note that the minimum weighted vertex cover problem over a given graph $G = \langle V, E \rangle$ can itself also be formulated as a Boolean WCSP over G . Here, a Boolean variable X_i is associated with each node $v_i \in V$; and $X_i = 1$ ($X_i = 0$) is indicative of v_i being included in (excluded from) the vertex cover. A unary weighted constraint is defined for each X_i ; and a cost equal to the weight of v_i is associated with $X_i = 1$ while a cost of 0 is associated with $X_i = 0$. Moreover, binary weighted constraints are defined for each edge $(v_i, v_j) \in E$. These constraints assign a cost of ∞ for the combination of values $\langle X_i = 0, X_j = 0 \rangle$, and a cost of 0 for every other combination of values to X_i and X_j . The constraint network of the resulting Boolean WCSP is exactly the same as G ; and therefore, the minimum weighted vertex cover for G can be computed in time exponential only in the treewidth of G .

3 Projections of Minimum Weighted Vertex Cover Problems onto Independent Sets

In this section, we will first introduce the idea of *projecting* the minimum weighted vertex cover problem onto an independent set of the given graph $G = \langle V, E \rangle$.⁴ We will then illustrate and prove a number of interesting computational properties of these projections. Our study of these projections motivates a special set of algorithmic techniques for solving (Boolean) WCSPs that considers each weighted constraint only *locally* [12].

⁴ An *independent set* of a graph is a subset of nodes no two of which are connected by an edge.

Consider an undirected graph $G = \langle V, E \rangle$. Let $U = \{u_1, u_2 \dots u_k\}$ be an independent set of G . We say that a k -bit vector t imposes the following restrictions: (a) the i^{th} bit $t_i = 0$ indicates that the node u_i is necessarily *excluded* from the minimum weighted vertex cover, and (b) the i^{th} bit $t_i = 1$ indicates that the node u_i is necessarily *included* in the minimum weighted vertex cover. The *projection* of the minimum weighted vertex cover problem onto the independent set U is then defined to be a table of size 2^k with entries corresponding to each of the 2^k possible k -bit vectors $t^{(1)}, t^{(2)} \dots t^{(2^k)}$; the value of the entry corresponding to $t^{(j)}$ is set to be equal to the weight of the minimum weighted vertex cover *conditioned* on the restrictions imposed by $t^{(j)}$. Figure 2 presents a simple example to illustrate this idea.⁵

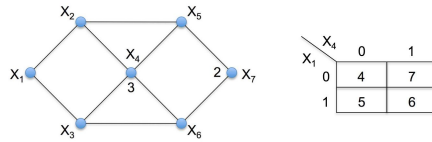


Fig. 2. The table on the right-hand side represents the projection of the minimum weighted vertex cover problem onto the independent set $\{X_1, X_4\}$ of the accompanying node-weighted undirected graph. (The weights on X_4 and X_7 are set to 3 and 2 respectively while all other nodes have unit weights.) Here, the entry ‘7’ written against $\langle X_1 = 0, X_4 = 1 \rangle$, for example, indicates that when X_1 is prohibited from being in the minimum weighted vertex cover but X_4 is necessarily included in it, then the weight of the minimum weighted vertex cover — $\{X_2, X_3, X_4, X_7\}$ or $\{X_2, X_3, X_4, X_5, X_6\}$ in this case — is equal to 7.

ALGORITHM: COMPUTE-PROJECTION-VALUE

INPUT: (a) a node-weighted undirected graph $G = \langle V, E \rangle$; (b) an independent set $U = \{u_1, u_2 \dots u_k\} \subseteq V$; (c) a k -bit vector t .

OUTPUT: the value of the projection $\mathcal{P}_{G,U}(t)$.

(1) $S_1 \leftarrow \{\}$.

(2) For $i = 1, 2 \dots k$:

(a) If $t_i = 0$: set the weight of u_i to ∞ .

(b) If $t_i = 1$: $S_1 \leftarrow S_1 \cup \{u_i\}$; remove u_i (and all edges incident on it) from the graph.

(3) Let S_2 be the minimum weighted vertex cover computed for the resulting graph.

(4) Let W be the sum of the weights on all the nodes in $S_1 \cup S_2$.

(5) RETURN: $\mathcal{P}_{G,U}(t) \leftarrow W$.

END ALGORITHM

Fig. 3. Shows a simple algorithm for computing $\mathcal{P}_{G,U}(t)$. The algorithm makes use of one call to the problem of computing the minimum weighted vertex cover.

⁵ It is worth noting that the projection is well defined only when U is an independent set. If this is not the case, then there exists some edge (u_{i_1}, u_{i_2}) for $u_{i_1}, u_{i_2} \in U$. The entry corresponding to any k -bit vector that disallows both u_{i_1} and u_{i_2} from being in the minimum weighted vertex cover then becomes undefined because the edge (u_{i_1}, u_{i_2}) cannot be covered in any way.

ALGORITHM: COMPUTE-MIN-PROJECTION

INPUT: (a) a node-weighted undirected graph $G = \langle V, E \rangle$; (b) an independent set $U = \{u_1, u_2 \dots u_k\} \subseteq V$.

OUTPUT: (a) the optimal t^* such that $t^* = \operatorname{argmin}_t \mathcal{P}_{G,U}(t)$; (b) the optimal value $\mathcal{P}_{G,U}(t^*)$.

(1) Compute the minimum weighted vertex cover on G . Let S be the set of nodes included in this cover, and let W be the total weight of the nodes in S .

(2) For all $u_i \in U$:

(a) If $u_i \in S$: set $t_i^* \leftarrow 1$.

(b) If $u_i \notin S$: set $t_i^* \leftarrow 0$.

(3) RETURN: (a) t^* : optimal assignment vector; (b) W : optimal value.

END ALGORITHM

Fig. 4. Shows an algorithm for computing the optimal t^* such that $t^* = \operatorname{argmin}_t \mathcal{P}_{G,U}(t)$. The optimal value $\mathcal{P}_{G,U}(t^*)$ is also returned. We note that the algorithm makes use of just one call to the problem of computing the minimum weighted vertex cover.

Let $\mathcal{P}_{G,U}$ denote the projection of the minimum weighted vertex cover problem onto U ; and let $\mathcal{P}_{G,U}(t)$ denote the value of the entry corresponding to the k -bit vector t . We now prove some basic algorithmic properties of $\mathcal{P}_{G,U}(t)$ (see Figures 3 and 4).

Lemma 1. The procedure ‘COMPUTE-PROJECTION-VALUE’ (Figure 3) computes $\mathcal{P}_{G,U}(t)$ for a given k -bit vector t .

Proof. In step 2(a) of the algorithm, we notice that if $t_i = 0$, then the weight of u_i is set to ∞ . This ensures the exclusion of u_i from the minimum weighted vertex cover computed in steps 3 and 4. In step 2(b), we notice that if $t_i = 1$, then u_i is included in the minimum weighted vertex cover (computed in step 4). Further, in this case, all the edges that are incident on u_i are removed from the graph; this reflects the fact that these edges would automatically be covered by the inclusion of u_i . The truth of the Lemma then follows simply from the definition of $\mathcal{P}_{G,U}(t)$.

Lemma 2. The procedure ‘COMPUTE-MIN-PROJECTION’ (Figure 4) computes $\min_t \mathcal{P}_{G,U}(t)$ and $\operatorname{argmin}_t \mathcal{P}_{G,U}(t)$.

Proof. First, we note that the conditions imposed by any k -bit vector t restricts the candidate space for optimization; therefore, $\mathcal{P}_{G,U}(t) \geq W$. Second, let the assignment returned by the algorithm in Figure 4 be $\hat{t}$. From step 2, $\hat{t}$ is consistent with S on the membership of $u_1, u_2 \dots u_k$ in the minimum weighted vertex cover. Conversely, S is a candidate vertex cover in the space for optimization associated with $\mathcal{P}_{G,U}(\hat{t})$ — establishing the condition that $\mathcal{P}_{G,U}(\hat{t}) \leq W$. Put together, we have that for any k -bit vector t , $\mathcal{P}_{G,U}(t) \geq \mathcal{P}_{G,U}(\hat{t})$. This proves that $\hat{t}$ is the required optimal vector of assignments; and clearly, this also proves that $W = \min_t \mathcal{P}_{G,U}(t)$.

4 Weighted Constraints as Projections: The Idea of the CCG

We will now present some important results that relate projections to (Boolean) WCSPs. In doing so, we will introduce the notion of the CCG and study some of its fundamental computational properties. As a first step, we make the simple observation that the result

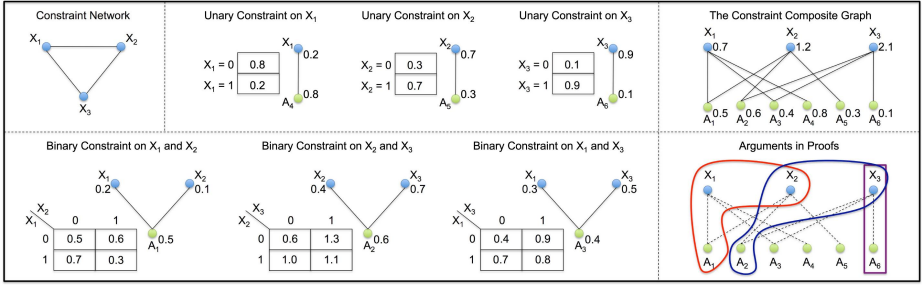


Fig. 5. Shows a WCSP over 3 Boolean variables. The constraint network is shown in the top-left cell, and the 6 unary/binary weighted constraints are shown along with their lifted graphical representations in the $1^{st}/2^{nd}$ rows. The CCG is shown in the top-right cell, and the arguments used in Lemma 3 are illustrated in the bottom-right cell. Here, the encircled subgraphs are indicative of the independence of the corresponding subproblems when all the X -variables are instantiated.

of projecting the minimum weighted vertex cover problem onto an independent set U of the given graph produces a table of size $2^{|U|}$. In some sense, this table can be viewed as a weighted constraint over $|U|$ Boolean variables. Conversely, given a (Boolean) weighted constraint, we can think about designing a “lifted” representation for it so as to be able to view it as the projection of a minimum weighted vertex cover problem in some intelligently constructed node-weighted undirected graph. Later in the paper, we will show how any given Boolean weighted constraint can be represented graphically using a tripartite graph. For now, however, we will concentrate only on the computational aspects of solving Boolean WCSPs when the lifted graphical representations for each of the individual weighted constraints are already given to us.

Figure 5 shows an example WCSP over 3 Boolean variables. Here, there are 3 unary weighted constraints and 3 binary weighted constraints; and their lifted representations (as projections of minimum weighted vertex cover problems) are shown next to each of them. The figure also illustrates how the CCG is obtained from the individual graphs representing each of the weighted constraints. In the CCG, nodes that represent the same variable are simply “merged” — along with their edges — and every “composite” node is given a weight equal to the sum of the individual weights. Figure 6 presents the procedure for constructing the CCG; and the following Lemmas prove some interesting properties of the CCG in the general case.

Lemma 3. Consider a complete assignment q (i.e., an assignment of values to all the variables from their respective domains). The cost of q can be computed simply by running the procedure ‘COMPUTE-PROJECTION-VALUE’ on the CCG.

Proof. We know that the cost of q is given by the sum of the costs defined locally by each weighted constraint. From Lemma 1, the cost defined locally by C_i can be computed by running ‘COMPUTE-PROJECTION-VALUE’ on H_i (see Figure 6). Therefore, it suffices for us to prove that running ‘COMPUTE-PROJECTION-VALUE’ on the CCG is equivalent to running it on each of the individual graphs $H_1, H_2 \dots H_M$ and summing the results. Consider the total weight contributed by the X -nodes — say, X_r ($1 \leq r \leq N$)

ALGORITHM: CONSTRUCT-CONSTRAINT-COMPOSITE-GRAPH

INPUT: (a) a Boolean WCSP with variables $X_1, X_2 \dots X_N$ and constraints $C_1, C_2 \dots C_M$;
 (b) lifted graphical representations $H_1, H_2 \dots H_M$ for each of the weighted constraints — the graph H_i corresponds to the weighted constraint C_i .

OUTPUT: The *constraint composite graph* (CCG) that provides a lifted representation for the entire Boolean WCSP.

(1) For $i = 1, 2 \dots M$:

(a) Give the auxiliary variables in H_i unique names.

(2) For $i = 1, 2 \dots N$:

(a) Simply “merge” all copies of X_i by doing the following:

(A) If X_i has an edge to an auxiliary variable A in any of the graphs $H_1, H_2 \dots H_M$, then introduce an edge between the “merged” copy of X_i and A in the CCG as well.

(B) Set the weight on the “merged” copy of X_i to be equal to the sum of the weights assigned to it in each of the individual graphs $H_1, H_2 \dots H_M$ that it appears in.

(3) RETURN: the resulting “composite” graph.

END ALGORITHM

Fig. 6. A straightforward procedure for building the CCG from the individual graphs that represent each of the weighted constraints in a Boolean WCSP

in particular. When $X_r = 0$, the total weight contributed by X_r in any H_i is 0, and this is also the case in the CCG. When $X_r = 1$, the total weight contributed by X_r is equal to the sum of the weights associated with it in each of the individual graphs that it appears in. By construction (step 2(a)(B) in Figure 6), this total weight is equal to the weight contributed by X_r in the CCG. Now consider the total weight contributed by the auxiliary nodes. It is easy to see that once the nodes $X_1, X_2 \dots X_N$ are instantiated in the CCG, the optimal values for the auxiliary variables coming from any graph are *independent* of the optimal values for the auxiliary variables coming from any other graph; and this establishes that any auxiliary node — say, coming from the graph H_j — is chosen to be in the minimum weighted vertex cover of the CCG if and only if it is chosen to be in the minimum weighted vertex cover of H_j . Therefore, the total weight contributed by the auxiliary nodes also remains the same in the CCG — hence proving the Lemma.

Lemma 4. The optimal (minimum) cost complete assignment q^* (for the given WCSP) can be computed simply by running the procedure ‘COMPUTE-MIN-PROJECTION’ on the CCG.

Proof. From Lemma 2, the assignment returned by running the procedure ‘COMPUTE-MIN-PROJECTION’ (on the CCG) is optimal with respect to the CCG. From Lemma 3, the cost of any complete assignment can be computed from the CCG. Put together, the returned assignment is optimal for the given (Boolean) WCSP — hence proving the Lemma.

5 Graphical Representations for Boolean Weighted Constraints

So far, we have studied how a given Boolean WCSP can be solved by computing a minimum weighted vertex cover for its associated CCG. This procedure, however, required

$X_i X_k$						$P(X_i, X_j, X_k) = a_{000} + a_{100}X_i + a_{010}X_j + a_{001}X_k + a_{110}X_iX_j + a_{101}X_iX_k + a_{011}X_jX_k + a_{111}X_iX_jX_k$				
X_i	0	00	01	11	10		$P(0,0,0) = c_{000}$	$P(0,1,0) = c_{010}$	$P(1,0,0) = c_{100}$	$P(1,1,0) = c_{110}$
	1	c_{100}	c_{101}	c_{111}	c_{110}		$P(0,0,1) = c_{001}$	$P(0,1,1) = c_{011}$	$P(1,0,1) = c_{101}$	$P(1,1,1) = c_{111}$

Fig. 7. Illustrates how a (Boolean) weighted constraint can be represented as a multivariate polynomial. Here, $P(X_i, X_j, X_k)$ is the required polynomial, and its coefficients can be computed simply by solving a system of 8 linear equations with 8 unknowns.

that the graphical representations for each of the weighted constraints already be given to us. In this section, we will present a simple polynomial-time algorithm for building a *tripartite* graph representation for any given Boolean weighted constraint.

The first step in our construction is to represent the given Boolean weighted constraint as a multivariate polynomial.⁶ Consider the example ternary weighted constraint shown in Figure 7. The weighted constraint can be encoded as a multivariate polynomial in X_i , X_j and X_k ; and the polynomial is of degree 1 in each of these variables.⁷ The coefficients of the polynomial can be computed by using a standard *Gaussian Elimination* procedure for solving systems of linear equations. The linear equations themselves arise from substituting different combinations of values to the variables X_i , X_j and X_k , and equating the results to the corresponding entries in the weighted constraint.⁸ One way to build a graphical representation for a given weighted constraint is therefore to simply do the following: (a) build the graphical representations for each of the individual terms in the multivariate polynomial (constructed as above), and (b) “merge” these individual graphical representations (as in Figure 6). The correctness of this procedure follows merely from the same arguments used in the proof of Lemma 3.

We will now show how to construct graphical representations for each of the individual terms in a multivariate polynomial. We will treat three different cases: (1) *linear terms*, (2) *negative nonlinear terms*, and (3) *positive nonlinear terms*. First, consider building a graphical representation for a given linear term (allowed to be +ve or -ve). It is easy to see that any such term can be represented using a single edge that connects the corresponding variable to an auxiliary node. The non-negative weights on the two nodes of this trivial graph are set appropriately as shown in Figure 8(a). Now consider building a graphical representation for a given negative nonlinear term — say, $-w \cdot (X_i \cdot X_j \cdot X_k)$ when $w > 0$. We claim that a simple “flower” structure (as shown in Figure 8(b)) serves the requirements. The “flower” structure makes use of one auxiliary node that is connected to all the variables appearing in the term. A unit weight is assigned to all the original variables while a weight of w is assigned to the auxiliary node. Since the only case in which the auxiliary node is excluded from the minimum weighted vertex cover is when all the original variables are set to 1, the weight of the minimum weighted vertex cover — for the example in Figure 8(b) — is $X_i + X_j + X_k + w - w \cdot (X_i \cdot X_j \cdot X_k)$.

⁶ This is a common technique used in coding/complexity theory.

⁷ In general, if the domain of a variable is $\{0, 1 \dots K\}$, then the multivariate polynomial is of degree K in this variable.

⁸ It is also easy to see that the number of terms in the polynomial is equal to the size of (number of entries in) the weighted constraint.

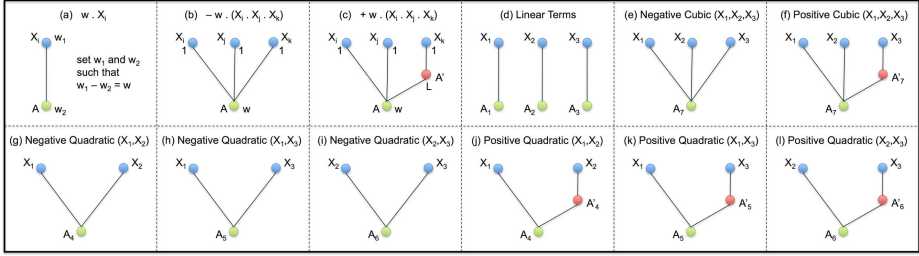


Fig. 8. The diagrams in (a), (b) and (c) illustrate the graphical representations used for linear, negative nonlinear and positive nonlinear terms respectively. The diagrams in (d)-(l) show the *basis* graphs for weighted constraints over the Boolean variables X_1, X_2 and X_3 . The blue nodes represent the original X -variables. The green and red nodes represent the auxiliary variables. The red nodes are referred to as “thorns” in the running text.

Now treating/cancelling the linear lower-order terms as explained before, we have a graphical representation for the term $-w \cdot (X_i \cdot X_j \cdot X_k)$ as required.

Finally, consider building a graphical representation for a given positive nonlinear term. Unlike negative nonlinear terms, positive nonlinear terms do not always have bipartite graph representations.⁹ However, it is easy to construct tripartite graph representations for such terms. First, we note that in the constructions presented so far, a simple graph-theoretic trick allows us to substitute $(1 - X_l)$ for X_l (where X_l is any of the original Boolean variables). Figure 8(c) shows how this is done for an example variable X_k by introducing an intermediate node with a large weight L on it. Assigning (as before) unit weights to all the original variables and a weight of w to the auxiliary node, a “flower” structure over the variables X_i, X_j and X_k that bears an intermediate node — referred to as the “thorn” — between X_k and the auxiliary node yields a minimum weighted vertex cover of weight $X_i + X_j + X_k + L \cdot (1 - X_k) + w - w \cdot (X_i \cdot X_j \cdot (1 - X_k))$. Treating lower-order terms as shown before and/or recursively, such a graphical structure essentially represents the positive nonlinear term $+w \cdot (X_i \cdot X_j \cdot X_k)$ as required ($w > 0$). In general, the graphical representation for a positive nonlinear term simply falls out of constructing a “flower” structure over the participating variables (using a single auxiliary node), and introducing a “thorn” (intermediate node of large weight) for one of the variables. We also note that by the introduction of a “thorn,” the graph no longer remains bipartite; instead, it becomes tripartite as shown in Figure 8(c).

Although the above constructions for individual terms allude to treating lower-order terms recursively, the size of the graph representing any given term is inconsequential. Instead, it is more important to study the size of the graph representing the entire weighted constraint. It can be easily observed that all of the graphs used for representing different terms in the multivariate polynomial are only of two kinds: (a) “flowers,” and (b) “flowers” with “thorns.”¹⁰ These graphs therefore constitute a *basis* for representing any Boolean weighted constraint (see Figure 8(d)-(l)). Further, although the number

⁹ The X -variables need to be in the same partition.

¹⁰ We can assume that the variables are ordered in some way, and that in any “flower” bearing a “thorn,” the “thorn” is always associated with the variable that appears lowest in this ordering.

of these different basis graphs is exponential in the number of variables, it only corresponds directly to the size of (number of entries in) the weighted constraint itself.

Theorem 5. Any given Boolean WCSP has a tripartite CCG associated with it; and the size of this CCG is only polynomial in the size of the WCSP.

Proof. We know that any Boolean weighted constraint can be cast as a multivariate polynomial. We also know that the -ve/+ve terms in this polynomial can be represented using bipartite/tripartite graphs. Further, from Figure 6, it is evident that when every weighted constraint in a Boolean WCSP has a lifted bipartite/tripartite graph representation with the X -variables belonging to the same partition, then the CCG is tripartite with all of the X -variables belonging to the same partition. The truth of the Theorem then follows from the observations made above.

6 Hybrid Computational Properties of the CCG

In the foregoing sections, we studied: (a) how to build graphical representations for each of the weighted constraints in a given Boolean WCSP, (b) how to build the CCG from these graphs, and (c) how computing the minimum weighted vertex cover for the CCG yields a solution to the original WCSP. While (a) and (b) are simple polynomial-time procedures, (c) is NP-hard in general.¹¹ Nonetheless, in this section, we will study the complexity of solving the minimum weighted vertex cover problem on the CCG associated with a given Boolean WCSP. In particular, we will exploit the fact that it can be efficiently solved in at least two different cases: (case 1) when the graph has bounded treewidth, and (case 2) when the graph is bipartite. We will show that (case 1) corresponds to the original constraint network having bounded treewidth, and that (case 2) corresponds to the presence of good numerical structure in the weighted constraints.¹²

6.1 CCG Captures the Topological Structure of the Constraint Network

In this subsection, we will constructively show that the treewidth of the CCG is essentially equal to that of the original constraint network — i.e., we will provide an elimination ordering on the nodes of the CCG that yields the same induced width as a proposed elimination ordering on the original variables in the constraint network.

Consider a proposed elimination ordering $\langle X_{i_1}, X_{i_2} \dots X_{i_N} \rangle$ on the original variables. Suppose that this ordering produces an induced width of w on the constraint network of the given Boolean WCSP. Let $\{\text{auxiliary variables}\}$ denote an arbitrary ordering on the auxiliary variables in the CCG. (Note that “thorns” are also auxiliary variables.) We claim that the following elimination ordering on the nodes of the CCG produces the same induced width w for it: $\langle \{\text{auxiliary variables}\}, X_{i_1}, X_{i_2} \dots X_{i_N} \rangle$.

¹¹ Although the CCG is not a general graph, but tripartite in nature, the minimum weighted vertex cover problem on tripartite graphs is also known to be NP-hard.

¹² Graphical structures called *microstructure complements* have been studied by other researchers in the context of identifying/exploiting symmetry in CSPs/WCSPs [16]. Unlike CCGs, however, microstructure complements do not exhibit the desired hybrid computational properties.

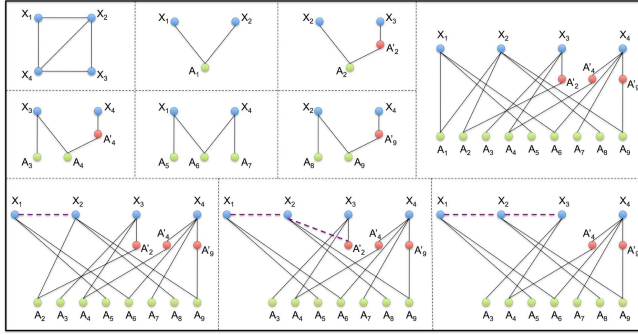


Fig. 9. Shows an example WCSP over 4 Boolean variables. The top-left cell shows the constraint network; and the graphical representations for the individual weighted constraints appear in the first two rows. The CCG is shown in the top-right cell. The graphs in the bottom row (from left to right) illustrate the process of variable-elimination when A_1 , A_2 and A'_2 are (respectively) eliminated. The newly induced edges are shown using purple dashed lines. After all the auxiliary variables are eliminated, the resulting graph is the same as the original constraint network.

The correctness of our claim follows from the “compositional” nature of the CCG. From Figure 6, we know that unique auxiliary variables are used for each of the weighted constraints. This means that there is no edge between any auxiliary variable and any other variable that does not explicitly participate in the corresponding constraint. Now consider what happens when all the auxiliary variables are eliminated before any of the X -variables are eliminated. In particular, consider what happens when all the auxiliary variables created for a weighted constraint C_i are eliminated. Because the only variables that these auxiliary variables are connected to are the ones that explicitly participate in C_i , the set of edges that are induced in the CCG after they are eliminated constitute a clique on the X -variables participating in C_i . However, this clique is the same as the one that is used in building the constraint network of the Boolean WCSP. After all the auxiliary variables are eliminated, therefore, the graphical structure that remains is the same as the constraint network on the original X -variables (see Figure 9). Moreover, following the proposed elimination ordering on these X -variables — viz., $\langle X_{i_1}, X_{i_2} \dots X_{i_N} \rangle$ — trivially results in the same induced width w .

Three things are worth noting in the above arguments. One, while the auxiliary variables are being eliminated, the size of the largest clique that we encounter is equal to the largest arity (K) of the weighted constraints. This factor can be safely ignored since the treewidth is lower-bounded by K , and is in fact typically much larger than K . Two, the number of auxiliary variables created for any weighted constraint may be exponential in the arity of that constraint. However, this does not come as a surprise since the size of (number of entries in) the weighted constraint is itself exponential in the arity of the constraint; and moreover, as argued before, this exponential factor can be ignored in comparison with the treewidth of the CCG. Three, an important outcome of the above arguments is that any algorithm/heuristic that works well in theory/practice for minimizing the induced width of the constraint network can be directly adapted to the CCG as well — just by eliminating all the auxiliary variables (in any order) before using the proposed elimination ordering on the X -variables.

6.2 CCG Captures the Numerical Structure of the Weighted Constraints

In this subsection, we will show how the CCG also captures the numerical structure of the weighted constraints. In particular, we will show that many classes of Boolean WCSPs that are tractable by virtue of the structure in their weighted constraints in fact have bipartite CCGs. We will begin with the following important Theorem.

Theorem 6. The language $\mathcal{L}_{bipartite}^{Boolean}$ of all Boolean weighted constraints that have bipartite graph representations is tractable.

Proof. From the procedure in Figure 6, it is clear that when every weighted constraint in a Boolean WCSP has a lifted bipartite graph representation with the X -variables belonging to the same partition, then the CCG is also bipartite with all of the X -variables belonging to the same partition. The truth of the Theorem then follows simply from the fact that in any bipartite graph, the minimum weighted vertex cover problem can be solved efficiently in polynomial time [5].

Drawing on the implications of the above Theorem, we will first consider Boolean WCSPs restricted to *binary* constraints. Even in this simple case, the kinds of problems that we can speak about significantly differ in their associated tractability results. For example, both the *min-st-cut* problem and the *max-cut* problem can be encoded as Boolean WCSPs with binary constraints;¹³ but while the former can be solved in polynomial time, the latter problem is NP-hard. Figure 10 sheds some light on such WCSPs. In particular, it shows that: (a) any Boolean unary weighted constraint has a simple bipartite graph representation; (b) the *min-st-cut* constraints are particular cases of weighted constraints that have simple bipartite graph representations as V -structures; and (c) the *max-cut* constraints are particular cases of weighted constraints that have simple graphical representations as U -structures (that are not bipartite). The following important conclusions can now be drawn immediately: (a) a generalization of the *min-st-cut* problem with arbitrary unary weighted constraints is tractable;¹⁴ (b) the entire space of weighted constraints resulting from varying the parameters w_1 , w_2 and w_3 (in the V -structures) is tractable; and (c) the absence of bipartite graph representations for the *max-cut* constraints is consistent with the intractability of the *max-cut* problem.¹⁵

As a next step, we present a simple example in Figure 10 to illustrate how we can generalize our techniques to *non-binary* constraints as well. The mere existence of a bipartite graph representation establishes the tractability of the kinds of ternary weighted constraints shown in the figure. Further, setting different values for w_1 and w_2 yields different kinds of tractable (convex) functions. In general, several parameters in the bipartite graph representations can be adjusted to yield a multitude of tractable classes of WCSPs. These include: (a) the weights on the original and auxiliary variables, (b) the graphical structure of the bipartite graphs, and (c) the encoding mechanism between the

¹³ For the *min-st-cut* problem, unary weighted constraints on X_s and X_t ensure that they are assigned the values 0 and 1 respectively; and for every edge $\langle v_i, v_j \rangle$ in the graph, a binary weighted constraint between X_i and X_j yields a value of 1 when $X_i \neq X_j$, and 0 otherwise. For the *max-cut* problem, the binary weighted constraints are reversed.

¹⁴ Similar tractable problems were identified in [11] using different combinatorial arguments.

¹⁵ Some other related useful results also appear in [6].

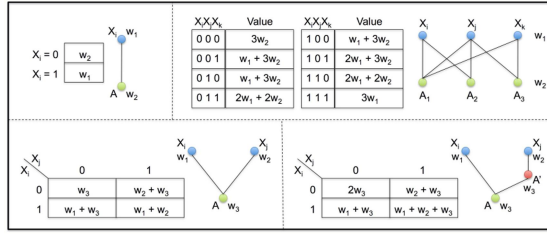


Fig. 10. The top-left cell shows that any Boolean unary weighted constraint has a trivial bipartite graph representation. The bottom-left/bottom-right cell shows the bipartite/tripartite graph representation (V -structure/ U -structure) for generalizations of the *min-st-cut*/*max-cut* constraints. The *min-st-cut* and *max-cut* constraints become apparent when $w_1 = w_2 = w_3/2$ (and the additive constants are factored out). The top-right cell shows a bipartite graph for a weighted constraint over the 3 Boolean variables X_i, X_j and X_k . Here, A_1, A_2 and A_3 are the auxiliary variables.

values of individual variables and the presence/absence of certain nodes in the minimum weighted vertex covers [12]. A very simple argument for establishing the tractability of the *hypergraph min-st-cut* problem, for example, is to note that when formulated as a Boolean WCSP, the weighted constraints have (bipartite) “flower” structure representations that generalize the V -structures of the *min-st-cut* problem.¹⁶ Another important result is that we can efficiently solve the minimization problem for any objective function that can be expressed as a *bounded-degree multivariate polynomial with the positive coefficients being restricted to linear terms*. This result follows directly from the discussions in Section 5 where we explicitly showed that all linear and negative nonlinear terms have simple bipartite graph representations.¹⁷

Finally, from Theorem 5, we see that the complexity of solving a given instance of the Boolean WCSP is exponential only in the size of the smallest partition — in terms of the number of nodes — of the tripartite graph constructed for it. This is because the minimum weighted vertex cover problem can be solved in polynomial time for a bipartite graph; and every possible combination of decisions to include/exclude the nodes of the smallest partition in the vertex cover can be evaluated to find the optimal one. We note that one of these partitions consists of the original N variables — leading us to the

¹⁶ The tractability of the *hypergraph min-st-cut* problem is also established in [13]. Nonetheless, it is worth noting that the simple arguments presented here generalize to the entire space of weighted constraints resulting from varying the parameters of the “flower” structures.

¹⁷ In turn, this result is also equivalent to the tractability of maximization problems defined for the so-called *negative-positive* pseudo-Boolean functions [15]. Similar and more general results appear in [1] and [3]. The techniques presented in [3] employ the *posiform* representations of pseudo-Boolean functions and work on their associated *conflict graphs*. Here, the equivalence between *posiform maximization* and *weighted graph stability* is exploited; and posiforms with corresponding conflict graphs having special characteristics are studied. Nonetheless, it is worthwhile to note that the arguments presented in this paper can be generalized very easily to non-Boolean domains [12]; and it is also important to note (once again) that our transformation techniques preserve the treewidth for general Boolean WCSPs — thereby providing for the desired hybrid computational properties of the CCG.

obvious upper bound of characterizing the problem to be exponential in N . However, this partition may not be the smallest — in which case our framework yields a much tighter characterization, and allows us to computationally leverage the numerical structure of the weighted constraints. (When there is sufficient numerical structure, for example, the CCG is only bipartite, and such WCSPs can be solved in polynomial time.)

7 Conclusions and Future Work

We introduced the notion of a *constraint composite graph* (CCG) associated with a given (Boolean) WCSP. We provided simple polynomial-time procedures for constructing it, and we showed that the given Boolean WCSP can be solved by computing a minimum weighted vertex cover for its CCG. We established that, unlike the constraint network, the CCG provides a unifying framework for characterizing/exploiting both the structure of the variable-interaction graph as well as the structure of the weighted constraints.¹⁸ As a consequence of these hybrid computational properties, we emphasize the importance of studying the CCG rather than the constraint network (for a given WCSP). Further, the compositional nature of the CCG is a highly attractive property that allows us to reason about the weighted constraints only *locally*. This property was used to provide simple arguments for establishing the tractability of the language $\mathcal{L}_{bipartite}^{Boolean}$ (that in turn captures many useful subclasses of WCSPs). For future work, we are interested in a more detailed study of CCGs — especially for general non-Boolean WCSPs.

References

1. Billionnet, A., Minoux, M.: Maximizing a Supermodular Pseudo-Boolean Function: A Polynomial Algorithm for Supermodular Cubic Functions. *Disc. Appl. Math.* (1985)
2. Bistarelli, S., Fargier, H., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G.: Semiring-Based CSPs and Valued CSPs: Basic Properties and Comparison. *Over-Constrained Systems* (1996)
3. Boros, E., Hammer, P.: Pseudo-Boolean Optimization. *Disc. Appl. Math* (2002)
4. Boutilier, C., Brafman, R., Domshlak, C., Hoos, H., Poole, D.: CP-nets: A Tool for Representing and Reasoning with Conditional Ceteris Paribus Preference Statements. *J. Artif. Intell. Res. (JAIR)* 21, 135–191 (2004)
5. Cormen, T., Leiserson, C., Rivest, R.: *Introduction to Algorithms*. MIT Press, Cambridge (1990)
6. Creignou, N., Khanna, S., Sudan, M.: *Complexity Classifications of Boolean Constraint Satisfaction Problems*. SIAM Monographs on Discrete Mathematics and Applications (2001)
7. Dechter, R.: *Constraint Networks (Survey)*. *Encyclopedia of Artificial Intelligence* (1992)
8. Do, M., Benton, J., van den Briel, M., Kambhampati, S.: Planning with Goal Utility Dependencies. In: *IJCAI 2007* (2007)

¹⁸ We implicitly showed that the complexity of solving a given Boolean WCSP is exponential only in $\min(s, t)$ where $s = \text{size of the smallest partition in the tripartite CCG constructed for it}$, and $t = \text{treewidth of its CCG/constraint network}$. Although more sophisticated hybrid tractability results can be derived from analyzing the CCG further, it is important to note that the CCG is also valuable merely because it provides an appropriate framework for doing so.

9. Hochbaum, D.: Efficient Bounds for the Stable Set, Vertex Cover and Set Packing Problems. *Disc. Appl. Math.* 6 (1983)
10. Kolmogorov, V.: Primal-Dual Algorithm for Convex Markov Random Fields. Microsoft Tech. Rep. MSR-TR-2005-117 (2005)
11. Kolmogorov, V., Zabih, R.: What Energy Functions can be Minimized via Graph Cuts? *Transactions on Pattern Analysis and Machine Intelligence* 26(2), 147–159 (2004)
12. Kumar, T.K.S.: Lifting Techniques for Weighted Constraint Satisfaction Problems. In: Tenth International Symposium on Artificial Intelligence and Mathematics (ISAIM 2008) (2008)
13. Mak, W., Wong, D.: A Fast Hypergraph Min-Cut Algorithm for Circuit Partitioning. *VLSI Journal* 30(1), 1–11 (2000)
14. Micali, S., Vazirani, V.: An $O(\sqrt{|V|}|E|)$ Algorithm for Finding Maximum Matching in General Graphs. In: FOCS 1980 (1980)
15. Rhys, J.: A Selection Problem of Shared Fixed Costs and Network Flows. *Management Sci.* 17(3), 200–207 (1970)
16. Rossi, F., van Beek, P., Walsh, T.: *Handbook of Constraint Programming*. Elsevier, Amsterdam (2006)

From High Girth Graphs to Hard Instances^{*}

Carlos Ansótegui, Ramón Béjar, César Fernández, and Carles Mateu

Dept. of Computer Science, Universitat de Lleida, Spain

{carlos,ramon,cesar,carlesm}@diei.udl.cat

Abstract. In this paper we provide a new method to generate hard k -SAT instances. Basically, we construct the bipartite incidence graph of a k -SAT instance where the left side represents the clauses and the right side represents the literals of our Boolean formula. Then, the clauses are filled by incrementally connecting both sides while keeping the girth of the graph as high as possible. That assures that the expansion of the graph is also high. It has been shown that high expansion implies high resolution width w . The resolution width characterizes the hardness of an instance F of n variables since if every resolution refutation of F has width w then every resolution refutation requires size $2^{\Omega(w^2/n)}$. We have extended this approach to generate hard n -ary CSP instances. Finally, we have also adapted this idea to increase the expansion of the system of linear equations used to generate XOR-SAT instances, being able to produce harder satisfiable instances than former generators.

1 Introduction

Providing challenging benchmarks for the SAT and the CSP problems is of a great significance for both the experimental evaluation of SAT and CSP solvers and for the theoretical computer science community. Every year, new benchmarks are submitted to the SAT and CSP competitions. Our aim is to provide a method for generating hard k -SAT and n -ary CSP instances.

In order to do that, we look at the field of propositional proof complexity, where it turns out that graph expansion has been established as a key to hard formulas for resolution (e.g. [1]), but also for other proof systems like the polynomial calculus [2]. Roughly speaking, an expander graph is a graph $G=(V,E)$ that, for any, not too big, subset of vertices S , its set of neighbors in $V \setminus S$ is big, compared with $|S|$, so S is well connected with the rest of the graph.

Basically, our approach is based on creating a bipartite graph with a high expansion, and then from this graph we generate the k -SAT or n -ary CSP instance. In particular, for the k -SAT instances one of the partitions of the graph represents the set of clauses and the other one the set of literals. Edges represent which literals belong to which clauses. We call this graph the literal incidence

^{*} Research partially supported by projects TIN2006-15662-C02-02, TIN2007-68005-C04-02 and José Castillejo 2007 program funded by the *Ministerio de Educación y Ciencia*.

graph of a SAT instance. Analogously, for the CSP instances, one partition represents nogood tuples and the other one pairs (variable, value).

The way our method tries to get a high expansion on the bipartite graph is to incrementally build the graph while keeping the girth as high as possible. The girth is the length of the shortest cycle of the graph. It is known that high girth implies high expansion [3].

The instances we generate with this method can be used to test the efficiency of SAT and CSP solvers. Moreover, expander graphs have many other applications, like efficient communication networks [4], linear-time decodable low density parity check codes [5] and cryptographic hash functions [6].

We have compared our approach against other methods in the SAT Community [7,8] which try to get hard SAT instances by balancing the occurrences of literals, and thus the degrees of the vertices at the literal incidence graph become also balanced. Previous results, e.g. [5], show that balanced bipartite graphs also tend to have a high expansion. Our empirical results confirm that our method generates harder instances. We have also introduced a modification of the generator of satisfiable SAT instances regular k -XORSAT [9], that seems to be very hard thanks to the expansion properties of its underlying system of linear equations. So, we modify the generation of this system by using our high girth bipartite graphs, instead of the original random regular bipartite graphs, for building the system, and show that the hardness of the instances increases by orders of magnitude.

In the CSP field there are four standard methods, denoted A, B, C and D, for generating hard random binary CSPs [10,11]. In [12] the model E was introduced in order to overcome some deficiencies of the previous models. For n -ary CSPs some extensions of random binary CSPs models have been defined [13]. At the section of experimental results we compare our method against the n -ary version of Model E, because the set of parameters in model E (domain size, number of variables and total number of nogoods) is the same as in our High-Girth model, thus giving a natural comparison.

The rest of the paper is organized as follows. First, we introduce a set of previous definitions. Second, we discuss the related work. Third we present our method for generating hard k -SAT and n -ary CSP instances. Finally, we show the experimental investigation on SAT and CSP solvers.

2 Preliminaries

We consider undirected bipartite graphs in this paper, although we mention some results about undirected general graphs. A *bipartite graph* G is a pair $(V_1 \cup V_2, E)$, where V_1 is the left partition and V_2 is the right partition of the set of vertices, and $E \subseteq V_1 \times V_2$. We say that G is (k_1, k_2) -regular if the degree of any vertex from V_1 is k_1 and the degree of any vertex from V_2 is k_2 . Observe that $|V_1|k_1 = |V_2|k_2$. In the same way, it is $(k_1, -)$ -regular if we only fix the degree of the vertices in V_1 .

Definition 1. *The girth of a graph G , $g(G)$, is the length of the shortest cycle in G . If G is acyclic then, by definition, $g(G) = \infty$.*

There is a limit on how large the girth can be, for a graph with V vertices and minimum degree d . This limit is $2 \log_{d-1}(|V|)$ [14].

Definition 2. *We say that a family of k -regular graphs $(G_m)_{m \geq 1}$, with $|V_m| \rightarrow \infty$ as $m \rightarrow \infty$, has high girth if, for some constant $0 < C < 2$, $g(G_m) \geq (C + o(1)) \log_{k-1} |V_m|$, where $o(1) \rightarrow 0$ as $m \rightarrow \infty$.*

Random k -regular graphs have expected girth slightly greater than 3 [15], but there exist constructions of graphs with high girth. The one with the highest girth is that of [14] where they achieve girth $(4/3) \log_{k-1}(|V|)$ and high expansion.

Definition 3. *The expansion of a subset X from the vertices of $G = (V_1 \cup V_2, E)$ is defined to be the ratio $|N(X)|/|X|$, where $N(X) = \{w \in (V_1 \cup V_2) \setminus X \mid \exists v \in X, \{v, w\} \in E\}$ is the set of outside neighbors of X .*

When all the neighbors of X are inside X , we have expansion 0. We consider a set high expanding when its expansion is greater than 1, that means that the set of different outside neighbors of X is larger than X , so it is well connected with the rest of the graph.

Definition 4. *A left (α, c) -expander is a bipartite graph $(V_1 \cup V_2, E)$ such that every subset of V_1 of size at most $\alpha|V_1|$ has expansion at least c .*

Usually, smaller sets will have better expansion, the limit being for $\alpha = 1.0$, where expansion cannot be greater than $|V_2|/|V_1|$. Observe that for a $(k, -)$ -regular bipartite graph, the left expansion cannot be greater than k . Analogously, we can also talk about right expanders or expanders in general if we consider the expansion of any possible subset of vertices.

For this work, the following concepts are the main tools used to link complexity with structural properties of k -SAT and n -ary CSP instances.

Definition 5. *Given a k -SAT instance F with set of clauses C , set of variables V and set of literals L , $G(F) = (C \cup V, E)$ is its bipartite variable incidence graph such that $(c, v) \in E$ if and only if variable v appears in clause c . $LG(F) = (C \cup L, E)$ is its bipartite literal incidence graph such that $(c, l) \in E$ if and only if literal l appears in clause c .*

Observe that if $LG(F) = (C \cup L, E)$ is a left (α, c) -expander, then $G(F) = (C \cup V, E)$, will be, at least, a left $(\alpha, c/2)$ -expander.

Definition 6. *Given a CSP instance $P = \langle X, D, C \rangle$, we define the literal incidence graph as the bipartite graph $LG(P) = (NG \cup L, E)$, where for every variable x_i and domain value $j \in \text{dom}(x_i)$ there is a vertex (x_i, j) in L and for every nogood tuple $ng_{i_j} = (v_{j_1} = d_1, v_{j_2} = d_2, \dots, v_{j_k} = d_k)$ associated with a constraint C_i of arity k there is a vertex ng_{i_j} in NG and k edges, one for every pair $(ng_{i_j}, (v_{j_j}, d_{j_j}))$.*

3 Related Work

In this section we survey some previous theoretical results about the expansion of random graphs, and the related work in the SAT and CSP communities.

3.1 Expansion of Random Graphs

The problem of checking whether a graph is an expander is co-NP complete [16]. However, lower and upper bounds on the expansion of a graph have been obtained using spectral graph theory results. Given the adjacency matrix of $G = (V, E)$ we denote its eigenvalues by $\lambda_0 \geq \lambda_1 \geq \dots \geq \lambda_{n-1}$, where $n = |V|$.

For k -regular graphs, we have lower bounds on expansion that depend on λ_1 , such that the lower λ_1 , the higher the expansion. Asymptotically (as $|V| \rightarrow \infty$), for k -regular graphs the best we can hope for λ_1 is to tend to $2\sqrt{k-1}$, and k -regular graphs with $\lambda_1 \leq 2\sqrt{k-1}$ are called Ramanujan graphs [14]¹. On the other hand, Kahale [17] gave a lower bound on expansion that shows that Ramanujan k -regular graphs can have expansion as high as $k/2$ for small sets.

Probabilistic methods have been used to show that regular graphs are almost surely very good expanders. The particular case of k -regular or (k_1, k_2) -regular bipartite graphs have received special attention in the communications community (e.g. [4,5]), and such bipartite graphs are good expanders almost always. So, it seems that regular graphs are promising towards obtaining good expanders, although we will see that *almost* regular graphs can also be excellent expanders, even better than regular graphs. For the case of bipartite graphs we have, for example, that a random $(k, -)$ -regular bipartite graph $(V_1 \cup V_2, E)$ with $|V_1| = |V_2|$ will be a good expander *with probability* $> 1/2$. So, when only the vertices of one part have the same degree, the expansion properties seem to degrade. Observe that this last graph can represent the incidence graph of a random k -SAT instance.

3.2 Related Results in SAT Community

Concerning the resolution complexity of a 3-SAT instance F with n variables, Ben-Sasson and Wigderson [18] proved that if every resolution refutation of F requires width w , then every resolution refutation of F requires size $2^{\Omega(w^2/n)}$. The width of a resolution refutation is the length of the longest clause in the refutation. Thus, lower bounds on width imply lower bounds on size. Finally, there is a connection between graph expansion and 3-SAT resolution complexity based on this width-size relationship. Consider a 3-SAT instance F with set of clauses C and set of variables V and its bipartite variable incidence graph $G(F) = (C \cup V, E)$. Results presented in [1] imply that any resolution refutation will have width lower bounded by $\lfloor (c-1)\alpha|C|/((2+c)d) \rfloor$, where d is the maximum right-degree of $G(F)$, if $G(F)$ is a left (α, c) -expander. So, any resolution refutation of F will have exponential size if $d = o(|C|)$ and $c > 1, \alpha > 0$, given

¹ $2\sqrt{k-1}$ is only an asymptotic limit on the minimum value of λ_1 , but actual graphs can have a smaller value.

the width-size relationship. So, the higher the expansion of the graph and the smaller the maximum right-degree d , the higher the refutation size lower bound. Moreover, the results also imply that more powerful proof algorithms based on strong k -consistency will also require exponential time for solving the 3-SAT instance under the same circumstances.

As we have discussed, regular graphs tend to be better expanders than general graphs. So, it is not surprising that previous random models for k -SAT based on balancing the literal and variable occurrences generate harder instances, as their incidence graph will tend to have higher expansion. The model described in [7] for 3-SAT (lit-bal-1), generates instances with m clauses by putting $\lfloor \frac{3m}{2n} \rfloor$ occurrences of each literal plus an additional random set of unique literals so that there are exactly $3m$ literals in it. To construct each clause, 3 literals on distinct variables are removed from the bag. If there are less than 3 distinct variables mentioned in literals remaining in the bag, additional distinct variables are randomly selected from the set of all variables and negated with probability $\frac{1}{2}$. This model easily generalizes for $k > 3$. The model described in [8] for k -SAT (lit-bal-2) is very similar, being the main difference that every literal in the resulting formula appears exactly $\lfloor \frac{k \cdot m}{2n} \rfloor$ or $\lfloor \frac{k \cdot m}{2n} \rfloor + 1$ times. By contrast with lit-bal-1 the occurrences of literals can be less balanced.

Recently, models of hard satisfiable k -SAT instances, based on variants of the XORSAT model, have been introduced [9,19,20,21]. The basic ingredient in all these models is that a system of linear equations (mod 2) with at least one solution is converted to an equivalent set of clauses, such that the solutions of the SAT formula correspond to the solutions of the linear system. All these models provide very challenging instances, being the hardest one regular k -XORSAT [9,21], where the running time of DPLL algorithms seems to scale exponentially in the number of variables. It seems that the key for the hardness of regular k -XORSAT is the high expansion they get in the system of linear equations, thanks to the use of a regular bipartite graph for building it, such that the resulting system has n variables and n equations with k variables per equation, and every variable appears in k equations. We will see that by using our High-Girth bipartite graph generation algorithm to generate the system of linear equations we increase the hardness of regular k -XORSAT instances even more.

3.3 Related Results in CSP Community

For binary CSPs, in [22] new methods for generating hard instances were presented, based on balancing both the constraint language and the constraint graph. Also, a method for generating a high girth constraint graph was introduced and it generated the hardest instances. In that work, they link the hardness of the instances to the fact that more balanced graphs tend to have a higher treewidth, thanks to the results that link the treewidth with the graph expansion [23]. Then, given the results that link the treewidth with the level of consistency needed to solve a CSP with k -consistency [24,25] we have that the higher the expansion on the constraint graph, the higher the complexity to solve the problem. Previous work has considered the generation of hard balanced

CSPs (see for example [26,27]), but without linking the balance of the constraint graphs to their treewidth.

We have also the results of [1], that link the complexity of a 3-SAT instance with the expansion of its incidence graph using an encoding of the 3-SAT instance as a CSP instance and using results that link the expansion of its incidence graph to the level of resources needed to win certain combinatorial games. These results are interesting, because one can consider also the same combinatorial games but directly on the incidence graph of a n -ary CSP to find a relation between its expansion and the level of k -consistency needed to solve the CSP, although the possible relation does not seem to be a straightforward generalization of the results in [1]. However, as we will see at the experimental results, our method increases the hardness of n -ary CSPs, so we believe that high lower bounds on complexity also hold for our model.

4 Hard SAT and n -ary CSP Instances

In this section we introduce our generation method for hard k -SAT and n -ary CSP instances.

4.1 Expansion, Balance, and Girth

To get an idea on which is the typical structure of a good expander graph, consider the expansion of subsets of the left partition of the bipartite graphs (a) and (b) of Fig. 1. As the vertices in the left partition of both graphs have degree 3, the expansion when $|S| = 1$ is 3. Consider now sets with $|S| = 2$. In the graph (a), the set $N(S)$ for any left subset S with $|S| = 2$ is always the entire right partition, so the expansion is $4/2$. But for graph (b) the set $N(\{1, 4\})$ does not contain the vertex 7, and so the expansion is only $3/2$ due to the poor connectivity of vertex 7. For $|S| = 3$ the situation is similar. For graph (a) any left subset with $|S| = 3$ is connected to the whole right partition (its expansion is $4/3$), but for graph (b) $N(\{2, 1, 4\}) = \{5, 6, 8\}$, so the expansion is 1. Thus, we observe that, due to the unbalanced degrees of the right partition of graph (b), the vertex expansion of the left subsets is not as good as in graph (a), where all the degrees are equal.

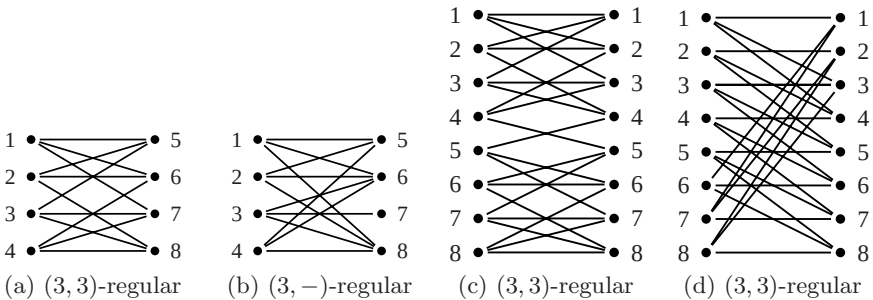


Fig. 1. Example of bipartite graphs with different expansion

Algorithm 1. Algorithm for generation of high girth $(k, -)$ -regular bipartite graphs $(V_1 \cup V_2, E)$

```

input :  $V_1, V_2, k$ 
output: a bipartite  $(k, -)$ -regular graph  $(V_1 \cup V_2, E)$ 
Initialize  $E$  with a random matching from  $V_1$  to  $V_2$ 
(every vertex from  $V_1$  will have degree 1)
for  $i = |V_1| + 1$  to  $k|V_1|$  do
     $L_T := \{u \in V_1 \mid \text{degree}(u) < k\}$ 
     $R_T := \{u \in V_2 \mid \text{degree}(u) \leq \text{degree}(v), \forall v \in V_2\}$ 
     $\text{maxdist} := 1$ 
    while ( $\text{maxdist} = 1$ ) do
         $T := \{(u, v) \mid (u, v) \in L_T \times R_T, \text{dist}(u, v) \geq \text{dist}(x, y) \forall (x, y) \in L_T \times R_T\}$ 
         $d_{\min} := \text{degree}(u), \text{ where } u \in R_T$ 
         $\text{maxdist} := \text{dist}(u, v), \text{ where } (u, v) \in T$ 
        if ( $\text{maxdist} = 1$ ) then
             $R_T := \{u \in V_2 \mid \text{degree}(u) = d_{\min} + 1\}$ 
     $E := E \cup (u, v), \text{ where } (u, v) \in T$ 

```

However, the balance of the degrees does not provide a complete characterization of good expander graphs. Consider the graphs (c) and (d) of Fig. 1, that are both equally balanced. Graph (c) has several cycles of length 4, and thus girth 4, and its expansion for some left subsets of size 4 is $5/4$. By contrast, in graph (d) the minimum expansion for left subsets of size 4 is $7/4$. The main structural difference with the graph (c) is actually its girth, that in this case is 6. Actually, Kahale [3] shows that high girth ($O(\log_{k-1}(|V|))$) implies high expansion, at least for subsets of size at most $|V|^\delta$, with $\delta < 1$. So, one way to obtain graphs with good expansion is to get high girth graphs. In [28] it is presented an algorithm for building graphs with degrees $k-1$, k and $k+1$ and high girth. The algorithm we present in the next subsection follows the same approach to build bipartite graphs with high girth. This graph will be used to build hard k -SAT and n -ary CSP instances .

4.2 High Girth Bipartite Graphs

The algorithm presented by Chandran in [28] works for general (non-bipartite) graphs. It builds the graph in a greedy fashion, introducing edges one by one, connecting vertices which are at large distances in the current graph, in such a way that the degrees are maintained almost balanced and the girth obtained is $O(\log_{k-1}(|V|))$. The algorithm initiates the construction by building a matching between the vertices, and then starts to insert edges between vertices with maximum distance between them.

For building the literal incidence graph of a k -SAT formula (and similarly for a k -ary CSP formula), we need to build a $(k, -)$ -regular bipartite graph $(V_1 \cup V_2, E)$, where V_1 is the set of clauses and V_2 is the set of literals. Algorithm 1 does this, but trying to keep the girth as high as possible, using the same technique of linking vertices which are at large distances in the current graph. It starts the

process by creating a random matching from V_1 to V_2 , such that every vertex from V_1 will have degree 1 and every vertex from V_2 will have degree either $\lfloor |V_1|/|V_2| \rfloor$ or $\lfloor |V_1|/|V_2| \rfloor + 1$. Because this matching does not create any cycle, it starts with girth equal to ∞ . Then, at every step it selects an edge from the subset of edges (u, v) with $u \in V_1$ and $v \in V_2$, such that $\text{degree}(u) < k$ and $\text{degree}(v)$ is minimum among all the current degrees in V_2 . From this subset of edges, it selects one (u', v') with the maximum distance between u' and v' , because this way the new created cycle is of maximum length. This process ends when the graph has $|V_1|k$ edges.

One of the keys of Chandran's algorithm is that at any stage of the process it maintains the degrees of the vertices almost balanced. Similarly, the algorithm in [29] creates a bipartite graph with balanced degrees and guaranteed high girth, but it only works for $|V_1| = |V_2|$. By contrast, for the purpose of using a high girth bipartite graph for generating SAT instances with any possible ratio $r = |C|/|V|$, we need a more general algorithm. That is, able to work with non-equal partition sizes. Observe that this implies that the degrees of the vertices on the right partition (literals) will be higher than on the left, the bigger the ratio r , the bigger the difference between the degrees on the left and right partitions.

However, we can show that the degrees of the right vertices (V_2) of our bipartite graph will be almost balanced.

Lemma 1. *For any fixed k and r and $|V_1| = r|V_2|$ this algorithm creates a $(k, -)$ -regular bipartite graph $(V_1 \cup V_2, E)$, where the degree of any vertex in V_2 will be asymptotically (as $|V_2| \rightarrow \infty$), from the set $\{d - 1, d, d + 1\}$, where $d = \lfloor rk \rfloor$.*

Proof. After inserting the initial matching from V_1 to V_2 , the degree of any vertex from V_2 will be $\lfloor r \rfloor$ or $\lfloor r \rfloor + 1$. If the algorithm always succeeds in selecting a minimum degree vertex from V_2 , then at the end of the process any vertex will have degree $\lfloor rk \rfloor = d$ or $d + 1$. We define a minimum degree vertex from V_2 that is linked with all the current available vertices from V_1 (vertices with less than k edges) as a blocked vertex.

Consider a blocked vertex v from V_2 . The number of available vertices from V_1 , but already linked to some vertex, when $k|V_1| - E$ edges are already inserted, will be, at least, $\lceil E/(k - 1) \rceil$. This situation corresponds to the extremal case when all the available vertices are linked with only one vertex, and the rest of vertices from V_1 have degree k .

So, the first time when a blocked minimum degree vertex v from V_2 can appear is when all the vertices from this minimum set of available vertices can appear linked with v . That is, when $\lceil E/(k - 1) \rceil$ coincides with the current minimum degree from V_2 :

$$\left\lceil \frac{E}{(k - 1)} \right\rceil = \left\lfloor \frac{k|V_1| - E}{|V_2|} \right\rfloor$$

Then, this will never occur before E satisfies:

$$E = \frac{k(k - 1)|V_1|}{|V_2| + (k - 1)} < \frac{k(k - 1)|V_1|}{|V_2|} = rk(k - 1) = O(1)$$

Table 1. Girth of bipartite graphs created by our algorithm, corresponding to 3-SAT, 4-SAT and 5-SAT literal incidence graphs for instances at the peak of hardness

$ V $	3-SAT				4-SAT		5-SAT	
	$ C $	g	$\log_d(2 V + C)$		$ C $	g	$\log_d(2 V + C)$	
62	221	8	5.6		539	6	3.8	
125	447	8	6.2		1,087	6	4.3	
250	895	10	6.9		2,175	6	4.6	
500	1,790	10	7.6		4,350	6	5.1	
1,000	3,560	10	8.3		-	-	-	-

That is, when only a constant number of vertices still wait for one more edge. At this time, we have two possibilities. On the one hand, if $R = kr|V_2| \bmod |V_2| > 0$ because k and r are fixed we have $R = O(f|V_2|)$ with $0 < f < 1$. Let i be the number of minimum degree vertices from V_2 that should receive one more edge. Then, the degree of $R - i$ vertices will be $d + 1$, and the degree of $|V_2| - (R - i)$ vertices will be d . Observe that a third degree $d + 2$ will only be introduced if all the minimum degree vertices from V_2 are blocked. But the maximum number of blocked vertices is always $k - 1$, so only if $|V_2| - R < k - 1$ will be possible to block, at most, the last $k - 1$ minimum degree vertices from V_2 . But $|V_2| - R = O(|V_2|) \gg k - 1$ and asymptotically no vertex of degree $d + 2$ will appear.

On the other hand, if $R = 0$ we will have $|V_2| - i$ vertices with degree d and i vertices with degree $d - 1$. In this case, only if $i < k$ we could have all the minimum degree vertices blocked, and this would lead to the introduction of at most $k - 1$ vertices of degree $d + 1$. $\square$

So, the degrees of the vertices in V_2 will be almost balanced. Actually, the instances we have obtained with this method in our experiments almost always have only two distinct degrees in V_2 , and only exceptionally three distinct degrees.

Regarding the girth, although we cannot ensure the same conditions that guarantee a logarithmic girth like in [28] and [29], our empirical results show that this is the case. Table 1 shows the girth for graphs obtained with our algorithm, compared with the minimum girth we would obtain for a general d -regular graph, with d equal to the floor of the average degree of our bipartite graphs ($d = \lfloor 2|V_1|k/(|V_1| + |V_2|) \rfloor$), if we used Chandran’s algorithm.

Table 1 does not show results for $|V| = 1,000$ for 4-SAT and 5-SAT because the size of the corresponding literal incidence graph is too big in such cases for our generation algorithm to work in reasonable time, even if we are using the best performing polynomial-time algorithm we have found in [30], for the dynamic all-pairs shortest distance problem (DAPSD). For our bipartite graph $(V_1 \cup V_2, E)$ with $|V_1| = |C|$, $|V_2| = 2|V|$ and $n = |V_1| + |V_2|$, the worst-case running time of our algorithm is $O((k - 1)^2 n^3)$, thanks to using the variant of the algorithm of [31] described in [32]. Theoretically, there is a best worst-case algorithm for DAPSD [33], but empirically for our particular graphs the chosen algorithm was the best performing.

By contrast, we also computed the girth for the literal incidence graphs obtained with the balanced SAT model Lit-bal-1, and the girth obtained in

all the instances was always 4, the minimum possible girth for a bipartite graph. Remember that actually there are theoretical results that imply that for random k -regular graphs the average girth is slightly greater than 3 [15].

We can easily use this algorithm to generate the literal incidence graphs of k -SAT and n -ary CSPs, with the goal of obtaining harder instances than the ones we obtain with regular (balanced) graph models. Moreover, we can also use it to generate the XORSAT linear equations system, instead of using a random regular bipartite graph like it is done in [9], with the left partition representing the equations, and the right partition the equations' variables.

5 Experimental Investigation

We have divided our experimental investigation into two subsections. The first one presents a comparison of our method against the most recent k -SAT generators and the classical random k -SAT generator. The second one shows a comparison between model E and our method High-Girth for n -ary CSPs.

5.1 Hard k -SAT Instances

Four methods have been used to generate k -SAT instances : the classical random k -SAT (Random), a generalization of method described in [7] (Lit-bal-1) for k -SAT, the method described in [8] (Lit-bal-2), and our method (High-Girth). We have solved the instances with five SAT solvers: satz [34], minisat [35], knufs [36], walksat [37] and adaptg2wsat [38].

Fig. 2 shows the results for the complete SAT solver knufs on 4-SAT and 5-SAT instances. As we can see High-Girth is the best generator, while Lit-bal-1 and Lit-bal-2 are almost identical, and the differences remain even when looking only at satisfiable instances. We only report the results for the SAT solver knufs since it was the fastest and it reported the least difference between the two best generators. Table 2 shows the ratios for the median time between High-Girth and Lit-bal-1 and between Lit-bal-1 and Random for different arities when both, all and only satisfiable instances, are considered. We observe that the higher the arity (k), the higher the ratio High-Girth/Lit-bal-1 results, particularly for larger number of variables. This can be due to the differences in the expansion of the bipartite graphs of the different models, because as we increase k , it is possible to obtain more drastic differences in the expansion of the bipartite graphs of the different models. That is, the higher the k , the higher the maximum expansion

Table 2. Ratio of median time to solve all/only_sat instances on peak hardness between High Girth Bipartite, Literal and Random generation methods

	3-SAT		4-SAT		5-SAT	
Num. vars.	300	330	130	150	70	100
High-Girth/Lit-bal-1	1.29/1.02	1.44/1.19	1.34/1.42	1.39/1.68	1.64/1.48	3.09/3.34
Lit-bal-1/Random	80.2/126	132/162	4.58/7.41	5.59/10.47	1.73/2.65	2.09/2.48

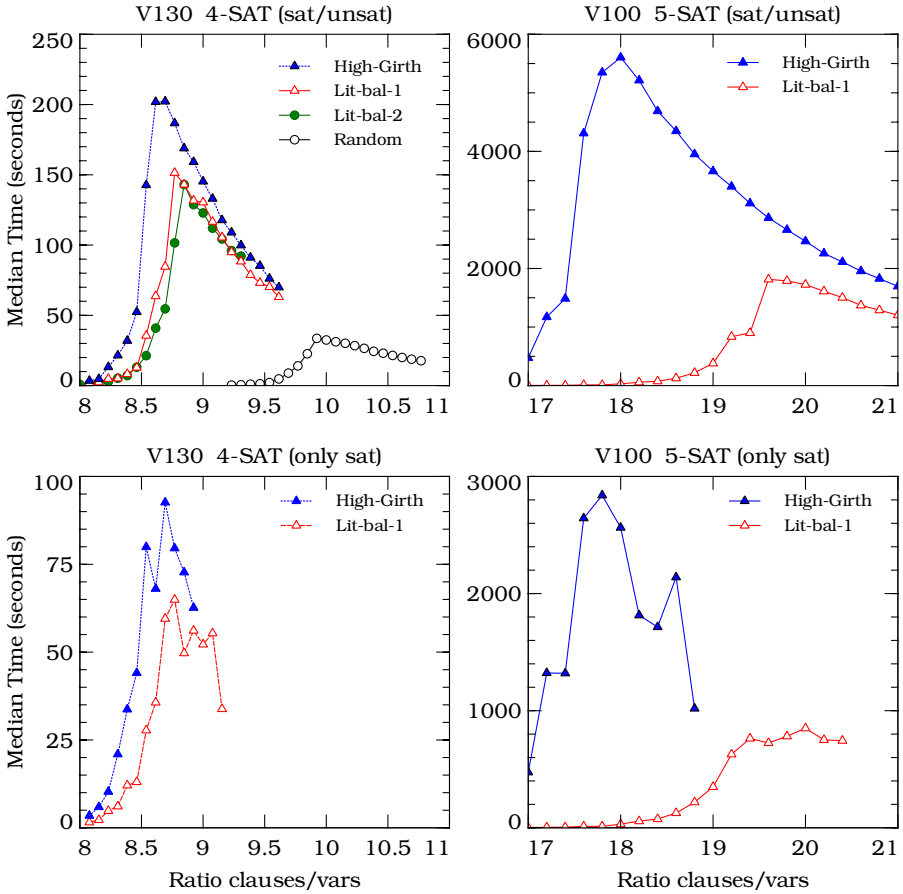


Fig. 2. Performance of knfs for the different SAT generators

of a subset of clauses S . At the same time, the ratio Lit-bal-1/Random seems to decrease, but this could be due to the fact that as we increase k , more variables may be needed in order to observe a difference for such ratio.

We also wanted to check if we could observe the same behavior when using a local search SAT solver. Table 3 reports results for 5-SAT. First, we run the solver walksat, on the 50 most difficult satisfiable instances at the underconstrained and the phase transition zones. We report the median time and the best noise parameter setting (median-time/best-noise at table) for the heuristics *best* and *novelty*.

Second, in order to solve instances with a higher number of variables, we run one of the best performing local search solvers at the SAT'07 solver competition, adaptg2wsat. We report the median time on the satisfiable instances at the peak of hardness. Since the unsatisfiable instances were too difficult to be filtered out by complete solvers, we applied a cutoff of 10,000 seconds for adaptg2wsat, and we assumed those instances lasting more to be unsatisfiable. We applied this

Table 3. Local search solvers on 5-SAT instances. Median time (in seconds). For walksat we give time/best-noise. Cutoff is 1,000 seconds.

walksat			
50 most	High-Girth	Lit-bal-1	Random
70 vars / best	1,000/—	0.12/29	0.08/27
70 vars / novelty	84/21	0.05/45	0.03/41
90 vars / best	1,000/—	0.17/29	0.07/27
90 vars / novelty	76/15	0.06/47	0.04/47

adaptg2wsat			
peak of hardness	High-Girth	Lit-bal-1	Random
130 vars	16	2.5	0.4
150 vars	53	8.7	0.6

Table 4. Median time (in seconds) for 3-SAT, 4-SAT, and 5-SAT. High-Girth solved with knfs, and regular XORSAT and HG-XORSAT solved with minisat. Results only for best solver among satz, minisat and knfs.

3-SAT						
Num. vars	200	250	270	300	330	350
High-Girth	0	7	14	91	368	1,125
XORSAT	14	386	2,322	19,778	>20,000	>20,000
HG-XORSAT	642	>20,000	>20,000	>20,000	>20,000	>20,000

4-SAT				5-SAT		
Num. vars	100	130	150	80	90	100
High-Girth	3	59	1,180	64	405	2,839
XORSAT	4	201	2,543	51	290	2,528
HG-XORSAT	66	8,018	>20,000	586	3,186	>20,000

process on sets of 100 instances. For all the generations methods we obtained around 50 satisfiable instances. As we can see, High-Girth dominates Lit-bal-1, and Lit-bal-1 dominates Random instances.

On why the expansion of the incidence graph also affects the performance of local search solvers, we can turn to results in [39], where the existence of what they call chains of short range connections between clauses is identified as a cause for bad performance of local search, due to the long range dependencies they create. It turns out that big cycles in the incidence graph could create these problematic chains on the formula.

Finally, we have also compared the hardness of the satisfiable instances obtained with High-Girth, at the peak of hardness, with the ones obtained with regular k -XORSAT and with our modification of k -XORSAT where we generate the system of linear equations with our High-Girth algorithm (HG-XORSAT). Table 4 shows the results when solving test-sets of 100 satisfiable instances with a cutoff of 20,000 seconds per instance. The instances from XORSAT are harder than the satisfiable ones from High-Girth, but when we use our High-Girth

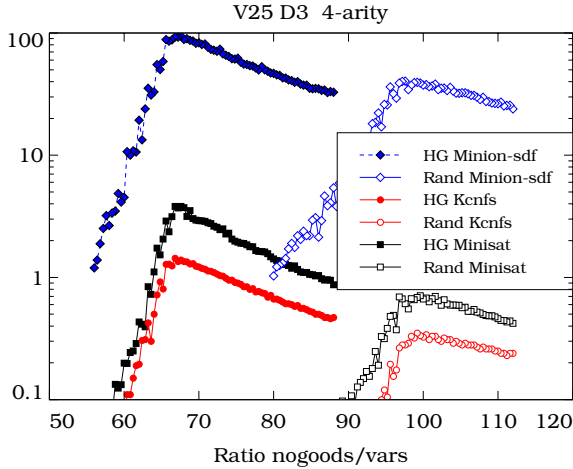


Fig. 3. Comparison of CSP generators

algorithm for the k -XORSAT instances is when we obtain the hardest instances, with orders of magnitude of difference, and even in some cases we have not been able to reach the median with our cutoff time (median $> 20,000$).

5.2 Hard n -ary CSP Instances

To generate the n -ary CSP instances we have used two methods: model E described in [12] and our method High-Girth. We have solved the n -ary CSP instances with the CSP solver minion [40] using the dynamic heuristic *sdf* (smaller domain first). We also report results on the direct SAT encoding [41] of the n -ary CSP instances for the SAT solvers minisat and kcnfs (some competitive solvers submitted to the CSP competition are built on top of minisat). We have generated two set of instances, one of 25 variables, domain 3, and arity 4, and the other one of 40 variables, domain 3 and arity 3. In Fig. 3 we plot the results for arity 4, showing again that our generation method, High-Girth (HG), produces the hardest instances. In this figure, results are shown in log-scale, in contrast with Fig. 2 for k -SAT, because here the differences are even more significant than in Fig. 2. However, observe that we do not have previous existing balanced models for n -ary CSPs, like Lit-bal-1 and Lit-bal-2 for k -SAT, that are the ones that are closer to our High-Girth model for k -SAT.

6 Conclusions

We have proposed a new method for generating hard k -SAT and n -ary CSP instances. This method is based on the results that link problem hardness with the expansion of the incidence graph of the instances. In particular, in our method we achieve high expansion by maintaining a high girth during the construction process of the incidence graph. We have also shown that our high girth graphs

can be used to increase the hardness of the satisfiable instances obtained with regular k -XORSAT, by building the system of linear equations with our high girth graphs, instead of using random regular graphs.

References

1. Atserias, A.: On sufficient conditions for unsatisfiability of random formulas. *Journal of the ACM* 51(2), 281–311 (2004)
2. Alekhovich, M., Razborov, A.: Lower bounds for polynomial calculus: non-binomial case. In: *Proceedings of 42nd Annual Symposium on Foundations of Computer Science*, pp. 190–199 (2001)
3. Kahale, N.: *Expander Graphs*. PhD thesis. MIT (1993)
4. Chung, F.R.K.: On concentrators, superconcentrators, generalizers and nonblocking networks. *Bell Systems Tech. Journal* 58, 1765–1777 (1978)
5. Sipser, M., Spielman, D.A.: Expander codes. *IEEE Trans. on Information Theory* 43(6), 1710–1722 (1996)
6. Charles, D.X., Goren, E.Z., Lauter, K.E.: Cryptographic hash functions from expander graphs. *Journal of Cryptology* (2007)
7. Bayardo, R., Schrag, R.: Using CSP look-back techniques to solve exceptionally hard sat instances. In: Freuder, E.C. (ed.) *CP 1996. LNCS*, vol. 1118, pp. 46–60. Springer, Heidelberg (1996)
8. Boufkhad, Y., Dubois, O., Interian, Y., Selman, B.: Regular random k -sat: Properties of balanced formulas. *Journal of Automated Reasoning* 35(1-3), 181–200 (2005)
9. Järvisalo, M.: Further investigations into regular xorsat. In: *Proceedings of the AAAI 2006*. AAAI Press / The MIT Press (2006)
10. Smith, B., Dyer, M.: Locating the Phase Transition in Binary Constraint Satisfaction Problems. *Artificial Intelligence* 81, 155–181 (1996)
11. Gent, I., MacIntyre, E., Prosser, P., Smith, B., Walsh, T.: Random constraint satisfaction: flaws and structure. *Constraints* 6(4), 345–372 (2001)
12. Achlioptas, D., Kirousis, L.M., Kranakis, E., Krizanc, D., Molloy, M.S.O., Stamatiou, Y.C.: Random Constraint Satisfaction: A More Accurate Picture. In: Smolka, G. (ed.) *CP 1997. LNCS*, vol. 1330, pp. 107–120. Springer, Heidelberg (1997)
13. 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)
14. Lubotzky, A., Phillips, R., Sarnak, P.: Ramanujan graphs. *Combinatorica* 8, 261–277 (1988)
15. McKay, B.D., Wormald, N.C., Wysocka, B.: Short cycles in random regular graphs. *Elect. J. Combinatorics* 11, R66 (2004)
16. Blum, M., Karp, R., Vornberger, O., Papadimitriou, C., Yannakakis, M.: The complexity of testing whether a graph is a superconcentrator. *Information Processing Letters* 13(4/5), 164–167 (1981)
17. Kahale, N.: Eigenvalues and expansion of regular graphs. *Journal of the ACM* 42(5), 1091–1106 (1995)
18. Ben-Sasson, E., Wigderson, A.: Short proofs are narrow-resolution made simple. *Journal of the ACM* 48(2), 149–169 (2001)
19. Ricci-Tersenghi, F., Weight, M., Zecchina, R.: Simplest random k -satisfiability problem. *Physical Review E* 63:026702 (2001)
20. Jia, H., Moore, C., Selman, B.: From spin glasses to hard satisfiable formulas. In: *Proceedings of SAT 2005. LNCS*, vol. 3452, pp. 199–210. Springer, Heidelberg (2005)

21. Haanpää, H., Järvisalo, M., Kaski, P., Niemelä, I.: Hard satisfiable clause sets for benchmarking equivalence reasoning techniques. *Journal on Satisfiability, Boolean Modeling and Computation* 2(1-4), 27–46 (2006)
22. Ansótegui, C., Béjar, R., Fernández, C., Mateu, C.: On balanced CSPs with high treewidth. In: *Proceedings of the AAAI 2007*. AAAI Press, Menlo Park (2007)
23. Chandran, L.S., Subramanian, C.: A spectral lower bound for the treewidth of a graph and its consequences. *Information Processing Letters* 87(4), 195–200 (2003)
24. 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. 310–326. Springer, Heidelberg (2002)
25. Atserias, A., Bulatov, A.A., Dalmau, V.: On the power of k -consistency. In: Arge, L., Cachin, C., Jurdziński, T., Tarlecki, A. (eds.) *ICALP 2007*. LNCS, vol. 4596, pp. 279–290. Springer, Heidelberg (2007)
26. Kautz, H.A., Ruan, Y., Achlioptas, D., Gomes, C.P., Selman, B., Stickel, M.E.: Balance and filtering in structured satisfiable problems. In: *Proceedings of the IJCAI 2001*, pp. 193–200 (2001)
27. Ansótegui, C., Béjar, R., Fernández, C., Gomes, C., Mateu, C.: The impact of balance in a highly structured problem domain. In: *Proceedings of the AAAI 2006*, pp. 438–443. AAAI Press / The MIT Press (2006)
28. Chandran, L.S.: A high girth graph construction. *SIAM journal on Discrete Mathematics* 16(3), 366–370 (2003)
29. Gudmundsson, J., Smid, M.: On spanners of geometric graphs. In: Arge, L., Freivalds, R. (eds.) *SWAT 2006*. LNCS, vol. 4059, pp. 388–399. Springer, Heidelberg (2006)
30. Demetrescu, C., Italiano, G.F.: Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Transactions on Algorithms* 2(4), 578–601 (2006)
31. Ramalingam, G., Reps, T.: An incremental algorithm for a generalization of the shortest-path problem. *Journal of Algorithms* 21(2), 267 (1996)
32. Demetrescu, C., Frigioni, D., Marchetti-Spaccamela, A., Nanni, U.: Maintaining shortest paths in digraphs with arbitrary arc weights: An experimental study. In: Näher, S., Wagner, D. (eds.) *WAE2000*. LNCS, vol. 1982. Springer, Heidelberg (2001)
33. Demetrescu, C., Italiano, G.: A new approach to dynamic all pairs shortest paths. *Journal of the Association for Computing Machinery (JACM)* 51(6), 968–992 (2004)
34. Li, C.M.: Anbulagan: Look-ahead versus look-back for satisfiability problems. In: *Principles and Practice of Constraint Programming*, pp. 341–355 (1997)
35. 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)
36. Dubois, O., Dequen, G.: A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In: *Proceedings of the IJCAI 2001*, pp. 248–253 (2001)
37. Selman, B., Kautz, H.A., Cohen, B.: Noise strategies for improving local search. In: *Proceedings of the AAAI 1994*, pp. 337–343 (1994)
38. Li, C.M., Wei, W., Zhang, H.: Combining adaptive noise and look-ahead in local search for SAT. In: Marques-Silva, J., Sakallah, K.A. (eds.) *SAT 2007*. LNCS, vol. 4501, pp. 121–133. Springer, Heidelberg (2007)
39. Wei, W., Selman, B.: Accelerating random walks. In: Van Hentenryck, P. (ed.) *CP 2002*. LNCS, vol. 2470, pp. 216–232. Springer, Heidelberg (2002)
40. 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)
41. Walsh, T.: SAT vs CSP. In: Dechter, R. (ed.) *CP 2000*. LNCS, vol. 1894, pp. 441–456. Springer, Heidelberg (2000)

Switching among Non-Weighting, Clause Weighting, and Variable Weighting in Local Search for SAT^{*}

Wanxia Wei¹, Chu Min Li², and Harry Zhang¹

¹ Faculty of Computer Science, University of New Brunswick, Fredericton, NB, Canada,
E3B 5A3

{wanxia.wei, hzhang}@unb.ca

² MIS, Université de Picardie Jules Verne 33 Rue St. Leu, 80039 Amiens Cedex 01, France
chu-min.li@u-picardie.fr

Abstract. One way to design a local search algorithm that is effective on many types of instances is allowing this algorithm to switch among heuristics. In this paper, we refer to the way in which non-weighting algorithm *adaptG²WSAT+* selects a variable to flip, as *heuristic adaptG²WSAT+*, the way in which clause weighting algorithm *RSAPS* selects a variable to flip, as *heuristic RSAPS*, and the way in which variable weighting algorithm *VW* selects a variable to flip, as *heuristic VW*. We propose a new switching criterion: the evenness or unevenness of the distribution of clause weights. We apply this criterion, along with another switching criterion previously proposed, to *heuristic adaptG²WSAT+*, *heuristic RSAPS*, and *heuristic VW*. The resulting local search algorithm, which adaptively switches among these three heuristics in every search step according to these two criteria to intensify or diversify the search when necessary, is called *NCVW* (Non-, Clause, and Variable Weighting). Experimental results show that *NCVW* is generally effective on a wide range of instances while *adaptG²WSAT+*, *RSAPS*, *VW*, and *gNovelty+* and *adaptG²WSAT0*, which won the gold and silver medals in the satisfiable random category in the SAT 2007 competition, respectively, are not.

1 Introduction

Intensification refers to search strategies that intend to greedily improve solution quality or the chances of finding a solution in the near future [5]. Diversification refers to search strategies that help achieve a reasonable coverage when exploring the search space in order to avoid search stagnation and entrapment in relatively confined regions of the search space that may contain only locally optimal solutions [5]. Generally speaking, there are three classes of local search algorithm: non-weighting, clause weighting, and variable weighting. A non-weighting algorithm does not use any weighting and mainly focuses on intensifying the search to greatly decrease the number of unsatisfied clauses. A clause weighting algorithm uses clause weighting to diversify the search while a variable weighting algorithm uses variable weighting to diversify the search.

^{*} The work of the first author is partially supported by NSERC PGS-D (Natural Sciences and Engineering Research Council of Canada Post-Graduate Scholarships for Doctoral students).

Efforts have been made to develop non-weighting local search algorithms [3,8,9,10,14,15]. Among these algorithms, *adaptG²WSAT_P* [9], the improved *adaptG²WSAT_P* [10], *adaptG²WSAT0* [8], and *adaptG²WSAT+* [15] combine the use of promising decreasing variables defined in [7] with the adaptive noise mechanism proposed in [4]. According to the definition of promising decreasing variables, flipping such variables not only decreases the number of unsatisfied clauses but also probably allows local search to explore new promising regions in the search space.

Clause weighting has been used in local search algorithms to help the search escape from local minima and diversify the search [6,11,12]. These algorithms include *Breakout* [11], *SAPS* (Scaling And Probabilistic Smoothing) [6], *RSAPS* (Reactive SAPS) [6], and *gNovelty+* [12]. Of these algorithms, *gNovelty+* combines the use of promising decreasing variables and clause weighting techniques. Rather than using clause weighting, the local search algorithm *VW* [13] employs variable weighting to diversify the search and guide local search out of local minima.

However, each single local search heuristic is usually ineffective on many types of instances, since the performance of a heuristic depends on the properties of the search space and the search spaces of different types of instances have different properties. One way to design a local search algorithm that is effective on many types of instances is allowing this algorithm to switch among heuristics in order to adapt to search spaces with different properties.

Several local search algorithms switch between two heuristics [2,16]. *UnitWalk* 0.98 [2] is improved by alternating between *WalkSAT*-like and *UnitWalk*-like fragments of the random walk. *Hybrid* [16] switches between *heuristic adaptG²WSAT_P* and *heuristic VW* according to the evenness or non-evenness of the distribution of variable weights.¹

Nevertheless, our experimental results show that the performance of *Hybrid* is poor on some instances for which a local search algorithm may result in imbalanced clause weights, and that the performance of clause weighting algorithm *RSAPS* [6] is poor on some instances for which a local search algorithm may result in balanced clause weights. In this paper, we propose a new switching criterion: the evenness or unevenness of the distribution of clause weights. We refer to the ways in which non-weighting algorithm *adaptG²WSAT+* [15], clause weighting algorithm *RSAPS*, and variable weighting algorithm *VW* [13] select a variable to flip, as *heuristic adaptG²WSAT+*, *heuristic RSAPS*, and *heuristic VW*, respectively. We apply this switching criterion together with another switching criterion, namely the evenness or non-evenness of the distribution of variable weights proposed in [16], to *heuristic adaptG²WSAT+*, *heuristic RSAPS*, and *heuristic VW*. The resulting local search algorithm, which adaptively switches among these three heuristics in every search step according to these two criteria to intensify or diversify the search when necessary, is called *NCVW* (Non-, Clause, and Variable Weighting).

Given a set of instances and a fixed cutoff for each instance, if an algorithm achieves a success rate greater than 50% for each instance, we say that this algorithm is generally effective on these instances. Our experimental results show that *NCVW* is

¹ The ways in which algorithms *adaptG²WSAT_P* [9] and *VW* [13] select a variable to flip, are referred to as *heuristic adaptG²WSAT_P* and *heuristic VW*, respectively [16].

generally effective on a wide range of instances while *adaptG²WSAT+*, *RSAPS*, *VW*, *Hybrid*, and *gNovelty+* and *adaptG²WSAT0*, which won the gold and silver medals in the satisfiable random category in the SAT 2007 competition,² respectively, are not.

Our work for *NCVW* provides a solution to the algorithm heuristic selection problem. Different approaches have been proposed for this problem. CBR (Case-Based Reasoning) was used to select a solution strategy for instances of a CP problem [1]. *SATzilla-07* [17] is a per-instance solver portfolio for SAT. This solver portfolio uses machine learning techniques to build an empirical hardness model that predicts an algorithm's runtime on a given instance based on the features of this instance and the past performance of this algorithm, and uses this model to choose among the constituent solvers of *SATzilla-07*. *NCVW* is different from *SATzilla-07* in that *NCVW* chooses heuristics for an instance dynamically during the search while *SATzilla-07* first chooses an algorithm for an instance and then runs this algorithm on this instance.

2 Review of Algorithms *adaptG²WSAT+*, *RSAPS*, *VW*, and *Hybrid*

The local search algorithm *adaptG²WSAT+* [15] combines the use of promising decreasing variables [7] and the adaptive noise mechanism [4]. As a result, noise p in this algorithm is adjusted during the search. Moreover, random walk probability wp is also adjusted and $wp = p/10$. This algorithm won the bronze medal in the satisfiable random category in the SAT 2007 competition. As presented in Section 1, we refer to the way in which algorithm *adaptG²WSAT+* selects a variable to flip, as *heuristic adaptG²WSAT+*.

SAPS [6] scales the weights of unsatisfied clauses and smoothes the weights of all clauses probabilistically. It performs a greedy descent search in which a variable is selected at random to flip, from the variables that appear in unsatisfied clauses and that lead to the maximum reduction in the total weight of unsatisfied clauses when flipped. *RSAPS* [6] is a reactive version of *SAPS* that adaptively tunes smoothing parameter P_{smooth} during the search. *RSAPS* has the other parameters α , ρ , and wp whose default values are $(\alpha, \rho, wp) = (1.3, 0.8, 0.01)$. As presented in Section 1, we refer to the way in which algorithm *RSAPS* selects a variable to flip, as *heuristic RSAPS*.

In *VW* [13], the weight of a variable reflects both the number of flips of this variable and the times when this variable is flipped. This algorithm initializes the weight of a variable x , $vw[x]$, to 0 and updates and smoothes $vw[x]$ each time x is flipped, using the following formula:

$$vw[x] = (1 - s)(vw[x] + 1) + s \times t \quad (1)$$

where s is a parameter and $0 \leq s \leq 1$, and t denotes the time when x is flipped, i.e., t is the number of search steps since the start of the search.

VW always flips a variable from a randomly selected unsatisfied clause c . If c contains freebie variables,³ *VW* randomly flips one of them. Otherwise, with probability

² <http://www.satcompetition.org/>

³ Flipping a freebie variable will not falsify any clause.

p (noise p), it flips a variable chosen randomly from c , and with probability $1 - p$, it flips a variable in c according to a unique variable selection rule, which generally favors variables with relatively low variable weights. As presented in Section 1, we refer to the way in which algorithm *VW* selects a variable to flip, as *heuristic VW*.

A switching criterion, namely the evenness or non-evenness of the distribution of variable weights, was proposed in [16]. This evenness or non-evenness is defined in [16] as follows. Assume that γ is a number. If the maximum variable weight is at least γ times as high as the average variable weight, the distribution of variable weights is considered *uneven*, and the step is called *an uneven step* in terms of variable weights. Otherwise, the distribution of variable weights is considered *even*, and the step is called *an even step* in terms of variable weights. An uneven or an even distribution of variable weights is used as a means to determine whether a search is undiversified in a step in terms of variable weights.

Hybrid [16] switches between *heuristic adaptG²WSAT_P* and *heuristic VW* according to the above switching criterion. More precisely, in each search step, *Hybrid* chooses a variable to flip according to *heuristic VW* if the distribution of variable weights is uneven, and selects a variable to flip according to *heuristic adaptG²WSAT_P* otherwise. In *Hybrid*, the default value of parameter γ is 10.0. *Hybrid* updates variable weights using Formula 1, and parameter s in this formula is fixed to 0.0.

3 Motivation

The search space of a hard SAT instance for a local search algorithm generally has a large number of local minima in each of which flipping any variable cannot decrease the number of unsatisfied clauses. Each local minimum is characterized by a subset of unsatisfied clauses, which cannot be satisfied without unsatisfying other clauses. The unsatisfied clauses in a local minimum can be considered as having attractions to draw a local search towards this local minimum. Different clauses in a SAT instance can have very different attractions for a local search. A local search can be frequently drawn towards the same local minima by the same unsatisfied clauses with strong attractions. In this case, the search is poorly diversified. Accordingly, clause weighting techniques are introduced to diversify this poorly diversified search.

Clause weighting in a local search algorithm has two purposes. The first one is to quantify the attraction of a clause for a local search. Different clause weights are defined to measure the attractions of clauses for local searches [6,11]. The second one is to modify the objective function, which is usually the number of unsatisfied clauses, during the search by minimizing the total weight of unsatisfied clauses instead of the number of unsatisfied clauses. As a result, a clause weighting algorithm usually first satisfies clauses that have the largest attractions to diversify the search.

We hypothesize firstly that, without clause weighting techniques, *Hybrid* [16] exhibits good performance, usually when clause weights are generally balanced, i.e., usually when all clauses have roughly equal attractions for a local search towards local minima. We hypothesize secondly that, with clause weighting techniques, *RSAPS* [6] exhibits good performance, usually when clause weights are unbalanced. To empirically verify our hypotheses, we conduct experiments with *Hybrid* and *RSAPS* on

two classes of representative instance, one of which leads to balanced clauses weights and the other of which leads to unbalanced clause weights.

In order to quantify the attraction of a clause towards a local minimum at a search point, we simply sum up the number of local minima, encountered so far, in which this clause is unsatisfied. In fact, these summations of the numbers of local minima are the clause weights defined in *Breakout* [11]. We refer to these clause weights as *clause weights defined by Breakout*. We calculate these clause weights in both *Hybrid* and *RSAPS* to make clause weights comparable for these two algorithms, although *RSAPS* has its own clause weighting techniques, which are more sophisticated. Note that the calculations of these clause weights in these two algorithms do not change the behavior of *Hybrid* or *RSAPS* in any way, i.e., these two algorithms do not consider the clause weights that we calculate, when choosing variables to flip.

We ran *RSAPS* and *Hybrid* on two classes of instance.⁴ The source code of *RSAPS* was downloaded from <http://www.satlib.org/ubcsat/>. When experimenting with these algorithms, we do not change the ways in which these algorithms adaptively adjust their parameters and do not change the default values of their other parameters. One class includes the 5 structured instances par16-1, par16-2, par16-3, par16-4, and par16-5 in PARITY from the SATLIB repository,⁵ and the other consists of the 5 structured instances f*3995, f*3997, f*3999, f*4001, and f*4003 in Ferry from the industrial category in the SAT 2005 competition benchmark.⁶ Each algorithm is run 100 times ($Maxtries = 100$). The cutoffs are set to 10^9 ($Maxsteps = 10^9$) and 10^8 ($Maxsteps = 10^8$) for an instance in PARITY and an instance in Ferry, respectively.

In Table 1, for each of the two algorithms *Hybrid* and *RSAPS*, we report the average coefficient of variation of distribution of clause weights (coefficient of variation = standard deviation / mean value) (“cv”) and the average division of the maximum clause weight by the average clause weight (“div”), over all search steps, for clause weights defined by *Breakout*. All reported values are then averaged over 100 runs ($Maxtries = 100$). A run is successful if an algorithm finds a solution within a cutoff ($Maxsteps$). The success rate of an algorithm is the number of successful runs divided by $Maxtries$. We also report success rates (“suc”). Moreover, in the last row for each group (“avg”) in this table, we present the average of the values in each column, for each algorithm, over all instances.

Generally speaking, the distribution of clause weights defined by *Breakout* reflects the history of the attractions of clauses for local searches towards local minima. If clauses in a small subset have drawn local searches towards local minima much more frequently than other clauses, the weights of these clauses should be much higher than those of others and the coefficient of variation of distribution of clause weights should be high. Otherwise, all clauses should have approximately equal weights, and the coefficient of variation of distribution of clause weights should be low. That is, the higher the coefficient of variation is, the more clause weights far from the mean value exist,

⁴ All experiments reported were conducted on a computer with Intel(R) Core(TM)2 CPU 6400 @ 2.13GHz and with 2GB of memory, under Linux.

⁵ <http://www.satlib.org/>

⁶ <http://www.lri.fr/~simon/contest/results/>

Table 1. Performance and distributions of clause weights defined by *Breakout* for *RSAPS* and *Hybrid* on PARITY and Ferry

	cutoff	<i>RSAPS</i>			<i>Hybrid</i>		
		cv	div	suc	cv	div	suc
par16-1	10 ⁹	0.82	7.22	0.19	10.20	127.06	1.00
par16-2	10 ⁹	0.82	7.31	0.05	10.40	131.58	1.00
par16-3	10 ⁹	0.82	7.27	0.12	10.30	129.04	1.00
par16-4	10 ⁹	0.82	7.22	0.16	10.23	127.75	1.00
par16-5	10 ⁹	0.82	7.28	0.09	10.43	131.88	1.00
avg	10 ⁹	0.82	7.26	0.12	10.31	126.46	1.00
f*3995	10 ⁸	2.72	22.38	1.00	41.05	1910.94	0.06
f*3997	10 ⁸	2.06	12.15	1.00	47.51	2632.37	0.02
f*3999	10 ⁸	2.36	17.39	1.00	48.23	2594.44	0.00
f*4001	10 ⁸	2.13	12.80	0.84	55.04	3489.75	0.00
f*4003	10 ⁸	2.11	13.57	1.00	55.35	3375.36	0.00
avg	10 ⁸	2.28	15.66	0.97	49.44	2800.57	0.02

the more unbalanced cause weights are, and the less well diversified, in terms of clause weights, the search is.

According to Table 1, the distributions of clause weights of PARITY are more balanced than the distributions of clause weights of Ferry, both for *Hybrid* and for *RSAPS*, meaning that when solving the Ferry instances, some clauses in the Ferry instances frequently draw local searches towards local minima so that the searches for Ferry are less diversified than those for PARITY. Nevertheless, with its own powerful clause weighting techniques, *RSAPS* diversifies the search better than does *Hybrid*. As a result, the distributions of clause weights in the searches of *RSAPS* are more balanced for each of the two classes of instance than the distributions of clause weights in the searches of *Hybrid*.

As shown in Table 1, the average success rate of *Hybrid* on PARITY is 1.00, suggesting that when the distributions of clause weights are generally balanced, *Hybrid*, which does not use clause weighting techniques, exhibits good performance. However, the average success rate of *Hybrid* on Ferry is only 0.02, suggesting that when the distributions of clause weights are generally unbalanced, the performance of *Hybrid* is poor. Conversely, *RSAPS* exhibits very good performance on Ferry but poor performance on PARITY. Therefore, the experimental results in this table indicate that an algorithm should ignore clause weights and concentrate on intensifying the search if clause weights are generally balanced, and that an algorithm should use clause weighting techniques, such as those introduced in *RSAPS*, to diversify the search to prevent a local search from being drawn towards the same local minima.

According to Table 1, on PARITY, the averages of the values for *div* in *RSAPS* and *Hybrid* are 7.26 and 126.46, respectively, while on Ferry, the averages of the values for *div* in *RSAPS* and *Hybrid* are 15.66 and 2800.57, respectively. That is, the maximum clause weight on Ferry usually deviates from the average clause weight to a greater degree than does the maximum clause weight on PARITY. Therefore, the results in this table suggest that, similar to the coefficient of variation of distribution of clause weights, the division of the maximum clause weight by the average clause weight also indicates whether clause weights are balanced. In fact, calculating the division is not time-consuming, but calculating the coefficient of variation is.

4 A New Switching Criterion

We propose a new switching criterion. Additionally, we introduce a new local search algorithm that uses this criterion along with another switching criterion.

4.1 Evenness or Unevenness of Distribution of Clause Weights

Assume that δ is a number. If the maximum clause weight is at least δ times as high as the average clause weight, the distribution of clause weights is considered *uneven*, and the step is called *an uneven step* in terms of clause weights. Otherwise, the distribution of clause weights is considered *even*, and the step is called *an even step* in terms of clause weights. An uneven distribution and an even distribution of clause weights correspond to the situations in which clause weights are unbalanced and balanced, respectively. We use an uneven or an even distribution of clause weights as a means to determine whether a search is undiversified in a step in terms of clause weights.

4.2 Algorithm *NCVW*

To evaluate the effectiveness of the proposed switching criterion, we apply it together with another switching criterion, namely the evenness or non-evenness of the distribution of variable weights proposed in [16], to *heuristic adaptG²WSAT+*, *heuristic RSAPS*, and *heuristic VW*. The resulting local search algorithm, which switches among these three heuristics according to these two criteria, is called *NCVW* (Non-, Clause, and Variable Weighting).

NCVW exploits the information about the structure of an instance when choosing a variable to flip by first examining the distribution of variable weights of this instance and then examining the distribution of clause weights of this instance. This algorithm adaptively switches among *heuristic adaptG²WSAT+*, *heuristic RSAPS*, and *heuristic VW* in every search step according to the distributions of variable and clause weights, to intensify or diversify the search when necessary. When the distribution of variable weights is uneven, i.e., when a search is undiversified in terms of variable weights, *NCVW* uses *heuristic VW* to choose a variable to flip to diversify the search by using variable weights. Otherwise, *NCVW* selects a variable to flip according to *heuristic RSAPS* or *heuristic adaptG²WSAT+*, depending on whether the distribution of clause weights is uneven. If the distribution of clause weights is uneven, i.e., if a search is undiversified in terms of clause weights, *NCVW* uses *heuristic RSAPS* to select a variable to flip to diversify the search by using clause weights; otherwise, i.e., if a search is diversified in terms of both variable and clause weights, *NCVW* uses *heuristic adaptG²WSAT+* to select a variable to flip to intensify the search.

NCVW is described in Fig. 1. In this figure, $flip\_time[i]$, $vw[i]$, max_vw , ave_vw , $cw[j]$, max_cw , and ave_cw are the time when variable i is flipped, the weight of variable i , maximum variable weight, average variable weight, the weight of clause j , maximum clause weight, and average clause weight, respectively.

NCVW has its own parameters γ , δ , and π , which are used to choose one heuristic from *NCVW*'s constituent heuristics in every step. Parameter γ determines whether

Algorithm: *NCVW*(SAT-formula $\mathcal{F}$)

```

1:  $A \leftarrow$  randomly generated truth assignment;
2: for each variable  $i$  do initialize  $flip\_time[i]$  and  $vw[i]$  to 0;
3: initialize  $max\_vw$  and  $ave\_vw$  to 0;
4: for each clause  $j$  do initialize  $cw[j]$  to 1; initialize  $max\_cw$  and  $ave\_cw$  to 1;
5: for  $flip \leftarrow 1$  to  $Maxsteps$  do
6:   if  $A$  satisfies  $\mathcal{F}$  then return  $A$ ;
7:   if ( $max\_vw \geq \gamma \times ave\_vw$ )
8:     then  $heuristic \leftarrow$  “ $VW$ ”;
9:   else
10:    if ( $(ave\_cw \leq \pi)$  or ( $max\_cw \geq \delta \times ave\_cw$ ))
11:      then  $heuristic \leftarrow$  “ $RSAPS$ ”;
12:    else  $heuristic \leftarrow$  “ $adaptG^2WSAT +$ ”;
13:   $y \leftarrow$  use  $heuristic$  to choose a variable;
14:  if ( $y \neq -1$ )
15:    then  $A \leftarrow A$  with  $y$  flipped; update  $flip\_time[y]$ ,  $vw[y]$ ,  $max\_vw$ , and  $ave\_vw$ ;
16:    if ( $heuristic =$  “ $RSAPS$ ”)
17:      then if ( $y = -1$ ) then update clause weights,  $max\_cw$ , and  $ave\_cw$ ;
18: return Solution not found;

```

Fig. 1. Algorithm *NCVW*

the distribution of variable weights is uneven, δ determines whether the distribution of clause weights is uneven, and π represents a threshold for average clause weight.

In algorithm *NCVW*, *heuristic RSAPS* is used both for gathering information about the distribution of clause weights and for selecting a variable to flip when the distribution of clause weights is uneven. In this algorithm, the n variables of an instance are represented as n integers from 0 to $n - 1$. When *NCVW* uses *heuristic RSAPS* and when this heuristic returns -1 , *NCVW* performs a null flip and updates clause weights in the same way as does *RSAPS*. To avoid frequent time-consuming clause weight updating, *NCVW* does not update clause weights if it uses *heuristic adaptG²WSAT+* or *heuristic VW*. *NCVW* uses Formula 1 to update variable weights after it selects any heuristic from its three constituent heuristics and after *NCVW* performs a non-null flip.

We have three objectives in *NCVW*. The first objective is to ensure that every variable in a SAT instance has an approximately equal chance of being flipped to diversify the search in terms of variable weights, i.e., to ensure that the distribution of variable weights is even. When the distribution of variable weights is uneven, *NCVW* chooses heuristic *VW* to balance variable weights. Whether the distribution of variable weights is even is determined at line 7 using the condition ($max\_vw \geq \gamma \times ave\_vw$), in which parameter $\gamma > 1.0$, in Fig. 1.

The second objective is to avoid the same local minima or to avoid exploring the same regions in the search space, i.e., to ensure that the distribution of clause weights is even. Since *NCVW* updates clause weights only when *heuristic RSAPS* is used to choose a variable to flip, parameter π , which represents a threshold for average clause weight ave_cw , is introduced. When the distribution of variable weights is even,

NCVW chooses *heuristic RSAPS* to build up the distribution of clause weights whenever the average clause weight is lower than parameter π . When the average clause weight is higher than π , clause weights are considered meaningful to determine whether the distribution of clause weights is even, and *heuristic RSAPS* is chosen to balance the distribution of clause weights and to diversify the search in terms of clause weights if the distribution of clause weights is uneven. This building up of distribution of clause weights and this balancing of distribution of clause weights are realized through line 10 using the condition $((ave\_cw \leq \pi) \text{ or } (max\_cw \geq \delta \times ave\_cw))$, in which parameter $\delta > 1.0$, in Fig. 1.

The third objective is to intensify the search when the distributions of both variable and clause weights are even. In this case, the search is considered diversified in terms of both variable and clause weights, and *heuristic adaptG²WSAT+* is used to intensify the search.

NCVW is an example that uses the proposed switching criterion along with the switching criterion proposed in [16]. These switching criteria can be used in other local search algorithms that combine intensification strategies with diversification strategies.

5 Evaluation

We present the default values of parameters in *NCVW*. Moreover, we evaluate *NCVW* on a wide range of instances and justify the switching strategy in *NCVW*.

5.1 Groups of Instances

We evaluate *NCVW* on 11 groups of benchmark SAT problems (36 instances shown in Table 2). They generally consist of hard problems from those widely used to evaluate local search algorithms in the literature and constitute a wide range of instances, including structured instances, instances from the industrial and crafted categories in a SAT competition benchmark, and hard random instances. Due to space limits, we do not present the performance of algorithms on the instances that are easy for most algorithms discussed in this paper. Structured problems come from the SATLIB repository and the SAT 2005 competition benchmark. The structured problems from SATLIB include instances in ais, blocksworld, Beijing, GCP, PARITY, and QG. The structured problems from the SAT 2005 competition benchmark include instances from the industrial category and instances from the crafted category. The former consist of f*3995, f*3997, f*3999, f*4001, and f*4003 in Ferry. The latter consist of g*1334, g*1337, g*1339, g*1340, and g*1341 in grid-pebbling/sat, and p*1318, p*1319, p*1320, p*1321, and p*1322 in random-pebbling/sat. Random problems come from the SAT 2007 competition benchmark,⁷ and they are hard problems, including *v10000*03, *v10000*04, *v10000*05, *v10000*06, and *v10000*10 in 3SAT/v10000, and *v1100*04, *v1100*06, *v1100*08, *v1100*10, and *v1100*14 in 5SAT/v1100.

Each instance is executed 100 times ($Maxtries = 100$). As shown in Table 2, the search step cutoff ($Maxsteps$) for each instance is set to a fixed value, to ensure that

⁷ <http://www.satcompetition.org/>

Table 2. Experimental results for *NCVW*, *adaptG²WSAT+*, *RSAPS*, and *VW* on the 11 groups of instances

	cutoff	<i>NCVW</i>			<i>adaptG²WSAT+</i>			<i>RSAPS</i>			<i>VW</i>		
		suc	#steps	time	suc	#steps	time	suc	#steps	time	suc	#steps	time
ais12	10 ⁷	0.93	249576	0.2	0.94	2568069	2.4	1.00	159617	0.2	1.00	962263	1.4
bw_large.d	10 ⁷	0.96	955185	2.5	0.70	5786347	8.1	0.06	> 10 ⁷	n/a	0.94	2554959	5.4
e0odr2*1	10 ⁷	1.00	119631	0.5	0.96	2219658	3.3	1.00	120139	0.7	0.71	6656635	10.7
g250.29	10 ⁷	0.79	3483743	76.9	1.00	728358	6.3	0.00	> 10 ⁷	n/a	0.15	> 10 ⁷	n/a
par16-1	10 ⁹	1.00	96371225	48.3	1.00	45395657	14.9	0.16	> 10 ⁹	n/a	0.00	> 10 ⁹	n/a
par16-2	10 ⁹	0.88	329217086	166.7	1.00	96745059	32.5	0.07	> 10 ⁹	n/a	0.00	> 10 ⁹	n/a
par16-3	10 ⁹	1.00	115526876	58.8	1.00	87203523	28.9	0.08	> 10 ⁹	n/a	0.00	> 10 ⁹	n/a
par16-4	10 ⁹	0.96	183854606	92.8	0.98	170022013	55.7	0.10	> 10 ⁹	n/a	0.01	> 10 ⁹	n/a
par16-5	10 ⁹	0.93	264988982	133.5	0.99	111417125	37.0	0.13	> 10 ⁹	n/a	0.00	> 10 ⁹	n/a
qg2-08	10 ⁷	0.83	2832886	12.7	1.00	1766643	4.9	0.08	> 10 ⁷	n/a	0.12	> 10 ⁷	n/a
qg7-13	10 ⁸	1.00	3663127	31.9	0.20	> 10 ⁸	n/a	0.07	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a
f*3995	10 ⁸	1.00	56405	0.1	0.04	> 10 ⁸	n/a	1.00	66907	0.1	1.00	5346194	4.2
f*3997	10 ⁸	0.95	4830154	3.3	0.01	> 10 ⁸	n/a	1.00	6348614	5.8	0.69	55768455	29.8
f*3999	10 ⁸	1.00	336040	0.4	0.00	> 10 ⁸	n/a	1.00	269670	0.3	0.38	> 10 ⁸	n/a
f*4001	10 ⁸	0.66	51971463	41.8	0.00	> 10 ⁸	n/a	0.80	39945421	40.3	0.18	> 10 ⁸	n/a
f*4003	10 ⁸	1.00	1514236	1.9	0.00	> 10 ⁸	n/a	1.00	1575665	1.9	0.05	> 10 ⁸	n/a
g*1334	10 ⁸	1.00	248515	0.2	0.11	> 10 ⁸	n/a	1.00	385746	0.2	1.00	165058	0.1
g*1337	10 ⁸	1.00	1411026	1.2	0.51	52247390	24.7	1.00	4188026	3.1	1.00	334304	0.2
g*1339	10 ⁸	1.00	2325027	3.5	0.70	1656358	1.9	1.00	19332961	27.5	1.00	1090780	1.1
g*1340	10 ⁸	0.76	3579295	7.3	0.00	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a	1.00	9592247	10.3
g*1341	10 ⁸	0.98	4507419	10.3	0.00	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a	1.00	6911453	8.5
p*1318	10 ⁸	1.00	1411992	6.2	0.19	> 10 ⁸	n/a	1.00	1109307	5.3	1.00	1023732	37.8
p*1319	10 ⁸	1.00	1039612	4.1	0.64	1283571	3.5	1.00	361676	0.9	1.00	202640	2.3
p*1320	10 ⁸	1.00	4264494	13.4	0.00	> 10 ⁸	n/a	1.00	3167791	8.7	1.00	555636	9.1
p*1321	10 ⁸	1.00	8401861	27.4	0.01	> 10 ⁸	n/a	1.00	3512466	6.8	1.00	925001	12.8
p*1322	10 ⁸	0.94	18686150	85.0	0.02	> 10 ⁸	n/a	0.99	11563065	39.5	1.00	1203442	45.8
*v10000*03	10 ⁹	0.93	195343618	310.3	0.38	> 10 ⁹	n/a	0.00	> 10 ⁹	n/a	1.00	51325928	70.1
*v10000*04	10 ⁹	0.78	445825231	703.0	0.12	> 10 ⁹	n/a	0.00	> 10 ⁹	n/a	1.00	73729009	99.9
*v10000*05	10 ⁹	0.56	928751054	1407.3	0.00	> 10 ⁹	n/a	0.00	> 10 ⁹	n/a	0.96	140754923	188.4
*v10000*06	10 ⁹	0.92	287233505	427.6	0.22	> 10 ⁹	n/a	0.00	> 10 ⁹	n/a	0.99	52498950	70.6
*v10000*10	10 ⁹	0.84	340420147	536.8	0.10	> 10 ⁹	n/a	0.00	> 10 ⁹	n/a	0.99	71314104	95.9
*v1100*04	10 ⁹	0.99	160716875	692.8	0.99	154199134	454.7	0.00	> 10 ⁹	n/a	0.00	> 10 ⁹	n/a
*v1100*06	10 ⁹	0.96	254443055	1082.3	0.97	181909223	538.3	0.00	> 10 ⁹	n/a	0.00	> 10 ⁹	n/a
*v1100*08	10 ⁹	0.94	271460435	1159.5	0.94	274281212	810.0	0.00	> 10 ⁹	n/a	0.00	> 10 ⁹	n/a
*v1100*10	10 ⁹	0.98	201335004	852.7	0.94	244056999	723.9	0.00	> 10 ⁹	n/a	0.00	> 10 ⁹	n/a
*v1100*14	10 ⁹	1.00	48218593	205.9	1.00	56308880	166.8	0.00	> 10 ⁹	n/a	0.02	> 10 ⁹	n/a

at least one algorithm discussed achieves a success rate greater than 50% in order to calculate median number of search steps and median run time based on these 100 runs. We report success rates (“suc”), median numbers of search steps (“#steps”), and median run times (“time”) in seconds. If an algorithm cannot achieve a success rate greater than 50% on an instance within the specified cutoff, we use “> *Maxsteps*” (greater than *Maxsteps*) and “n/a” to denote the median number of search steps and median run time, respectively. Results in bold indicate the best results for an instance.

5.2 Default Values of Parameters in *NCVW*

Like *VW*, *NCVW* updates variable weights using Formula 1. To adapt to *NCVW*, which is set up to solve a wide range of instances, parameter s in this formula is fixed to 0.0. When s is 0.0, the weight of a variable is just a counter of the number of flips of this variable. Conversely, in *VW*, s is adjusted during the search ($s > 0.0$). That

is, *NCVW* does not smooth variable weights while *VW* does. This non-smoothing of variable weights makes uneven distributions of variable weights and even distributions of variable weights more distinguishable.

In addition to its own parameters γ , δ , and π , *NCVW* has all parameters from its constituent heuristics. According to our experiments, on a wide range of instances, *NCVW* with $(\gamma, \delta, \pi, s, wp) = (7.5, 3.0, 15.0, 0.0, 0.05)$ (*wp* is from *RSAPS*) exhibits generally good performance. Thus, in *NCVW*, the default values of γ , δ , π , s , and wp are $(\gamma, \delta, \pi, s, wp) = (7.5, 3.0, 15.0, 0.0, 0.05)$. For the other parameters in *NCVW* from its constituent heuristics, *NCVW* adaptively adjusts these parameters as do the constituent heuristics or uses the same default values as do the constituent heuristics.

5.3 Comparison of Performance

We compare the performance of *NCVW*, *adaptG²WSAT+*, *RSAPS*, and *VW* on the 11 groups of instances (36 instances) in Table 2, and compare the performance of *NCVW*, *gNovelty+*, *adaptG²WSAT0*, and *Hybrid* on these instances in Table 3. The source code of *adaptG²WSAT+*, *gNovelty+*, and *adaptG²WSAT0* was downloaded from <http://www.satcompetition.org/>, and that of *RSAPS* was downloaded from <http://www.satlib.org/ubcsat/>. The source code of *VW* was obtained from the organizer of the SAT 2005 competition. When experimenting with these algorithms, we do not change the ways in which these algorithms adaptively adjust their parameters and do not change the default values of the other parameters in these algorithms either.

According to our experiments, the 36 instances include those that, for *NCVW*, usually lead to the following four combinations of the distributions of variable and clause weights: the distributions of both variable and clause weights are even, the distributions of variable weights are even while the distributions of clause weights are uneven, the distributions of variable weights are uneven while the distributions of clause weights are even, and the distributions of both variable and clause weights are uneven. Specifically, for *NCVW*, the instances in *PARITY* and *5SAT/v1100* generally result in even distributions of both variable and clause weights. The instances in *Ferry* and *QG* usually lead to even distributions of variable weights but uneven distributions of clause weights. The instances *g*1340* and *g*1341* in *grid-pebbling/sat* usually result in uneven distributions of variable weights but even distributions of clause weights. The instances in *blocksworld* and *Beijing* generally lead to uneven distributions of both variable and clause weights.

According to Table 2, within the specified cutoffs, *NCVW* is generally effective on these 11 groups. Conversely, within the designated cutoffs, *adaptG²WSAT+*, *RSAPS*, and *VW* are effective on only 6, 4, and 6 groups, respectively.

NCVW exhibits good performance on *qg7-13* although *adaptG²WSAT+*, *RSAPS*, *VW* all show poor performance on this instance. There are two reasons for this good performance. First, like *adaptG²WSAT+*, *NCVW* conducts preprocessing using unit propagation to simplify an instance before searching. Second, *NCVW* usually chooses heuristic *RSAPS* automatically to select a variable to flip for the simplified *qg7-13* because this simplified *qg7-13* results in even distribution of variable

Table 3. Experimental results for *NCVW*, *gNovelty+*, *adaptG²WSAT0*, and *Hybrid* on the 11 groups of instances

	cutoff	<i>NCVW</i>			<i>gNovelty+</i>			<i>adaptG²WSAT0</i>			<i>Hybrid</i>		
		suc	#steps	time	suc	#steps	time	suc	#steps	time	suc	#steps	time
ais12	10 ⁷	0.93	249576	0.2	0.32	> 10 ⁷	n/a	1.00	1181980	1.1	1.00	1534609	2.0
bwLarge.d	10 ⁷	0.96	955185	2.5	0.60	6561933	16.7	0.49	> 10 ⁷	n/a	0.96	661253	2.1
e0ddr2*1	10 ⁷	1.00	119631	0.5	0.00	> 10 ⁷	n/a	0.96	2013651	3.0	1.00	117320	1.7
g250.29	10 ⁷	0.79	3483743	76.9	1.00	590941	10.7	1.00	806380	7.0	0.88	1360491	22.6
par16-1	10 ⁹	1.00	96371225	48.3	0.51	985564819	231.1	1.00	78289718	26.1	1.00	69932292	29.6
par16-2	10 ⁹	0.88	329217086	166.7	0.38	> 10 ⁹	n/a	0.99	105017111	35.8	0.99	119600333	52.2
par16-3	10 ⁹	1.00	115526876	58.8	0.52	955707228	224.5	1.00	113814551	39.0	1.00	92166515	40.2
par16-4	10 ⁹	0.96	183854606	92.8	0.26	> 10 ⁹	n/a	0.98	142059581	47.7	0.99	95694408	40.8
par16-5	10 ⁹	0.93	264988982	133.5	0.55	924073354	219.7	1.00	113484583	38.8	0.99	81990858	35.4
qg2-08	10 ⁷	0.83	2832886	12.7	0.02	> 10 ⁷	n/a	0.98	1757920	4.8	0.99	1440339	6.1
qg7-13	10 ⁸	1.00	3663127	31.9	0.00	> 10 ⁸	n/a	0.20	> 10 ⁸	n/a	0.40	> 10 ⁸	n/a
f*3995	10 ⁸	1.00	56405	0.1	0.00	> 10 ⁸	n/a	0.06	> 10 ⁸	n/a	0.15	> 10 ⁸	n/a
f*3997	10 ⁸	0.95	4830154	3.3	0.00	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a	0.01	> 10 ⁸	n/a
f*3999	10 ⁸	1.00	336040	0.4	0.00	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a
f*4001	10 ⁸	0.66	51971463	41.8	0.00	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a
f*4003	10 ⁸	1.00	1514236	1.9	0.00	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a
g*1334	10 ⁸	1.00	248515	0.2	0.04	> 10 ⁸	n/a	0.01	> 10 ⁸	n/a	1.00	69867	0.1
g*1337	10 ⁸	1.00	1411026	1.2	0.47	> 10 ⁸	n/a	0.31	> 10 ⁸	n/a	1.00	146414	0.2
g*1339	10 ⁸	1.00	2325027	3.5	0.67	23436378	32.7	0.49	> 10 ⁸	n/a	1.00	602498	2.0
g*1340	10 ⁸	0.76	3579295	7.3	0.00	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a	1.00	2542104	6.4
g*1341	10 ⁸	0.98	4507419	10.3	0.00	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a	1.00	2875184	9.8
p*1318	10 ⁸	1.00	1411992	6.2	0.49	> 10 ⁸	n/a	0.05	> 10 ⁸	n/a	0.85	828436	4.2
p*1319	10 ⁸	1.00	1039612	4.1	0.16	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a	0.26	> 10 ⁸	n/a
p*1320	10 ⁸	1.00	4264494	13.4	0.63	58348721	135.6	0.03	> 10 ⁸	n/a	0.79	507350	2.2
p*1321	10 ⁸	1.00	8401861	27.4	0.02	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a	0.15	> 10 ⁸	n/a
p*1322	10 ⁸	0.94	18686150	85.0	0.09	> 10 ⁸	n/a	0.00	> 10 ⁸	n/a	0.17	> 10 ⁸	n/a
*v10000*03	10 ⁹	0.93	195343618	310.3	1.00	55787237	93.0	0.12	> 10 ⁹	n/a	0.98	162732471	229.1
*v10000*04	10 ⁹	0.78	445825231	703.0	1.00	69007926	111.5	0.02	> 10 ⁹	n/a	0.88	262272862	345.4
*v10000*05	10 ⁹	0.56	928751054	1407.3	1.00	134839615	219.3	0.00	> 10 ⁹	n/a	0.77	516289640	735.8
*v10000*06	10 ⁹	0.92	287233505	427.6	1.00	53678135	84.1	0.10	> 10 ⁹	n/a	1.00	134103486	208.1
*v10000*10	10 ⁹	0.84	340420147	536.8	1.00	69535298	110.0	0.00	> 10 ⁹	n/a	0.92	260197067	370.4
*v1100*04	10 ⁹	0.99	160716875	692.8	0.04	> 10 ⁹	n/a	1.00	134579805	390.1	0.88	298792644	1410.6
*v1100*06	10 ⁹	0.96	254443055	1082.3	0.08	> 10 ⁹	n/a	0.98	189797513	544.4	0.75	468379966	2194.6
*v1100*08	10 ⁹	0.94	271460435	1159.5	0.08	> 10 ⁹	n/a	0.94	219521877	627.1	0.69	643453270	3039.1
*v1100*10	10 ⁹	0.98	201335004	852.7	0.06	> 10 ⁹	n/a	0.96	219480345	629.1	0.73	547026001	2556.3
*v1100*14	10 ⁹	1.00	48218593	205.9	0.26	> 10 ⁹	n/a	1.00	54801707	157.2	1.00	112659869	525.2

weights and uneven distribution of clause weights, and *heuristic RSAPS* is effective on the simplified qg7-13 although this heuristic is not effective on the original qg7-13.

As shown in Table 3, within the specified cutoffs, *NCVW* is generally effective on these 11 groups of instances while *gNovelty+*, *adaptG²WSAT0*, and *Hybrid* are effective on only 3, 5, and 8 groups, respectively.

5.4 Justification for Switching Strategy Used in *NCVW*

To justify the proposed switching strategy used in *NCVW*, we implement two other switching strategies in two algorithms *NCVW_diff* and *NCVW_rand*, which are described as follows. If the distribution of variable weights is uneven or the distribution of clause weights is uneven, *NCVW_diff* chooses a variable to flip according to *heuristic adaptG²WSAT+*. Otherwise, i.e., if the distributions of both variable and

Table 4. Experimental results for *NCVW*, *NCVW_diff*, and *NCVW_rand* on the hardest instances in the 11 groups

	cutoff	<i>NCVW</i>			<i>NCVW_diff</i>			<i>NCVW_rand</i>		
		suc	#steps	time	suc	#steps	time	suc	#steps	time
ais12	10^7	0.93	249576	0.2	0.94	3068730	4.5	1.00	340597	0.5
bw_large.d	10^7	0.96	955185	2.5	0.70	7040081	16.2	0.90	2061289	5.2
e0ddr2*1	10^7	1.00	119631	0.5	0.96	2206478	4.3	1.00	193154	0.7
g250.29	10^7	0.79	3483743	76.9	1.00	771854	11.6	1.00	994184	13.9
par16-2	10^9	0.88	329217086	166.7	1.00	131330419	65.8	0.98	198219412	118.2
qg7-13	10^8	1.00	3663127	31.9	0.20	$> 10^8$	n/a	0.32	$> 10^8$	n/a
f*4001	10^8	0.66	51971463	41.8	0.00	$> 10^8$	n/a	0.00	$> 10^8$	n/a
g*1341	10^8	0.98	4507419	10.3	0.00	$> 10^8$	n/a	1.00	28817283	66.2
p*1322	10^8	0.94	18686150	85.0	0.01	$> 10^8$	n/a	1.00	1932060	18.5
*v10000*05	10^9	0.56	928751054	1407.3	0.02	$> 10^9$	n/a	0.14	$> 10^9$	n/a
*v1100*08	10^9	0.94	271460435	1159.5	0.94	253006754	1056.5	0.39	$> 10^9$	n/a

clause weights are even, *NCVW_diff* first randomly selects a heuristic from *heuristic RSAPS* and *heuristic VW*, and then chooses a variable to flip according to the randomly selected heuristic. In each search step, *NCVW_rand* randomly selects a heuristic from *heuristic adaptG²WSAT+*, *heuristic RSAPS*, and *heuristic VW*, and then uses this randomly selected heuristic to choose a variable to flip.

Both *NCVW_diff* and *NCVW_rand* update variable and clause weights in the same ways as does *NCVW*. For an instance that leads to even distributions of variable weights, *NCVW_diff* will build up the distribution of clause weights, after a small number of search steps compared with the total search steps that *NCVW_diff* performs for this instance. Thus, *NCVW_diff* does not need parameter π . For each of the other parameters in *NCVW_diff*, *NCVW_diff* adjusts this parameter as does *NCVW* or uses the same default value as does *NCVW*. As opposed to *NCVW*, *NCVW_rand* does not need parameters γ , δ , and π . For each of the other parameters in *NCVW_rand*, *NCVW_rand* adjusts this parameter as does *NCVW* or uses the same default value as does *NCVW*. In Table 4, we compare the performance of *NCVW*, *NCVW_diff*, and *NCVW_rand* on the hardest instances in the 11 groups for most algorithms discussed in this paper. Within the specified cutoffs, *NCVW* is generally effective on these 11 instances, but *NCVW_diff* and *NCVW_rand* are effective on only 6 and 7 instances, respectively.

6 Conclusion

We have proposed a new switching criterion: the evenness or unevenness of the distribution of clause weights. We apply this criterion, along with another switching criterion, to *heuristic adaptG²WSAT+*, *heuristic RSAPS*, and *heuristic VW*. The resulting algorithm, which combines intensification strategies with diversification strategies, is called *NCVW* (Non-, Clause, and Variable Weighting). Experimental results show that *NCVW* is generally effective on a wide range of instances whereas *adaptG²WSAT+*, *RSAPS*, *VW*, *gNovelty+*, *adaptG²WSAT0*, and *Hybrid* are not.

References

1. Gebruers, C., Hnich, B., Bridge, D.G., Freuder, E.C.: Using CBR to Select Solution Strategies in Constraint Programming. In: Muñoz-Ávila, H., Ricci, F. (eds.) ICCBR 2005. LNCS (LNAI), vol. 3620, pp. 222–236. Springer, Heidelberg (2005)
2. Hirsch, E.A., Kojevnikov, A.: UnitWalk: A New SAT Solver that Uses Local Search Guided by Unit Clause Elimination. *Ann. Math. Artif. Intell.* 43(1), 91–111 (2005)
3. Hoos, H.H.: On the Run-Time Behavior of Stochastic Local Search Algorithms for SAT. In: *Proceedings of AAAI 1999*, pp. 661–666. AAAI Press, Menlo Park (1999)
4. Hoos, H.H.: An Adaptive Noise Mechanism for WalkSAT. In: *Proceedings of AAAI 2002*, pp. 655–660. AAAI Press, Menlo Park (2002)
5. Hoos, H.H., Stützle, T.: *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, San Francisco (2004)
6. Hutter, F., Tompkins, D.A.D., Hoos, H.H.: Scaling and Probabilistic Smoothing: Efficient Dynamical Local Search for SAT. In: Van Hentenryck, P. (ed.) *CP 2002*. LNCS, vol. 2470, pp. 233–248. Springer, Heidelberg (2002)
7. Li, C.M., Huang, W.Q.: Diversification and Determinism in Local Search for Satisfiability. In: Bacchus, F., Walsh, T. (eds.) *SAT 2005*. LNCS, vol. 3569, pp. 158–172. Springer, Heidelberg (2005)
8. Li, C.M., Wei, W., Zhang, H.: Combining Adaptive Noise and Promising Decreasing Variables in Local Search for SAT,
<http://www.satcompetition.org/2007/contestants.html>
9. Li, C.M., Wei, W., Zhang, H.: Combining Adaptive Noise and Look-Ahead in Local Search for SAT. In: Benhamou, F., Jussien, N., O’Sullivan, B. (eds.) *Trends in Constraint Programming*, ch. 14, pp. 261–267. ISTE (2007)
10. Li, C.M., Wei, W., Zhang, H.: Combining Adaptive Noise and Look-Ahead in Local Search for SAT. In: Marques-Silva, J., Sakallah, K.A. (eds.) *SAT 2007*. LNCS, vol. 4501, pp. 121–133. Springer, Heidelberg (2007)
11. Morris, P.: The Breakout Method for Escaping from Local Minima. In: *Proceedings of AAAI 1993*, pp. 40–45. AAAI Press, Menlo Park (1993)
12. Pham, D.N., Gretton, C.: gnovelty+,
<http://www.satcompetition.org/2007/contestants.html>
13. Prestwich, S.: Random Walk with Continuously Smoothed Variable Weights. In: Bacchus, F., Walsh, T. (eds.) *SAT 2005*. LNCS, vol. 3569, pp. 203–215. Springer, Heidelberg (2005)
14. Selman, B., Kautz, H., Cohen, B.: Noise Strategies for Improving Local Search. In: *Proceedings of AAAI 1994*, pp. 337–343. AAAI Press, Menlo Park (1994)
15. Wei, W., Li, C.M., Zhang, H.: Deterministic and Random Selection of Variables in Local Search for SAT,
<http://www.satcompetition.org/2007/contestants.html>
16. Wei, W., Li, C.M., Zhang, H.: Criterion for Intensification and Diversification in Local Search for SAT. In: *Proceedings of LSCS 2007*, pp. 2–16 (2007)
17. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla-07: The Design and Analysis of an Algorithm Portfolio for SAT. In: Bessière, C. (ed.) *CP 2007*. LNCS, vol. 4741, pp. 712–727. Springer, Heidelberg (2007)

CPBPV: A Constraint-Programming Framework for Bounded Program Verification

Hélène Collavizza¹, Michel Rueher¹, and Pascal Van Hentenryck²

¹ Université de Nice–Sophia Antipolis, France
{helen,rueher}@polytech.unice.fr

² Brown University, Box 1910, Providence, RI 02912
pvh@cs.brown.edu

Abstract. This paper studies how to verify the conformity of a program with its specification and proposes a novel constraint-programming framework for bounded program verification (CPBPV). The CPBPV framework uses constraint stores to represent the specification and the program and explores execution paths nondeterministically. The input program is partially correct if each constraint store so produced implies the post-condition. CPBPV does not explore spurious execution paths as it incrementally prunes execution paths early by detecting that the constraint store is not consistent. CPBPV uses the rich language of constraint programming to express the constraint store. Finally, CPBPV is parametrized with a list of solvers which are tried in sequence, starting with the least expensive and less general. Experimental results often produce orders of magnitude improvements over earlier approaches, running times being often independent of the variable domains. Moreover, CPBPV was able to detect subtle errors in some programs while other frameworks based on model checking have failed.

1 Introduction

This paper is concerned with software correctness, a critical issue in software engineering. It proposes a novel constraint-programming framework for bounded program verification (CPBPV), i.e., when the program inputs (e.g., the array lengths and the variable values) are bounded. The goal is to verify the conformity of a program with its specification, that is to demonstrate that the specification is a consequence of the program. The key idea of CPBPV is to use constraint stores to represent the specification and the program, and to non-deterministically explore execution paths over these constraint stores. This non-deterministic constraint-based symbolic execution incrementally refines the constraint store, which initially consists of the precondition. Non-determinism occurs when executing conditional or iterative instructions and the non-deterministic execution refines the constraint store by adding constraints coming from conditions and from assignments. The input program is partially correct if each constraint store produced by the symbolic execution implies the post-condition. It is important to emphasize that CPBPV considers programs with complete specifications and

that verifying the conformity between a program and its specification requires to check (explicitly or implicitly) all executables paths. This is not the case in model-checking tools designed to detect violations of some specific property, e.g., safety or liveness properties.

The CPBPV framework has a number of fundamental benefits. First, contrary to earlier work using constraint programming or SMT [2,11,12], CPBPV does not use predicate abstraction or explore spurious execution paths, i.e., paths that do not correspond to actual executions over inputs satisfying the pre-condition. CPBPV incrementally prunes execution paths early by detecting that the constraint store is not consistent. Second, CPBPV uses the rich language of constraint programming to express the constraint store, including arbitrary logical and threshold combination of constraints, the *element* constraint, and global/combinatorial constraints that express complex relationships on a set of variables. Finally, CPBPV is parametrized with a list of solvers which are tried in sequence, starting with the least expensive and less general.

The CPBPV framework was evaluated experimentally on a series of benchmarks from program verification. Experimental results of our (slow) prototype often produce orders of magnitude improvements over earlier approaches, and indicate that the running times are often independent of the variable domains. Moreover, CPBPV was able to found subtle errors in some programs that some other verification frameworks based on model-checking could not detect.

The rest of the paper is organized as follows. Section 2 illustrates how CPBPV handles constraints store on a motivating example. Section 3 formalizes the CPBPV framework for a small programming language and Section 4 discusses the implementation issues. Section 5 presents experimental results on a number of verification problems, comparing our approach with state of the art model-checking based verification frameworks. Section 6 discusses related work in test generation, bounded program verification and software model checking. Section 7 summarizes the contributions and presents future research directions.

2 The Constraint-Programming Framework at Work

This section illustrates the CPBPV verifier on a motivating example, the binary search program. CPBPV uses Java programs and JML specifications for the pre- and post-conditions, appropriately enhanced to support the expressivity of constraint programming. Figure 1 depicts a binary search program to determine if a value v is present in a sorted array t . (Note that `\result` in JML corresponds to the value returned by the program). To verify this program, our prototype implementation requires a bound on the length of array t , on its elements, and on v . We will verify its correctness for specific lengths and simply assume that the values are signed integers on a number of bits.

The initial constraint store of the CPBPV verifier, assuming an input array of length 8, is the precondition¹ $c_{pre} \equiv \forall 0 \leq i < 7 : t^0[i] \leq t^0[i + 1]$ where t^0 is an array of constraint variables capturing the input. The constraint variables

¹ We omit the domain constraints on the variables for simplicity.

```

/*@ requires (\forall int i; i>=0 && i<t.length-1;t[i]<=t[i+1])
   @ ensures
   @   (\result != -1 ==> t[\result] == v) &&
   @   (\result == -1 ==> \forall int k; 0 <= k < t.length ; t[k] != v)@*/
1 static int binary_search(int[] t, int v) {
2   int l = 0;
3   int u = t.length-1;
4   while (l <= u) {
5     int m = (l + u) / 2;
6     if (t[m]==v)
7       return m;
8     if (t[m] > v)
9       u = m - 1;
10    else
11      l = m + 1;  }   // ERROR else u = m - 1;
12  return -1; }

```

Fig. 1. The Binary Search Program

are annotated with a version number as CPBPV performs a SSA-like renaming [10] on the fly since each assignment generates constraints possibly linking the old and the new values of the assigned variable. The assignments in lines 2–3 add the constraints $l^0 = 0 \wedge u^0 = 7$. CPBPV then considers the loop instruction. Since $l^0 \leq u^0$, it enters the loop body, adds the constraint $m^0 = (l^0 + u^0)/2$, which simplifies to $m^0 = 3$, and considers the conditional statement on line 6. The execution of the statement is nondeterministic: Indeed, both $t^0[3] = v^0$ and $t^0[3] \neq v^0$ are consistent with the constraint store, so that the two alternatives, which give rise to two execution paths, must be explored. Note that these two alternatives correspond to actual execution paths in which $t[3]$ in the input is equal to, or different from, input v . The first alternative adds the constraint $t^0[3] = v^0$ to the store and executes line 7 which adds the constraint $result = m^0$. CPBPV has thus obtained an execution path p whose final constraint store c_p is: $c_{pre} \wedge l^0 = 0 \wedge u^0 = 7 \wedge m^0 = (l^0 + u^0)/2 \wedge t^0[m^0] = v^0 \wedge result = m^0$

CPBPV then checks whether this store c_p implies the post-condition c_{post} by searching for a solution to $c_p \wedge \neg c_{post}$. This test fails, indicating that the computation path p , which captures the set of actual executions in which $t[3] = v$, satisfies the specification. CPBPV then explores the other alternatives to the conditional statement in line 6. It adds the constraint $t^0[m^0] \neq v^0$ and executes the conditional statement in line 8. Once again, this statement is nondeterministic. Its first alternative assumes that the test holds, generating the constraint $t^0[m^0] > v^0$ and executing the instruction in line 9. Since u is (re-)assigned, CPBPV creates a new variable u^1 and posts the constraint $u^1 = m^0 - 1 = 2$. The execution returns to line 4, where the test now reads $l^0 \leq u^1$, since CPBPV always uses the most recent version for each variable. Since the constraint store entails $l^0 \leq u^1$, the only extension to the current path consists of executing line 5, adding the constraint $m^1 = (l^0 + u^1)/2$, which actually simplifies to $m^1 = 1$. Another complete execution path is then obtained by executing lines 6 and 7.

Consider now a version of the program in which line 11 is replaced by $u = m-1$. To illustrate the CPBPV verifier, we specify partial execution paths by indicating which alternative is selected for each nondeterministic instruction. For instance, $\langle T_4, F_6, T_8, T_5, T_6 \rangle$ denotes the last execution path discussed above in which the true alternative is selected for the first execution of the instruction in line 4, the false alternative for the first execution of instruction 6, the true alternative for the first instruction of instruction 8, the true alternative of the second execution of instruction 5, and the true alternative of the second execution of instruction 6. Consider the partial path $\langle T_4, F_6, F_8 \rangle$ and let us study how it can be extended. The partial path $\langle T_4, F_6, F_8, T_4, T_6 \rangle$ is not explored, since it produces a constraint store containing

$$c_{pre} \wedge t^0[3] \neq v^0 \wedge t^0[3] \leq v^0 \wedge t^0[1] = v^0$$

which is clearly inconsistent. Similarly, the path $\langle T_4, F_6, F_8, T_4, F_6, T_8 \rangle$ cannot be extended. The output of CPBPV on this incorrect program when executed on an array of length 8 (with integers coded on 8-bits to make it readable) produces, in 0.025 seconds, the counterexample:

$$v^0 = -126 \wedge t^0 = [-128, -127, -126, -125, -124, -123, -122, -121] \wedge result = -1.$$

This example highlights a few interesting benefits of CPBPV.

1. The verifier only considers paths that correspond to collections of actual inputs (abstracted by constraint stores). The resulting execution paths must all be explored since our goal is to prove the partial correctness of the program.
2. The performance of the verifier is independent of the integer representation on this application: it only requires a bound on the length of the array.
3. The verifier returns a counter-example for debugging the program.

Note that *CBMC* and *ESC/Java2*, two state-of-the-art model checkers fail to verify this example as discussed in Section 5.

3 Formalization of the Framework

This section formalizes the CPBPV verifier on a small abstract language using a small-step SOS semantics. The semantics primarily specifies the execution paths over constraint stores explored by the verifier. It features **assert** and **enforce** constructs which are necessary for modular composition.

Syntax. Figure 2 depicts the syntax of the programs and the constraints generated by the verifier. In the following, we use s , possibly subscripted, to denote elements of a syntactic entity S .

Renamings. CPBPV creates variables and arrays of variables “on-the-fly” when they are needed. This process resembles an SSA normalization but does not introduce the join nodes, since the results of different execution paths are not merged. Similar renamings are used in model checking. The renaming uses mappings of type $V \cup A \rightarrow \mathcal{N}$ which maps variables and arrays into a natural numbers

L : list of instructions; I : instructions; B : Boolean expressions
 E : integer expressions; A : arrays; V : variables

$L ::= I; L \mid \epsilon$
 $I ::= A[E] \leftarrow E \mid V \leftarrow E \mid \text{if } B \text{ } I \mid \text{while } B \text{ } I \mid \text{assert}(B) \mid \text{enforce}(B) \mid \text{return } E \mid \{L\}$
 $B ::= \text{true} \mid \text{false} \mid E > E \mid E \geq E \mid E = E \mid E \neq E \mid E \leq E \mid E < E$
 $B ::= \neg B \mid B \wedge B \mid B \vee B \mid B \Rightarrow B$
 $E ::= V \mid A[E] \mid E + E \mid E - E \mid E \times E \mid E / E \mid$

C : constraints E^+ : solver expressions
 $V^+ = \{v^i \mid v \in V \ \& \ i \in \mathcal{N}\}$: solver variables
 $A^+ = \{a^i \mid a \in A \ \& \ i \in \mathcal{N}\}$: solver arrays

$C ::= \text{true} \mid \text{false} \mid E^+ > E^+ \mid E^+ \geq E^+ \mid E^+ = E^+ \mid E^+ \neq E^+ \mid E^+ \leq E^+ \mid E^+ < E^+$
 $C ::= \neg C \mid C \wedge C \mid C \vee C \mid C \Rightarrow C$
 $E^+ ::= V \mid A[E^+] \mid E^+ + E^+ \mid E^+ - E^+ \mid E^+ \times E^+ \mid E^+ / E^+ \mid$

Fig. 2. The Syntax of Programs and Constraints

denoting their current “version numbers”. In the semantics, the version number is incremented each time a variable or an array element is assigned. We use $\sigma_\perp$ to denote the uniform mapping to zero (i.e., $\forall x \in V \cup A : \sigma_\perp(x) = 0$) and $\sigma[x/i]$ the mapping σ where x now maps to i , i.e., $\sigma[x/i](y) = \text{if } x = y \text{ then } i \text{ else } \sigma(y)$. These mappings are used by a polymorphic renaming function ρ to transform program expressions into constraints. For example, $\rho \ \sigma \ b_1 \oplus b_2 = (\rho \ \sigma \ b_1) \oplus (\rho \ \sigma \ b_2)$ (where $\oplus \in \{\wedge, \vee, \Rightarrow\}$) is the rule used to transform a logical expression.

Configurations. The CPBCV semantics mostly uses configurations of the type $\langle l, \sigma, c \rangle$, where l is the list of instructions to execute, σ is a version mapping, and c is the set of constraints generated so far. It also uses configurations of the form $\langle \top, \sigma, c \rangle$ to denote final states and configurations of the form $\langle \perp, \sigma, c \rangle$ to denote the violation of an assertion. The semantics is specified by rules of the form $\frac{\text{conditions}}{\gamma_1 \xrightarrow{\gamma_2} \gamma_2}$ stating that configuration γ_1 can be rewritten into γ_2 when the conditions hold.

Conditional Instructions. The conditional instruction *if* b i considers two cases. If the constraint c_b associated with b is consistent with the constraint store, then the store is augmented with c_b and the body is executed. If the negation $\neg c_b$ is consistent with the store, then the constraint store is augmented with $\neg c_b$. Both rules may apply, since the store may represent some memory states satisfying the condition and some violating it.

$$\frac{c \wedge (\rho \ \sigma \ b) \text{ is satisfiable}}{\langle \text{if } b \ i; l, \sigma, c \rangle \mapsto \langle i; l, \sigma, c \wedge (\rho \ \sigma \ b) \rangle} \quad \frac{c \wedge \neg(\rho \ \sigma \ b) \text{ is satisfiable}}{\langle \text{if } b \ i; l, \sigma, c \rangle \mapsto \langle l, \sigma, c \wedge \neg(\rho \ \sigma \ b) \rangle}$$

Iterative Instructions. The while instruction *while* b i also considers two cases. If the constraint c_b associated with b is consistent with the constraint store, then the constraint store is augmented with c_b , the body is executed, and

the while instruction is reconsidered. If the negation $\neg c_b$ is consistent with the constraint store, then the constraint store is augmented with $\neg c_b$.

$$\frac{c \wedge (\rho \sigma b) \text{ is satisfiable}}{\langle \text{while } b \ i ; l, \sigma, c \rangle \mapsto \langle i; \text{while } b \ i ; l, \sigma, c \wedge (\rho \sigma b) \rangle}$$

$$\frac{c \wedge \neg(\rho \sigma b) \text{ is satisfiable}}{\langle \text{while } b \ i ; l, \sigma, c \rangle \mapsto \langle l, \sigma, c \wedge \neg(\rho \sigma b) \rangle}$$

Scalar Assignments. Scalar assignments create a new constraint variable for the program variable to be assigned and add a constraint specifying that the variable is equal to the right-hand side. A new renaming mapping is produced.

$$\frac{\sigma_2 = \sigma_1[v/\sigma_1(v) + 1] \ \& \ c_2 \equiv (\rho \sigma_2 v) = (\rho \sigma_1 e)}{\langle v \leftarrow e ; l, \sigma_1, c_1 \rangle \mapsto \langle l, \sigma_2, c_1 \wedge c_2 \rangle}$$

Assignments of Array Elements. The assignment of an array element creates a new constraint array, add a constraint for the index being indexed and posts constraints specifying that all the new constraint variables in the array are equal to their earlier version, except for the element being indexed. Note that the index is an expression which may contain variables as well, giving rise to the well-known *element* constraint in constraint programming [25].

$$\frac{\begin{array}{l} \sigma_2 = \sigma_1[a/\sigma_1(a) + 1] \\ c_2 \equiv (\rho \sigma_2 a)[\rho \sigma_1 e_1] = (\rho \sigma_1 e_2) \\ c_3 \equiv \forall i \in 0..a.length : (\rho \sigma_1 e_1) \neq i \Rightarrow (\rho \sigma_2 a)[i] = (\rho \sigma_1 a)[i] \end{array}}{\langle a[e_1] \leftarrow e_2, \sigma_1 ; l, c_1 \rangle \mapsto \langle l, \sigma_2, c_1 \wedge c_2 \wedge c_3 \rangle}$$

Assert Statements. An assert statement checks whether the assertion is implied by the control store in which case it proceeds normally. Otherwise, it terminates the execution with an error.

$$\frac{c \Rightarrow (\rho \sigma b)}{\langle \text{assert } b ; l, \sigma, c \rangle \mapsto \langle l, \sigma, c \rangle} \qquad \frac{c \wedge \neg(\rho \sigma b) \text{ is satisfiable}}{\langle \text{assert } b ; l, \sigma, c \rangle \mapsto \langle \perp, \sigma, c \rangle}$$

Enforce Statements. An enforce statement adds a constraint to the constraint store if it is satisfiable.

$$\frac{c \wedge (\rho \sigma b) \text{ is satisfiable}}{\langle \text{enforce } b ; l, \sigma, c \rangle \mapsto \langle l, \sigma, c \wedge (\rho \sigma b) \rangle}$$

Block Statements. Block statements simply remove the braces.

$$\langle \{l_1\} ; l_2, \sigma, c \rangle \mapsto \langle l_1 : l_2, \sigma, c \rangle$$

Return Statements. A return statement simply constrains the *result* variable.

$$\frac{c_2 \equiv (\rho \sigma_1 \text{result}) = (\rho \sigma_1 e)}{\langle \text{return } e ; l, \sigma_1, c_1 \rangle \mapsto \langle \sigma_1, c_1 \wedge c_2 \rangle}$$

Termination. Termination also occurs when no instruction remains.

$$\langle \epsilon, \sigma, c \rangle \mapsto \langle \top, \sigma, c \rangle$$

The CPBPV Semantics. Let $\mathcal{P}$ be program $b_{pre} \ l \ b_{post}$ in which b_{pre} denotes the precondition, l is a list of instructions, and b_{post} the post-condition. Let $\mapsto^*$ be the transitive closure of $\mapsto$. The final states are specified by the set

$$SFN(b_{pre}, \mathcal{P}) = \{ \langle f, \sigma, c \rangle \mid \langle i, \sigma_\perp, \rho \ \sigma_\perp \ b_{pre} \rangle \mapsto^* \langle f, \sigma, c \rangle \wedge f \in \{\perp, \top\} \}$$

The program violates an assertion if the set

$$SFE(b_{pre}, \mathcal{P}, b_{post}) = \{ \langle \perp, \sigma, c \rangle \in SFN(b_{pre}, \mathcal{P}) \}$$

is not empty. It violates its specification if the set

$$SFE(b_{pre}, \mathcal{P}, b_{post}) = \{ \langle \top, \sigma, c \rangle \in SFN(b_{pre}, \mathcal{P}) \mid c \wedge (\rho \ \sigma \neg b_{post}) \text{ satisfiable} \}$$

is not empty. It is partially correct otherwise.

4 Implementation Issues

The CPBPV framework is parametrized by a list of solvers $(S_1, \dots, S_k)$ which are tried in sequence, starting with the least expensive and less general. When checking satisfiability, the verifier never tries solver $S_{i+1}, \dots, S_k$ if solver S_i is a decision procedure for the constraint store. If solver S_i is not a decision procedure, it uses an abstraction α of the constraint store c satisfying $c \Rightarrow \alpha$ and can still detect failed execution paths quickly. The last solver in the sequence is a constraint-programming solver (CP solver) over finite domains which iterates pruning and searching to find solutions or prove infeasibility. When the CP solver makes a choice, the earlier solvers in the sequence are called once again to prune the search space or find solutions if they have become decision procedures. Our prototype implementation uses a sequence (MIP, CP) , where MIP is the mixed integer-programming tool ILOG CPLEX² and CP is the constraint-programming tool Ilog JSOLVER. Our Java implementation also performs some trivial simplifications such as constant propagation but is otherwise not optimized in its use of the solvers and in its renaming process whose speed and memory usage could be improved substantially. Practically, simplifications are done on the fly and the MIP solver is called at each node of the executable paths. The CP solver is only called at the end of the executable paths when the complete post condition is considered. Currently, the implementation use a depth-first strategy for the CP solver, but modern CP languages now offer high-level abstractions to implement other exploration strategies. In practice, when CPBPV is used for model checking as discussed below, it is probably advisable to use a depth-first iterative deepening implementation.

² See <http://www.ilog.com/products>

5 Experimental Results

In this section, we report experimental results for a set of traditional benchmarks for program verification. We compare CPBVP with the following frameworks:

- ESC/Java is an Extended Static Checker for Java to find common run-time errors in JML-annotated Java programs by static analysis of the code and its annotations. See <http://kind.ucd.ie/products/opensource/ESCJava2/>.
- CBMC is a Bounded Model Checker for ANSI-C and C++ programs. It allows for the verification of array bounds (buffer overflows), pointer safety, exceptions, and user-specified assertions. See <http://www.cprover.org/cbmc/>.
- BLAST, the Berkeley Lazy Abstraction Software Verification Tool, is a software model checker for C programs. See <http://mtc.epfl.ch/software-tools/blast/>.
- EUREKA is a C bounded model checker which uses an SMT solver instead of an SAT solver. See <http://www.ai-lab.it/eureka/>.
- Why is a software verification platform which integrates many existing provers (proof assistants such as Coq, PVS, HOL 4,...) and decision procedures such as Simplify, Yices, ...). See <http://why.lri.fr/>.

Of course, neither the expressiveness nor the objectives of all these systems are the same as the one of CPBPV. For instance, some of them can handle CTL/LTL constraints whereas CPBPV does not yet support this kind of constraints. Nevertheless, this comparison is useful to illustrate the capabilities of CPBPV.

All experiments were performed on the same machine, an Intel(R) Pentium(R) M processor 1.86GHz with 1.5G of memory, using the version of the verifiers that can be downloaded from their web sites (except for EUREKA for which the execution times given in [2,3] are reported.) For each benchmark program, we describe the data entries and the verification parameters. In the tables, “UNABLE” means that the corresponding framework is unable to validate the program either because a lack of expressiveness or because of time or memory limitations, “NOT_FOUND” that it does not detect an error, and “FALSE_ERROR” that it reports an error in a correct program. Complete details of the experiments, including input files and error traces, can be found in [13].

Binary Search. We start with the binary search program presented in figure 1. ESC/Java is applied on the program described in Figure 1. ESC/Java requires a limit on the number of loop unfoldings, which we set to $\log(n) + 1$ which is the worst case complexity of binary search algorithm for an array of length n . Similarly, CBMC requires an overestimate of the number of loop unfoldings. Since CBMC does not support first-order expressions such as JML *forall* statement, we generated a C program for each instance of the problem (i.e., each array length). For example, the postcondition for an array of length 8 is given by

```
(result!=-1 && a[result]==x)||
(result==-1 && (a[0]!=x&&a[1]!=x&&a[2]!=x&&a[3]!=x&&a[4]!=x&&a[5]!=x&&a[6]!=x&&a[7]!=x))
```

For the Why framework, we used the binary search version given in their distribution. This program uses an assert statement to give a loop invariant.

Table 1. Comparison table for binary search

CPBPV	array length	8	16	32	64	128	256
	time	1.081s	1.69s	4.043s	17.009s	136.80s	1731.696s
CBMC	array length	8	16	32	64	128	256
	time	1.37s	1.43s	UNABLE	UNABLE	UNABLE	UNABLE
Why	with invariant	11.18s					
	without invariant	UNABLE					
ESC/Java	FALSE_ERROR						
BLAST	UNABLE						

Note that CPBPV does not require any additional information: no invariant and no limits on loop unfoldings. During execution, it selects a path by nondeterministically applying the semantic rules for conditional and loop expressions.

Table 1 reports the experimental results. Execution times for CPBPV are reported as a function of the array length for integers coded on 31 bits.³ Our implementation is neither optimized for time or space at this stage and times are only given to demonstrate the feasibility of the CPBPV verifier.

The “Why” framework [16] was unable to verify the correctness without the loop invariant; 60% of the proof obligations remained unknown.

The CBMC framework was not able to do the verification for an instance of length 32 (it was interrupted after 6691,87s).

ESC/Java was unable to verify the correctness of this program unless complete loop invariants are provided⁴.

An Incorrect Binary Search. Table 2 reports experimental results for an incorrect *binary search* program (see Figure 1, line 11) for CPBPV, ESC/Java, CBMC, and Why using an invariant. The error trace found with CPBPV has been described in Section 2. The error traces provided by CBMC and ESC/Java only show the decisions taken along the faulty path can be found in [13]. In contrast to CPBPV, they do not provide any value for the array nor the searched data. Observe that CPBPV provides orders of magnitude improvements in efficiency over CBMC and also outperforms ESC/Java by almost a factor 8 on the largest instance.

The Tritype Program. The tritype program is a standard benchmark in test case generation and program verification since it contains numerous non-feasible paths: only 10 paths correspond to actual inputs because of complex conditional statements in the program. The program takes three positive integers as inputs (the triangle sides) and returns 2 if the inputs correspond to an isosceles triangle, 3 if they correspond to an equilateral triangle, 1 if they correspond to some other triangle, and 4 otherwise. The `tritype` program in Java with its specification in JML can be found in [13]. Table 3 depicts the experimental results for CPBPV,

³ The commercial MIP solver fails with 32-bit domains because of scaling issues.

⁴ A version with loop invariants that allows to show the correctness of this program has been written by David Cok, a developer of ESC/Java, after we contacted him.

Table 2. Experimental Results for an Incorrect Binary Search

	CPBPV	ESC/Java	CBMC	WHY with invariant	BLAST
length 8	0.027s	1.21 s	1.38s	NOT_FOUND	UNABLE
length 16	0.037s	1.347 s	1.69s	NOT_FOUND	UNABLE
length 32	0.064s	1.792 s	7.62s	NOT_FOUND	UNABLE
length 64	0.115s	1.886 s	27.05s	NOT_FOUND	UNABLE
length 128	0.241s	1.964 s	189.20s	NOT_FOUND	UNABLE

Table 3. Experimental Results on the Tritype Program

	CPBPV	ESC/Java	CBMC	Why	BLAST
time	0.287s	1.828s	0.82s	8.85s	UNABLE

ESC/Java, CBMC, BLAST and Why. BLAST was unable to validate this example because the current version does not handle linear arithmetic. Observe the excellent performance of CPBPV and note that our previous approach using constraint programming and Boolean abstraction to abstract the conditions, validated this benchmark in 8.52 seconds when integers were coded on 16 bits [12]. It also explored 92 spurious paths.

An Incorrect Tritype Program. Consider now an incorrect version of *Tritype* program in which the test “if ((trityp==2) && (i+k>j))” in line 22 (see [13]) is replaced by “if ((trityp==1) && (i+k>j))”. Since the local variable *trityp* is equal to 2 when $i=k$, the condition $(i+k)>j$ implies that (i,j,k) are the sides of an isosceles triangle (the two other triangular inequalities are trivial because $j>0$). But, when $trityp=1$, $i=j$ holds and this incorrect version may answer that the triangle is isosceles while it may not be a triangle at all. For example, it will return 2 when $(i,j,k)=(1,1,2)$. Table 4 depicts the experimental results. Execution times correspond to the time required to find the first error. The error found with CPBPV corresponds to input values $(i,j,k) = (1,1,2)$ mentioned earlier. Once again, observe the excellent behavior of CPBPV compared to the remaining tools.⁵

Bubble Sort with Initial Condition. This benchmark (see [13]) is taken from [2] and performs a bubble sort of an array t which contains integers from 0 to $t.length$ given in decreasing order. Table 5 shows the comparative results for this benchmark. CPBPV was limited on this benchmark because its recursive implementation uses up all the JAVA stack space. This problem should be remedied by removing recursion in CPBPV.

Selection Sort. We now present a benchmark to highlight both modular verification and the `element` constraint of constraint programming to index arrays

⁵ For CBMC, we have contacted D. Kroening who has recommended to use the option `CPROVER_assert`. If we do so, CBMC is able to find the error, but we must add some assumptions to mean that there is no overflow into the sums, in order to prove the correct version of *tritype* with this same option.

Table 4. Experimental Results for the Incorrect Tritype Program

	CPBPV	ESC/Java	CBMC	WHY
time	0.056s	1.853s	NOT_FOUND	NOT_FOUND

Table 5. Experimental Results for Bubble Sort

	CPBPV	ESC/Java	CBMC	EUREKA
length 8	1.45s	3.778 s	1.11s	91s
length 16	2.97s	UNABLE	2.01s	UNABLE
length 32	UNABLE	UNABLE	6.10s	UNABLE
length 64	UNABLE	UNABLE	37.65s	UNABLE

with arbitrary expressions. The benchmark described in [13]. Assume that function `findMin` has been verified for arbitrary integers. When encountering a call to `findMin`, CPBPV first checks if its precondition is entailed by the constraint store, which requires a consistency check of the constraint store with respect to the negation of the precondition. Then CPBPV replaces the call by the postcondition where the formal parameters are replaced by the actual variables. In particular, for the first iteration of the loop and an array length of 40, CPBPV generates the conjunction $0 \leq k^0 < 40 \wedge t^0[k^0] \leq t^0[0] \wedge \dots \wedge t^0[k^0] \leq t^0[39]$ which features `element` constraint [25]. Indeed, k^0 is a variable and a constraint like $t^0[k^0] \leq t^0[0]$ indexes the array t^0 of variables using k^0 .

The modular verification of the selection sort explores only a single path, is independent of the integer representation, and takes less than 0.01s for arrays of size 40. The bottleneck in verifying selection sort is the validation of function `findMin`, which requires the exploration of many paths. However the complete validation of selection sort takes less than 4 seconds for an array of length 6. Once again, this should be contrasted with the model-checking approach of Eureka [2]. On a version of selection sort where all variables are assigned specific values (contrary to our verification which makes no assumptions on the inputs), Eureka takes 104 seconds on a faster machine. Reference [2] also reports that CBMC takes 432.6 seconds, that BLAST cannot solve this problem, and that SATABS [9] only verifies the program for an array with 2 elements.

Sum of Squares. Our last benchmark is described in [13] and computes the sum of the square of the n first integers stored in an array. The precondition states that n is the size of the array and that t must contain any possible permutation of the n first integers. The postcondition states that the result is $n \times (n + 1) \times (2 \times n + 1)/6$. The benchmark illustrates two functionalities of constraint programming: the ability of specifying combinatorial constraints and of solving nonlinear problems. The `alldifferent` constraint [23] in the pre-condition specifies that all the elements of the array are different, while the program constraints and postcondition involves quadratic and cubic constraints. The maximum instance that we were able to solve with CPBPV was an array of size 10 in 66.179s.

CPLEX, the MIP solver, plays a key role in all these benchmarks. For instance, the CP solver is never called in the Tritype benchmark. For the Binary search benchmark, there are length calls to the CP solver but almost 75% of the CPU time is spent in the CP solver. Since there is only path in the Buble sort benchmark, the CP solver is only called once. In the Sum of squares example, 80% of the CPU time is spent in the CP solver.

6 Discussion and Related Work

We briefly review recent work in constraint programming and model checking for software testing, validation, and verification. We outline the main differences between our CPBPV framework and existing approaches.

Constraint Logic Programming. Constraint logic programming (CLP) was used for test generation of programs (e.g., [17,20,24,19]) and provides a nice implementation tool extending symbolic execution techniques [4]. Gotlieb et al. showed how to represent imperative programs as constraint logic programs and used predicate abstraction (from model checking) and conditional constraints within a CLP framework. Flanagan [15] formalized the translation of imperative programs into CLP, argued that it could be used for bounded model checking, but did not provide an implementation. The test-generation methodology was generalized and applied to bounded program verification in [11,12]. The implementation used dedicated predicate abstractions to reduce the exploration of spurious execution paths. However, as shown in the paper, the CPBPV verifier is significantly more efficient and often avoids the generation of spurious execution paths completely.

Model Checking. It is also useful to contrast the CPBPV verifier with model checking of software systems. SAT-based bounded model checking for software [6] consists in building a propositional formula whose models correspond to execution paths of bounded length violating some properties and in using SAT solvers to check whether the resulting formula is satisfiable. SAT-based model checking platforms [6] have been widely popular thanks to significant progress in SAT solvers. A fundamental issue faced by model checkers is the state space explosion of the resulting model. Various techniques have been proposed to address this challenge, including generalized symbolic execution (e.g., [21]), SMT-based model checking, and abstraction/refinement techniques. SMT-based model checking is the idea of representing and checking quantifier-free formulas in a more general decidable theory (e.g. [14,18,22]). These SMT solvers integrate dedicated solvers and share some of the motivations of constraint programming. Predicate abstraction is another popular technique to address the state space explosion. The idea consists in abstracting the program to obtain an abstract program on which model checking is performed. The model checker may then generate an abstract counterexample which must be checked to determine if it corresponds to a concrete execution path. If the counterexample is spurious, the abstract program is refined and the process is iterated. A successful predicate

abstraction consists of abstracting the concrete program into a Boolean program (e.g., [5,7,8]). In recent work [3,2], Armando & al proposed to abstract concrete programs into linear programs and used an abstraction of sets of variables and array indices. They showed that their tool compares favourably and, on some of the programs considered in this paper, outperforms model checkers based on predicate abstraction.

Our CPBPV verifier contrasts with SAT-based model checkers, SMT-based model checkers and predicate abstraction based approaches: It does not abstract the program and does not generate spurious execution paths. Instead it uses a constraint-solver and nondeterministic exploration to incrementally construct abstractions of execution paths. The abstraction uses constraint stores to represent sets of concrete stores. On many bounded verification benchmarks, our preliminary experimental results show significant improvements over the state-of-the-art results in [2]. Model checking is well adapted to check low-level C program and hardware applications with numerous Boolean constraints and bit-wise operations: It was successfully used to compare an ANSI C program with a circuit given as design in Verilog [7]. However, it is important to observe that in model checking, one is typically interested in checking some specific properties such as buffer overflows, pointer safety, or user-specified assertions. These properties are typically much less detailed than our post-conditions and abstracting the program may speed up the process significantly. In our CPBPV verifier, it is critical to explore all execution paths and the main issue is how to effectively abstract memory stores by constraints and how to check satisfiability incrementally. It is an intriguing issue to determine whether an hybridization of the two approaches would be beneficial for model checking, an issue briefly discussed in the next section. Observe also that this research provides convincing evidence of the benefits of Nieuwenhuis' challenge [22] aiming at extending SMT⁶ with CP techniques.

7 Perspectives and Future Work

This paper introduced the CPBPV framework for bounded program verification. Its novelty is to use constraints to represent sets of memory stores and to explore execution paths over these constraint stores nondeterministically and incrementally. The CPBPV verifier exploits the fact that, when variables and arrays are bounded, the constraint store can always be checked for feasibility. As a result, it never explores spurious execution path contrary to earlier approaches combining constraint programming and predicate abstraction [11,12] or integrating SMT solvers and the abstraction/refinement approach from model checking [2]. We demonstrated the CPBPV verifier on a number of standard benchmarks from model checking and program checking as well as on nonlinear programs and functions using complex array indexings, and showed how to perform modular verification. The experimental results demonstrate the potential of the approach:

⁶ See also [1] for a study of the relations between constraint programming and Satisfiability Modulo Theories (SMT).

The CPBPV verifier provides significant gain in performance and functionalities compared to other tools.

Our current work aims at improving and generalizing the framework and implementation. In particular, we would like to include tailored, light-weight solvers for a variety of constraint classes, the optimization of the array implementation, and the integration of Java objects and references. There are also many research avenues opened by this research, two of which are reviewed now.

Currently, the CPBPV verifier does not check for variable overflows: the constraint store enforces that variables take values inside their domains and execution paths violating these constraints are thus not considered. It is possible to generalize the CPBPV verifier to check overflows as the verification proceeds. The key idea is to check before each assignment if the constraint store entails that the value produced fits in the selected integer representation and generate an error otherwise. (Similar assertions must in fact be checked for each subexpression in the right hand-side in the language evaluation order. Interval techniques on floats [4] may be used to obtain conservative checking of such assertions.

An intriguing direction is to use the CPBPV approach for properties checking. Given an assertion to be verified, one may perform a backward execution from the assertion to the function entry point. The negation of the assertion is now the pre-condition and the pre-condition becomes the post-condition. This requires to specify inverse renaming and executions of conditional and iterative statements but these have already been studied in the context of test generation.

Acknowledgements. Many thanks to Jean-Francois Couchot for many helps on the use of the *Why* framework.

References

1. Aït-Kaci, H., Berstel, B., Junker, U., Leconte, M., Podelski, A.: Satisfiability Modulo Structures as Constraint Satisfaction: An Introduction. In: *Procs of JFLA 2007* (2007)
2. Armando, A., Benerecetti, M., Montovani, J.: Abstraction Refinement of Linear Programs with Arrays. In: Grumberg, O., Huth, M. (eds.) *TACAS 2007*. LNCS, vol. 4424, pp. 373–388. Springer, Heidelberg (2007)
3. Armando, A., Mantovani, J., Platania, L.: Bounded Model Checking of C Programs using a SMT solver instead of a SAT solver. In: Valmari, A. (ed.) *SPIN 2006*. LNCS, vol. 3925, pp. 146–162. Springer, Heidelberg (2006)
4. Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability* 16(2), 97–121 (2006)
5. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian Abstraction for Model Checking C Programs. In: Margaria, T., Yi, W. (eds.) *TACAS 2001*. LNCS, vol. 2031. Springer, Heidelberg (2001)
6. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded Model Checking using Satisfiability Solving. *FMSD* 19(1), 7–34 (2001)
7. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) *TACAS 2004*. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)

8. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate abstraction of ANSI-C Programs using SAT. *FMSD* 25, 105–127 (2004)
9. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwegs, N., Zuck, L.D. (eds.) *TACAS 2005*. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
10. Cytron, R., Ferrante, J., Rosen, B., Wegman, M., Zadeck, K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *Transactions on Programming Languages and Systems* 13(4), 451–490 (1991)
11. Collavizza, H., Rueher, M.: Software Verification using Constraint Programming Techniques. In: Hermanns, H., Palsberg, J. (eds.) *TACAS 2006*. LNCS, vol. 3920, pp. 182–196. Springer, Heidelberg (2006)
12. Collavizza, H., Rueher, M.: Exploring different constraint-based modelings for program verification. In: *Procs. of CP 2007*. LNCS, vol. 3920, pp. 182–196 (2007)
13. Collavizza H. Rueher M., Van Hentenryck P. : Comparison between CPBPV with ESC/Java, CBMC, Blast, EUREKA and Why,
<http://www.i3s.unice.fr/~rueher/verificationBench.pdf>
14. Dutertre, B., de Moura, L.M.: 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)
15. Flanagan, C.: Automatic software model checking via constraint logic. *Science of Computer Programming* 50(1-3), 253–270 (2004)
16. Fillitre, J.C., Marché, C.: The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In: Damm, W., Hermanns, H. (eds.) *CAV 2007*. LNCS, vol. 4590, pp. 173–177. Springer, Heidelberg (2007)
17. Gotlieb, A., Botella, B., Rueher, M.: Automatic Test Data Generation using Constraint Solving Techniques. In: *Proc. ISSTA 1998; ACM SIGSOFT (2)* (1998)
18. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): Fast Decision Procedures. In: Alur, R., Peled, D.A. (eds.) *CAV 2004*. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)
19. Godefroid, P., Levin, M.Y., Molnar, D.: Automated Whitebox Fuzz Testing. In: *NDSS 2008, Network and Distributed System Security Symposium* (2008)
20. Jackson, D., Vaziri, M.: Finding Bugs with a Constraint Solver. In: *ACM SIGSOFT Symposium on Software Testing and Analysis*, pp. 14–15 (2000)
21. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized Symbolic Execution for Model Checking and Testing. In: Garavel, H., Hatcliff, J. (eds.) *TACAS 2003*. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003)
22. Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E., Rubio, A.: Challenges in Satisfiability Modulo Theories. In: Baader, F. (ed.) *RTA 2007*. LNCS, vol. 4533, pp. 2–18. Springer, Heidelberg (2007)
23. Régim, J.-C.: A filtering algorithm for constraints of difference in CSPs. In: *AAAI 1994, Seattle, WA, USA*, pp. 362–367 (1994)
24. Sy, N.T., Deville, Y.: Automatic Test Data Generation for Programs with Integer and Float Variables. In: *Proc. of 16th IEEE ASE 2001* (2001)
25. VanHentenryck, P.: *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge (1989)
26. Van Hentenryck, P., Michel, L., Deville, Y.: *Numerica: A Modeling Language for Global Optimization*. MIT Press, Cambridge (1997)

Exploiting Common Subexpressions in Numerical CSPs

Ignacio Araya, Bertrand Neveu, and Gilles Trombettoni

INRIA, Université de Nice-Sophia, Certis

{Ignacio.Araya,Bertrand.Neveu,Gilles.Trombettoni}@sophia.inria.fr

Abstract. It is acknowledged that the symbolic form of the equations is crucial for interval-based solving techniques to efficiently handle systems of equations over the reals. However, only a few automatic transformations of the system have been proposed so far. Vu, Schichl, Sam-Haroud, Neumaier have exploited common subexpressions by transforming the equation system into a unique directed acyclic graph. They claim that the impact of common subexpressions elimination on the gain in CPU time would be only due to a reduction in the number of operations.

This paper brings two main contributions. First, we prove theoretically and experimentally that, due to interval arithmetics, exploiting certain common subexpressions might also bring additional filtering/contraction during propagation. Second, based on a better exploitation of n-ary plus and times operators, we propose a new algorithm I-CSE that identifies and exploits *all* the “useful” common subexpressions. We show on a sample of benchmarks that I-CSE detects more useful common subexpressions than traditional approaches and leads generally to significant gains in performance, of sometimes several orders of magnitude.

1 Introduction

Granvilliers et al. [9] show in a survey several ways to combine symbolic and interval methods to improve performance of solvers. They noticed that Gröbner basis computation [3] introduces redundancies that often improve the pruning effect of interval techniques. The use of several forms of the equations together in the same system (e.g., the natural and centered forms) has the same effect.

The presence of multiple occurrences of the same variable in a given equation is well-known to lower the power of interval arithmetics [17]. Thus, several practitioners apply by hand symbolic transformations of their systems, such as factorizations, to limit the number of occurrences of variables [9,15].

Common subexpression elimination (CSE) is an important feature of compiler optimization [16]. CSE searches in the code for common subexpressions with identical evaluation and replaces them by auxiliary variables. It generally fasten the program by decreasing the number of instructions. Symbolic tools like Mathematica [2] or Maple [11] represent equations by directed acyclic graphs (DAGs), where nodes with several parents correspond to *common subexpressions* (CSs). This decreases the number of evaluations and also stores all the expressions with

less memory. Ceberio and Granvilliers in [4] use Gaussian elimination to reduce the number of non-linear terms in equations. As a side effect, their algorithm identifies some CSs.

Following the representation used by symbolic tools, Schichl and Neumaier have proposed a unique DAG to represent a system of equations handled by interval analysis techniques [18]. CSs are the nodes of the DAG with several parents and the main interval analysis operators are redefined on this data structure: evaluation of functions, computation of derivatives, etc. Vu, Schichl and Sam-Haroud have described in [20] how to carry out propagation in the DAG. In particular, an interval is attached to internal nodes and the propagation is performed in a sophisticated way: two queues are managed, one for the evaluation, the other for the narrowing/propagation (see below), and the top-down narrowing operations have priority over the bottom-up evaluation. All the researchers who have exploited CSs manually or automatically [9,20] think that the gain in performance due to common subexpressions would be only implied by a reduction of the number of operations.

The first good news is that CSE in interval analysis might bring *a stronger contraction/filtering power*. Section 3 clearly states which types of CSs are useful for bringing additional filtering. Section 4 presents a new algorithm I-CSE (Interval CSE) to detect CSs and generate a new system of equations. For a given form of the equations, I-CSE is able to find *all* the “useful” CSs, because it finds all the n -ary maximal CSs corresponding to sums and products, modulo the commutativity and the associativity of these operators, including overlapping CSs. In addition, I-CSE is not intrusive in that it produces a new system that can be handled by any interval solver using a classical propagation scheme. Finally, experiments shown in Section 6 highlight that the CSs are extracted very quickly. The new system of equations then leads solving algorithms using HC4 to significant gains in performance (of sometimes several orders of magnitude).

2 Background

The algorithms presented in this paper aim at solving systems of equations or, more generally, numerical CSPs.

Definition 1. A numerical CSP (NCSP) $P = (V, C, B)$ contains a set of constraints C and a set V of n variables. Every variable $x_i \in V$ can take a real value in the interval X_i and B is the cartesian product (called a **box**) $X_1 \times \dots \times X_n$. A solution to P is an assignment of the variables in V satisfying all constraints in C .

Finding all the solutions to an NCSP follows a scheme analogous to branch and prune for CSPs. *Branch*: Bisections divide the domain of one variable into two sub-domains in a combinatorial way. *Prune*: Two types of algorithms are used. Algorithms from interval analysis, like interval Newton [17], contract/filter the current box in all the dimensions simultaneously and can often guarantee that a box contains a unique solution. Algorithms from constraint programming are

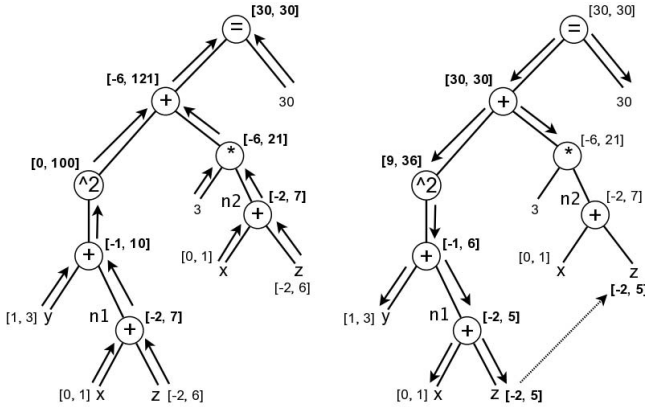


Fig. 1. Evaluation and narrowing in the HC4-revise algorithm. The tree represents the constraint: $(x + y + z)^2 + 3(x + z) = 30$.

also useful. HC4 follows a propagation loop like that of AC3 and handles the constraints individually with a procedure HC4-revise that removes inconsistent values on the bounds of intervals [1,12]. Stronger consistencies like 3B [13], similar to SAC [7] for finite domain CSPs, often obtain a better performance. At the end, the solving algorithm finds an approximation of all the solutions of the NCSP. The algorithm HC4-revise uses a tree representation of one constraint, where leaves are constants or variables, and internal nodes correspond to *primitive operators* like +, $\times$, *sinus*. An interval is associated with every node. HC4-revise works in two phases. The *evaluation* phase is performed bottom-up from the leaves (variables and constants) to the root. Using the *natural extension* of primitive functions, this phase evaluates the intervals of the sub-expressions represented by the tree nodes (see Fig. 1-left). The *narrowing* phase traverses the tree top-down from root to leaves and applies in every node a narrowing operator (also called projection; see Fig. 1-right). The narrowing operator contracts the intervals of the nodes eliminating inconsistent values w.r.t. the corresponding unary or binary primitive operator. In Fig. 1, the intervals in bold have been narrowed. If an empty interval is obtained during the narrowing phase, this means that the constraint is inconsistent w.r.t. the initial domains. The intervals computed in the internal nodes are not stored from one call to HC4-revise to another, as opposed to the intervals of the leaves (i.e., the variables).

3 Properties of HC4 and CSE

We call *Common Subexpression* (in short CS) a numerical expression that occurs several times in one or several constraints.

If we observe carefully the HC4-revise algorithm, we can note that the contraction obtained by a narrowing operator on a given expression f is in general partially lost in the next evaluation of f . Consider for instance a sum $x + z$

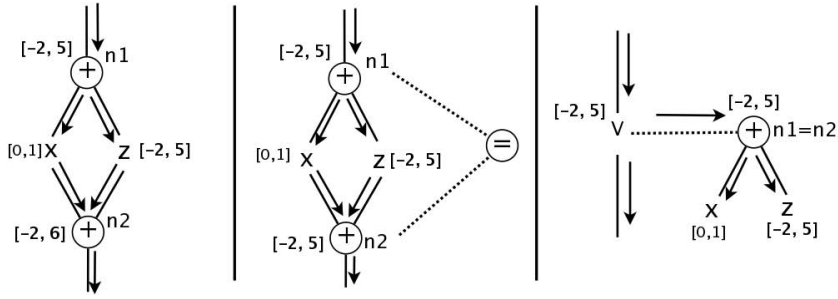


Fig. 2. Narrowing/Evaluation without and with CSE

that is shared by two expressions n_1 and n_2 . Following Fig. 1, the narrowing phase of **HC4-revise** applied to n_1 contracts its interval to $[-2, 5]$. Then, when the evaluation phase of **HC4-revise** applies to n_2 , its interval is set to $[-2, 6]$ (Fig. 2-left). Clearly, the interval of n_2 is larger than that of n_1 . To avoid this loss of information, the idea is to replace n_1 and n_2 by a common variable v , and to add a new constraint $v = x + z$. The new system is equivalent to the original one (both have the same solutions) while it improves the contraction power of **HC4**. The introduction of v (Fig. 2-right) amounts to adding a redundant equation $n_1 = n_2$ (Fig. 2-center). If one applies evaluation and narrowing phases of **HC4-revise** until the fixed-point on the new system, one will obtain the interval $[0.19, 4.14]$ for z , instead of $[-0.36, 4.58]$.

3.1 Additional Propagation

Proposition 1 underlines that **HC4** might obtain a better filtering when new auxiliary variables and equations corresponding to CSs are added in the system.

Proposition 1. *Let S be a NCSP and S' be the NCSP obtained by replacing in S one CS f in common between two expressions (belonging to constraints in S) by an auxiliary variable v , and by adding the new equation $v = f$. Then, **HC4** (with a floating-point precision) applied to S' produces a contracted box B' that is smaller than or equal to the box B produced by **HC4** applied to S .*

Proof. One first produces a system S_1 by replacing in S the first occurrence of f by an auxiliary variable v_1 and the second one by v_2 . We add the equations $v_1 = f$ and $v_2 = f$. Because **HC4-revise** works on acyclic graphs, **HC4** computes the 2B-consistency of the decomposed system (i.e., ternary system equivalent to S where all the operators are replaced by auxiliary variables). It is thus well-known that S and S_1 are equivalent: **HC4** applied to S_1 and **HC4** applied to S produce the same contracted box B [6]. Finally, creating S' amounts to adding the constraint $v_1 = v_2$ to S_1 . Thus, the box B' is smaller than or equal to B . $\square$

Of course, this result is useless if the box B' is equal to B , and we want to determine conditions for obtaining a box B' that might be *strictly* smaller than B . Among the set of *primitive operators* that are defined in a standard

implementation of HC4, the analysis presented below highlights that the following subset of non-monotonic or non-continuous operators might bring additional contraction when they occur several times (as CS) in the same system: $\sin(x)$, $\cos(x)$, $\tan(x)$ with non-monotonic domains, x^{2^c} (c positive integer and $0 \in X$), $\cosh(x)$ with $0 \in X$, $1/x$ with $0 \in X$ and binary operators $(+, -, \times, /)$.

3.2 Unary Operators

Let us first introduce some definitions. An *evaluation* function associated with a function f computes a *conservative* interval, i.e., the application of f on any tuple of values picked inside the input intervals falls inside the computed interval.

Definition 2. Let IR be the set of all the intervals over the reals. $F : IR \rightarrow IR$, $Y = [\underline{y}, \overline{y}] = F(X)$ is an **evaluation operator** associated with a unary primitive operator f if: $\forall x \in X, \exists y \in Y$ such that $f(x) = y$.

A *narrowing* operator N_F^x associated with a function f allows us to filter/contract the domain of a variable x .

Definition 3. Let X be the domain of a variable x , let F be an evaluation operator associated with f , and let Y be an interval. N_F^x is a **narrowing operator** of F on x , if $X' = [\underline{x}, \overline{x}] = N_F^x(Y)$ verifies:

$$f(\underline{x}) \in Y \wedge f(\overline{x}) \in Y \wedge \forall y \in Y, \forall x \in (X - X') : f(x) \neq y$$

Definition 4. Let f be a function defined on $I(f)$. f is a **monotonic function** on an interval X if: $\forall x_1, x_2 \in (X \cap I(f))^2, x_1 \leq x_2 : f(x_1) \leq f(x_2)$ or $\forall x_1, x_2 \in (X \cap I(f))^2, x_1 \leq x_2 : f(x_1) \geq f(x_2)$

As said above, a necessary condition to replace a CS is when the contraction obtained by a narrowing operator on a given expression f is partially lost in the next evaluation of f . More formally:

Condition 1. $\exists Y \subseteq F(X), X' = N_F^x(Y) : F(X') \not\subseteq Y$ where X is the domain of variable x , F is the evaluation operator associated with f , and N_F^x is the projection narrowing operator of F on x .

The following proposition indicates a simple condition to identify a *useless CS* for which no filtering is expected.

Proposition 2. Let F be the evaluation operator associated with a unary operator f . Let N_F^x be the narrowing operator of F on a variable x of domain X . If f is a monotonic and continuous function, then:

$$\forall Y \subseteq F(X), X' = N_F^x(Y) : F(X') \subseteq Y$$

Proof. WLOG we suppose that f is monotonically increasing. $X' = N_F^x(Y)$, then using Def. 3: $f(\underline{x}), f(\overline{x}) \in Y$, where $X' = [\underline{x}, \overline{x}]$. Finally, with Defs. 2 and 4, $F(X') = [f(\underline{x}), f(\overline{x})] \subseteq Y$. $\square$

Proposition 3. *With the same notations as above, if f is a non-monotonic function, then: $\exists Y \subseteq F(X), X' = N_F^x(Y) : F(X') \not\subseteq Y$*

Proof. The non monotonicity of f means:

$$\exists x_1, x_2, x_3 \subseteq X^3, x_1 \leq x_2 \leq x_3 \text{ s.t. } f(x_2) > f(x_1) \wedge f(x_2) > f(x_3)$$

Using values x_1, x_2 and x_3 that satisfy the existency condition, we can suppose that $Y = [f(x_1), f(x_3)]$. As $(f(x_2) > f(x_1)) \wedge (f(x_2) > f(x_3))$, $f(x_2) \notin Y$. $X' = N_F(Y)$, then, with Def. 1, $[x_1, x_3] \subseteq X'$. Since $x_1 \leq x_2 \leq x_3$, $x_2 \in X'$, with Def. 2, $f(x_2) \in F(X')$. Finally, $F(X') \not\subseteq Y$. $\square$

Example. Let $X = [-1, 3]$ be the domain of a variable x , and x^2 be an expression shared by two or more constraints. Suppose that in the narrowing phase of HC4-revise, the node corresponding to one of the expressions x^2 is contracted to: $Y = [3, 4]$. Applying the narrowing operator on x produces $X' = [-1, 2]$. In the next evaluation of the expression, $F(X') = [0, 4] \not\subseteq Y$.

Considering the standard operators managed in HC4 (except operators like floor), the *useful CSs* do not satisfy Proposition 2 and satisfy Proposition 3.

3.3 N-Ary Operators (Sums, Products)

For binary (n-ary) primitive functions, Condition 1 above can be extended to the following **Condition 2**:

$$\exists Z \subseteq F(X, Y), X' = N_F^x(Z, Y), Y' = N_F^y(Z, X) : F(X', Y') \not\subseteq Z$$

where X, Y are the domains of variables x and y respectively, F is the evaluation operator associated with f , N_F^x and N_F^y are the narrowing/projections operators on x and y resp. This condition 2 is generally satisfied by the n-ary operators $+$ and $\times$ (resp. $-$ and $/$). Many examples prove this result (see below). The result is due to intrinsic “bad” properties of interval arithmetics. First, the set of intervals IR is not a group for addition. That is, let I be an interval: $I - I \neq [0, 0]$ (in fact, $[0, 0] \subset I - I$). Second, $IR \setminus \{0\}$ is not a group for multiplication, i.e., $I/I \neq [1, 1]$.

The proposition 4 provides a *quantitative idea* of *how much* we can win when replacing additive CSs. It estimates the width Δ that is lost in binary sums (when an additive CS is not replaced by an auxiliary variable). Note that an upper bound of Δ is $2 \times \min(\text{Diam}(X), \text{Diam}(Y))$ and depends only on the initial domains of the variables.

Proposition 4. *Let $x + y$ be a sum related to a node n inside the tree representation of a constraint. The domains of x and y are the intervals X and Y resp. Suppose that HC4-revise is carried out on the constraint: in the evaluation phase, the interval of n is set to $V = X + Y$; in the narrowing phase, the interval V is contracted to $V_c = [\underline{V} + \alpha, \overline{V} - \beta]$ (with $\alpha, \beta \geq 0$ being the decrease in left and right bounds of V); X and Y are contracted to X_c and Y_c resp. The difference Δ between the diameter of V_c (current projection) and the diameter of the sum $X_c + Y_c$ (computed in the next evaluation) is:*

$$\Delta = \min(\alpha, \text{Diam}(X), \text{Diam}(Y), \text{Diam}(V) - \alpha) + \min(\beta, \text{Diam}(X), \text{Diam}(Y), \text{Diam}(V) - \beta)$$

Example. Consider $X = [0, 1]$ and $Y = [2, 4]$. Thus, $V = X + Y = [2, 5]$. Suppose that after applying HC4-revise we obtain $V_c = [2 + \alpha, 5 - \beta] = [4, 4]$ ($\alpha = 2$, $\beta = 1$). With Proposition 4, the narrowing operator yields $X_c = [0, 1]$ and $Y_c = [3, 4]$. Finally, $X_c + Y_c = [3, 5]$ is $\Delta = 2$ units larger than $V_c = [4, 4]$.

The properties related to multiplication are more difficult to establish. Concise results (not reported here) have been obtained only in the cases when 0 does not belong to the domains or when 0 is a bound of the domains.

4 The I-CSE Algorithm

The novelty of our algorithm I-CSE lies in the way additive and multiplicative CSs are taken into account.

First, I-CSE manages the commutativity and associativity of $+$ and $\times$ in a simple way thanks to *intersections* between expressions. An **intersection** between two sums (resp. multiplications) f_1 and f_2 produces the sum (resp. multiplication) of their common terms. For example: $+(x, \times(y, +(z, x^2)), \times(5, z)) \cap +(x^2, x, \times(5, z)) = +(x, \times(5, z))$. Consider two expressions $w_1 \times x \times y \times z_1$ and $w_2 \times y \times x \times z_2$ that share the CS $x \times y$. We are able to view these two expressions as $w_1 \times (x \times y) \times z_1$ and $w_2 \times (x \times y) \times z_2$ since $\times(w_1, x, y, z_1) \cap \times(w_2, y, x, z_2) = \times(x, y)$.

Second, contrarily to existing CSE algorithms, I-CSE handles *conflictive* subexpressions. Two CSs f_a and f_b included in f are **in conflict** (or **conflictive**) if $f_a \cap f_b \neq \emptyset$, $f_a \not\subseteq f_b$ and $f_b \not\subseteq f_a$. An example of conflictive CSs occurs in the expression $f : x \times y \times z$ that contains the conflictive CSs $f_a : x \times y$ and $f_b : y \times z$. Since $x \times y$ and $y \times z$ have a non empty intersection y , it is not possible to directly replace both f_a and f_b in f .

I-CSE works with the n-ary trees encoding the original equations¹ and produces a DAG. The roots of this DAG correspond to the initial equations; the leaves correspond to the variables and constants; every internal node f corresponds to an *operator* ($+$, $\times$, *sin*, *exp*, etc) applied to its *children* $t_1, t_2, \dots, t_n$. f represents the expression $f(t_1, t_2, \dots, t_n)$ and $t_1, \dots, t_n$ are the *terms* of the expression. *The CSs extracted by I-CSE are the nodes with several parents.*

We illustrate I-CSE with the following system made of two equations.

$$\begin{aligned} x^2 + y + (y + x^2 + y^3 - 1)^3 + x^3 &= 2 \\ \frac{(x^2 + y^3)(x^2 + \cos(y)) + 14}{x^2 + \cos(y)} &= 8 \end{aligned}$$

4.1 Step 1: DAG Generation

This step follows a standard algorithm that traverses simultaneously the n-ary trees corresponding to the equations in a bottom-up way (see e.g. [8]). By labelling nodes with identifiers, two nodes with common children and with the same operator are identified equivalent, i.e., they are CSs.

¹ The $+$ and $\times$ operators are viewed as **n-ary** operators. They include $-$ and $/$. For example, the 3-ary expression $x^2 y / (2 - x)$ is viewed as $*(x^2, y, 1/(2 - x))$.

4.2 Step 2: Pairwise Intersection between Sums and Products

Step 2 pairwise intersects, in any order, the nodes corresponding to n -ary sums on one hand, and to n -ary products on the other hand. This step creates *intersection nodes* corresponding to CSs. *Inclusion arcs* link their parents to intersection nodes. If the intersection expression is already present in the DAG, an inclusion arc is just added from each of the two intersected parents to this node.

For instance, on Fig. 3-a, the node 1.4 is obtained by intersecting the nodes 1 and 4, and we create inclusion arcs from the nodes 1 and 4 to the node 1.4. This means that 2 (i.e., y and x^2) among the 3 terms of the sum/node 4 are in common with 2 among the 4 terms of the sum/node 1. (Note that the two terms are in different orders in the intersected nodes.) The node 10 corresponds to the intersection between nodes 4 and 10 (in fact the node 10 is included in the node 4), but it has already been created at the first step.

Step 2 is a key step because it makes appear CSs modulo the commutativity and associativity of $+$ and $\times$ operators, and creates at most a quadratic number of CSs. By storing the maximal expressions obtained by intersection, intersection nodes and inclusion arcs enable I-CSE to compute all the CSs before adding them in the DAG in the next step².

4.3 Step 3: Integrating Intersection Nodes into the DAG

In this step, all the intersection nodes are integrated into the DAG, creating the definitive DAG. The routine is top-down and follows the inclusion arcs. Every node f is processed to incorporate into the DAG its “children” reached by an inclusion arc.

If f has no conflictive child, the inclusion arcs outgoing from f are transformed into plain arcs. Also, to preserve the equivalence between the DAG and the system of equations, one removes arcs from f to the children of its intersection terms/children. For instance, on Fig. 3-b, Step 3 modifies the inclusion arc $1 \rightarrow 1.4$ and removes the arcs $1 \rightarrow 12$ and $1 \rightarrow y$.

Otherwise, f has children/CSs in conflict, i.e., there exist at least two CSs f_a and f_b , included in f , such that $f_a \cap f_b \neq \emptyset$, $f_a \not\subseteq f_b$ and $f_b \not\subseteq f_a$. In this case, one or several nodes $(f_1, f_2, \dots, f_r)$ equivalent to f are added such that: the set $f, f_1, f_2, \dots, f_r$ cover all the CSs included in f . To maintain the DAG equivalent to the original system, a node *equal* is created with the children: $f, f_1, f_2, \dots, f_r$. For example on Fig. 3, the node 4 associated with the expression $y + x^2 + y^3 - 1$ has two CSs in conflict. Step 3 creates a node 4b (attaching the conflictive CS $x^2 + y^3$) redundant to the node 4 (attaching the other conflictive CS $y + x^2$).

A greedy algorithm has been designed to generate a small number r of redundant nodes, with a small number of children. r is necessarily smaller than the number of CSs included in f . We illustrate our greedy algorithm handling conflictive CSs on a more complicated example, in which an expression (node) $u = s + t + x + y + z$ contains 3 CSs in conflict: $v_1 = s + t$, $v_2 = t + x$, $v_3 = y + z$

² If one did not want to manage conflictive expressions, one would incorporate directly, in Step 2, the new CSs into the DAG.

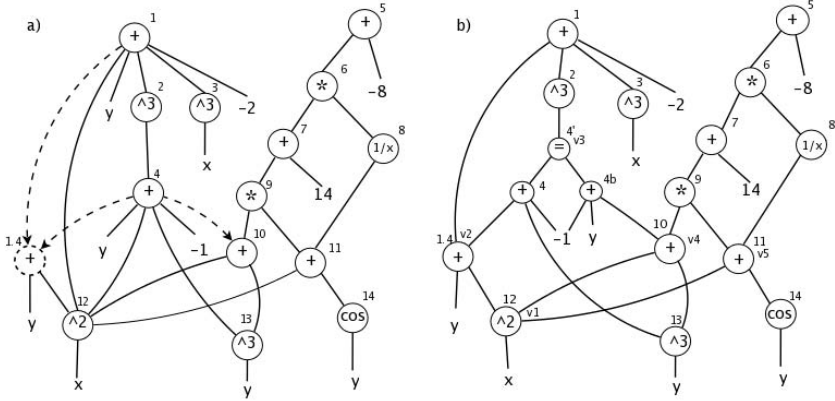


Fig. 3. (a) DAG obtained after the first two steps of I-CSE. **Step 1:** the system is transformed into a DAG including all the nodes, excepting 1.4, together with the arcs in plain lines. (For the sake of clarity, we have not merged the variables with multiple occurrences.) **Step 2:** the $\times$ and $+$ nodes are pairwise intersected, resulting in the creation of the node 1.4 and the three *inclusion arcs* in dotted lines. (b) DAG obtained after **Step 3:** all the inclusion arcs have been integrated into the DAG. For the conflictive subexpressions (nodes 1.4 and 10), a redundant node 4b and an equality node 4' have been created. The node 1.4 is attached to 4 whereas the node 10 is attached to 4b. **Step 4** generates the auxiliary variables corresponding to the useful CSs ($v1, v2, v4$ and $v5$) and to the equality nodes ($v3$).

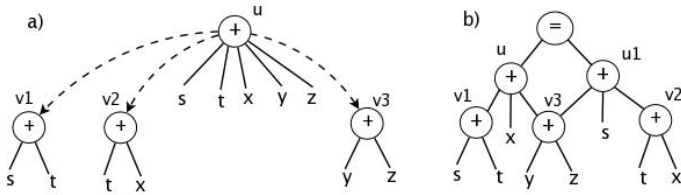


Fig. 4. Integrating intersection nodes into the DAG. a) The node u has three CSs in conflict. b) The DAG, with an equality node, obtained by the greedy algorithm.

(Fig. 4-a). The greedy algorithm works in two phases. In the first phase, several occurrences of u are generated until all the CSs are replaced. On the example, $u = v_1 + x + v_3$ and $u_1 = s + v_2 + y + z$ are created. The second phase handles all the redundant equations that have been created in the first phase. In a greedy way, it tries to introduce CSs into every equation to obtain a shorter equation that improves filtering. On the example, it transforms $u_1 = s + v_2 + y + z$ into $u_1 = s + v_2 + v_3$. Finally, an equality node ($=$) is associated with the node u and the redundant node u_1 (Fig. 4-b).

4.4 Step 4: Generation of the New System

A first way to exploit CSs for solving an NCSP is to use the DAG obtained after Step 3. As shown by Vu et al. in [20], the propagation phase cannot still be carried out by a pure HC4, and a more sophisticated propagation algorithm must consider the unique DAG corresponding to the whole system.

Alternatively, in order to still be able to use HC4 for propagation, and thus to be compatible with existing interval-based solvers, Step 4 generates a new system of equations in which an auxiliary variable v and an equation $v = f$ are added for every *useful* CS. Avoiding the creation of new equations for useless CSs, which cannot provide additional contraction, decreases the size of the new system. In addition, redundant expressions $(f, f_1, f_2, \dots, f_r)$ linked by an equality node, add a new auxiliary variable v' and the equations $v' = f, v' = f_1, \dots, v' = f_r$. To achieve these tasks, Step 4 traverses the DAG bottom-up and generates variables and equations in every node.

Finally, the new system will be composed by the modified equations (in which the CSs are replaced by their corresponding auxiliary variable), by the auxiliary variables and by the new constraints $v = f$ corresponding to CSs. The new system corresponding to the example in Fig. 3 is the following:

$$\begin{array}{lll} v_2 + (v_3)^3 + x^3 - 2 = 0 & v_1 = x^2 & v_3 = -1 + y + v_4 \\ \frac{v_4 \times v_5 + 14}{v_5} - 8 = 0 & v_2 = y + v_1 & v_4 = v_1 + y^3 \\ & v_3 = v_2 + y^3 - 1 & v_5 = v_1 + \cos(y) \end{array}$$

For a given system of equations, our interval-based solver manages two systems: the new system generated by I-CSE is used only for HC4 and the original system is used for the other operations (bisections, interval Newton). The intervals in both systems must be synchronized during the search of solutions. First, this allows us to clearly validate the interest of I-CSE for HC4. Second, carrying out Newton or bisection steps on auxiliary variables would need to be validated both in theory and in practice. Finally, this implementation is similar to the DAG-based solving algorithm proposed by Vu et al. which also considers only the initial variables for bisections and interval Newton computations, the internal nodes corresponding to CSs being only used for propagation [20].

4.5 Time Complexity

The time complexity of I-CSE mainly depends on the number n of variables, on the number k of a -ary operators and on the maximum arity a of an a -ary sum or multiplication expression in the system. $k + n$ is the size of the DAG created in Step 1, so that the time complexity of Step 1 is $O(k + n)$ on average if the identifiers are maintained using hashing. In Step 2, the number i of intersections performed is quadratic in the number of sums (or products) in the DAG, i.e., $i = O(k^2)$. Every intersection requires $O(a)$ on average using hashing (a worst-case complexity $O(a \log(a))$ can be reached with sets encoded by trees/heaps). The worst-case for Step 3 depends on the maximum number of inclusion arcs which

Table 1. Time complexity of I-CSE on three representative scalable systems of equations (see Section 6). The CPU times have been obtained with a processor Intel 2.40 GHz. The CPU time increases linearly in the size $k + n$ of the DAG for **Trigexp1** and **Katsura** while the time complexity for **Brown** reaches the worst-case one.

Benchmark	Trigexp1			Katsura			Brown		
Number n of variables	10	20	40	5	10	20	10	20	40
Number k of operators	46	96	196	15	55	208	10	20	40
I-CSE time in second	0.19	0.28	0.63	0.08	0.19	0.91	0.05	0.20	1.26

is $O(k^2)$. Step 4 is linear in the size of the final DAG and is $O(k + n + i)$. Overall, I-CSE is thus $O(n + a \log(a) k^2)$. Table 1 illustrates how the time complexity evolves in practice with the size of the system.

5 Implementation of I-CSE

I-CSE has been implemented using Mathematica version 6. Mathematica first automatically transforms the equations into a canonical form, where additions and multiplications are n-ary and where are performed reductions, i.e., factorizations by a constant. For instance, the expression $2x - y + x + z$ is transformed into $+(\times(3, x), -y, z)$. The n-ary representation of equations is useful for the pairwise intersections of I-CSE (Step 2).

The solving algorithms are developed in the open source interval-based library in C++ called **Ibex** [5]. A given benchmark is solved by a branch and prune process: the variables are bisected in a round-robin manner and contracted by constraint propagation (HC4 only, or 3BCID using HC4 – 3BCID is a variant of 3B [19]) and interval Newton. As mentioned above, **Ibex** offers facilities to create two systems of equations in memory for which domains of variables are synchronized during the search of solutions.

I-CSE-B and I-CSE-NC

We have proven theoretically that the interest of I-CSE resides in the additional pruning it permits and not only in a decrease of the number of operations. To confirm in practice this significant result, we have designed two variants of I-CSE that compute fewer CSs. I-CSE-B (Basic I-CSE) simply ignores the step 2 of I-CSE. The commutativity and associativity of $+$ and $\times$ are not taken into account. Additive and multiplicative n-ary expressions are considered in a fixed binary form in which only a few subexpressions can be detected. For instance, the CS $x + y$ is detected in two expressions $x + y + z_1$ and $x + y + z_2$, but not in expressions $x + z_1 + y$ and $x + z_2 + y$.

I-CSE-NC (I-CSE with No Conflicts) completely exploits the commutativity and associativity of $+$ and $\times$, but does not take into account conflictive CSs. I-CSE-NC lowers the worst case time complexity of I-CSE, but does not replace all the CSs. If a given system does not contain CSs in conflict, I-CSE and I-CSE-NC

return the same new system (with no redundant equations). In the example, I-CSE-NC does not create the redundant equation $v_3 = y + v_7$, so that the equation $v_7 = v_9 + y^3$ is not created either. The second (initial) equation finally becomes: $\frac{(x^2+y^3) \times v_8 + 14}{v_8} = 8$.

Existing CSE algorithms take place between I-CSE-B and I-CSE-NC in terms of number of detected useful CSs. We assume here that the algorithm by Vu et al. [20] is similar to I-CSE-NC.

6 Experiments

Benchmarks have been taken in the first two sections (polynomial and non-polynomial systems) of the COPRIN page³. The selected sample fulfills systematic criteria: every tested benchmark is an NCSP with a finite number of isolated solutions (no optimization); all the solutions can be found by the ALIAS system [14] in a time comprised between one second and one hour; selected systems are written with the following primitive operators: `+`, `-`, `×`, `/`, `sin`, `cos`, `tan`, `exp`, `log`, `power`. With these criteria, we have selected 40 benchmarks. The I-CSE algorithm detects no CS in 16 of them. There are also two more benchmarks (*Fourbar* and *Dipole2*) for which no test has finished before the timeout (one hour), providing no indication. 9 of the remaining 22 benchmarks are scalable, that is, can be defined with any number of variables. Table 2 provides information about the selected benchmarks. When there is no conflictive CS, I-CSE and I-CSE-NC return the same new system and there is no redundant constraints (`#rc=0`). The interval-based solver results will be the same.

For all the benchmarks, the CPU time required by I-CSE (and variants) is often negligible and always less than 1 second.

Remark. In the benchmarks marked with a star (*), the equations have not been initially rewritten into the canonical form by *Mathematica* (see Section 5). This leads to fewer CSs, but these CSs correspond to larger subexpressions shared by more expressions, providing generally better results.

Tables 3 and 4 compare the CPU times required by *Ibex* to solve the initial system (Init) and the systems generated by I-CSE-B, I-CSE-NC and I-CSE. Table 3 reports results obtained by a standard branch and prune approach with bisection, Newton and **HC4**. Table 4 reports results obtained by a branch and prune approach with bisection, Newton and **3BCID** (using **HC4** as a refutation algorithm). Both tables report CPU times in seconds obtained on a 2.40 GHz Intel Core 2 processor with 1 Gb of RAM, and the corresponding gain w.r.t. the solving of the original system. The time limit has been set to 3600 seconds. The tables also report the number of generated boxes (`#Boxes`) during the search. This corresponds to the number of nodes in the tree search and highlights the additional pruning due to I-CSE. The precision of solutions has been set to 10^{-8} for all the benchmarks. The parameter used by **HC4** has been set to 1% in Table 3. The parameters used by **HC4** and **3BCID** have been set to

³ www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/benches.html

Table 2. Selected benchmarks. The columns yield the name of the benchmark, the number of solutions ($\#s$), the number of variables (n), the number of useful CSs ($\#cs$) found by I-CSE-B, I-CSE-NC, I-CSE, the number of redundant constraints created by I-CSE due to conflictive CSs ($\#rc$).

Benchmark	$\#s$	n	I-CSE-B $\#cs$	ICSE-NC $\#cs$	I-CSE $\#cs$	$\#rc$	Benchmark	$\#s$	n	I-CSE-B $\#cs$	ICSE-NC $\#cs$	I-CSE $\#cs$	$\#rc$
6body	5	6	2	3	3	0	Katsura-20	7	21	90	90	90	0
Bellido	8	9	0	1	1	0	Kin1	16	6	13	13	19	3
Brown-7	3	7	3	7	21	24	Pramanik	2	8	0	15	15	0
Brown-7*	3	7	3	1	1	0	Prolog	0	21	0	7	7	0
Brown-30	2	30	26	53	435	783	Rose	16	3	5	5	5	0
BroyBand-20	1	20	22	37	97	73	Trigexp1-30	1	30	29	29	29	0
BroyBand-100	1	100	102	119	479	473	Trigexp1-50	1	50	49	49	49	0
Caprasse	18	4	6	7	11	2	Trigexp2-11	0	11	15	15	15	0
Design	1	9	3	3	3	0	Trigexp2-19	0	19	27	27	27	0
Dis-Integral-6	1	6	4	6	18	9	Trigonom-5	2	5	7	9	20	14
Dis-Integral-20	3	20	18	34	207	171	Trigonom-5*	2	5	7	6	6	0
Eco9	16	8	0	3	7	1	Trigonom-10	24	10	15	15	26	15
EqCombustion	4	5	7	8	11	1	Trigonom-10*	24	10	15	12	12	0
ExtendWood-4	3	4	2	2	2	0	Yamamura-8	7	8	5	10	36	48
Geneig	10	6	11	14	14	0	Yamamura-8*	7	8	5	1	1	0
Hayes	1	8	9	8	8	0	Yamamura-12	9	12	9	18	78	119
I5	30	10	3	4	10	5	Yamamura-12*	9	12	9	1	1	0
Katsura-19	5	20	81	81	81	0	Yamamura-16	9	16	13	26	136	224

Table 3. Results obtained with HC4 and interval Newton

Benchmark	Time in second				Time(Init) / Time			#Boxes		
	Init	ICSE-B	ICSE-NC	I-CSE	ICSE-B	ICSE-NC	I-CSE	Init	ICSE-NC	I-CSE
EqCombustion	>3600	26.1	0.35	0.14	>137	>10000	> 25000	>1e+08	3967	1095
Rose	>3600	500	101	101	>7.2	> 35	> 35	>3e+07	865099	865099
Hayes	141	51.9	15.7	15.7	2.7	9	9	550489	44563	44563
6-body	0.22	0.07	0.07	0.07	3.1	3.1	3.1	4985	495	495
Design	176	65.2	63.2	63.2	2.7	2.8	2.8	425153	122851	122851
I5	>3600	>3600	1534	1565	?	> 2.3	>2.3	>3e+07	7e+06	7e+06
Geneig	3323	2910	2722	2722	1.14	1.22	1.22	7e+08	4e+08	4e+08
Kin1	8.52	8.32	8.32	8.01	1.02	1.02	1.06	905	909	905
Pramanik	89.3	92.1	84.9	84.9	0.97	1.05	1.05	487255	378879	378879
Bellido	15.7	15.9	15.6	15.6	0.99	1.01	1.01	29759	29319	29319
Eco9	23.9	23.9	24	24.1	1.00	1.00	0.99	126047	117075	110885
Caprasse	1.56	1.81	1.68	2.16	0.86	0.93	0.72	8521	7793	7491
Brown-7*	500	350	0.01	0.01	1.42	49500	49500	6e+06	95	95
Dis-Integral-6	201	0.46	1.3	0.03	437	155	6700	653035	4157	47
ExtendWood-4	29.9	0.03	0.03	0.03	997	997	997	422705	353	353
Brown-7	500	350	30.7	1.49	1.42	16.1	332	6e+06	258601	3681
Trigexp2-11	1118	208	56.2	56.2	5.38	19.9	19.9	1e+06	316049	316049
Yamamura-8*	13	13.3	0.75	0.75	0.98	17.3	17.3	29615	2161	2161
Broy-Banded-20	778	759	261	58.1	1.03	2.98	13.4	172959	46761	12623
Trigonometric-5*	15.8	12.3	1.49	1.49	1.28	10.6	10.6	10531	1503	1503
Trigonometric-5	15.8	12.3	8.94	6.97	1.28	1.77	2.27	10531	7369	5307
Yamamura-8	13	13.3	44.6	10.8	0.98	0.3	1.20	29615	115211	13211
Katsura-19	1430	1583	1583	1583	0.90	0.90	0.90	145839	153193	153193
Trigexp1-30	2465	3244	3244	3244	0.76	0.76	0.76	1e+07	1e+07	1e+07

10% in Table 4. We have put at the end of both tables the results corresponding to scalable benchmarks. To return a fair comparison between algorithms, we have selected for the scalable systems the instance with the largest number of variables n such that the solver on the original system finds the solutions in less

Table 4. Results obtained with 3BCID using HC4 and interval Newton

Benchmark	Time in second				Time(Init) / Time			#Boxes		
	Init	ICSE-B	ICSE-NC	I-CSE	ICSE-B	ICSE-NC	I-CSE	Init	ICSE-NC	I-CSE
Rose	2882	5.17	4.04	4.04	557	713	713	4e+06	5711	5711
Prolog	38.5	60	0.14	0.14	0.64	275	275	4647	11	11
EqCombustion	0.42	0.37	0.06	0.06	1.35	7	7	427	23	23
Hayes	32.6	27.2	5.67	5.67	1.13	5.7	5.7	17455	1675	1675
Design	52	17.9	13.3	13.3	2.9	3.9	3.9	16359	4401	4401
I5	33.5	41.1	17.9	17.8	0.81	1.9	1.9	10619	4387	4281
6-body	0.14	0.08	0.1	0.1	1.75	1.4	1.4	173	51	51
Kin1	1.66	2.66	1.76	1.23	0.62	0.94	1.35	85	161	197
Bellido	10.3	10.4	9.98	9.98	1	1.03	1.03	4487	4341	4341
Eco9	11.6	11.6	12.4	13.2	1	0.94	0.88	6205	6045	5749
Pramanik	73.8	114	96.8	96.8	0.65	0.76	0.76	124663	95305	95305
Caprasse	1.96	2.51	2.5	2.92	0.74	0.78	0.67	1285	1311	1219
Geneig	696	1050	1050	1050	0.66	0.66	0.66	362225	362045	362045
Trigexp2-19	2308	2.23	0.03	0.03	1035	77000	77000	250178	7	7
Brown-7*	600	318	0.01	0.01	1.88	60000	60000	662415	9	9
ExtendWood-4	185	0.03	0.03	0.03	6167	6167	6167	669485	35	35
Dis-Integral-6	135	0.18	0.51	0.03	750	264	4500	86487	185	7
Brown-7	600	318	4.75	0.22	1.88	126	2700	662415	2035	23
Yamamura-12*	1751	1842	1.01	1.01	0.95	1700	1700	364105	307	307
Yamamura-12	1751	1842	31.1	8.72	0.95	56.3	200	364105	5647	445
Trigonometric-10*	1344	506	19.4	19.4	2.67	69	69	140512	2033	2033
Trigonometric-10	1344	506	156	49.6	2.67	8.62	27	140512	19883	3339
Broy-Banded-100	9.96	20.3	14.8	8.21	0.49	0.67	1.21	13	23	11
Trigexp1-50	0.15	0.19	0.17	0.17	0.79	0.88	0.88	1	1	1
Katsura20	3457	5919	5919	5919	0.58	0.58	0.58	62451	120929	120929
Brown-30	>3600	>3600	>3600	22.9	?	?	>150	>210021	>151527	31
Dis-Integral-20	>3600	>3600	>3600	1.12	?	?	>3200	>111512	>75640	39
Yamamura-16	>3600	>3600	681	35.6	?	>5	>100	>522300	96341	919

than one hour. This number n is greater with 3BCID (Table 4) than with only HC4 (Table 3) because 3BCID is generally more efficient than HC4.

Tables 3 and 4 clearly highlight that I-CSE is very interesting in practice. We observe a gain in performance greater than a factor 2 on 15 among the 24 lines (on both tables). The gain is of two orders of magnitude (or more) for 5 benchmarks with HC4 (corresponding to 4 different systems) and for 10 benchmarks with 3BCID (corresponding to 8 different systems).

I-CSE clearly outperforms the variants extracting fewer useful CSs, as shown on Table 3 (see Brown-7, Dis-Integral-6, Broyden-Banded-20) and Table 4 (see Brown-7, Dis-Integral-6, Yamamura-12, Trigonometric-10). In these cases, the gains in CPU time are significant. They are sometimes of several orders of magnitude. The few exceptions for which I-CSE is worse than its simpler variants give only a slight advantage to I-CSE-NC or I-CSE-B.

The number of boxes is generally decreasing from the left to the right of tables. This confirms our theoretical analysis that expects gains in filtering when a system has additional equations due to CSs. This experimentally proves that exploiting conflictive CSs is useful. This confirms an intuition shared by a lot of practitioners of partial consistency algorithms that redundant constraints are often useful because they allow a better pruning effect [10]. Benchmarks like Brown-30, Dis-Integral-20 and Yamamura-16, have been added at the end of Table 4 to highlight this trend: I-CSE produces a gain in performance of 3 orders of magnitude while it adds hundreds of redundant equations.

Most of the obtained results are good or very good, but four benchmarks observe a loss of performance lying between 20% and 42%: **Caprasse** with both strategies, and **Pramanik**, **Geneig**, **Katsura** with **3BCID**. The loss in performance observed for **Katsura-20** (10% or 42% according to the strategy) is due to the domains of the variables that are initialized to $[0,1]$. Without detailing, such domains imply that the pruning in the search tree is due to the evaluation (bottom-up) phase and not to the (top-down) narrowing phase of **HC4-revise**.

7 Conclusion

This paper has presented the algorithm **I-CSE** for exploiting common subexpressions in numerical CSPs. A theoretical analysis has shown that gains in filtering can only be expected when CSs do not correspond to monotonic and continuous operators like x^3 or $\log$. Contrarily to a belief in the community, this means that CSs can bring significant gains in filtering/contraction, and not only a decrease in the number of operations. These are good news for the significance of this line of research.

Experiments have been performed on 40 benchmarks among which 24 contain CSs. Significant gains of one or several orders of magnitude have been observed on 10 of them. **I-CSE** differs from existing CSEs in that it also detects conflictive CSs. As compared to **I-CSE-NC** (similar to existing CSEs), the additional contraction involved by the corresponding redundant equations leads to improvements of one or several orders of magnitude on 4 benchmarks (**Brown**, **Dis-Integral**, **Yamamura** and, only for **HC4**, **BroyBanded**).

A future work is to compare our implementation based on the standard **HC4** algorithm (and the management of two systems), with the sophisticated propagation algorithm carried out on the elegant DAG-based structure proposed by Vu, Schichl and Sam-Haroud. However, our experimental results have underlined that the gain in contraction has a greater impact on efficiency than the time required to reach the fixed-point of propagation. Thus, we suspect that both implementations will show similar performances.

References

1. Benhamou, F., Goualard, F., Granvilliers, L., Puget, J.-F.: Revising Hull and Box Consistency. In: Proc. ICLP, pp. 230–244 (1999)
2. Brown, D.P.: Calculus and Mathematica. Addison Wesley, Reading (1991)
3. Buchberger, B.: Gröbner Bases: an Algorithmic Method in Polynomial Ideal Theory. Multidimensional Systems Theory, 184–232 (1985)
4. Ceberio, M., Granvilliers, L.: Solving Nonlinear Equations by Abstraction, Gaussian Elimination, and Interval Methods. In: Armando, A. (ed.) FroCos 2002. LNCS (LNAI), vol. 2309. Springer, Heidelberg (2002)
5. Chabert, G. (2008), <http://ibex-lib.org>
6. Collavizza, H., Delobel, F., Rueher, M.: Comparing partial consistencies. Reliable Computing 5(3), 213–228 (1999)
7. Debruyne, R., Bessière, C.: Some Practicable Filtering Techniques for the Constraint Satisfaction Problem. In: Proc. IJCAI, pp. 412–417 (1997)

8. Flajolet, P., Sipala, P., Steyaert, J.-M.: Analytic variations on the common subexpression problem. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 220–334. Springer, Heidelberg (1990)
9. Granvilliers, L., Monfroy, E., Benhamou, F.: Symbolic-Interval Cooperation in Constraint Programming. In: Proc. ISSAC, pp. 150–166. ACM, New York (2001)
10. Harvey, W., Stuckey, P.J.: Improving Linear Constraint Propagation by Changing Constraint Representation. *Constraints* 7, 173–207 (2003)
11. Heck, A.: Introduction to Maple. Springer, Heidelberg (2003)
12. Lebbah, Y.: Contribution à la Résolution de Contraintes par Consistance Forte. Phd thesis, Université de Nantes (1999)
13. Lhomme, O.: Consistency Tech. for Numeric CSPs. In: IJCAI, pp. 232–238 (1993)
14. Merlet, J.-P.: ALIAS: An Algorithms Library for Interval Analysis for Equation Systems. Technical report, INRIA Sophia (2000), <http://www-sop.inria.fr/coprin/logiciels/ALIAS/ALIAS.html>
15. Merlet, J.-P.: Interval Analysis and Robotics. In: Symp. of Robotics Research (2007)
16. Muchnick, S.: Advanced Compiler Design and Implem. M. Kauffmann (1997)
17. Neumaier, A.: Interval Methods for Systems of Equations. Cambridge University Press, Cambridge (1990)
18. Schichl, H., Neumaier, A.: Interval analysis on directed acyclic graphs for global optimization. *Journal of Global Optimization* 33(4), 541–562 (2005)
19. Trombettoni, G., Chabert, G.: Constructive Interval Disjunction. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 635–650. Springer, Heidelberg (2007)
20. Vu, X.-H., Schichl, H., Sam-Haroud, D.: Using Directed Acyclic Graphs to Coordinate Propagation and Search for Numerical Constraint Satisfaction Problems. In: Proc. ICTAI 2004, pp. 72–81. IEEE, Los Alamitos (2004)

A Soft Constraint of Equality: Complexity and Approximability^{*}

Emmanuel Hebrard, Barry O’Sullivan, and Igor Razgon

Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{e.hebrard,b.osullivan,i.razgon}@4c.ucc.ie

Abstract. We introduce the `SOFTALLEQUAL` global constraint, which maximizes the number of equalities holding between pairs of assignments to a set of variables. We study the computational complexity of propagating this constraint, showing that it is intractable in general, since maximizing the number of pairs of equally assigned variables in a set is NP-hard. We propose three ways of coping with NP-hardness. Firstly, we develop a greedy linear-time algorithm to approximate the maximum number of equalities within a factor of 2. Secondly, we identify a tractable (polynomial) class for this constraint. Thirdly, we identify a parameter based on this class and show that the `SOFTALLEQUAL` constraint is fixed-parameter tractable with respect to this parameter.

1 Introduction

Constraints for reasoning on the number of differences within a set of variables are ubiquitous in constraint programming. One of the most commonly used global constraints is the `ALLDIFFERENT` constraint [11], which enforces that *all* variables take pair-wise different values. Petit et al. have introduced a soft version of the `ALLDIFFERENT` constraint, `SOFTALLDIFF` [10]. They proposed two types of costs, which are to be minimized: *graph*- and *variable*-based costs (see Definitions 1 and 2). The former counts the number of equalities, whilst the latter counts the number of variables violating an `ALLDIFFERENT` constraint. The algorithms for filtering these two constraints, introduced in the same paper, were then improved by Hoeve et al. [15]. In both cases the constraint can be represented as a flow problem, leading to polynomial time algorithms for achieving generalised arc consistency.

Another closely related constraint dealing with equalities between variables, `ATMOSTNVALUE`, received some attention recently. Achieving bounds consistency on this constraint can be done in polynomial time [2] whilst achieving GAC is NP-hard [3]. This latter constraint ensures that no more than k distinct values are assigned to a set of n variables. It is therefore the dual of

^{*} This work was supported by Science Foundation Ireland (Grant Number 05/IN/I886). Hebrard is also supported by an Embark Initiative (IRCSET) Post-doctoral Fellowship.

Table 1. Complexity of optimizing inequalities

	Minimizing Cost	Maximizing Cost
Variable Cost	$\mathcal{O}(n\sqrt{m})$	NP-hard
Graph Cost	$\mathcal{O}(nm)$	?

SOFTALLDIFF for the variable-based cost, i.e. an assignment with k distinct values among n variables violates ALLDIFFERENT on $n - k$ variables. To impose a cost of k for ATMOSTNVALUE is thus equivalent to imposing a cost of $n - k$ for SOFTALLDIFF.

The complexities of minimizing both variable- and graph-based costs, as well as maximizing the variable-based cost, for SOFTALLDIFF are known. However, the complexity of maximizing the graph-based cost (i.e. maximizing the number of pairs of variables assigned with the same value) is still an open problem. Table 1 summarizes the known results for these constraints. Interestingly, whereas minimizing this cost can be mapped to a flow problem and therefore solved in polynomial time, maximizing it does not correspond in a straightforward way to any known problem. In this paper, we fill this gap by providing a number of algorithmic results on the problem of maximizing the number of pair-wise equalities amongst a set of variables. We first show that this problem is NP-hard in general, then we introduce an approximation algorithm, a tractable class and a fixed parameter tractable algorithm.

We call SOFTALLEQUAL_G the global constraint defined with the same *graph-based* cost as SOFTALLDIFF, albeit where this cost is to be maximized instead of minimized. This constraint has many applications. For instance, consider the problem of scheduling a number of meetings so that every person attends exactly one meeting, and the number of interactions is to be maximized. Each person is assigned to a timeslot, and two people interact only if they attend the same meeting, i.e., are assigned the same value. This can be modelled using a single SOFTALLEQUAL_G constraint. As another example, consider a map coloring problem where we want each continent to be colored as homogeneously as possible. One could post, besides inequalities corresponding to borders, as many SOFTALLEQUAL_G constraints as continents, ensuring that whilst neighboring countries are distinguishable, continents also appear as entities.

However, the original motivation for this work comes from our desire to formulate the problem of finding sets of similar and diverse solutions to CSPs as a constraint optimization problem. Similarity and diversity play fundamental roles in theories of knowledge and behaviour [14]. Reasoning about the distance between solutions is an important problem in artificial intelligence [1,4,7,8,13]. For example, in belief update one might wish to minimize Hamming distance between states [5], in case-based reasoning one often seeks solutions to similar problems while achieving diversity amongst the alternatives [13], in preference-based search one may express preferences in terms of a set of ideal or non-ideal solutions [8]. For instance, let $\mathcal{P}_1, \dots, \mathcal{P}_m$, be m CSPs with the same number of variables n . In [7] the *diversity* of a set of solutions was defined as the sum

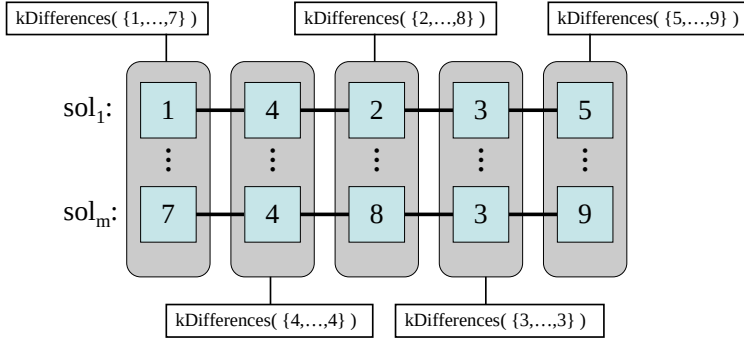


Fig. 1. Finding diverse solutions as a CSP

of the Hamming distances between each pair of solutions. Let $kDifferences(M)$ be the number of pairs of distinct elements in the multiset M . The previously defined diversity of a set of solutions $\{sol_1, \dots, sol_m\}$ is equivalent to the sum of $kDifferences(\{sol_1[i], \dots, sol_m[i]\})$ over all indices $1 \leq i \leq n$, as illustrated in Figure 1. When either minimizing or maximizing this sum, achieving GAC on each number of differences ($kDifferences$) is enough to obtain GAC on the whole, since the hypergraph is Berge-acyclic. The complexity of computing lower and upper bounds for the number of differences on variables is, therefore, key to this problem.

Our contribution in this paper is to present an indepth study of the complexity of SOFTALLEQUAL_G . While achieving generalized arc consistency on the SOFTALLDIFF constraint is known to be polynomial, the complexity of filtering the SOFTALLEQUAL_G constraint is more intriguing. In Section 3, we show that SOFTALLEQUAL_G is intractable in general, since maximizing the number of pairs of equally assigned variables in a set is NP-complete. We propose three ways of coping with NP-hardness. Firstly, in Section 4, we show that a natural greedy algorithm approximates the maximum number of equalities within a factor of 2, and that its complexity can be brought down to linear time. Secondly, in Section 5, we identify a polynomial class for this constraint. Thirdly, in Section 6, we identify a parameter based on this class and show that the SOFTALLEQUAL_G constraint is fixed-parameter tractable with respect to this parameter.

2 Formal Background

Constraint Satisfaction. A constraint satisfaction problem (CSP) is a triplet $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{X}$ is a set of variables, $\mathcal{D}$ a mapping of variables to sets of values and $\mathcal{C}$ a set of constraints that specify allowed combinations of values for subsets of variables. A constraint $C \in \mathcal{C}$ is *generalized arc consistent* (GAC) iff, when a variable in the scope of C is assigned any value, there exists an assignment of the other variables in C such that C is satisfied. This satisfying assignment is called a *support* for the value. Given a CSP $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, we shall

use the following notation throughout the paper: n shall denote the number of variables, i.e., $n = |\mathcal{X}|$; m shall denote the number of distinct unary assignments, i.e., $m = \sum_{X \in \mathcal{X}} |\mathcal{D}(X)|$; λ shall denote the total number of distinct values, i.e., $\lambda = |\bigcup_{X \in \mathcal{X}} \mathcal{D}(X)|$.

Soft Global Constraints. Adding a cost variable to a constraint to represent its degree of violation is now common practice in constraint programming. This model was introduced in [12]. It offers the advantage of unifying hard and soft constraints since generalized arc consistency, along with other types of consistencies, can be applied to such constraints with no extra effort. As a consequence, classical constraint solvers can solve over-constrained problems modelled in this way without modification. This approach was refined and applied to a number of other constraints in [15].

Two natural cost measures have been explored for the ALLDIFFERENT and for a number of other constraints. The *variable-based cost* counts how many variables need to change in order to obtain a valid assignment of the hard constraint. The *graph-based cost* counts how many times a component of a decomposition of the constraint is violated. Typically these components correspond to edges of a decomposition graph, e.g. for an ALLDIFFERENT constraint, the decomposition graph is a clique and an edge is violated if and only if both variables connected by this edge share the same value (see Definitions 1 and 2). The SOFTALLDIFF constraint was, thus, given the following definitions in [10]:

Definition 1 (SOFTALLDIFF_V – variable-based cost)

$$\text{SOFTALLDIFF}_V(\{X_1, \dots, X_n\}, N) \Leftrightarrow N \geq n - |\{v \mid X_i = v\}|.$$

Definition 2 (SOFTALLDIFF_G – graph-based cost)

$$\text{SOFTALLDIFF}_G(\{X_1, \dots, X_n\}, N) \Leftrightarrow N \geq |\{\{i, j\} \mid X_i = X_j \text{ \& } i \neq j\}|.$$

Consider each of the violation costs for the following two solutions of a CSP involving four variables $X_1, \dots, X_4$ each with domain $\{a, b\}$:

$$S_1 : X_1 = a, X_2 = b, X_3 = a, X_4 = b$$

$$S_2 : X_1 = a, X_2 = b, X_3 = b, X_4 = b$$

In both solutions, at least two variables need to change (for example X_3 and X_4) to obtain a valid solution. Therefore, the variable-based cost is two for S_1 and S_2 . However, in S_1 only two edges are violated $\{X_1, X_3\}$ and $\{X_2, X_4\}$ whilst in S_2 , three edges are violated $\{X_2, X_3\}$, $\{X_2, X_4\}$ and $\{X_3, X_4\}$. Therefore, the graph-based cost of S_1 is two whereas it is three for S_2 .

Parameterized Complexity. We shall use the notion of parameterized complexity in the last section of this paper. For a comprehensive introduction the reader is referred to [9]. Given a problem $\mathbf{A}$, a parameterized version of $\mathbf{A}$ is obtained by specifying a parameter of this problem and getting as additional

input a non-negative integer k which restricts the value of this parameter. The resulting parameterized problem $\langle \mathbf{A}, k \rangle$ is *fixed-parameter tractable* (FPT) with respect to k if it can be solved in time $f(k) * n^{\mathcal{O}(1)}$, where $f(k)$ is a function depending only on k .

3 The SOFTALLEQUAL Constraint

We define the constraint that we study in this paper, SOFTALLEQUAL_G , by reversing the graph-based cost of the SOFTALLDIFF constraint (Definition 2).

Definition 3 (SOFTALLEQUAL_G – dual of SOFTALLDIFF_G)

$$\text{SOFTALLEQUAL}_G(\{X_1, \dots, X_n\}, N) \Leftrightarrow N \leq |\{\{i, j\} \mid X_i = X_j \text{ \& } i \neq j\}|.$$

Interestingly, the same inversion of the definition for the variable-based cost of SOFTALLDIFF leads to the ATMOSTNVALUE constraint [2]. The focus of this paper, however, is on the graph-based cost. We first show that solving a CSP with the SOFTALLEQUAL_G constraint is intractable using a reduction from $\mathbf{3dMatching}$ [6].

Definition 4 ($\mathbf{3dMatching}$)

Data: An integer K , three disjoint sets X, Y, Z , and $T \subseteq X \times Y \times Z$.

Question: Does there exist $M \subseteq T$ such that $|M| \geq K$ and $\forall m_1, m_2 \in M, \forall i \in \{1, 2, 3\}, m_1[i] \neq m_2[i]$.

Theorem 1 (The Complexity of SOFTALLEQUAL_G). *Finding a satisfying assignment for the SOFTALLEQUAL_G constraint is NP-complete even if no value appears in more than three domains.*

Proof. The problem is clearly in NP: checking the number of equalities in an assignment can be done in $\mathcal{O}(n^2)$ time.

We use a reduction from $\mathbf{3dMatching}$ to show completeness. Let $P = (X, Y, Z, T, K)$ be an instance of $\mathbf{3dMatching}$, where: K is an integer; X, Y, Z are three disjoint sets such that $X \cup Y \cup Z = \{x_1, \dots, x_n\}$; and $T = \{t_1, \dots, t_m\}$ is a set of triplets over $X \times Y \times Z$. We build an instance I of SOFTALLEQUAL_G as follows:

1. Let $n = |X| + |Y| + |Z|$, we build n variables $\{X_1, \dots, X_n\}$.
2. For each $t_l = \langle x_i, x_j, x_k \rangle \in T$, we have $l \in \mathcal{D}(X_i)$, $l \in \mathcal{D}(X_j)$ and $l \in \mathcal{D}(X_k)$.
3. For each pair (i, j) such that $1 \leq i < j \leq n$, we put the value $(|T| + (i - 1) * n + j)$ in both $\mathcal{D}(X_i)$ and $\mathcal{D}(X_j)$.

We show there exists a matching of P of size K if and only if there exists a solution of I with $\lfloor \frac{3K+n}{2} \rfloor$ equalities. We refer to “a matching of P ” and to a “solution of I ” as “a matching” and “a solution” throughout this proof, respectively.

$\Rightarrow$: We show that if there exists a matching of cardinality K then there exists a solution with at least $\lfloor \frac{3K+n}{2} \rfloor$ equalities. Let M be a matching of cardinality K .

We build a solution as follows. For all $t_l = \langle x_i, x_j, x_k \rangle \in M$ we assign X_i, X_j and X_k to l (2). Observe that there remain exactly $n - 3K$ unassigned variables after this process. We pick an arbitrary pair of unassigned variables and assign them with their common value (3), until at most one variable is left (if one variable is left we assign it to an arbitrary value). Therefore, the solution obtained in this way has exactly $\lfloor \frac{3K+n}{2} \rfloor$ equalities, $3K$ from the variables corresponding to the matching and $\lfloor \frac{n-3K}{2} \rfloor$ for the remaining variables.

$\Leftarrow$: We show that if the cardinality of the maximal matching is K , then there is no solution with more than $\lfloor \frac{3K+n}{2} \rfloor$ equalities. Let S be a solution. Furthermore, let L be the number of values appearing three times in S . Observe that this set of values corresponds to a matching. Indeed, a value l appears in three domains $\mathcal{D}(X_i), \mathcal{D}(X_j)$ and $\mathcal{D}(X_k)$ if and only if there exists a triplet $t_l = \langle x_i, x_j, x_k \rangle \in T$ (2). Since a variable can only be assigned to a single value, the values appearing three times in a solution form a matching. Moreover, since no value appears in more than three domains, all other values can appear at most twice. Hence the number of equalities in S is less than or equal to $\lfloor \frac{3L+n}{2} \rfloor$, where L is the size of a matching. It follows that if there is no matching of cardinality greater than K , there is no solution with more than $\lfloor \frac{3K+n}{2} \rfloor$ equalities. $\square$

It is worth noting that if one similarly reverses the alternative, variable-based, cost of `SOFTALLDIFF`, the result corresponds to the `ATMOSTNVALUE` constraint. Interestingly, this is *not* equivalent to applying the variable-based cost on a constraint `ALLEQUAL`. For instance, consider n variables $X_1, \dots, X_n$ and suppose that half are assigned to a whilst the other half are assigned to b . One needs to change $n/2$ variables in order to make them all equal, and $n - 2$ to make them all different. In this paper we consider only the costs as defined for `SOFTALLDIFF` in [10], when reasoning about both lower and upper bounds. On the other hand, the graph-based cost on `ALLEQUAL` is indeed equivalent to the opposite of `SOFTALLDIFFG`.

4 Approximation Algorithm

In this section and in the rest of the paper we consider the optimization version of `SOFTALLEQUALG` where the objective is to *maximize* the number of pairs of variables assigned with the same value. We first study a natural greedy algorithm for approximating the maximum number of equalities in a set of variables. This algorithm picks the value that occurs in the largest number of domains, and assigns as many variables as possible to this value (this can be achieved in $\mathcal{O}(m)$). Then it recursively repeats the process on the resulting sub-problem until all variables are assigned (at most $\mathcal{O}(n)$ times). We show that this algorithm approximates the maximum number of equalities with a factor 2 in the worst case. Moreover, we can implement it to run in linear amortized time (that is, $\mathcal{O}(m)$) by using the following data structures:

- $var : \Lambda \mapsto 2^{\mathcal{X}}$ maps every value v to the set of variables whose domains contain v .
- $val : \mathbb{N} \mapsto 2^A$ maps every integer $i \in [0..n]$ to the set of values appearing in exactly i domains.

These data structures are initialized in Lines 1 and 2 of Algorithm 1, respectively.

Algorithm 1. GreedyValue

Data: A set of variables $\mathcal{X}$

Result: Lower bound on the maximum number of equalities

```

1  $var(v) \leftarrow \emptyset, \forall v \in \bigcup_{X \in \mathcal{X}} \mathcal{D}(X);$ 
  foreach  $X \in \mathcal{X}$  do
    foreach  $v \in \mathcal{D}(X)$  do
       $\text{add } X \text{ to } var(v);$ 
2  $val(k) \leftarrow \emptyset, \forall k \in [0..|\mathcal{X}|];$ 
  foreach  $v \in \bigcup_{X \in \mathcal{X}} \mathcal{D}(X)$  do
     $\text{add } v \text{ to } val(|var(v)|);$ 
  return  $\text{AssignAndRecurse}(var, val, |\mathcal{X}|);$ 
```

Algorithm 2. AssignAndRecurse

Data: a mapping $var : \Lambda \mapsto 2^{\mathcal{X}}$, a mapping $val : \mathbb{N} \mapsto 2^A$, an integer k ;

```

1 while  $val(k) \neq \emptyset$  do  $k \leftarrow k - 1;$ 
  if  $k \leq 1$  then
     $\text{return } 0;$ 
  else
2    $\text{pick and remove any } v \in val(k);$ 
3   foreach  $X \in var(v)$  do
     foreach  $w \neq v \in \mathcal{D}(X)$  do
        $occ_w \leftarrow |var(w)|;$ 
        $\text{remove } w \text{ from } val(occ_w);$ 
        $\text{add } w \text{ to } val(occ_w - 1);$ 
4    $\text{assign } X \text{ with } w \text{ and remove } X \text{ from } var(w);$ 
   $\text{return } \frac{k(k-1)}{2} + \text{AssignAndRecurse}(var, val, k);$ 
```

The above algorithm returns the number of pairs of equal values of an assignment of the given CSP, which is constructed on Line 4.

Theorem 2 (Algorithm Correctness). *The algorithm GreedyValue approximates the optimal satisfying assignment of the SOFTALLEQUAL_G constraint within a factor of 2 and runs in $\mathcal{O}(m)$.*

Proof. We first prove the correctness of the approximation ratio, the soundness of the algorithm and then the complexity of the algorithm.

Approximation Factor. We proceed using induction on n . Let lb be the value returned by **GreedyValue** and let eq^* be the maximum possible number of equalities. We denote $P(n)$ the proposition “If there are no more than n values in the union of the domains of $\mathcal{X}$, then $lb \geq eq^*/2$ ”. $P(1)$ implies that every variable can all be assigned to a unique value v . Algorithm **GreedyValue** therefore chooses this value and assigns all variables to it. In this case $lb = eq^*$.

Now we suppose that $P(n)$ holds and we show that $P(n+1)$ also holds. Let $\mathcal{X}$ be a set of variables such that $|\bigcup_{X \in \mathcal{X}} \mathcal{D}(X)| = n+1$ and let v be the first value chosen by **GreedyValue**. We denote by $\mathcal{X}_v = \{X \in \mathcal{X} \mid v \notin \mathcal{D}(X)\}$ the set of variables whose domains do not contain v , $\bar{\mathcal{X}}_v = \mathcal{X} \setminus \mathcal{X}_v$ the complementary set of variables assigned to v and we let $k = |\bar{\mathcal{X}}_v|$. We denote eq_v^* the maximal number of equalities on the set of variables $\mathcal{X}_v$, that is where both variables in the equality belong to $\mathcal{X}_v$. Consider any variable $X \in \bar{\mathcal{X}}_v$. Given any value w in $\mathcal{D}(X)$, there are no more than k variables in $\mathcal{X}$ containing w . Indeed, v was chosen for maximizing this criterion and belongs to the domains of exactly k variables. Therefore, the total number of equalities involving at least one variable in $\bar{\mathcal{X}}_v$ is at most $k(k-1)$, since there are k variables in $\bar{\mathcal{X}}_v$ and each can only be involved in $k-1$ equalities. Since an equality either involves at least one variable $\bar{\mathcal{X}}_v$, or none of them, we can bound the maximal total number of equalities as follows:

$$eq^* \leq k(k-1) + eq_v^*.$$

Now, observe that the total number of values in $\mathcal{X}_v$ is less than or equal to n since every variable whose domain contains v is in $\bar{\mathcal{X}}_v$. Since we suppose that $P(n)$ holds, we know that the value returned by **GreedyValue** for $\mathcal{X}_v$ is greater than or equal to $eq_v^*/2$. Hence the value returned for $\mathcal{X}$ is greater than or equal to $k(k-1)/2 + eq_v^*/2$. We can therefore conclude that $lb \geq eq^*/2$ and hence $P(n+1)$ is true.

Correctness. Here we show that the mapping *var* and *val* are correctly updated in a call to **AssignAndRecurse**. Let $\mathcal{X}$ be the set of variables given as initial input of **GreedyValue**. We define $\mathcal{X}_V$, as the set of variables remaining after greedily choosing and assigning the set of values V . ($\mathcal{X}_V = \{X \mid X \in \mathcal{X} \ \& \ \mathcal{D}(X) \cap V = \emptyset\}$). We say that *val* and *var* are correct for $\mathcal{X}_V$ iff both of the following invariants hold:

$$\forall v \in \bigcup_{X \in \mathcal{X}_V} \mathcal{D}(X), \text{ var}(v) = \{X \mid v \in \mathcal{D}(X)\} \quad (1)$$

$$\forall k \in [1..n], \text{ val}(k) = \{v \mid k = |\text{var}(v)|\} \quad (2)$$

This is clearly the case after the initialisation phase. Now we suppose that it is the case at the i^{th} call to **AssignAndRecurse** and we show that it still holds at the $(i+1)^{th}$ call. We assume that value w is chosen in Line 2.

We suppose first that invariant 1 does not hold. That is, there exists $X \in \mathcal{X}_V$, v such that either $v \in \mathcal{D}(X)$ and $X \notin \text{var}(v)$ or $v \notin \mathcal{D}(X)$ and $X \in \text{var}(v)$. The latter case is not possible since we only remove values from $\text{var}(v)$. The former case can only arise if X was removed from $\text{var}(v)$ in **AssignAndRecurse** (Line 4). However this can only happen if $X \in \text{var}(w)$, hence $X \notin \mathcal{X}_{V \cup \{w\}}$.

Then we suppose that invariant 2 does not hold, i.e., there exists k, v such that either $v \in \text{val}(k)$ and $k \neq |\text{var}(v)|$ or $v \notin \text{val}(k)$ and $k = |\text{var}(v)|$. However, the cardinality k of $\text{var}(v)$ can only decrease by one at Line 4, and in that case v is removed from $\text{val}(k)$ and added to $\text{val}(k - 1)$.

Complexity. The mapping var is in $\mathcal{O}(m)$ space, and val is in $\mathcal{O}(\lambda)$, where λ denotes the number of distinct values. Initialising both mappings is done in linear time since exactly one element is added to either val or var at every step. Hence the initialisation is in $\mathcal{O}(m) + \mathcal{O}(\lambda)$ time, i.e., $\mathcal{O}(m)$. In Line 1 in **AssignAndRecurse**, k can be decremented at most n times in total. In Loop 3, every iteration remove exactly one element in var , the amortised time complexity for this loop therefore is $\mathcal{O}(m)$. The overall time complexity is thus $\mathcal{O}(m)$. $\square$

Theorem 3 (Tightness of the Approximation Ratio). *The approximation factor of 2 for GreedyValue is tight.*

Proof. Let $\{X_1, \dots, X_4\}$ be a set of four variables with domains as follows:

$$X_1 \in \{a\}; \quad X_2 \in \{b\}; \quad X_3 \in \{a, c\}; \quad X_4 \in \{b, c\}.$$

Every value appears in exactly two domains, hence **GreedyValue** can choose any value. We suppose that the value c is chosen first. At this point no other value can contribute to an equality, hence **GreedyValue** returns 1. However, it is possible to achieve two equalities with the following solution: $X_1 = a$, $X_3 = a$, $X_2 = b$, $X_4 = b$. $\square$

5 Tractable Class

In this section we explore further the connection between the SOFTALLEQUAL_G constraint and vertex matching. We showed earlier that the general case was linked to **3dMatching**. We now show that the particular case where no value appears in more than two domains solving the SOFTALLEQUAL_G constraint is equivalent to the vertex matching problem on general graphs, and therefore can be solved by a polynomial time algorithm. We shall then use this tractable class to show that SOFTALLEQUAL_G is NP-hard only if an unbounded number of values appear in more than two domains.

Definition 5 (The VertexMatching Problem)

Data: An integer K , an undirected graph $G = (V, E)$.

Question: Does there exist $M \subseteq E$ such that $|M| \geq K$ and $\forall e_1, e_2 \in M$, e_1 and e_2 do not share a vertex.

Theorem 4 (Tractable Class of SOFTALLEQUAL_G). *If all triplets of variables $X, Y, Z \in \mathcal{X}$ are such that $\mathcal{D}(X) \cap \mathcal{D}(Y) \cap \mathcal{D}(Z) = \emptyset$ then finding an optimal satisfying assignment to SOFTALLEQUAL_G is in P .*

Proof. In order to solve this problem, we build a graph $G = (V, E)$ with a vertex x_i for each variable $X_i \in \mathcal{X}$, that is, $V = \{x_i \mid X_i \in \mathcal{X}\}$. Then for each pair

$\{i, j\}$ such that $\mathcal{D}(X_i) \cap \mathcal{D}(X_j) \neq \emptyset$, we create an undirected edge $\{i, j\}$; let $E = \{\{i, j\} \mid i \neq j \text{ \& } \mathcal{D}(X_i) \cap \mathcal{D}(X_j) \neq \emptyset\}$.

We first show that if there exists a matching of cardinality K , then there exists a solution with at least K equalities. Let M be a matching of cardinality K of G , for each edge $e = (i, j) \in M$ we assign X_i and X_j to any value $v \in \mathcal{D}(X_i) \cap \mathcal{D}(X_j)$ (by construction, we know that there exists such a value). Observe that no variable is counted twice since it would mean that two edges of the matching have a common vertex. The obtained solution therefore has at least $|M|$ equalities.

Now we show that if there are K equalities in S , then there exists a matching of cardinality K . Let S be a solution, and let $M = \{\{i, j\} \mid S[X_i] = S[X_j]\}$. Observe that M is a matching of G . Indeed, suppose that two edges sharing a vertex (say $\{i, j\}, \{j, k\}$) are both in M . It follows that $S[X_i] = S[X_j] = S[X_k]$, however this is in contradiction with the hypothesis. We can therefore compute a solution S maximizing the number of equalities by computing a maximal matching in G . $\square$

This tractable class can be generalized by restricting the number of occurrences of values in the domains of variables. The notion of *heavy values* is key to this result.

Definition 6 (Heavy Value). *A heavy value is a value that occurs more than twice in the domains of the variables of the problem.*

Theorem 5 (Tractable Class with Heavy Values). *If the domain $\mathcal{D}(X_i)$ of each variable X_i contains at most one heavy value then finding an optimal satisfying assignment of SOFTALLEQUAL_G is in P .*

Proof. Consider a two stage algorithm. In the first stage it explores all values w that have three or more appearances and assigns w to all the variables whose domains contain it. Notice that no variable will be assigned with two values. In the second stage the CSP created by the domains of unassigned variables consists of only values having at most two occurrences, so we solve this CSP by transforming it to the matching problem as suggested in the proof of Theorem 4.

We show that there exists an optimal solution where each variable that can be assigned to an heavy value is assigned to this value. Let s^* be an optimal solution and w be an heavy value over a set T of variables of cardinality t . We suppose that only $z < t$ of them are assigned to w in s^* . Consider the solution s' obtained by assigning all these t variables to w : we add exactly $t(t-1)/2 - z(z-1)/2$ equalities. However, we potentially remove $t-z$ equalities since values other than w do not appear more than twice. We therefore have $\text{obj}(s') - \text{obj}(s^*) \geq t^2 - 3t - z^2 + 3z$, which is non-negative for $t \geq 3$ and $z < t$. By iteratively applying this transformation, we obtain an optimal solution where each variable that can be assigned to an heavy value is assigned to this value. The first stage of the algorithm is thus correct. The second stage is correct by Theorem 4. $\square$

6 Parameterized Complexity

We further advance our analysis of the complexity of the SOFTALLEQUAL_G constraint by introducing a fixed-parameter tractable (FPT) algorithm with respect to the number of values. This result is important because it shows that the complexity of propagating this constraint grows only polynomially in the number of variables. It may therefore be possible to achieve GAC at a reasonable computational cost even for a very large set of variables, providing that the total number of distinct values is relatively small.

We first show that the SOFTALLEQUAL_G problem is FPT with respect to the number of values λ . We use the tractable class introduced in the previous section to generalize this result, showing that the problem is FPT with respect to the number of *heavy* values occurring in domains containing two or more heavy values. We begin with a definition.

Definition 7 (Solution from a Total Order). *A solution $s_{\prec}$ is induced by a total order $\prec$ over the values if and only if*

$$s[X] = v \Rightarrow \forall w \prec v, w \notin \mathcal{D}(X).$$

We now prove the following key lemma.

Lemma 1. *Let s^* be an optimal solution, v be a value, and $\text{occ}(s^*, v)$ be the number of variables assigned to v in s^* . Moreover, let $\prec_{\text{occ}}$ be a total order such that values are ranked by decreasing number of occurrences (ties are broken arbitrarily). We claim that $\prec_{\text{occ}}$ induces s^* .*

Proof. Consider, without loss of generality, a pair of values v, w such that $v \prec_{\text{occ}} w$. By definition we have $\text{occ}(s^*, v) \geq \text{occ}(s^*, w)$. We suppose that the hypothesis is falsified and show that this leads to a contradiction. Suppose that there exists a variable X such that $\{v, w\} \subseteq \mathcal{D}(X)$ and $s^*[X] = w$ (that is, $\prec_{\text{occ}}$ does not induce s^*). The objective value of the solution s' such that $s'[X] = v$ and $s'[Y] = s^*[Y] \forall y \neq x$ is given by: $\text{obj}(s') = \text{obj}(s^*) + \text{occ}(s^*, v) - (\text{occ}(s^*, w) - 1)$. Therefore, $\text{obj}(s') > \text{obj}(s^*)$. However, s^* is optimal, hence this is a contradiction. $\square$

This lemma has two interesting consequences. The first consequence is expressed by the following corollary.

Corollary 1. *There exists a total order $\prec$ over the set of values, such that the solution $s_{\prec}$ induced by $\prec$ is optimal.*

Proof. Direct consequence of Lemma 1.

The fixed-parameter tractability of the SOFTALLEQUAL_G constraint follows easily from Corollary 1.

Theorem 6 (FPT – number of values). *Finding an optimal satisfying assignment of the SOFTALLEQUAL_G constraint is fixed-parameter tractable with respect to λ , the number of values in the domains of the constrained variables.*

Proof. Explore all possible $\lambda!$ permutations of values. For each permutation create a solution induced by this permutation. Compute the cost of this solution. Return the solution having the highest cost. According to Corollary 1, this solution is optimal. Creating an induced solution can be done by selecting for each domain the first value in the order. Clearly, this can be done in $\mathcal{O}(m)$. Computing the cost of the given solution can be done by computing the number of occurrences $occ(w)$ and then summing up $occ(w) * (occ(w) - 1)/2$ for all values w . Clearly, this can be done in $\mathcal{O}(m)$ as well. Hence the theorem follows. $\square$

The second corollary from Lemma 1 is much more surprising.

Corollary 2. *The number of optimal solutions of the CSP with the SOFTALLEQUAL_G is at most $\lambda!$*

Proof. According to Lemma 1, each optimal solution is induced by an order over the values of the given problem. Clearly each order induces exactly one solution. Thus the number of optimal solution does not exceed the number of total orders which is at most $\lambda!$. $\square$

Corollary 2 claims that the number of optimal solutions of the considered problem *does not* depend on the number of variables and they all can be explored by considering all possible orders of values. We believe this fact is interesting from the practical point of view because in essence it means that even enumerating all optimal solutions is *scalable* with respect to the number of variables. Moreover, we can show that SOFTALLEQUAL_G is fixed-parameter tractable with respect to the number of bad values, defined as follows.

Definition 8 (Bad Value). *A value w of a given CSP is a bad value if and only if it is an heavy value and there is a domain $\mathcal{D}(X)$ that contains w and another heavy value.*

Theorem 7 (FPT – number of bad values). *Let k be the number of bad values of a CSP comprising only one SOFTALLEQUAL_G constraint. Then the CSP can be solved in time $\mathcal{O}(k! * n^2 * \lambda)$, hence SOFTALLEQUAL_G is fixed-parameter tractable with respect to k .*

Proof. Consider all the permutations of the bad values. For each permutation perform the following two steps. In the first step for each variable X where there are two or more bad values, remove all the bad values except the one which is the first in the order among the bad values of $\mathcal{D}(X)$ according to the given permutation. In the second stage we obtain a problem where each domain contains exactly one heavy value. Solve this problem polynomially by the algorithm provided in the proof of Theorem 5.

Let s be the solution obtained by this algorithm. We show that this solution is optimal. Let p^* be a permutation of *all* the values of the considered CSP so that the solution s^* induced by p^* has the highest possible cost. By Corollary 1, s^* is an optimal solution. Let p_1 be the permutation of the bad values which is induced by p^* and let s_1 be the solution obtained by the above algorithm with respect

to p_1 . By definition of s , $obj(s) \geq obj(s_1)$. We show that $obj(s_1) \geq obj(s^*)$ from which the optimality of s immediately follows.

Observe that there is no X such that $s^*[X] = w$ and w was removed from $\mathcal{D}(X)$ in the first stage of the above algorithm where the permutation p_1 is considered. Indeed, w can only be removed from $\mathcal{D}(X)$ if it is preceded in p_1 by a value $v \in \mathcal{D}(X)$. It follows that w is also preceded in p^* by v and consequently $s^*(X) \neq w$. Thus s^* is a solution of the CSP obtained as a result of the first stage. However s_1 is an *optimal* solution of that CSP by Theorem 5 and, consequently, $obj(s_1) \geq obj(s^*)$ as required. $\square$

This result shows that the complexity of propagating the SOFTALLEQUAL_G constraint comes primarily from the number of (bad) values, whereas other factors, such as the number of variables, have little impact. Observe that the “exponential” part of this algorithm is based on the exploration of all possible orders over the given set of bad values. In fact the ordering relation between two values matters only if these values belong to a domain of the same variable. In other words consider a graph H on values of the given CSP instance. Two values a and b are connected by an edge if and only if they belong to the domain of the same variable. Instead of considering all possible orders over the given set of values we may consider all possible ways of transforming the given graph into an acyclic digraph. The upper bound on the number of possible transformations is $2^{E(H)}$ where $E(H)$ is the number of edges of H . For sparse graphs such a bound is much more optimistic than $k!$. For example, if the average degree of a vertex is 4 then the number of considered partial orders is $2^{2k} = 4^k$.

7 Conclusion and Future Work

We showed that achieving GAC for the SOFTALLEQUAL_G constraint is NP-complete. Then we introduced a simple linear greedy algorithm and showed that it approximates the maximum number of pairs of variables that can be assigned equally within a factor of 2. Moreover, we showed that the hardness of the problem could be encapsulated by the number of “bad” values, irrespective of the size of the instance.

We believe one can combine our parameterized and approximation algorithms in order to design a practical algorithm for solving the SOFTALLEQUAL_G constraint. Firstly, Theorem 7 allows us to search in the space of permutations of *a subset of values*, which is usually much smaller than the space of partial assignments. Therefore we can design a branch-and-bound algorithm searching in the space of permutations. Secondly, we can use our approximation algorithm to more effectively prune the branches of the search tree. In particular, if the number of equalities guessed by the approximation algorithm is at most half of the current *upper bound* maintained by the branch-and-bound algorithm, then the algorithm may safely backtrack.

To the best of our knowledge, the problem we are tackling in this paper does not have a straightforward equivalent formulation in the algorithmic literature.

We therefore focused on complexity and approximability issues, laying down theoretical foundations for future work on this constraint. The design of a filtering algorithm for SOFTALLEQUAL_G , besides the trivial application of the bounds provided in this paper, is left as a challenge. A very important avenue of research is to study the complexity of achieving *bounds consistency* on SOFTALLEQUAL_G . The complexity of SOFTALLEQUAL_G when domains are intervals on $\mathbb{N}$ is still open, while bounds consistency on the ATMOSTNVALUE constraint can be done in polynomial time [2]. The problem is equivalent to finding a *clique cover* of minimal cardinality for the intersection graph of the domains [3], which is by definition an interval graph. The restriction of SOFTALLEQUAL_G to intervals, on the other hand, leads to a similar problem, but requiring a clique cover of the same graph that maximizes the sum of cardinalities of the cliques.

References

1. Bailleux, O., Marquis, P.: Some Computational Aspects of Distance-SAT. *Journal of Automated Reasoning* 37(4), 231–260 (2006)
2. 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)
3. Bessière, C., Hebrard, E., Hnich, B., Kiziltan, Z., Walsh, T.: Filtering Algorithms for the NValue Constraint. In: Barták, R., Milano, M. (eds.) *CPAIOR 2005*. LNCS, vol. 3524, pp. 79–93. Springer, Heidelberg (2005)
4. Crescenzi, P., Rossi, G.: On the Hamming Distance of Constraint Satisfaction Problems. *Theor. Comput. Sci.* 288(1), 85–100 (2002)
5. Forbus, K.D.: Introducing Actions into Qualitative Simulation. In: *IJCAI*, pp. 1273–1278 (1989)
6. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-completeness*. W.H. Freeman and Company, New York (1979)
7. Hebrard, E., Hnich, B., O’Sullivan, B., Walsh, T.: Finding Diverse and Similar Solutions in Constraint Programming. In: *AAAI*, pp. 372–377 (2005)
8. Hebrard, E., O’Sullivan, B., Walsh, T.: Distance Constraints in Constraint Satisfaction. In: *IJCAI*, pp. 106–111 (2007)
9. Niedermeier, R.: *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, Oxford (2006)
10. Petit, T., Régin, J.-C., Bessière, C.: Specific Filtering Algorithms for Over-Constrained Problems. In: *CP*, pp. 451–463 (2001)
11. Régin, J.-C.: A Filtering Algorithm for Constraints of Difference in CSPs. In: *AAAI*, pp. 362–367 (1994)
12. Régin, J.-C., Petit, T., Bessière, C., Puget, J.-F.: An Original Constraint Based Approach for Solving over Constrained Problems. In: *CP*, pp. 543–548 (2000)
13. Smyth, B., McClave, P.: Similarity vs. Diversity. In: Aha, D.W., Watson, I. (eds.) *ICCBR 2001*. LNCS (LNAI), vol. 2080, pp. 347–361. Springer, Heidelberg (2001)
14. Tversky, A.: Features of Similarity. *Psychological Review* 84, 327–352 (1977)
15. van Hoeve, W.-J., Pesant, G., Rousseau, L.-M.: On Global Warming: Flow-Based Soft Global Constraints. *Journal of Heuristics* 12(4-5), 347–373 (2006)

Structural Tractability of Propagated Constraints

Martin J. Green^{1,*} and Christopher Jefferson^{2,**}

¹ Department of Computer Science, Royal Holloway, University of London, UK

² Computing Laboratory, Oxford University, UK

Abstract. Modern constraint solvers do not require constraints to be represented using any particular data structure. Instead, constraints are given as black boxes known as propagators. Propagators are given a list of current domains for variables and are allowed to prune values not consistent with these current domains.

Using propagation as the only primitive operation on constraints imposes restrictions on the operations that can be performed in polynomial time. In the extensional representation of constraints (so-called positive table constraints) join and project are primitive polynomial-time operations. This is not true for propagated constraints.

The question we pose in this paper is: If propagation is the only primitive operation, what are the structurally tractable classes of constraint programs (whose instances can be solved in polynomial time)?

We consider a hierarchy of propagators: arbitrary propagators, whose only ability is consistency checking; partial assignment membership propagators, which allow us to check partial assignments; and generalised arc consistency propagators, the strongest form of propagator.

In the first two cases, we answer the posed question by establishing dichotomies. In the case of generalised arc consistency propagators, we achieve a useful dichotomy in the restricted case of acyclic structures.

1 Background

Over recent years the research into the constraint satisfaction problem (CSP) has led to the development of many practically useful general purpose constraint solvers. Solving a *CSP instance*, or constraint program, involves assigning *values* to *variables* which are consistent with a set of restrictions, known as *constraints*. The identification of *tractable* classes of CSP instances (solvable in polynomial time) has become an important and fruitful area of research in the constraint community. Generally, tractable classes are described by limiting either the interaction of the constraints, called a *structural* restriction, or the type of constraint allowed, called a *relational* restriction.

Unfortunately, most of the identified classes [1,2] rely on a somewhat impractical (*positive*) *extensional* representation for the constraints: an explicit relation, or table, of allowed assignments. In this representation, *join* and *project*

* This work was supported by EPSRC Grant EP/C525949/1.

** This work was supported by EPSRC Grant EP/D032636/1.

are primitive operations that can be performed in polynomial time. This notion of primitive operations allows large classes of CSP instances to be structurally tractable. For example, it is well-known that the class of CSP instances with *acyclic* structure is tractable in this representation [3]. *Structural decompositions*, such as *hypertrees* [1], aim to exploit this result, reducing CSP instances with bounded *width* to acyclic instances.

The advantage of the extensional representation is that it allows us to readily specify any constraint we wish on finite domains without issues of *expressibility*. Unfortunately, it also has a number of significant drawbacks in the context of constraint programming. Consider any constraint which forbids precisely one assignment on r variables each of domain size d . Such constraints can be used to represent clauses for SAT instances. However the (positive) extensional representation of this constraint requires listing all $d^r - 1$ allowed assignments.

This leads to the anomalous tractable class of all CSP instances which contain a *universal* (over all variables) constraint allowing all assignments. The extensional representation of this class is tractable: the size of the universal constraint dominates the number of solutions to the rest of the instance. However, it would be somewhat absurd to encode this universal constraint extensionally, and more succinct representations may affect the tractability of such classes.

Recently, researchers have started to determine those structural classes (derived from families of *hypergraphs* or *relational structures*) that remain tractable when we allow specific succinct representations to be used instead of the extensional representation. Both Houghton et al. [4] and Chen and Grohe [5] approach this task from above. Houghton et al. prove that a large class of structures, specified by two width parameters rather than just one, define a tractable structural class of CSP instances when we allow each constraint to be represented as the smaller of the allowed or forbidden assignments. Chen and Grohe specify two succinct representations: one based on a generalised form of DNF allowing larger domain sizes (called the GDNF representation), the other based on a form of decision diagrams (called the DD representation). In each case, they find a dichotomy between the tractable and intractable structural classes.

In this paper, we approach this task from below. Modern constraint solvers implement constraints as *oracles* known as *propagators*. Normally, only two primitive operations are allowed: consistency checking (membership) and propagation (pruning inconsistent values from domains). A propagator for a constraint is an algorithm. It takes as input a set of possible values for each variable and returns a subset of these values for each variable. Values may be removed from a set only when they cannot occur in any assignment to the constraint which is consistent with the given sets. In order to use a propagator as a complete representation of a constraint, we combine the two allowed operations into one, imposing a (natural) membership condition on the propagators we consider.

Propagators have proved to be very successful in practice. Efficient propagators have been designed for a large range of constraint types [6] and implementations are provided in many available solvers [7,8,9,10]. Unfortunately, it is often the case that propagators align poorly with existing tractability theory, simply

because they are able to provide very compact representations for many common constraint types. The standard assumption of polynomial-time join and project operators is no longer valid.

In this paper, we pose the question: If propagation is the only primitive operation, what are the structurally tractable classes of CSP instances?

We consider a hierarchy of propagators. In Sect. 3 we begin with arbitrary propagators, whose only ability is consistency checking (membership). In this representation, we find very small tractable structural classes. We then examine *partial assignment membership* (PAM) propagators in Sect. 4, which allow us to check partial assignments. In this representation we find that there are much larger, more useful tractable structural classes, defined by restricting the overlapping of constraints. Finally, in Sect. 5, we investigate *generalised arc consistency* propagators, the strongest form of propagator: here we find still larger tractable structural classes with less restrictive overlapping.

In the first two cases, we completely answer the posed question by establishing dichotomies. In the case of generalised arc consistency propagators, we achieve a useful dichotomy in the restricted case of acyclic structures.

2 Theoretical Introduction

In this section we introduce fundamental theory and definitions. We begin by considering a constraint satisfaction problem (CSP).

Definition 1. A CSP instance is a triple $\langle V, D, C \rangle$, where V is a finite set of **variables**, D is a function which maps each element of V to a finite set of possible values, called its **domain**, and C is a finite set of **constraints**.

Each constraint $c \in C$ is a pair, $\langle \sigma, \rho \rangle$, where σ is a sequence of variables from V , called the **scope**. The length of σ , denoted $|\sigma|$, is called the **arity** of c . The **relation**, ρ , is a subset of $D(\sigma[1]) \times \cdots \times D(\sigma[r])$, where $r = |\sigma|$, and defines the allowed assignments for this constraint.

A **solution** to a CSP instance is a function which maps each variable to a value from its domain which is consistent with all of the constraints.

A CSP is a decision problem. In this paper, we consider the *search problem* for a class of CSP instances. However, for any class of CSP instances for which we are allowed to add *constant constraints*, which define assignments to some subsets of the variables, these two notions coincide [11]: we find a solution (backtrack-free) in at most $|V| \times d$ steps (where V is the set of variables and d is the size of the largest domain) by adding unary constant constraints and calling the decision algorithm at each step. For all the classes we consider in this paper, we show that adding constant constraints does not affect complexity.

CSP instances are mathematical objects. They tell us what the instances are, but not how they should be *represented* for feeding to a constraint solver.

Definition 2. The *extensional (Pos) representation* of a CSP instance $\langle V, D, C \rangle$ specifies the size of V , the size of the union of the domains, and then

a list of constraints. Each constraint is represented extensionally as a list of the variables in the scope followed by a list of the allowed tuples. The **size** of the extensional representation of the CSP instance $\langle V, D, C \rangle$ is

$$\log |V| + \log d + \sum_{\langle \sigma, \rho \rangle \in C} (|\sigma| (|\rho| \log d + \log |V|)), \text{ where } d = \left| \bigcup_{v \in V} D(v) \right|.$$

It is this representation that has been assumed in most tractability results for classes of CSP instances [1,2]. However, this leads to anomalies since it imposes an artificial limitation on practitioners who use other representations such as forbidden assignments, equations, SAT clauses or *propagators*.

Definition 3. Given a CSP instance $\langle V, D, C \rangle$, a **sub-domain** of a variable $v \in V$ is a subset of $D(v)$.

Let $\langle \sigma, \rho \rangle \in C$ be a constraint of arity r . A **list of sub-domains**, S , for σ is a list containing a sub-domain for each element of σ in order. We denote by $\times S$ the product of the sets in S , so that $\times S = S[1] \times \dots \times S[r]$.

A **propagator** prop for $\langle \sigma, \rho \rangle$ is a function which maps a list of sub-domains for σ to another list of sub-domains for σ satisfying the following conditions:

1. $\times \text{prop}(S) \subseteq \times S$ (**inclusion**)
2. $\times S \subseteq \times T \Rightarrow \times \text{prop}(S) \subseteq \times \text{prop}(T)$ (**monotonicity**)
3. $a \in (\times S \cap \rho) \Rightarrow a \in \times \text{prop}(S)$ (**validity**)
4. If $\times S = \{a\}$ and $a \notin \rho$, then $\times \text{prop}(S) = \emptyset$ (**membership**)

The first three of these conditions are generally accepted requirements which ensure that applying a list of propagators results in a well-defined fixed point. The fourth condition is introduced to provide a method of checking if an assignment satisfies a constraint, allowing us to use a propagator as the complete representation of a constraint. In this paper, we will consider the class of all such propagators, Prop , as well as two restricted types: *partial assignment membership* (PAM) and *generalised arc consistency* (GAC).

Definition 4. A **partial assignment membership (PAM) propagator** satisfies the condition that when all variables have either their complete sub-domain or a single value in their sub-domain, it will empty the sub-domains if there is no allowed assignment in the current sub-domains.

A **generalised arc consistency (GAC)** [12,13] propagator removes as many values as possible from the sub-domain of each variable.

Propagators can naturally be intersected. As we will see in Proposition 2, it is not always possible to intersect (or join) propagators in polynomial time, even on the same scope. GAC propagators provide the greatest level of propagation that can be achieved for domain filtering: they can be seen as the intersection of all possible propagators for a constraint.

Example 1. The most famous GAC propagator is for the ‘AllDiff’ constraint, which imposes that a list of variables take different values [14].

Definition 5. Let $\mathcal{R}$ be any class of propagators (such as Prop, PAM or GAC). An $\mathcal{R}$ -**representation** of a constraint $\langle \sigma, \rho \rangle$ is a propagator for $\langle \sigma, \rho \rangle$ from $\mathcal{R}$.

An $\mathcal{R}$ -**representation** of the CSP instance $\langle V = \{v_1, \dots, v_n\}, D, C \rangle$ is a list of the indices of the variables in V , that is $1, \dots, n$, followed by a list of the union of the domains (again, as indices), followed by a list of $\mathcal{R}$ -representations of the constraints in C . The **size** of any $\mathcal{R}$ -propagator representation of the CSP instance $\langle V, D, C \rangle$ is

$$|V| \log |V| + d \log d + |C|, \text{ where } d = \left| \bigcup_{v \in V} D(v) \right|.$$

There are some differences between the size of a propagator representation of a CSP instance and its extensional representation (Definition 2). Firstly, we assume in a propagator representation that each variable is explicitly named (as the integers $1, \dots, n$), as are the domain values (as the integers $1, \dots, d$). This is because, unlike the extensional representation, we are not given the scopes and relations explicitly in the representation. We assume that constraints are represented in unit space (even their scopes). This is a very harsh property, but we make this assumption because we want to treat constraints as black boxes.

Definition 6. Let $\mathcal{C}$ be a class of CSP instances and Θ be a method for representing CSP instances. We denote by $\Theta(\mathcal{C})$ the set of Θ representations of the instances in $\mathcal{C}$. We call the class of CSP instance representations $\Theta(\mathcal{C})$ **tractable** if there is a solution algorithm for $\Theta(\mathcal{C})$ that runs in time polynomial in the size of the representation: otherwise, we call the class **intractable**.

In this paper, we are aiming for classes of propagator representations of CSP instances that are tractable. Strictly speaking, a class of propagator representations would be intractable if the propagators were not polynomial time. Instead, we assume that calling any propagator can be considered as one time step, which therefore limits any solution algorithm to a polynomial number of propagator calls in the size of the propagator representation (as given in Definition 5).

In order to prove intractability results in this paper, we will reduce intractable problems to propagator representations of CSP instances for which the propagators run in polynomial time. This will imply that the solution algorithm for these CSP instance representations must require exponential time (under a standard assumption from complexity theory). Using propagator representations allow us to drastically reduce the size of representation for some CSP instances.

Example 2. Recall Ex. 1. In any propagator representation, AllDiff is compact. However, we are unaware of any other representation that will succinctly represent an AllDiff constraint over r variables with $\geq r$ domain values. This includes Pos, forbidden tuples, GDNF [5] and DD [5].

Whilst it is possible to decompose an AllDiff constraint into a clique of binary inequalities, this decomposition does not allow for simple GAC propagation and is therefore less practical. Therefore, there is presently no method of identifying

practically applicable tractable classes of propagator representations that contain AllDiff constraints (of unbounded arity), despite the existence of such classes.

Definition 7. A **multi-hypergraph** is a pair $H = \langle V, E \rangle$ where V is a set of vertices and E is a multi-set of subsets of V , called the hyperedges of H . We define a **list of structures**¹ to be a list of multi-hypergraphs. We say a list of structures is of **bounded arity** if there is a bound on the size of all hyperedges in all multi-hypergraphs in the list.

For any tuple t of arity r we define $\text{set}(t) = \{t[1], \dots, t[r]\}$. The **structure** of a CSP instance $P = \langle V, D, C \rangle$ is the multi-hypergraph $\langle V, E \rangle$ where E is the multi-set containing all of the scopes of the constraints in C , abstracted to be sets. That is, $E = \{\{\text{set}(\sigma) \mid (\sigma, \rho) \in C\}\}$, where $\{\{\dots\}\}$ denotes a multiset.

A class of CSP instances is called **structural** if it is defined only by restricting the allowed structure of its instances to a given list of structures. For a list of structures $\mathcal{H}$ we denote by $\Psi(\mathcal{H})$ the structural class of CSP instances whose structures are all in $\mathcal{H}$.

The definition of structure we give here differs from the two usual definitions in the literature. When constraints are represented extensionally, it is a cheap operation to join constraints occurring on the same scope. As such, instances may be treated as if they have only a single constraint on each scope, which leads to the *hypergraph structure* of a CSP instance. Many structural tractability results use this notion [1]. Our definition of structure as a multi-hypergraph does not alter complexity results for the extensional representation, since subsumed hyperedges (which includes multiple occurrences of the same hyperedge) are always removed as an initial step (called *normalisation*). Multi-hypergraphs are a natural generalisation of the hypergraph structure of CSP instances and are necessary for our complexity results. As Proposition 2 will show, there is no simple method of combining constraints defined using propagators — multiple constraints on the same scope must be handled with care.

Alternatively, some results on structural tractability use the notion of *relational structures*, which are more restrictive than multi-hypergraphs: they impose that a particular set of (ordered) hyperedges must take the same constraint relation. Use of relational structures is outside the scope of this paper. However, we can induce a relational structure from a multi-hypergraph by placing each occurrence of a hyperedge in a different relation of the relational structure.

An important result in the area of structural tractability for bounded arity CSP instances is reproduced in Theorem 1. Due to insufficient space, we will not define the terms used in this theorem, but instead refer the reader to [15].

Theorem 1 (Corollary 19 of Grohe [15]). *Given a list, $\mathcal{H}$, of (relational) structures of bounded arity, the class $\text{Pos}(\Psi(\mathcal{H}))$ of extensionally represented CSP instances with structure in $\mathcal{H}$ is tractable if and only if the following requirement is satisfied: $\mathcal{H}$ has bounded treewidth modulo homomorphic equivalence.*

¹ We require a list of structures as there must be a mapping from the structures to the natural numbers, which arises naturally from a list. We also require this list of structures to be recursively enumerable.

Due to space limitations, we give the following result without proof.

Proposition 1. *Constraint solvers employing 2-way branching search cannot solve all tractable structural classes identified by Theorem 1 in polynomial time.*

The aim of this paper is to identify properties of families of structures which give tractability results for certain propagator representations. We rely heavily on Theorem 1 to produce complexity results by producing tractable CSP instances of bounded arity. Unfortunately, Proposition 1 shows that with current constraint solvers, employing only 2-way branching search, we cannot utilise these tractability results. We will be forced to modify the constraint solvers by providing new solution algorithms. However, the advantage of the results we give in this paper are that we need *only* provide new, simple, solution algorithms — we do not require *any* new representations of constraints to be implemented, instead utilising the current framework of propagated constraints.

For our hardness results we require some results from parameterised complexity theory. We limit ourselves to a very small explanation of the fundamental definitions here: for a more complete discussion we refer the reader to [16,17].

Definition 8. A **parameterised problem** is a pair $(\mathcal{P}, \kappa)$ where $\mathcal{P}$ is a problem and κ is a function which maps each member of $\mathcal{P}$ to $\mathbb{N}$.

Example 3. An instance of the parameterised problem p-CLIQUE is a pair (G, k) where G is a graph and k is an integer. The parameter of (G, k) is k . The question for (G, k) is: Does there exist a clique in G of size k ?

Classes of CSP instances can be parameterised. In particular, the parameterisation we use throughout this paper is given in Ex. 4.

Example 4. Given a (possibly infinite) list of multi-hypergraphs $\mathcal{H}$, any class of CSP instances whose structures are all in $\mathcal{H}$ can be parameterised by mapping each CSP instance to the index of the first occurrence of its structure in $\mathcal{H}$.

Definition 9. A parameterised problem $(\mathcal{P}, \kappa)$ is **fixed-parameter tractable (FPT)** if there is a (computable) function $f : \mathbb{N} \rightarrow \mathbb{N}$ and an algorithm which decides if the instance $p \in \mathcal{P}$ has a solution in time

$$f(\kappa(p)) \cdot |p|^{O(1)}.$$

Definition 10. An **FPT-reduction** from a parameterised problem $(\mathcal{P}, \kappa)$ to another parameterised problem $(\mathcal{P}', \kappa')$ is a mapping R from instances of $\mathcal{P}$ to instances of $\mathcal{P}'$ such that:

- $p \in \mathcal{P}$ has a solution if and only if $R(p)$ has a solution.
- There is a (computable) function $m : \mathbb{N} \rightarrow \mathbb{N}$ and an algorithm that, given $p \in \mathcal{P}$, computes $R(p)$ in time $m(\kappa(p)) \cdot |p|^{O(1)}$.
- There is a (computable) function $g : \mathbb{N} \rightarrow \mathbb{N}$ such that for all $p \in \mathcal{P}$ we have that $\kappa'(R(p)) \leq g(\kappa(p))$.

Parameterised complexity is a large and complex field of research. Instead of the traditional distinction between complexity classes P and NP, there is a hierarchy $FPT \subseteq W[1] \subseteq W[2] \subseteq \dots$ of complexity classes. Downey and Fellows [18] define this hierarchy and conjecture that it is strict. In this paper, we are only interested in $W[1]$ and make the standard assumption that $FPT \neq W[1]$ throughout (including previously in Theorem 1). For our proofs, we will make extensive use of Theorem 2.

Theorem 2 (Corollary 3.2 of [19]). *The parameterised problem p-CLIQUE is $W[1]$ -complete under FPT-reductions.*

3 Arbitrary Propagators

In this section, we will consider the question of tractable structural classes of CSP instances where each constraint is represented by an arbitrary propagator. Arbitrary propagators need only check if a complete assignment satisfies the constraint. As such, constant constraints can be integrated into a call to a propagator by intersecting them with the given sub-domains.

Example 5. Consider any CSP instance of arity bounded by k . We can convert any Prop representation of this instance into an extensional representation by testing the membership of all d^k possible assignments for each constraint, where d is the size of the union of the domains. Since the union of the domains is listed as part of the input and k is fixed, this is polynomial time. Alternatively, building a GAC propagator (which is the strongest possible propagator type) for an extensionally represented constraint is straightforward. It is equivalent to enforcing GAC on an extensionally represented constraint, which may be done in polynomial time [13], as can listing the variables and (used) domain values. Hence, for bounded arity, propagators are equivalent to extensionally represented constraints.

Lemma 1. *Let $\mathcal{H}$ be a list of structures of bounded arity. The class $\text{Prop}(\Psi(\mathcal{H}))$ is tractable if and only if $\mathcal{H}$ satisfies the requirement of Theorem 1.*

Proof. Ex. 5 shows that for bounded arity there exist polynomial-time reductions between $\text{Prop}(\Psi(\mathcal{H}))$ and $\text{Pos}(\Psi(\mathcal{H}))$, so Theorem 1 applies to $\text{Prop}(\Psi(\mathcal{H}))$. $\square$

Unfortunately, as our next result shows, *any* list of structures of *unbounded* arity generates a class of CSP instances that is intractable when represented using arbitrary propagators.

Lemma 2. *Let $\mathcal{H}$ be any list of structures of unbounded arity. The search problem for the class $\text{Prop}(\Psi(\mathcal{H}))$ is $W[1]$ -hard, and therefore intractable.*

Proof. We give an FPT-reduction from p-CLIQUE to $\text{Prop}(\Psi(\mathcal{H}))$. Given any instance (G, k) of p-CLIQUE, the parameter k is mapped to the index of the first element H of $\mathcal{H}$ which includes a hyperedge of arity at least k . We construct a Prop representation of a CSP instance with structure H as follows.

The domain of all variables is the vertices of G . For a single occurrence of a hyperedge of arity at least k , order this hyperedge and generate the Prop representation of the constraint with this scope such that the first k variables in its scope satisfy p-CLIQUE for (G, k) . All other constraints are true for all assignments. The size of this mapping is the size of (G, k) , plus a constant (polynomial in the size of H) to express the other domains and constraints, and is therefore a FPT-reduction. By Theorem 2, this proves the result. $\square$

Lemmas 1 and 2 provide the following dichotomy for arbitrary propagator representations of structural classes of CSP instances.

Theorem 3. *Let $\mathcal{H}$ be any list of structures. The class $\text{Prop}(\Psi(\mathcal{H}))$ is tractable if and only if $\mathcal{H}$ is of bounded arity and satisfies the requirement of Theorem 1.*

4 Partial Assignment Membership Propagators

Partial assignment membership (PAM) propagators exist for many constraint types. For instance, most arithmetic constraints have polynomial-time PAM propagators but yet have no polynomial-time GAC propagator. PAM propagators provide a useful level of propagation. For example, all propagators provided by the Minion CSP solver [10] are PAM propagators.

One feature of the (positive) extensional representation of constraints is that it is possible to join two constraints in polynomial time with respect to the size of their representation. This does not in general hold for propagators.

Proposition 2. *Given two lists of constraints A and B with polynomial time PAM propagators (or GAC propagators), checking if the constraints $A[i] \wedge B[i]$ have a consistent assignment is, in general, an NP-complete problem.*

Proof. Consider the constraints $\sum_{i=1}^n a_i v_i \leq 0$ and $\sum_{i=1}^n a_i v_i \geq 0$ defined by integer constants a_i and variables v_i with domain $\{0, 1\}$. Both of these families of constraint have polynomial time GAC propagators [20], which are of course PAM propagators. Given a set of integers $S = \{s_1, \dots, s_n\}$, the problem of finding a subset of S which sums to exactly 0 is a well-known NP-complete problem [21]. However, this problem is exactly equivalent to the intersection of the two constraints above by setting $a_i = s_i$ for each i . $\square$

We observe that a PAM propagator exactly captures the notion of applying constant constraints to some subset of the variables. As such, we can freely integrate constant constraints into calls to PAM propagators.

Many more families of structures derive tractable classes for PAM propagators than with general propagators. For example, the class of CSP instances which contain only a single constraint is tractable for the PAM propagator representation. However, there are still simple classes which remain intractable.

Definition 11. *Given any constraint, c , the **Free- c** constraint is defined as follows: Free- c is a constraint over the same (number of) variables, each containing*

one extra domain value, denoted *Free*. An assignment to the scope of *Free-c* is true either if any variable is assigned *Free* or the assignment satisfies *c*.

For a set of constraints, C , the set of constraints $\{\text{Free-}c \mid c \in C\}$ is denoted *Free- C* . For a problem, $\mathcal{P}$, and a set of constraints, $C(\mathcal{P})$, that model the instances of $\mathcal{P}$, we denote by *Free- $\mathcal{P}$* the set of constraints *Free- $C(\mathcal{P})$* .

PAM propagators for *Free-c* constraints only ever have to check if complete assignments satisfy *c*, because given a list of sub-domains with any unassigned variables, *Free-c* is satisfied by assigning any unassigned variable *Free*.

We assume the following model of p-CLIQUE as a set of constraints.

Definition 12. Let (G, k) be an instance of *p-CLIQUE*, where $G = \langle V, E \rangle$. We generate a constraint on k variables each with domain V and satisfying assignments defined by cliques of size k in G .

By imposing *Free-p-CLIQUE*, along with unary constraints of the type “variable is not assigned *Free*”, we derive classes of CSP instances which model p-CLIQUE and are therefore $W[1]$ -hard for the PAM representation.

Definition 13. Let $H = \langle V, E \rangle$ be a multi-hypergraph. A vertex $v \in V$ is called *isolated* if it exists in (at most) one occurrence of a hyperedge in E .

Lemma 3. Given a list of structures $\mathcal{H}$, generate a new list of structures $\mathcal{H}'$ by removing from members of $\mathcal{H}$ all isolated vertices. The search problem for $\text{PAM}(\Psi(\mathcal{H}))$ is $W[1]$ -hard if and only if it is $W[1]$ -hard for $\text{PAM}(\Psi(\mathcal{H}'))$.

Proof. Given a PAM propagator, prop , for a constraint c , over a list of variables $\langle x_1, \dots, x_r \rangle$ each with domain D , consider the projection, c_k , of c onto the variables $\langle x_1, \dots, x_k \rangle$, for $k \leq r$. A PAM propagator, prop_k , for c_k can be constructed from prop using at most $|D| \times (r - k)$ extra time: take the partial assignment to c_k and attach the complete sub-domain D for x_{k+1} to x_r .

Further, we observe that projecting out isolated variables preserves solutions. Therefore, we can solve CSP instances whose structure is $\mathcal{H}[i]$, for some i , by projecting down to $\mathcal{H}'[i]$ using the above transformation. Given a CSP instance whose structure is $\mathcal{H}'[i]$, we extend this to an instance on $\mathcal{H}[i]$ by giving all the extra variables a single domain value and have the constraints ignore the assignment to these variables. This gives FPT-reductions between $\text{PAM}(\Psi(\mathcal{H}))$ and $\text{PAM}(\Psi(\mathcal{H}'))$, proving the result. $\square$

We may now prove the main result of this section: a complete dichotomy of the tractable structural classes of CSP instances for the PAM representation.

Theorem 4. Given any list of structures $\mathcal{H}$, generate the list of structures $\mathcal{H}'$ by removing from members of $\mathcal{H}$ all isolated vertices. Then $\text{PAM}(\Psi(\mathcal{H}))$ is tractable if and only if $\mathcal{H}'$ is of bounded arity and satisfies the requirement of Theorem 1.

Proof. By Lemma 3 we know that the search problem for $\text{PAM}(\Psi(\mathcal{H}))$ is $W[1]$ -hard if and only if it is for $\text{PAM}(\Psi(\mathcal{H}'))$. We therefore consider only $\text{PAM}(\Psi(\mathcal{H}'))$.

If $\mathcal{H}'$ is not of bounded arity then we map p-CLIQUE to PAM($\Psi(\mathcal{H}')$) (using a similar mapping to that used in the proof of Lemma 2). Given any instance (G, k) of p-CLIQUE, the parameter k is mapped to the index of the first element H of $\mathcal{H}'$ which includes a hyperedge of arity at least k . Then, we construct a PAM representation of a CSP instance with structure H as follows.

For a single occurrence of a hyperedge of arity at least k , order this hyperedge and generate the PAM representation of the constraint with this scope such that the first k variables in its scope impose the *Free*-p-CLIQUE constraint for (G, k) , extending the constraint to ignore the assignment to the other variables. The domain for these k variables is the vertices of G together with the value *Free*. By definition of $\mathcal{H}'$ no variable on which we imposed *Free*-p-CLIQUE can be isolated, so for each constraint meeting the first k variables of the *Free*-p-CLIQUE constraint we impose that no variable in the constraint is assigned *Free*. All other variables in the CSP instance are given a single value (which is not *Free*) in their domains, and all other constraints allow all assignments. This CSP instance has a solution if and only if G has a clique of size k . The size of this mapping is the size of (G, k) , plus a constant (polynomial in the size of H) to express the other domains and constraints, and therefore is a FPT-reduction.

Otherwise, $\mathcal{H}'$ is of bounded arity. We use Ex. 5 to map in polynomial time between PAM($\Psi(\mathcal{H}')$) and Pos($\Psi(\mathcal{H}')$) and Lemma 3 completes the result. $\square$

5 Generalised Arc Consistency Propagators

Generalised arc consistency (GAC) propagators provide the highest level of domain filtering. Many famous GAC propagators have been found for highly practical *global constraints*. For an overview, including the following constraint types, see, for example, [6].

- AllDiff($\langle v_1, \dots, v_r \rangle$) : All of the v_i take distinct values.
- Element($\langle M_1, \dots, M_n \rangle, x, y$) : $M_x = y$.
- GCC($\langle v_1, \dots, v_r \rangle, \langle \langle l_1, u_1 \rangle, \dots, \langle l_r, u_r \rangle \rangle$) : For constants $\langle l_i, u_i \rangle$, there are between l_i and u_i occurrences of i in $\langle v_1, \dots, v_r \rangle$.

In this paper, we have not considered the cost of applying a propagator for a constraint, instead treating them as oracles taking constant time. That said, the tractability results generated in this paper would be less useful if low-cost propagation was not possible for many common constraint types. While there exist many constraint types for which GAC propagation is itself NP-hard, noticeably those involving arithmetic (such as bin packing), many constraints do have simple GAC propagators, including the previous examples. As with PAM propagators, calls to GAC propagators can integrate constant constraints without affecting complexity. In fact, they can integrate any unary constraints on their variables. This power of GAC propagation allows for significantly more tractable structural classes than with either arbitrary or PAM propagators.

In the special case of Boolean domains, Proposition 3 shows that PAM and GAC propagators have comparable complexity. However, for non-Boolean domains this is not the case, as our subsequent results show.

Proposition 3. *For a constraint c of arity r and Boolean domain, a GAC propagator for c can be constructed using $2r$ calls to a PAM propagator for c .*

Proof. We observe that non-empty sub-domains of a Boolean domain must either be complete or contain only a single value. This satisfies the condition for PAM propagators. GAC propagators must remove all values which cannot be extended to a solution. We can do this in at most $2r$ calls to the PAM propagator for c . $\square$

Theorem 4 says that the structures which derive tractable classes for the PAM representation are essentially bounded arity: there must be a bounded number of non-isolated vertices in each hyperedge. However, for the GAC representation, this restriction is lifted. The following example seems to be known in the constraints community, though the authors failed to find a reference in the literature.

Example 6. We call a multi-hypergraph $\langle V, E \rangle$ a **tree with single vertex overlap** if the following algorithm creates an empty multi-hypergraph: Choose any hyperedge $e \in E$ for which $|e \cap \bigcup_{f \in E \setminus \{e\}} f| \leq 1$, remove it from E , and repeat whilst some hyperedge has been removed.

Let $\mathcal{T}$ be any list of trees with single vertex overlaps. The class $\text{GAC}(\Psi(\mathcal{T}))$ is tractable: GAC propagation decides. This result is a straightforward extension of (binary) tree-structured instances being decided by *arc consistency*.

In this section, we generalise this result to provide a dichotomy for lists of *acyclic* multi-hypergraphs.

Definition 14. *Given any constraint, c , with scope $\langle v_1, \dots, v_r \rangle$, the **Pair- c** constraint is defined as follows: Pair- c is a constraint over $\langle x_1, \dots, x_r, y_1, \dots, y_r \rangle$, where for all i , x_i and y_i have the same domain as v_i . Pair- c is true either if $x_i \neq y_i$, for any i , or the assignment to $\langle x_1, \dots, x_r \rangle$ satisfies c .*

For a set of constraints, C , the set of constraints $\{\text{Pair-}c \mid c \in C\}$ is denoted Pair- C . For a problem, $\mathcal{P}$, and a set of constraints, $C(\mathcal{P})$, that model the instances of $\mathcal{P}$, we denote by Pair- $\mathcal{P}$ the set of constraints Pair- $C(\mathcal{P})$.

If assignments to a constraint c can be checked in polynomial time, then it is straightforward to show that Pair- c has a simple polynomial-time GAC propagator. The class of Pair-p-CLIQUE constraints will be used to construct intractable structural classes of CSP instances for the GAC representation.

Proposition 4. *The search problem for the following class of CSP instances, parameterised by structure, is $W[1]$ -hard for the GAC representation:*

- Variables labelled $x_1, \dots, x_n$ and $y_1, \dots, y_n$.
- An arbitrary Pair-p-CLIQUE constraint on scope $\langle x_1, \dots, x_n, y_1, \dots, y_n \rangle$.
- The constraint $x_i = y_i$ for $i = 1, \dots, n$.

Proof. There is a simple FPT-reduction from p-CLIQUE to this class. $\square$

Proposition 4 shows one important feature of the tractability for the GAC representation is the *size* of overlaps with other constraints, rather than simply the number of overlaps or the total number of variables contained in overlaps.

Definition 15. Given a multi-hypergraph H and a hyperedge e in H , a **non-trivial overlap** of e in H is a minimally sized set of vertices which, if removed from H , leaves every other hyperedge (including other occurrences of the same hyperedge) meeting e at no more than one vertex.

Definition 16. A multi-hypergraph H is **acyclic** if the following algorithm results in an empty multi-hypergraph: Remove all isolated vertices from (occurrences of) hyperedges, then remove all (occurrences of) hyperedges contained in some other hyperedge, and repeat whilst some change has been made.

A **join tree** of a multi-hypergraph $\langle V, E \rangle$ is a tree $J = \langle E, F \rangle$ which satisfies the condition: each $v \in V$ induces a connected subtree of J . Not all multi-hypergraphs have join trees. A multi-hypergraph is **acyclic** if it has a join tree.

For any list, $\mathcal{H}$, of acyclic structures, it is well-known that $\text{Pos}(\Psi(\mathcal{H}))$ is tractable [3]. In the extensional representation, we solve acyclic instances by identifying a join tree using the algorithm of Definition 16: the parent of a hyperedge is the hyperedge that subsumes it. We apply *pairwise consistency* along the join tree: two constraints are pairwise consistent if an assignment to their common variables can be extended to an assignment over their join. For the extensional representation, pairwise consistency can be achieved in polynomial time [22]. If each constraint still has allowed assignments, there is a solution, and we can find it in a backtrack-free way. This solution technique does not work for the GAC representation: for example, the intractable class of instances defined in Proposition 4 has only acyclic structures.

We are now in a position to provide the main result of this section: a dichotomy for the GAC representation in the restricted case of acyclic structures.

Theorem 5. Let $\mathcal{H}$ be any list of acyclic structures. $\text{GAC}(\Psi(\mathcal{H}))$ is tractable if and only if $\mathcal{H}$ has bounded non-trivial overlap.

Proof (sketch). If $\mathcal{H}$ does not have bounded non-trivial overlap then, for every l , the following property holds: there exists a structure H in $\mathcal{H}$ for which there is an occurrence of a hyperedge e containing $2l$ distinct vertices which may be partitioned into l pairs such that each pair exists in the intersection of e with an occurrence of some other hyperedge (possibly another occurrence of e).

We give an FPT-reduction from p-CLIQUE to $\text{GAC}(\Psi(\mathcal{H}))$. For any instance (G, k) of p-CLIQUE we choose the first structure H in $\mathcal{H}$ containing a hyperedge e with k pairs of vertices (as defined above). We generate an instance in $\text{GAC}(\Psi(\mathcal{H}))$ as follows. We impose a *Pair-p-CLIQUE* constraint for (G, k) on the k pairs of variables of e . Other constraints allow all assignments except for enforcing that each of the selected pairs of variables are equal. This provides a FPT-reduction from p-CLIQUE to $\text{GAC}(\Psi(\mathcal{H}))$.

Otherwise, $\mathcal{H}$ has bounded non-trivial overlap. Let k be this bound. We solve instances from $\text{GAC}(\Psi(\mathcal{H}))$ in polynomial time in the following way. Identify a join tree, J , for the structure of the instance, labelling each node with the constraint on this (ordered) hyperedge. Then, we identify a non-trivial overlap for each hyperedge (node of J). We determine the set of allowed assignments to the non-trivial overlaps: at most d^k assignments for maximum domain size d .

Finally, we show that pairwise consistency can be achieved using GAC on these allowed assignments to the non-trivial overlaps. Consider any adjacent pair of nodes in J . Apart from their non-trivial overlaps, they must meet in at most one additional variable. We remove an assignment to a non-trivial overlap if it cannot extend to one of the two labels (constraints) of these hyperedges. This can be tested using the GAC propagators for these two constraints, since there is at most one additional variable in their overlap that is not assigned a constant.

Hence, we can use GAC to achieve pairwise consistency along the join tree, in polynomial time, just as with a positive representation. If at any point we empty a set of assignments to a non-trivial overlap, there is no solution. Otherwise, after pairwise consistency has been achieved, there is a solution which can be found in a backtrack-free way, testing unary extensions with GAC propagation. $\square$

Theorem 5 includes the tractable class whose structures are trees with single vertex overlaps (Ex. 6): these structures have *empty* non-trivial overlaps. Unfortunately, we do not yet know what can be said about bounded width structural decompositions, such as hypertrees [1]. To finish, we observe a simple extension to the tractable structural classes of Theorem 5 for non-acyclic structures.

Corollary 1. *Let k be a constant and $\mathcal{H}$ be any list of structures, with bounded non-trivial overlap, that can be made acyclic by removal of at most k variables. The class $\text{GAC}(\Psi(\mathcal{H}))$ is tractable.*

6 Conclusion

In this paper we have made a major step towards making the tractability of structural classes of CSP instances applicable to modern constraint solvers. We have shown the unfortunately weak tractability classes for general propagators and the much stronger tractability classes for partial assignment membership propagators. We have given a dichotomy for generalised arc consistency propagators in the restricted case of acyclic structures. While we have not yet provided a complete dichotomy for generalised arc consistency propagators, we have shown the existence of large classes, both tractable and intractable, providing immediately useful results and guidance towards where a future dichotomy may exist.

Acknowledgements. The authors are extremely grateful to Peter Jeavons and David Cohen for many insightful discussions about the contents of this paper.

References

1. Gottlob, G., Leone, N., Scarcello, F.: A comparison of structural CSP decomposition methods. *Artif. Intell.* 124(2), 243–282 (2000)
2. Cohen, D., Jeavons, P.: The complexity of constraint languages. In: Rossi, F., van Beek, P., Walsh, T. (eds.) *Handbook of Constraint Programming*. Elsevier, Amsterdam (2006)

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. Houghton, C., Cohen, D.A., Green, M.J.: The effect of constraint representation on structural tractability. In: Benhamou, F. (ed.) *CP 2006*. LNCS, vol. 4204, pp. 726–730. Springer, Heidelberg (2006)
5. Chen, H., Grohe, M.: Constraint satisfaction with succinctly specified relations. In: Creignou, N., Kolaitis, P., Vollmer, H. (eds.) *Complexity of Constraints*. Dagstuhl Seminar Proceedings, Dagstuhl, Germany, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), vol. 06401. Schloss Dagstuhl, Germany, (2006)
6. van Hoeve, W.J., Katriel, I.: Global constraints. In: Rossi, F., van Beek, P., Walsh, T. (eds.) *Handbook of Constraint Programming*. Elsevier, Amsterdam (2006)
7. Cheadle, A., Harvery, W., Sadler, A.J., Schimpf, J., Shen, K., Wallace, M.: *ECLiPSe: An introduction*. Technical Report IC-Parc-03-1, Imperial College, London (2003)
8. The Gecode team: Generic constraint development environment, <http://www.gecode.org>
9. ILOG S.A.: *ILOG Solver 5.3 Reference and User Manual* (2002)
10. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: Brewka, G., Coradeschi, S., Perini, A., Traverso, P. (eds.) *ECAI*, pp. 98–102. IOS Press, Amsterdam (2006)
11. Cohen, D.: Tractable decision for a constraint language implies tractable search. *Constraints* 9(3), 219–229 (2004)
12. Mackworth, A.: Consistency in networks of relations. *Artificial Intelligence* 8, 99–118 (1977)
13. Bessière, C., Régin, J.C.: Arc consistency for general constraint networks: preliminary results. In: *Proceedings of IJCAI 1997*, Nagoya, Japan, pp. 398–404 (1997)
14. Régin, J.C.: A filtering algorithm for constraints of difference in CSPs. In: *AAAI 1994: Proceedings of the twelfth national conference on Artificial intelligence*, vol. 1, pp. 362–367. American Association for Artificial Intelligence, Menlo Park (1994)
15. Grohe, M.: The complexity of homomorphism and constraint satisfaction problems seen from the other side. *J. ACM* 54(1), 1 (2007)
16. Downey, R.G., Fellows, M.R.: *Parameterized Complexity*. Springer, Heidelberg (1999)
17. Flum, J., Grohe, M.: *Parameterized Complexity Theory* (Texts in Theoretical Computer Science. An EATCS Series). Springer, New York (2006)
18. Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness I: Basic results. *SIAM J. Comput.* 24(4), 873–921 (1995)
19. Downey, R.G., Fellows, M.R.: Fixed-parameter tractability and completeness II: On completeness for $W[1]$. *Theoretical Computer Science* 141(1–2), 109–131 (1995)
20. Schulte, C., Stuckey, P.J.: When do bounds and domain propagation lead to the same search space? *ACM Trans. Program. Lang. Syst.* 27(3), 388–425 (2005)
21. Rivest, R.L., Leiserson, C.E.: *Introduction to Algorithms*. McGraw-Hill, Inc., New York (1990)
22. Janssen, P., Jegou, P., Nougier, B., Vilarem, M.: A filtering process for general constraint satisfaction problems: achieving pair-wise consistency using an associated binary representation. In: *Proceedings of the IEEE Workshop on Tools for Artificial Intelligence*, pp. 420–427 (1989)

Connecting ABT with Arc Consistency^{*}

Ismael Brito and Pedro Meseguer

IIIA, Institut d'Investigació en Intel·ligència Artificial
CSIC, Consejo Superior de Investigaciones Científicas
Campus UAB, 08193 Bellaterra, Spain
{ismel, pedro}@iiia.csic.es

Abstract. ABT is the reference algorithm for asynchronous distributed constraint satisfaction. When searching, ABT produces nogoods as justifications of deleted values. When one of such nogoods has an empty left-hand side, the considered value is eliminated unconditionally, once and for all. This value deletion can be propagated using standard arc consistency techniques, producing new deletions in the domains of other variables. This causes substantial reductions in the search effort required to solve a class of problems. We also extend this idea to the propagation of conditional deletions, something already proposed in the past. We provide experimental results that show the benefits of the proposed approach, especially considering communication cost.

1 Introduction

In recent years, there is an increasing interest for solving problems in which information is distributed among different agents. Most of the work in constraint reasoning assumes centralized solving, so it is inadequate for problems requiring a true distributed resolution. This has motivated the new Distributed CSP (*DisCSP*) framework, where constraint problems with elements (variables, domains, constraints) distributed among automated agents which cannot be centralized for different reasons (prohibitive translation costs or security/privacy issues) are modelled and solved.

When solving a *DisCSP* instance, all agents cooperate to find a globally consistent solution. To achieve this, agents assign their variables and exchange messages on these assignments, which allows them to check their consistency with respect to problem constraints. Several synchronous and asynchronous solving algorithms have been proposed [2, 5, 8, 12, 13]. While synchronous algorithms are easier to understand and implement, asynchronous ones are more robust. If some agents disconnect, an asynchronous algorithm is still able to provide a solution for the connected part, while this is not true in general for synchronous ones. Asynchronous algorithms exhibit a high degree of parallelism, but the information exchanged among agents is less up to date than in synchronous ones.

ABT [12, 13] is the reference algorithm for asynchronous distributed constraint solving, playing a role similar to backtracking algorithm in the centralized case. Several ideas to improve its efficiency and privacy have been proposed.

^{*} Supported by the Spanish project TIN2006-15387-C03-01.

In this paper we study the idea of propagating value deletions in ABT. When searching, ABT produces nogoods as justifications of deleted values. When one of such nogoods has an empty left-hand side, the considered value is eliminated unconditionally, once and for all. This value deletion can be propagated using standard arc consistency techniques, producing new deletions in the domains of other variables. This causes substantial reductions in the search effort required to solve a class of problems, especially on communication cost. We extend this idea to the propagation of conditional deletions.

The idea of including consistency maintenance in ABT is not new. It was proposed by [9, 10]. However, the specialization to unconditional value deletions (nogoods with empty left-hand side) is new. Previous experimental results [9, 10], considered AAS [8] with bound consistency. Here, we provide experimental results for ABT with directional and full (both directions) arc consistency on a set of random *DisCSP* instances.

This paper is organized as follows. First, we recall the *DisCSP* definition and the *ABT* description. Then, we present the idea of propagating unconditional deletions, in the ABT-UAC algorithm. We extend this idea to conditional deletions, in the ABT-DAC algorithm. We present experimental results for both approaches on random *DisCSP* instances. Finally, we extract some conclusions and directions for further research.

2 Preliminaries

2.1 Distributed Constraint Satisfaction

A *Constraint Satisfaction Problem* $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ involves a finite set of variables $\mathcal{X}$, each taking values in a finite domain, and a finite set of constraints $\mathcal{C}$. A constraint on a subset of variables forbids some combinations of values that these variables can take. A *solution* is an assignment of values to variables which satisfies every constraint. Formally,

- $\mathcal{X} = \{x_1, \dots, x_n\}$ is a set of n variables;
- $\mathcal{D} = \{D(x_1), \dots, D(x_n)\}$ is a set of finite domains; $D(x_i)$ is value set for x_i ;
- $\mathcal{C}$ is a finite set of constraints. A constraint C_i on the ordered subset of variables $var(C_i) = (x_{i_1}, \dots, x_{i_{r(i)}})$ specifies the relation $prm(C_i)$ of the *permitted* combinations of values for the variables in $var(C_i)$, $prm(C_i) \subseteq \prod_{x_{i_k} \in var(C_i)} D(x_{i_k})$.
An element of $prm(C_i)$ is a tuple $(v_{i_1}, \dots, v_{i_{r(i)}})$, $v_{i_k} \in D(x_{i_k})$.

A *Distributed Constraint Satisfaction Problem* (*DisCSP*) is a *CSP* where variables, domains and constraints are distributed among automated agents. Formally, a finite *DisCSP* is defined by a 5-tuple $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \phi)$, where $\mathcal{X}$, $\mathcal{D}$ and $\mathcal{C}$ are as before, and

- $\mathcal{A} = \{1, \dots, p\}$ is a set of p agents,
- $\phi : \mathcal{X} \rightarrow \mathcal{A}$ is a function that maps each variable to its agent.

Each variable belongs to one agent. The distribution of variables divides $\mathcal{C}$ in two disjoint subsets, $\mathcal{C}_{intra} = \{C_i | \forall x_j, x_k \in var(C_i), \phi(x_j) = \phi(x_k)\}$, and $\mathcal{C}_{inter} = \{C_i | \exists x_j, x_k \in var(C_i), \phi(x_j) \neq \phi(x_k)\}$, called intraagent and interagent constraint sets, respectively. An intraagent constraint C_i is known by the agent owner of $var(C_i)$, and it is unknown by the other agents. Usually, it is considered that an interagent constraint C_j is known by every agent that owns a variable of $var(C_j)$ [13].

A *solution* of a *DisCSP* is an assignment of values to variables satisfying every constraint. *DisCSPs* are solved by the coordinated action of agents, which communicate by exchanging messages. It is assumed that the delay of a message is finite but random. For a given pair of agents, messages are delivered in the order they were sent. For simplicity, we assume that each agent owns exactly one variable, and the agent number is the variable index ($\forall x_i \in \mathcal{X}, \phi(x_i) = i$). Furthermore, we assume that all constraints are binary. A constraint C_{ij} indicates that it binds variables x_i and x_j .

2.2 Asynchronous Backtracking

ABT [12, 13] is the reference algorithm for asynchronous distributed constraint solving, with a role similar to backtracking in the centralized case. An *ABT* agent makes its own decisions, informs other agents about them, and no agent has to wait for the others' decisions. The algorithm computes a global consistent solution (or detects that no solution exists) in finite time; its correctness and completeness have been proved [2, 13]. *ABT* requires constraints to be directed. A binary constraint causes a directed link between the two constrained agents: the value-sending agent, from which the link starts, and the constraint-evaluating agent, at which the link ends. To make the network cycle-free, there is a total order among agents, which is followed by the directed links.

Each *ABT* agent keeps its own agent view and nogood store. The agent view of *self*, a generic agent, is the set of values that *self* believes are assigned to higher priority agents (connected to *self* by incoming links). Its nogood store keeps nogoods as justifications of inconsistent values. Agents exchange four message types:

- **ok?**: A high priority agent informs lower priority ones about its assignment.
- **ngd**: A lower priority agent inform a higher priority one of a new nogood.
- **addl**: A lower priority agent requests a higher priority one to set up a link.
- **stop**: The empty nogood has been generated. There is no solution.

When the algorithm starts, each agent assigns its variable, and sends the assignment to its neighboring agents with lower priority. When *self* receives an assignment, *self* updates its agent view with the new assignment, removes inconsistent nogoods and checks the consistency of its current assignment with the updated agent view.

When *self* receives a nogood, it is accepted if the nogood is consistent with *self*'s agent view (for the variables in the nogood, their values in the nogood and in *self*'s agent view are equal). Otherwise, *self* discards the nogood as obsolete. If the nogood is accepted, the nogood store is updated, causing *self* to search for a new consistent value (since the received nogood forbids its current value). If an unconnected agent i appears in the nogood, it is requested to set up a new link with *self*. From this point on, *self* will receive i values. When *self* cannot find any value consistent with its agent view, either because of the original constraints or because of the received nogoods, new nogoods are generated from its agent view and each one sent to the closest agent involved in it. This operation causes backtracking. There are several forms of how new nogoods are generated. In [2], when an agent has no consistent values, it resolves its nogoods following a procedure described in [1]. In this paper we consider this last version. The *ABT* code is in Figure 1 for the *self* agent. Γ_0^+ and Γ_0^- are the sets of agents initially constrained with *self* which are above and below it in the agent ordering.


```

procedure ABT()
 $\Gamma = \Gamma_0^- \cup \Gamma_0^+$ ;  $\Gamma^- = \Gamma_0^-$ ;  $\Gamma^+ = \Gamma_0^+$ ;  $myValue \leftarrow \text{empty}$ ;  $end \leftarrow \text{false}$ ; CheckAgentView();
while ( $\neg end$ ) do
   $msg \leftarrow \text{getMsg}()$ ;
  switch( $msg.type$ )
     $Ok?:ProcessInfo(msg)$ ;  $Ngd:Conflict(msg)$ ;  $Stop: end \leftarrow \text{true}$ ;  $AddL:SetLink(msg)$ ;

procedure CheckAgentView()
if  $\neg \text{consistent}(myValue, myAgentView)$  then
   $myValue \leftarrow \text{ChooseValue}()$ ;
  if ( $myValue$ ) then for each  $k \in \Gamma^+$  do  $sendMsg:Ok?(k, myValue)$ ; else Backtrack();

procedure ProcessInfo( $msg$ )
Update( $myAgentView, msg.assig$ ); CheckAgentView();

procedure Conflict( $msg$ )
if Coherent( $msg.nogood, \Gamma^- \cup \{self\}$ ) then
  CheckAddLink( $msg$ ); add( $msg.nogood, myNogoodStore$ );
   $myValue \leftarrow \text{empty}$ ; CheckAgentView();
else if Coherent( $msg.nogood, self$ ) then  $sendMsg:Ok?(msg.sender, myValue)$ ;

procedure SetLink( $msg$ )
add( $msg.sender, \Gamma^+$ );  $sendMsg:Ok?(msg.sender, myValue)$ ;

procedure CheckAddLink( $msg$ )
for each ( $var \in \text{lhs}(msg.nogood)$ )
  if ( $var \notin \Gamma^-$ ) then  $sendMsg:AddL(var, self)$ ; add( $var, \Gamma^-$ );
  Update( $myAgentView, msg.nogood[var]$ );

procedure Backtrack()
 $newNogood \leftarrow \text{solve}(myNogoodStore)$ ;
if ( $newNogood = \text{empty}$ ) then  $end \leftarrow \text{true}$ ;  $sendMsg:Stop(system)$ ;
else  $sendMsg:Ngd(newNogood)$ ; Update( $myAgentView, rhs(newNogood) \leftarrow \text{unknown}$ );
  CheckAgentView();

function ChooseValue()
for each  $v \in D(self)$  not eliminated by  $myNogoodStore$  do
  if consistent( $v, myAgentView$ ) then return ( $v$ ); else add( $nogood(v), myNogoodStore$ );
return ( $\text{empty}$ );

procedure Update( $myAgentView, newAssig$ )
add( $newAssig, myAgentView$ );
for each  $ng \in myNogoodStore$  do
  if  $\neg \text{Coherent}(\text{lhs}(ng), myAgentView)$  then remove( $ng, myNogoodStore$ );

function Coherent( $nogood, agents$ )
for each  $var \in nogood \cup agents$  do if  $nogood[var] \neq myAgentView[var]$  then return false;
return true;

```

Fig. 1. The ABT algorithm for asynchronous backtracking search

3 Propagating Unconditional Deletions

During search, ABT produces nogoods in *self* as result of the reception of **ok?** and **ngd** messages. A *nogood* is a conjunction of individual assignments, which has been found inconsistent, either because the initial constraints or because searching all possible combinations. For instance, the following nogood,

$$x_1 = a \wedge x_2 = b \wedge x_3 = c$$

means that these three assignments cannot happen simultaneously because they cause an inconsistency: either they violate a constraint or any extension including the remaining variables violates a constraint. Often, a nogood is written in *directed form*,

$$x_1 = a \wedge x_2 = b \Rightarrow x_3 \neq c$$

meaning that x_3 cannot take value c because of the values of x_1 and x_2 . In a directed nogood, " $\Rightarrow$ " separates the *left-hand side* (lhs) from the *right-hand side* (rhs). Since variables are ordered at each branch of the search tree, it is useful to write nogoods in a directed form, where the last variable in the branch order appears in the rhs. A nogood is a necessary *justification* to eliminate a value. In the previous example, the directed nogood is a justification to eliminate value c of $D(x_3)$. A nogood is *active* if the assignments in its lhs hold. To assure polynomial space usage, ABT only keeps one active nogood per eliminated value. As soon as a nogood becomes no active, it is removed (and the corresponding eliminated value is again available).

When all values of $D(\text{self})$ are eliminated by some nogood, the justifying nogoods are resolved generating a new nogood *ng* (see [1] for a detailed description of this process). *ng* is sent to *var*, the variable that appears in $\text{rhs}(\text{ng})$ (which always has the form $\text{var} \neq \text{val}$). This means that *val* can be eliminated from $D(\text{var})$, conditioned to the assignments of $\text{lhs}(\text{ng})$. It may happen that $\text{lhs}(\text{ng})$ is empty. In this case, *val* can be deleted from $D(\text{var})$ unconditionally, once and for all. After removal, *val* will never be available again: no matter which new assignments may be explored in the future, the empty $\text{lhs}(\text{ng})$ always holds.

An unconditional deletion may generate further unconditional deletions in other domains, if it happens that the initial deletion causes a constraint (or a subset of constraints) to become locally inconsistent, and the corresponding local consistency is enforced afterwards. In this paper, we only consider arc consistency, although other local consistencies removing single values could also be analyzed [6]. We assume that domains are initially arc consistent (if not, this can be easily done by a preprocess, explained below). If value a of variable x_i is deleted, this has to be notified to all agents connected with i . These agents will check their constraints with i to enforce arc consistency after a deletion (for instance, using the popular **revise-2001** function [3]). If more deletions occur, they are propagated in the same way, until reaching a fix point.

Values deleted in this way are removed once and for all. Let us assume three sequentially constrained agents i, j and k , $i < j < k$, connected by two constraints C_{ij} and C_{jk} , such that j receives a nogood from k eliminating unconditionally value b . If it happens that b was the only support for value $a \in D(x_i)$, after the deletion of b in $D(x_j)$, a must be deleted from $D(x_i)$ because a will not be in any solution. Value b will never be available again, so a would never have a support and its deletion is unconditional.

3.1 ABT-UAC

The idea of propagating unconditional deleted values can be included in ABT, producing the new algorithm ABT-UAC. It exploits the idea that a constraint C_{ij} is known by both agents i and j . ABT-UAC requires the some minor changes with respect to ABT:

- In addition to its own domain, the domain of every variable constrained with *self* is also represented in *self*. Assuming that a constraint between *self* and j does not contain irrelevant values, domain computation can be done by projecting the constraint on x_j . This constraint will be arc consistent after the preprocess.
- A new message type, **del**, is required. When *self* deletes value a in $D(\text{self})$, it sends a **del** message to every agent initially constrained with it, except the agent that sent the message that caused a deletion. When *self* receives a **del** message, it registers that the message value has been deleted from the domain of sender, and it enforces arc consistency on the constraint between *self* and sender. If, as result of this enforcing, some value is deleted in $D(\text{self})$ it is propagated as above.

Including the propagation of unconditionally deleted values does not changes the semantic of original ABT messages. It is worth noting that ABT-UAC keeps the good ABT properties, namely correctness, completeness and termination: since we are eliminating values which are unconditionally arc inconsistent, their removal will not cause to miss any solution. If the value assigned to *self* is found to be arc inconsistent, it is removed and another value is tried for *self*. Any value removal is propagated to agents initially constrained with *self*.

It is mentioned above that initial domains are assumed to be arc consistent. If not, this can be easily done by a preprocess depicted in Figure 2, executed on each agent. First, it initially enforces arc consistency between *self* and each constrained agent. Second, value deletions are propagated as described above, until reaching quiescence,

```

procedure AC-preprocess ()
  compute  $\Gamma_0 = \Gamma_0^- \cup \Gamma_0^+$ ;  $end \leftarrow \text{false}$ ; init structures of revise-2001
  for each  $j \in \Gamma_0$  do AC(self,  $j$ );
  while ( $\neg end$ ) do
     $msg \leftarrow \text{getMsg}()$ ;
    switch( $msg.type$ )
      Del: ValueDeletedPre( $msg.sender, msg.value$ );           Stop:  $end \leftarrow \text{true}$ ;

procedure ValueDeletedPre( $j, a$ )
   $D(j) \leftarrow D(j) - \{a\}$ ; AC(self,  $j$ );

procedure AC(self,  $j$ )
  if revise-2001(self,  $j$ ) then
    if  $D(\text{self}) = \emptyset$  then sendMsg:Stop(system);
    else DEL is the set of deleted values in  $D(\text{self})$  by the last revise-2001(self,  $j$ ) call
      for each  $v \in DEL$  and  $k \in \Gamma_0, k \neq j$  do sendMsg:Del(self,  $v$ );

```

Fig. 2. The AC algorithm for preprocessing DisCSP

```

procedure ABT-UAC ()
   $\Gamma = \Gamma_0^- \cup \Gamma_0^+$ ;  $\Gamma^- = \Gamma_0^-$ ;  $\Gamma^+ = \Gamma_0^+$ ;
   $myValue \leftarrow \text{empty}$ ;  $end \leftarrow \text{false}$ ; CheckAgentView();
  while ( $\neg end$ ) do
     $msg \leftarrow \text{getMsg}()$ ;
    switch( $msg.type$ )
       $Ok?: \text{ProcessInfo}(msg)$ ;  $Ngd: \text{Conflict}(msg)$ ;  $Stop: end \leftarrow \text{true}$ ;
  new    $AddL: \text{SetLink}(msg)$ ;  $Del: \text{ValueDeleted}(msg.sender, msg.value)$ ;

procedure Conflict( $msg$ )
  if Coherent( $msg.nogood, \Gamma^- \cup \{self\}$ ) then
new   if lhs( $msg.nogood$ ) =  $\text{empty}$  then DeleteValue( $myValue, msg.sender$ );
      else CheckAddLink( $msg$ ); add( $msg.nogood, myNogoodStore$ );
       $myValue \leftarrow \text{empty}$ ; CheckAgentView();
  else if Coherent( $msg.nogood, self$ ) then sendMsg:Ok?( $msg.sender, myValue$ );

new   procedure ValueDeleted( $j, a$ )
new    $D(j) \leftarrow D(j) - \{a\}$ ; AC( $self, j$ );
new   if  $myValue \notin D(self)$  then  $myValue \leftarrow \text{empty}$ ; CheckAgentView();

new   procedure DeleteValue( $a, j$ )
new    $D(self) \leftarrow D(self) - \{a\}$ ;
new   if  $D(self) = \emptyset$  then sendMsg:Stop(system);
new   else for each  $k \in \Gamma_0, k \neq j$  do sendMsg:Del( $self, a$ ); CheckAgentView();

procedure Backtrack()
   $newNogood \leftarrow \text{solve}(myNogoodStore)$ ;
  if ( $newNogood = \text{empty}$ ) then  $end \leftarrow \text{true}$ ; sendMsg:Stop(system);
  else sendMsg:Ngd( $newNogood$ ); Update( $myAgentView, rhs(newNogood) \leftarrow \text{ukn}$ );
new   if lhs( $newNogood$ ) =  $\text{empty}$  then ValueDeleted( $rhs(newNogood)$ );
  else CheckAgentView();

```

Fig. 3. New lines/procedures of ABT-UAC with respect to ABT

when ABT-UAC execution begins. Value deletions in the preprocessing phase are unconditional. Differences between ABT-UAC and ABT appear in Figure 3. They are,

- **ABT - UAC.** It includes the *Del* message, which notifies that a value has been deleted in some domain. Upon reception, the *ValueDeleted* procedure is called.
- **Conflict.** After accepting a *Ngd* message with empty lhs, the *DeleteValue* procedure is called.
- **ValueDeleted(j, a).** Agent j has deleted value a of its domain. $self$ registers this in its $D(j)$ copy, and enforces AC on the constraint between $self$ and j . If the value of $self$ is deleted in this process, the *CheckAgentView* procedure is called (looking for a new compatible value; if none exists performs backtracking). Any deletion in $D(self)$ is propagated.
- **DeleteValue(a, j).** Agent $self$ must delete its currently assigned value a because a nogood with empty lhs has been received from agent j . Value a is deleted

from $D(self)$. If, as consequence of a 's deletion, $D(self)$ becomes empty, there is no solution so a *Stop* message is produced. Otherwise, a 's deletion is notified to all agents constrained with $self$ except j via *Del* messages, and the procedure *CheckAgentView* is called.

- **Backtrack.** After $self$ computes and sends a *newNogood*, it checks if its lhs is empty. If so, $self$ knows that the value that forbids *newNogood* will be removed in the domain of the variable that appears in $rhs(newNogood)$. Therefore, $self$ calls *ValueDeleted*, as if it would have received a *Del* message.

3.2 Example

A simple example of the benefits of this approach appears in Figure 4. It is a graph coloring instance, with seven agents and the indicated domains. This instance has no solution (realize that there are two available values a, b for the clique formed by the agents 5, 6 and 7, mutually connected). We assume that agents are ordered lexicographically and values are tried in the order indicated for each domain. Agent 1 assigns $x_1 \leftarrow a$, to discover after a while that there is no solution with this assignment. A nogood with empty lhs will reach agent 1, forbidding value a . From this point on, ABT and ABT-UAC behave differently.

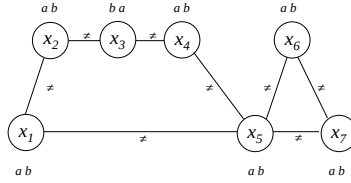


Fig. 4. Instance of graph coloring with 7 agents, each holding a variable. Domains are indicated

t/a	1	2	3	4	5	6	7
1	$x_1 \leftarrow b$ 1 ok? to x_2 1 ok? to x_5	$x_2 \leftarrow a$ 1 ok? to x_3	$x_3 \leftarrow b$ 1 ok? to x_4	$x_4 \leftarrow a$ 1 ok? to x_5	$x_5 \leftarrow a$ 2 ok? to x_6, x_7	$x_6 \leftarrow a$ 1 ok? to x_7	$x_7 \leftarrow a$
2					$x_1 = b \Rightarrow x_5 \neq b$ $x_4 = a \Rightarrow x_5 \neq a$ 1 ngd to x_4 $x_5 \leftarrow a$ 2 ok? to x_6, x_7	$x_6 \leftarrow b$ 1 ok? to x_7	$x_7 \leftarrow b$
3				$x_1 = b \Rightarrow x_4 \neq a$ $x_3 = b \Rightarrow x_4 \neq b$ 1 ngd to x_3 $x_4 \leftarrow b$ 1 ok? to x_5			$x_5 = a \Rightarrow x_7 \neq a$ $x_6 = b \Rightarrow x_7 \neq b$ 1 ngd to x_6 $x_7 \leftarrow b$
4			$x_1 = b \Rightarrow x_3 \neq b$ $x_2 = a \Rightarrow x_3 \neq a$ 1 ngd to x_2 $x_3 \leftarrow a$ 1 ok? to x_4			$x_5 = a \Rightarrow x_6 \neq a$ $x_5 = a \Rightarrow x_6 \neq b$ 1 ngd to x_5 $x_6 \leftarrow a$ 1 ok? to x_7	
5		$x_1 = b \Rightarrow x_2 \neq a$ $x_1 = b \Rightarrow x_2 \neq b$ 1 ngd to x_1 $x_2 \leftarrow a$ 1 ok? to x_3			$x_1 = b \Rightarrow x_5 \neq b$ $\Rightarrow x_5 \neq a$ 1 ngd to x_1 $x_5 \leftarrow a$ 2 ok? to x_6, x_7		
6	$\Rightarrow x_1 \neq a$ $\Rightarrow x_1 \neq b$ empty nogood stop				$x_6 \leftarrow b$ 1 ok? to x_7		

Fig. 5. Trace of ABT in the example, after discarding value a for x_1

time/agent	1	2	3	4	5	6	7
1	$D_1 = \{\neq b\}$ 2 del to x_2, x_5 $x_1 \leftarrow b$ 2 ok? to x_2, x_5	$x_2 \leftarrow a$ 1 ok? to x_3	$x_3 \leftarrow b$ 1 ok? to x_4	$x_4 \leftarrow a$ 1 ok? to x_5	$x_5 \leftarrow a$ 2 ok? to x_6, x_7	$x_6 \leftarrow a$ 1 ok? to x_7	$x_7 \leftarrow a$
2					$D_5 = \{a, \neq\}$ 2 del to x_6, x_7 $x_1 = b \Rightarrow x_5 \neq b$ $x_4 = a \Rightarrow x_5 \neq a$ 1 ngd to x_4 $x_5 \leftarrow a$ 2 ok? to x_6, x_7	$x_6 \leftarrow b$ 1 ok? to x_7	$x_7 \leftarrow b$
3				$x_1 = b \Rightarrow x_4 \neq a$ $x_3 = b \Rightarrow x_4 \neq b$ 1 ngd to x_3 $x_4 \leftarrow b$ 1 ok? to x_5		$D_6 = \{\neq b\}$ 1 del to x_7	$D_7 = \{\neq b\}$ 1 del to x_6 $x_5 = a \Rightarrow x_7 \neq a$ $x_6 = b \Rightarrow x_7 \neq b$ 1 ngd to x_6 $x_7 \leftarrow b$
4			$x_1 = b \Rightarrow x_3 \neq b$ $x_2 = a \Rightarrow x_3 \neq a$ 1 ngd to x_2 $x_3 \leftarrow a$ 1 ok? to x_4			$D_6 = \emptyset$ stop	$D_7 = \emptyset$ stop

Fig. 6. Trace of ABT-UAC in the example, after discarding value a for x_1

ABT behavior is summarized in Figure 5, while ABT-UAC behavior is summarized in Figure 6. Since tracing asynchronous algorithms is difficult, we assume that all messages sent in a time period are read in the next time period. The main difference between ABT and ABT-UAC is that the latter propagates unconditional deletions via **del** messages. As consequence of that, it detects two empty domains at period 4 (D_4 and D_5), so there is no solution. For this, it exchanges 23 messages. ABT performs just search and it requires 25 messages to deduce that the instance has no solution.

It is worth noting that the detection of empty domains by ABT-UAC is done by the unique action of **del** messages, and **ok?** and **ngd** messages are useless.

4 Propagating Any Deletion

The idea of propagating unconditional deletions can be extended to propagate any deletion, including conditional ones. A value is conditionally deleted when the reason for its removal is a nogood with a non-empty lhs. This value remains deleted as long as its justifying nogood is active. When this nogood becomes no active, it has to be removed and the value is available again. Propagating any deletion means that any deleted value and its justifying nogood of any agent has to be sent to any other agent constrained with it. This was already proposed in [9, 10]. Agents have to store the received nogoods while they are active, but the space complexity remains polynomial [9]. As in the previous case, ABT propagating any deletion remains sound, complete and terminates.

When a value $a \in D(self)$ is removed, we differentiate between,

- Unconditional deletion. Value a is removed when,
 1. a nogood with empty lhs has been accepted, or
 2. all values initially consistent with a in the domain of a constrained variable have been unconditionally eliminated.

Then, a is eliminated from $D(self)$ once and for all ($D(self) \leftarrow D(self) - \{a\}$).

- Conditional deletion. Value a is removed when,
 1. $self$ produces a nogood with non-empty lhs for a when looking for a consistent value for x_{self} ; it is the justification for the conditional a deletion, or

2. a nogood with non-empty lhs has been accepted; this nogood is the justification for the conditional a deletion, or
3. all values initially consistent with a in the domain of a constrained variable have been eliminated, and this removal is conditional for at least one of these values. The nogood of a deletion is the conjunction of the lhs of the nogoods of the conditionally removed values which were initially consistent with a .

Nogoods justifying deletions of values in $D(self)$ have to be sent to constrained agents, which will enforce arc consistency in their constraints with $self$. This may produce further deletions in the domains of those agent variables, which have to be propagated, etc. Each time a value is conditionally deleted, a nogood is added that justifies its deletion. When this nogood is no longer active, it has to be removed and the deleted value becomes available again. Because of that, if an agent stores a nogood, it must have direct link with all agents owners of the variables that appear in its lhs, to be notified if one of these variables changes its value (which could render the nogood no active). To perform propagation, $self$ has to send all nogoods of its values to all agents constrained with $self$. If ng is a nogood to propagate, it is sent to all constrained agents that are below the last variable (lv) in the static agent ordering of ABT agents, that appears in $lhs(ng)$. The reason is clear: if ng is sent to agent $k < lv(lhs(ng))$, agent k has no way to determine if ng is active or not, because there are variables in $lhs(ng)$ which are below k (so their values will never be sent to k).

Propagating any deletion has a clear drawback: the *huge* number of messages that should be exchanged. This large number of **del** messages may overcome the benefits of propagation, which are reduction of the search effort (because there are less available values) which causes a reduction in the number of **ok?** and **ngd** messages. To mitigate this drawback, we suggest to propagate any deletion directionally, following the ABT static order of agents. If a value is deleted in $self$, the **del** message goes to agents above $self$ in the ordering. If agent $j, j < self$ is constrained with $self$, upon the reception of **del** message, j enforces arc consistency in the constraint between j and $self$.

Assuming that the instance is initially arc consistent, directional arc consistency is maintained. These ideas are implemented in the ABT-DAC algorithm. This algorithm presents the following changes with respect to ABT-UAC:

- Agent $self$, in addition to store the nogoods for the values of $D(self)$, it has to store the nogoods for values of other agents. Because of that, *myNogoodStore* becomes a vector indexed by agent, *myNogoodStore*[k].
- Message **del** contains a nogood (instead of a pair (variable,value)).

Differences of ABT-DAC with ABT-UAC appear in Figure 7. In that code, when value a is unconditionally deleted, it is removed from its domain ($D \leftarrow D - \{a\}$). When a is conditionally deleted, a nogood justifying its deletion is added to the nogood store. Checking for empty domain ($D = \emptyset$) means if all values of D have been unconditionally removed. New lines are explained in the following,

- **ValueDeleted**(*msg*). Agent *msg.sender* has deleted a value of its domain with *msg.nogood*. First, $self$ checks if this message is up to date, by comparing $lhs(msg.nogood)$ with its *agentView*. If the message is accepted, then it

```

procedure ValueDeleted(msg)
  new if Coherent (msg.nogood,  $\Gamma^- \cup \{self\}$ ) then
  new   if lhs(msg.nogood) = empty then  $D(msg.sender) \leftarrow D(msg.sender) - \{a\}$ ;
  new   else add(msg.nogood, myNogoodStore[msg.sender]); CheckAddLink(msg);
  new   DAC(self, msg.sender);
  new   if myValue  $\notin D(self)$  then myValue  $\leftarrow$  empty; CheckAgentView();

procedure DeleteValue(a, j)
   $D(self) \leftarrow D(self) - \{a\}$ ;
  if  $D(self) = \emptyset$  then sendMsg:Stop(system);
  new else for each  $k \in \Gamma_0^-, k \neq j$  do sendMsg:Del("  $\Rightarrow x_{self} \neq a$ "); CheckAgentView();

procedure Backtrack()
  new newNogood  $\leftarrow$  solve(myNogoodStore[self]);
  if (newNogood = empty) then end  $\leftarrow$  true; sendMsg:Stop(system);
  else sendMsg:Ngd(newNogood); Update(myAgentView, rhs(newNogood)  $\leftarrow$  ukn);
  new ValueDeleted(newNogood);

  new procedure DAC(self, j)
  new if revise-2001(self, j) then
  new   if  $D(self) = \emptyset$  then sendMsg:Stop(system); /* empty by unconditional deletions */
  new   else DEL is the set of deleted values in  $D(self)$  by the last revise-2001(self, j) call
  new     for each  $a \in DEL$ , ng(a) justifies deletion,  $k \in \Gamma_0^-, k > lv(lhs(ng(a)))$  do
  new       sendMsg:Del(ng(a));

function ChooseValue ()
  new for each  $v \in D(self)$  not eliminated by myNogoodStore[self] do
  new   if consistent(v, myAgentView) then return (v);
  new   else add(nogood(v), myNogoodStore[self]);
  new   for each  $k \in \Gamma_0^-, k > lv(lhs(nogood(v)))$  do sendMsg:Del(nogood(v));
  new   return (empty);

procedure Update(myAgentView, newAssig)
  add(newAssig, myAgentView);
  new for each  $k \in \Gamma^+ \cup \{self\}$  do
  new   for each ng  $\in$  myNogoodStore[k] do
  new     if  $\neg$ Coherent(lhs(ng), myAgentView) then remove(ng, myNogoodStore[k]);

```

Fig. 7. New lines/procedures for ABT-DAC, with respect to ABT-UAC. Deletions are directionally propagated with DAC.

differentiates between unconditional or conditional deletion. In the first case, the value is removed from $D(msg.sender)$. Otherwise, the nogood is stored and analyzed for possible new links. In both cases, directed arc consistency between *self* and *msg.sender* is enforced. If the value of *self* is deleted in this process, the CheckAgentView procedure is called (looking for a new compatible value; if none exists performs backtracking). Any deletion in $D(self)$ is directionally propagated.

- **DeleteValue**(a, j). The only difference with the same procedure of ABT-UAC is in the last line, where an unconditional deletion is propagated. The new format of **del** message requires to form a directed nogood.
- **Backtrack**. Differences are in the first and last lines. In the first line, because *myNogoodStore* is now a vector, *myNogoodStore*[*self*] is solved, instead of *myNogoodStore*. In the last line, the **ValueDeleted** procedure is always called.
- **DAC**(*self*, *j*). Arc consistency is enforced in the constraint between *self* and *j*. If, as result of this enforcing, *D*(*self*) becomes empty (all values have been unconditionally deleted), the problem has no solution, stop. Otherwise, any deletion is propagated to agents above *self* and below *lv(lhs)* of the nogood sent.
- **ChooseValue**(*self*). A new value for *self*, consistent with *myAgentView*, is fetched. If a particular value is not consistent with *myAgentView* (because constraints with other agents), a nogood justifying its conditional deletion is computed, stored and directionally propagated.
- **Update**(*myAgentView*, *newAssig*). *myNogoodStore*[*k*], now a vector indexed by agent, causes the only difference. To remove nogoods which may become no active by *newAssig*, all stored nogoods of agents below *self* should be checked.

It may occur that a **del** message includes a nogood more up to date than the agent view of the receiving agent. In that case, [9] used a time-stamp system to determine which message was earlier, and how to update correctly the agent view. We take a simpler approach here: if this happens, the **del** message is considered obsolete and discarded (see **ValueDeleted**).

5 Experimental Results

We experimentally evaluate the performance of ABT-UAC and ABT-DAC algorithms with respect to ABT on uniform binary random *DisCSP*. A binary random *DisCSP* class is characterized by $\langle n, d, p_1, p_2 \rangle$, where n is the number of variables, d the number of values per variable, p_1 the network *connectivity* defined as the ratio of existing constraints, and p_2 the constraint *tightness* defined as the ratio of forbidden value pairs. The constrained variables and the forbidden value pairs are randomly selected [11]. A problem class will be referred to as a $\langle n, d, p_1, p_2 \rangle$ network. Each agent is assigned one variable. Neighboring agents are connected by constraints.

Using this model, we have tested random instances on four sets of experiments. The first three ones consist of instances of 16 agents and 8 values per agent, considering three connectivity classes, sparse ($p_1 = .2, p_2 = .7$), medium ($p_1 = .5, p_2 = .4$) and dense ($p_1 = .8, p_2 = .3$) while the four experiment considers instances of 50 agents and 50 values per agent, considering two dense connectivity classes, ($p_1 = 1, p_2 = .875, p_1 = .7, p_2 = .8$). Tightness were selected at the complexity peak, where the differences among algorithms are more explicit. Tables 1 and 2 present the results of the algorithms according to two parameters: the communication cost, in terms of the number of messages, and the computation effort, in terms of the number of non concurrent constraint checks (*necc*) [7], respectively. In addition to these parameters, we also report the number messages sent for each message type, the number of unconditionally deleted values and the number obsolete messages (obsolete **ngd** for ABT and

Table 1. Results of ABT, ABT-UAC and ABT-DAC on random instances of 16 agents, 8 values per agent and sparse, medium and dense connectivities

	<i>alg</i>	#ok?	#ngd	#addl	#del	#msg	#del-val	#obsolete	nccc
$p_1 = .2$	ABT	4,866	1,829	27	0	6,722	0	446	5,689
	ABT-UAC	1,566	573	19	154	2,312	66	142	2,502
	ABT-DAC	2,983	1,132	26	1,938	6,080	22	653	57,785
$p_1 = .5$	ABT	28,055	8,070	39	0	36,164	0	2,702	39,923
	ABT-UAC	27,487	7,902	39	125	35,553	19	2,642	40,039
	ABT-DAC	28,042	8,094	39	11,281	47,456	3	5,138	345,933
$p_1 = .8$	ABT	53,400	14,999	19	0	68,418	0	5,909	98,330
	ABT-UAC	52,304	14,670	19	805	67,798	76	5,783	101,948
	ABT-DAC	53,238	14,932	19	21,484	89,674	7	11,049	670,803

Table 2. Results of ABT, ABT-UAC and ABT-DAC on random instances of 50 agents, 50 values per agent and two connectivities

	<i>alg</i>	#ok?	#ngd	#addl	#del	#msg	#del-val	#obsol	nccc
$p_1 = 1$	ABT	267,046	153,909	0	0	420,955	0	100,632	557,242
	ABT-UAC	44,312	25,378	0	45,640	115,331	1,130	16,277	1,708,130
	ABT-DAC	66,596	38,270	0	124,931	229,797	499	65,762	131,596,664
$p_1 = .7$	ABT	1,191,937	468,069	301	0	1,660,307	0	321,994	1,876,059
	ABT-UAC	1,130,235	443,916	301	47,954	1,622,407	1,587	305,807	4,535,344
	ABT-DAC	210,963	87,129	275	277,351	575,718	52	151,288	391,778,623

ABT-UAC, obsolete **ngd** plus obsolete **del** for ABT-DAC). All parameters are averaged over 50 executions.

Table 1 presents the first three sets of experiments for random instances of 16 agents and 8 values per agent. The upper set corresponds to the sparse instances. Regarding the number of messages, we observe that ABT-UAC and ABT-DAC always dominate the standard ABT. ABT-UAC is the algorithm that shows the best results, reducing approximately three times the number of **ok?**, **ngd** and total messages sent. As expected, ABT-UAC sends a lower number of arc consistent messages than ABT-DAC. However, ABT-UAC discards more unconditional arc inconsistent values than ABT-DAC (since ABT-UAC and ABT-DAC perform different propagations of different deletions, they may cause different numbers of removed values). Regarding message obsolescence, results show that agents in ABT-UAC are better informed about others' assignments than agents in ABT.

The second and third sets of experiments in the same table correspond to medium and dense connected binary instances of 16 agents and 8 values. In both cases the tightness of the constraints is low at the complexity peak. Therefore, there is a little propagation to reach an arc consistent state. Although the impact of maintaining arc consistent domains on ABT is minor, ABT-UAC is always more economic than ABT with respect to the total number of messages. In contrast, ABT-DAC sends more messages than ABT.

Considering the three experiments, propagating deletions algorithms send **del** messages, which cause to delete some values. These deletions cause to diminish the search

effort, decreasing the number of **ok?** and **ngd** messages exchanged. When the number of saved **ok?** and **ngd** messages is larger than the number of **del** messages, propagation pays off and causes an overall message decrement. However, if the number of saved **ok?** and **ngd** messages is smaller than the number of **del** messages, propagation is harmful. In the sparse class, both ABT-UAC and ABT-DAC are beneficial, while for the medium and dense classes only ABT-UAC is beneficial while ABT-DAC is harmful. In these two classes, the number of **ok?** and **ngd** is practically the same for ABT and ABT-DAC, so the effect of propagation is practically unnoticed. In terms of *nccc*, propagating deletions algorithms are clearly more costly than ABT, since they perform full or directed arc consistency, which implies more constraint checks. ABT-DAC is always more costly than ABT-UAC because it performs more effort, propagating also conditional deletions.

Since the decrement in the number of messages caused by ABT-UAC in the medium and dense connectivity classes of 16 agents and 8 values is minor, one might think that the proposed approach is not beneficial on any medium or dense classes. To evaluate this hypothesis, we have performed the fourth set of experiments for random instances of 50 agents and 50 values per agent, with $p_1 = 1$ and $p_1 = 0.7$. Results appear in Table 2. Regarding communication cost, results of $p_1 = 1$ show a significant improvement of ABT-UAC with respect to ABT: the number of messages it sends is 3.6 times lower than ABT. We can note larger gains in the number of messages for each ABT message type. We observe that ABT-DAC also needs lower number of messages than ABT, even when it discards less than the half of the arc inconsistent values that ABT-UAC. Results for $p_1 = .7$ show that the winner here is ABT-DAC, requiring a number of messages that divides by 2.9 the number required by ABT. ABT-UAC shows some minor improvements. Regarding computation effort, once again *nccc* reflect the high local effort that agents must pay in order to have consistent domains.

6 Conclusions

From this work we can extract some conclusions. According to experimental results, propagation of unconditional deletions is not harmful, and it provides substantial benefits for some problem instances, reducing substantially ABT communication requirements among agents. Directional propagation of any deletion provides a less clear picture: it can be harmful in some instances, but also beneficial in others. More experimental work is needed to assess their relative importance in different problem classes.

As future work, many ideas remain to be explored. On one hand, it has to be found the right degree of arc consistency when propagating any deletion. In more general terms, other local consistencies that remove individual values [6] could replace arc consistency in the proposed approach; it remains to analyze how this can be done and their cost. On the other hand, the proposed approach could be combined with other strategies that improve ABT efficiency, like dynamic variable ordering [14, 15], or the hybrid ABT version [4]. Finally, privacy deserves a special mention. Obviously, the proposed approach is less private than ABT, since deleted values (and the reasons for their deletion) are exchanged among agents. How privacy could be improved inside the framework of the proposed approach is an open question for further research.

Acknowledgements

Authors sincerely thank reviewers for their comments; they helped us to make a better paper.

References

1. Baker, A.B.: The hazards of fancy backtracking. In: Proc. of AAAI 1994, pp. 288–293 (1994)
2. Bessiere, C., Brito, I., Maestre, A., Meseguer, P.: The Asynchronous Backtracking without adding links: a new member in the ABT family. *Artificial Intelligence* 161, 7–24 (2005)
3. Bessiere, C., Regin, J.C.: Refining the basic constraint propagation algorithm. In: Proc. IJCAI 2001, pp. 309–315 (2001)
4. Brito, I., Meseguer, P.: Improving ABT performance by adding synchronization points. In: Recent Advances in Constraints (in press, 2008)
5. Dechter, R., Pearl, J.: Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence* 34, 1–38 (1988)
6. Debruyne, R., Bessiere, C.: Domain filtering consistencies. *JAIR* 14, 205–230 (2001)
7. Meisels, A., Kaplansky, E., Razgon, I., Zivan, R.: Comparing Performance of Distributed Constraint Processing Algorithms. In: AAMAS Workshop on Distributed Constr. Reas., pp. 86–93 (2002)
8. Silaghi, M.C., Sam-Haroud, D., Faltings, B.: Asynchronous Search with Aggregations. In: Proc. of AAAI 2000, pp. 917–922 (2000)
9. Silaghi, M.C., Sam-Haroud, D., Faltings, B.: Consistency Maintenance for ABT. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 271–285. Springer, Heidelberg (2001)
10. Silaghi, M.C., Sam-Haroud, D., Faltings, B.: Asynchronous consistency and maintenance in distributed constraint satisfaction. *Artificial Intelligence* 161, 25–53 (2005)
11. Smith, B.: Phase Transition and the Mushy Region in Constraint Satisfaction Problems. In: Proc. of the 11th ECAI, pp. 100–104 (1994)
12. Yokoo, M., Durfee, E., Ishida, T., Kuwabara, K.: Distributed Constraint Satisfaction for Formalizing Distributed Problem Solving. In: Proc. of the 12th. DCS, pp. 614–621 (1992)
13. Yokoo, M., Durfee, E., Ishida, T., Kuwabara, K.: The Distributed Constraint Satisfaction Problem: Formalization and Algorithms. *IEEE Trans. Know. and Data Engin.* 10, 673–685 (1998)
14. Zivan, R., Meisels, A.: Dynamic ordering for asynchronous backtracking on DisCSPs. *Constraints* 11, 179–197 (2006)
15. Zivan, R., Zazone, M., Meisels, A.: Min-domain ordering for asynchronous backtracking. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 758–772. Springer, Heidelberg (2007)

Elicitation Strategies for Fuzzy Constraint Problems with Missing Preferences: Algorithms and Experimental Studies

Mirco Gelain¹, Maria Silvia Pini¹, Francesca Rossi¹, K. Brent Venable¹,
and Toby Walsh²

¹ Dipartimento di Matematica Pura ed Applicata, Università di Padova, Italy
{mgelain,mpini,frossi,kvenable}@math.unipd.it

² NICTA and UNSW Sydney, Australia
Toby.Walsh@nicta.com.au

Abstract. Fuzzy constraints are a popular approach to handle preferences and over-constrained problems in scenarios where one needs to be cautious, such as in medical or space applications. We consider here fuzzy constraint problems where some of the preferences may be missing. This models, for example, settings where agents are distributed and have privacy issues, or where there is an ongoing preference elicitation process. In this setting, we study how to find a solution which is optimal irrespective of the missing preferences. In the process of finding such a solution, we may elicit preferences from the user if necessary. However, our goal is to ask the user as little as possible. We define a combined solving and preference elicitation scheme with a large number of different instantiations, each corresponding to a concrete algorithm which we compare experimentally. We compute both the number of elicited preferences and the "user effort", which may be larger, as it contains all the preference values the user has to compute to be able to respond to the elicitation requests. While the number of elicited preferences is important when the concern is to communicate as little information as possible, the user effort measures also the hidden work the user has to do to be able to communicate the elicited preferences. Our experimental results show that some of our algorithms are very good at finding a necessarily optimal solution while asking the user for only a very small fraction of the missing preferences. The user effort is also very small for the best algorithms. Finally, we test these algorithms on hard constraint problems with possibly missing constraints, where the aim is to find feasible solutions irrespective of the missing constraints.

1 Introduction

Constraint programming is a powerful paradigm for solving scheduling, planning, and resource allocation problems. A problem is represented by a set of variables, each with a domain of values, and a set of constraints. A solution is an assignment of values to the variables which satisfies all constraints and which optionally maximizes/minimizes an objective function. Soft constraints are a way to model optimization problems by allowing for several levels of satisfiability, modelled by the use of preference or cost values that represent how much we like an instantiation of the variables of a constraint.

It is usually assumed that the data (variables, domains, (soft) constraints) is completely known before solving starts. This is often unrealistic. In web applications and multi-agent systems, the data is frequently only partially known and may be added to at a later date by, for example, elicitation. Data may also come from different sources at different times. In multi-agent systems, agents may release data reluctantly due to privacy concerns.

Incomplete soft constraint problems can model such situations by allowing some of the preferences to be missing. An algorithm has been proposed and tested to solve such incomplete problems [7]. The goal is to find a solution that is guaranteed to be optimal irrespective of the missing preferences, eliciting preferences if necessary until such a solution exists. Two notions of optimal solution are considered: *possibly optimal* solutions are assignments that are optimal in *at least one way* of revealing the unspecified preferences, while *necessarily optimal* solutions are assignments that are optimal in *all ways* that the unspecified preferences can be revealed. The set of possibly optimal solutions is never empty, while the set of necessarily optimal solutions can be empty.

If there is no necessarily optimal solution, the algorithm proposed in [7] uses branch and bound to find a “promising solution” (specifically, a complete assignment in the best possible completion of the current problem) and elicits the missing preferences related to this assignment. This process is repeated till there is a necessarily optimal solution.

Although this algorithm behaves reasonably well, it make some specific choices about solving and preference elicitation that may not be optimal in practice, as we shall see in this paper. For example, the algorithm only elicits missing preferences after running branch and bound to exhaustion. As a second example, the algorithm elicits all missing preferences related to the candidate solution. Many other strategies are possible. We might elicit preferences at the end of every complete branch, or even at every node in the search tree. Also, when choosing the value to assign to a variable, we might ask the user (who knows the missing preferences) for help. Finally, we might not elicit all the missing preferences related to the current candidate solution. For example, we might just ask the user for the worst preference among the missing ones.

In this paper we consider a general algorithm scheme which greatly generalizes that proposed in [7]. It is based on three parameters: *what* to elicit, *when* to elicit it, and *who* chooses the value to be assigned to the next variable. We test all 16 possible different instances of the scheme (among which is the algorithm in [7]) on randomly generated fuzzy constraint problems. We demonstrate that some of the algorithms are very good at finding necessarily optimal solution without eliciting too many preferences. We also test the algorithms on problems with hard constraints. Finally, we consider problems with fuzzy temporal constraints, where problems have more specific structure.

In our experiments, we compute the elicited preferences, that is, the missing values that the user has to provide to the system because they are requested by the algorithm. Providing these values usually has a cost, either in terms of computation effort, or in terms of privacy decrease, or also in terms of communication bandwidth. Thus knowing how many preferences are elicited is important if we care about any of these issues. However, we also compute a measure of the user’s effort, which may be larger than the number of elicited preferences, as it contains all the preference values the user has to consider to be able to respond to the elicitation requests. For example, we may ask the user for the worst preference value among k missing ones: the user will communicate

only one value, but he will have to consider all k of them. While knowing the number of elicited preferences is important when the concern is to communicate as little information as possible, the user effort measures also the hidden work the user has to do to be able to communicate the elicited preferences. This user's effort is therefore also an important measure.

As a motivating example, recommender systems give suggestions based on partial knowledge of the user's preferences. Our approach could improve performance by identifying some key questions to ask before giving recommendations. Privacy concerns regarding the percentage of elicited preferences are motivated by eavesdropping. User's effort is instead related to the burden on the user.

Our results show that the choice of preference elicitation strategy is crucial for the performance of the solver. While the best algorithms need to elicit as little as 10% of the missing preferences, the worst one needs much more. The user's effort is also very small for the best algorithms. The performance of the best algorithms shows that we only need to ask the user a very small amount of additional information to be able to solve problems with missing data.

Several other approaches have addressed similar issues. For example, open CSPs [4,6] and interactive CSPs [9] work with domains that can be partially specified. As a second example, in dynamic CSPs [2] variables, domains, and constraints may change over time. However, the incompleteness considered in [5,6] is on domain values as well as on their preferences. Working under this assumption means that the agent that provides new values/costs for a variable knows all possible costs, since they are capable of providing the best value first. If the cost computation is expensive or time consuming, then computing all such costs (in order to give the most preferred value) is not desirable. We assume instead, as in [7], that all values are given at the beginning, and that only some preferences are missing. Because of this assumption, we don't need to elicit preference values in order, as in [6].

2 Background

In this section we give a brief overview of the fundamental notions and concepts on Soft Constraints and Incomplete Soft Constraints.

Incomplete Soft Constraints problems (ISCSPs) [7] extend Soft Constraint Problems (SCSPs) [1] to deal with partial information. We will focus on a specific instance of this framework in which the soft constraints are fuzzy.

Given a set of variables V with finite domain D , an *incomplete fuzzy constraint* is a pair $\langle \text{idef}, \text{con} \rangle$ where $\text{con} \subseteq V$ is the scope of the constraint and $\text{idef} : D^{|\text{con}|} \rightarrow [0, 1] \cup \{?\}$ is the preference function of the constraint associating to each tuple of assignments to the variables in con either a preference value ranging between 0 and 1, or ?. All tuples mapped into ? by idef are called *incomplete tuples*, meaning that their preference is unspecified. A fuzzy constraint is an incomplete fuzzy constraint with no incomplete tuples.

An *incomplete fuzzy constraint problem* (IFCSP) is a pair $\langle C, V, D \rangle$ where C is a set of incomplete fuzzy constraints over the variables in V with domain D . Given an IFCSP P , $IT(P)$ denotes the set of all incomplete tuples in P . When there are no incomplete tuples, we will denote a fuzzy constraint problem by FSCP.

Given an IFCSP P , a *completion* of P is an IFCSP P' obtained from P by associating to each incomplete tuple in every constraint an element in $[0, 1]$. A completion is *partial* if some preference remains unspecified. $C(P)$ denotes the set of all possible completions of P and $PC(P)$ denotes the set of all its partial completions.

Given an assignment s to all the variables of an IFCSP P , $pref(P, s)$ is the preference of s in P , defined as $pref(P, s) = \min_{\langle idef, con \rangle \in C \mid idef(s \downarrow_{con}) \neq ? idef(s \downarrow_{con})}$. It is obtained by taking the minimum among the known preferences associated to the projections of the assignment, that is, of the appropriated sub-tuples in the constraints.

In the fuzzy context, a complete assignment of values to all the variables is an optimal solution if its preference is maximal. The optimality notion of FCSPs is generalized to IFCSPs via the notions of *necessarily and possibly optimal solutions*, that is, complete assignments which are maximal in all or some completions. Given an IFCSP P , we denote by $NOS(P)$ (resp., $POS(P)$) the set of necessarily (resp., possibly) optimal solutions of P . Notice that $NOS(P) \subseteq POS(P)$. Moreover, while $POS(P)$ is never empty, $NOS(P)$ may be empty. In particular, $NOS(P)$ is empty whenever the revealed preferences do not fix the relationship between one assignment and all others.

In [7] an algorithm is proposed to find a necessarily optimal solution of an IFCSP based on a characterization of $NOS(P)$ and $POS(P)$. This characterization uses the preferences of the optimal solutions of two special completions of P , namely the 0-completion of P , denoted by P_0 , obtained from P by associating preference 0 to each tuple of $IT(P)$, and the 1-completion of P , denoted by P_1 , obtained from P by associating preference 1 to each tuple of $IT(P)$. Notice that, by monotonicity of $\min$, we have that $pref_0 \leq pref_1$. When $pref_0 = pref_1$, $NOS(P) = Opt(P_0)$; thus, any optimal solution of P_0 is a necessary optimal solution. Otherwise, $NOS(P)$ is empty and $POS(P)$ is a set of solutions with preference between $pref_0$ and $pref_1$ in P_1 . The algorithm proposed in [7] finds a necessarily optimal solution of the given IFCSP by interleaving the computation of $pref_0$ and $pref_1$ with preference elicitation steps, until the two values coincide. Moreover, the preference elicitation is guided by the fact that only solutions in $POS(P)$ can become necessarily optimal. Thus, the algorithm only elicits preferences related to optimal solutions of P_1 .

3 A General Solver Scheme

We now propose a more general schema for solving IFCSPs based on interleaving branch and bound (BB) search with elicitation. This schema generalizes the concrete solver presented in [7], but has several other instantiations that we will consider and compare experimentally in this paper. The scheme uses branch and bound. This considers the variables in some order, choosing a value for each variable, and pruning branches based on an upper bound (assuming the goal is to maximize) on the preference value of any completion of the current partial assignment. To deal with missing preferences, branch and bound is applied to both the 0-completion and the 1-completion of the problem. If they have the same solution, this is a necessarily optimal solution and we can stop. If not, we elicit some of the missing preferences and continue branch and bound on the new 1-completion.

Preferences can be elicited after each run of branch and bound (as in [7]) or during a BB run while preserving the correctness of the approach. For example, we can elicit

preferences at the end of every complete branch (that is, regarding preferences of every complete assignment considered in the branch and bound algorithm), or at every node in the search tree (thus considering every partial assignment). Moreover, when choosing the value for the next variable to be assigned, we can ask the user (who knows the missing preferences) for help. Finally, rather than eliciting all the missing preferences in the possibly optimal solution, or the complete or partial assignment under consideration, we can elicit just one of the missing preferences. For example, with fuzzy constraint problems, eliciting just the worst preference among the missing ones is sufficient since only the worst value is important to the computation of the overall preference value. More precisely, the algorithm schema we propose is based on the following parameters:

1. **Who** chooses the value of a variable: the algorithm can choose the values in decreasing order either w.r.t. their preference values in the 1-completion (Who=dp) or in the 0-completion (Who=dpi). Otherwise, the user can suggest this choice. To do this, he can consider all the preferences (revealed or not) for the values of the current variable (*lazy user*, Who=lu for short); or he considers also the preference values in constraints between this variable and the past variables in the search order (*smart user*, Who=su for short).
2. **What** is elicited: we can elicit the preferences of all the incomplete tuples of the current assignment (What=all) or only the worst preference in the current assignment, if it is worse than the known ones (What=worst);
3. **When** elicitation takes place: we can elicit preferences at the end of the branch and bound search (When=tree), or during the search, when we have a complete assignment to all variables (When=branch) or whenever a new value is assigned to a variable (When=node).

By choosing a value for each of the three above parameters in a consistent way, we obtain in total 16 different algorithms, as summarized in Figure 1, where the circled instance is the concrete solver used in [7].

Figures 2 and 3 show the pseudo-code of the general scheme for solving IFCSPs. There are three algorithms: ISCSP-SCHEME, BBE and BB. ISCSP-SCHEME takes as input an IFCSP P and the values for the three parameters: Who, What and When.

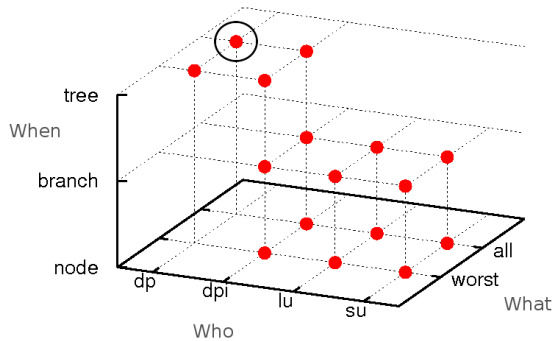


Fig. 1. Instances of the general scheme

```

IFCSP-SCHEME( $P, Who, What, When$ )
 $Q \leftarrow P_0$ 
 $s_{max}, pref_{max} \leftarrow BB(P_0, -)$ 
 $Q', s_1, pref_1 \leftarrow BBE(P, 0, Who, What, When, s_{max}, pref_{max})$ 
If ( $s_1 \neq nil$ )
     $s_{max} \leftarrow s_1, pref_{max} \leftarrow pref_1, Q \leftarrow Q'$ 
Return  $Q, s_{max}, pref_{max}$ 

```

Fig. 2. Algorithm IFCSP-SCHEME

```

BBE( $P, nInstVar, Who, What, When, sol, lb$ )
 $sol' \leftarrow sol, pref' \leftarrow lb$ 
 $currentVar \leftarrow nextVariable(P_1)$ 
While ( $nextValue(currentVar, Who)$ )
    If ( $When = node$ )
         $P, pref \leftarrow Elicit@Node(What, P, currentVar, lb)$ 
         $ub \leftarrow UpperBound(P_1, currentVar)$ 
        If ( $ub > lb$ )
            If ( $nInstvar = number\ of\ variables\ in\ P$ )
                If ( $When = branch$ )
                     $P, pref \leftarrow Elicit@branch(What, P, lb)$ 
                If ( $pref > lb$ )
                     $sol \leftarrow getSolution(P_1)$ 
                     $lb \leftarrow pref(P_1, sol)$ 
            else
                 $BBE(P, nInstVar + 1, Who, What, When, sol, lb)$ 
    If ( $When = tree\ and\ nInstVar = 0$ )
        If ( $sol = nil$ )
             $sol \leftarrow sol', pref \leftarrow pref'$ 
        else
             $P, pref \leftarrow Elicit@tree(What, P, sol, lb)$ 
            If ( $pref > pref'$ )
                 $BBE(P, 0, Who, What, When, sol, pref)$ 
            else  $BBE(P, 0, Who, What, When, sol', pref')$ 

```

Fig. 3. Algorithm BBE

It returns a partial completion of P that has some necessarily optimal solutions, one of these necessarily optimal solutions, and its preference value. It starts by computing via branch and bound (algorithm BB) an optimal solution of P_0 , say s_{max} , and its preference $pref_{max}$. Next, procedure BBE is called. If BBE succeeds, it returns a partial completion of P , say Q , one of its necessarily optimal solutions, say s_1 , and its associated preference $pref_1$. Otherwise, it returns a solution equal to nil . In the first case the output of IFCSP-SCHEME coincides with that of BBE , otherwise IFCSP-SCHEME returns P_0 , one of its optimal solutions, and its preference.

Procedure BBE takes as input the same values as IFCSP-SCHEME and, in addition, a solution sol and a preference lb representing the current lower bound on the optimal

preference value. Function *nextVariable*, applied to the 1-completion of the IFCSP, returns the next variable to be assigned. The algorithm then assigns a value to this variable. If the Boolean function *nextValue* returns true (if there is a value in the domain), we select a value for *currentVar* according to the value of parameter *Who*.

Function *UpperBound* computes an upper bound on the preference of any completion of the current partial assignment: the minimum over the preferences of the constraints involving only variables that have already been instantiated.

If When=tree, elicitation is handled by procedure *Elicit@tree*, and takes place only at the end of the search over the 1-completion. The user is not involved in the value assignment steps within the search. At the end of the search, if a solution is found, the user is asked either to reveal all the preferences of the incomplete tuples in the solution (if What=all), or only the worst one among them (if What=worst). If such a preference is better than the best found so far, BBE is called recursively with the new best solution and preference.

If When=branch, BB is performed only once. The user may be asked to choose the next value for the current variable being instantiated. Preference elicitation, which is handled by function *Elicit@branch*, takes place during search, whenever all variables have been instantiated and the user can be asked either to reveal the preferences of all the incomplete tuples in the assignment (What=all), or the worst preference among those of the incomplete tuples of the assignment (What=worst). In both cases the information gathered is sufficient to test such a preference value against the current lower bound.

If When=node, preferences are elicited every time a new value is assigned to a variable and it is handled by procedure *Elicit@node*. The tuples to be considered for elicitation are those involving the value which has just been assigned and belonging to constraints between the current variable and already instantiated variables. If What=all, the user is asked to provide the preferences of all the incomplete tuples involving the new assignment. Otherwise if What=worst, the user provides only the preference of the worst tuple.

Theorem 1. *Given an IFCSP P and a consistent set of values for parameters When, What and Who, Algorithm IFCSP-SCHEME always terminates, and returns an IFCSP $Q \in PC(P)$, an assignment $s \in NOS(Q)$, and its preference in Q .*

Proof. Let us first notice that, as far as correctness and termination concern, the value of parameter Who is irrelevant.

We consider two separate cases, i.e., When=tree and When=branch or node.

Case 1: When =tree.

Clearly IFCSP-SCHEME terminates if and only if BBE terminates. If we consider the pseudocode of procedure BBE shown in Algorithm 3, we see that if When = tree, BBE terminates when $sol = nil$. This happens only when the search fails to find a solution of the current problem with a preference strictly greater than the current lower bound. Let us denote with Q^i and Q^{i+1} respectively the IFCSPs given in input to the i -th and $i+1$ -th recursive call of BBE. First we notice that only procedure *Elicit@tree* modifies the IFCSP in input by possibly adding new elicited preferences. Moreover, whatever the value of parameter What is, the returned IFCSP is either the same as the one in input or it is a (possibly partial) completion of the one in input. Thus we have $Q^{i+1} \in PC(Q^i)$ and

$Q^i \in PC(P)$. Since the search is always performed on the 1-completion of the current IFCSP, we can conclude that for every solution s , $pref(Q^{i+1}, s) \leq pref(Q^i, s)$. Let us now denote with lb^i and lb^{i+1} the lower bounds given in input respectively to the i -th and $i+1$ -th recursive call of BBE. It is easy to see that $lb^{i+1} \geq lb^i$. Thus, since at every iteration we have that the preferences of solutions can only get lower, and the bound can only get higher, and since we have a finite number of solutions, we can conclude that BBE always terminates.

The reasoning that follows relies on the fact that value $pref$ returned by function *Elicit@tree* is the final preference after elicitation of assignment sol given in input. This is true since either What = all and thus all preferences have been elicited and the overall preference of sol can be computed or only the worst preference has been elicited but in a fuzzy context where the overall preference coincide with the worst one. If called with When = tree IFCSP-SCHEME exits when the last branch and bound search has ended returning $sol = nil$. In such a case sol and $pref$ are updated to contain the best solution and associated preference found so far, i.e., sol' and $pref'$. Then, the algorithm returns the current IFCSP, say Q , and sol and $pref$. Following the same reasoning as above done for Q^i we can conclude that $Q \in PC(P)$.

At the end of every while loop execution, assignment sol either contains an optimal solution sol of the 1-completion of the current IFCSP or $sol = nil$. $sol = nil$ iff there is no assignment with preference higher than lb in the 1-completion of the current IFCSP. In this situation, sol' and $pref'$ are an optimal solution and preference of the 1-completion of the current IFCSP. However, since the preference of sol' , $pref'$ is independent of unknown preferences and since due to monotonicity the optimal preference value of the 1-completion is always greater than or equal to that of the 0-completion we have that sol' and $pref'$ are an optimal solution and preference of the 0-completion of the current IFCSP as well.

By Theorems 1 and 2 of [7] we can conclude that $NOS(Q)$ is not empty. If $pref = 0$, then $NOS(Q)$ contains all the assignments and thus also sol . The algorithm correctly returns the same IFCSP given in input, assignment sol and its preference $pref$. If instead $0 < pref$, again the algorithm is correct, since by Theorem 1 of [7] we know that $NOS(Q) = Opt(Q_0)$, and we have shown that $sol \in Opt(Q_0)$.

Case 2: When=branch or node.

In order to prove that the algorithm terminates, it is sufficient to show that BBE terminates. Since the domains are finite, the labeling phase produces a number of finite choices at every level of the search tree. Moreover, since the number of variables is limited, then, we have also a finite number of levels in the tree. Hence, BBE considers at most all the possible assignments, that are a finite number. At the end of the execution of IFCSP-SCHEME, sol , with preference $pref$ is one of the optimal solutions of the current P_1 . Thus, for every assignment s' , $pref(P_1, s') \leq pref(P_1, sol)$. Moreover, for every completion $Q' \in C(P)$ and for every assignment s' , $pref(Q', s') \leq pref(P_1, s')$. Hence, for every assignment s' and for every $Q' \in C(P)$, we have that $pref(Q', s') \leq pref(P_1, sol)$. In order to prove that $sol \in NOS(P)$, now it is sufficient to prove that for every $Q' \in C(P)$, $pref(P_1, sol) = pref(Q', sol)$. This is true, since sol has a preference that is independent from the missing preferences of P , both when eliciting all the missing preferences, and when eliciting only the worst one either

at branch or node level. In fact, in both cases, the preference of *sol* is the same in every completion. Q.E.D.

If *When=tree*, then we elicit after each BB run, and it is proven in [7] that IFCSP-SCHEME never elicits preferences involved in solutions which are not possibly optimal. This is a desirable property, since only possibly optimal solutions can become necessarily optimal. However, the experiments will show that solvers satisfying such a desirable property are often out-performed in practice.

4 Problem Generator and Experimental Design

To test the performance of these different algorithms, we created IFCSPs using a generator which is a simple extension of the standard random model for hard constraints to soft and incomplete constraints. The generator has the following parameters:

- *n*: number of variables;
- *m*: cardinality of the variable domains;
- *d*: density, that is, the percentage of binary constraints present in the problem w.r.t. the total number of possible binary constraints that can be defined on *n* variables;
- *t*: tightness, that is, the percentage of tuples with preference 0 in each constraint and in each domain w.r.t. the total number of tuples (m^2 for the constraints, since we have only binary constraints, and *m* in the domains);
- *i*: incompleteness, that is, the percentage of incomplete tuples (that is, tuples with preference ?) in each constraint and in each domain.

Given values for these parameters, we generate IFCSPs as follows. We first generate *n* variables and then *d*% of the $n(n-1)/2$ possible constraints. Then, for every domain and for every constraint, we generate a random preference value in $(0, 1]$ for each of the tuples (that are *m* for the domains, and m^2 for the constraints); we randomly set *t*% of these preferences to 0; and we randomly set *i*% of the preferences as incomplete.

Our experiments measure the *percentage of elicited preferences* (over all the missing preferences) as the generation parameters vary. Since some of the algorithm instances require the user to suggest the value for the next variable, we also show the *user's effort* in the various solvers, formally defined as the number of missing preferences the user has to consider to give the required help.

Besides the 16 instances of the scheme described above, we also considered a "baseline" algorithm that elicits preferences of randomly chosen tuples every time branch and bound ends. All algorithms are named by means of the three parameters. For example, algorithm DPI.WORST.BRANCH has parameters *Who=dpi*, *What=worst*, and *When=branch*. For the baseline algorithm, we use the name DPI.RANDOM.TREE.

For every choice of parameter values, 100 problem instances are generated. The results shown are the average over the 100 instances. Also, when it is not specified otherwise, we set $n = 10$ and $m = 5$. However, we have similar results (although not shown in this paper for lack of space) for $n = 5, 8, 11, 14, 17$, and 20. All our experiments have been performed on an AMD Athlon 64x2 2800+, with 1 Gb RAM, Linux operating system, and using JVM 6.0.1.

5 Results

In this section we summarize and discuss our experimental comparison of the different algorithms. We first focus on incomplete fuzzy CSPs. We then consider two special cases: incomplete CSPs where all constraints are hard, and incomplete fuzzy temporal problems. In all the experimental results, the association between an algorithm name and a line symbol is shown below.

DP.ALL.TREE	+	DPI.WORST.NODE	---△---	SU.ALL.BRANCH	---○---
DP.WORST.TREE	*	DPI.WORST.TREE	▲	SU.ALL.NODE	---●---
DPI.ALL.BRANCH	---□---	LU.ALL.BRANCH	---▽---	SU.WORST.BRANCH	---○---
DPI.ALL.NODE	---■---	LU.ALL.NODE	---▼---	SU.WORST.NODE	---○---
DPI.ALL.TREE	---○---	LU.WORST.BRANCH	---◇---	DPI.RANDOM.TREE	---■---
DPI.WORST.BRANCH	---●---	LU.WORST.NODE	---◆---		

5.1 Incomplete Fuzzy CSPs

Figure 4 shows the percentage of elicited preferences when we vary the incompleteness, the density, and the tightness respectively. For reasons of space, we show only the results for specific values of the parameters. However, the trends observed here hold in general. It is easy to see that the best algorithms are those that elicit at the branch level. In particular, algorithm SU.WORST.BRANCH elicits a very small percentage of missing preferences (less than 5%), no matter the amount of incompleteness in the problem,

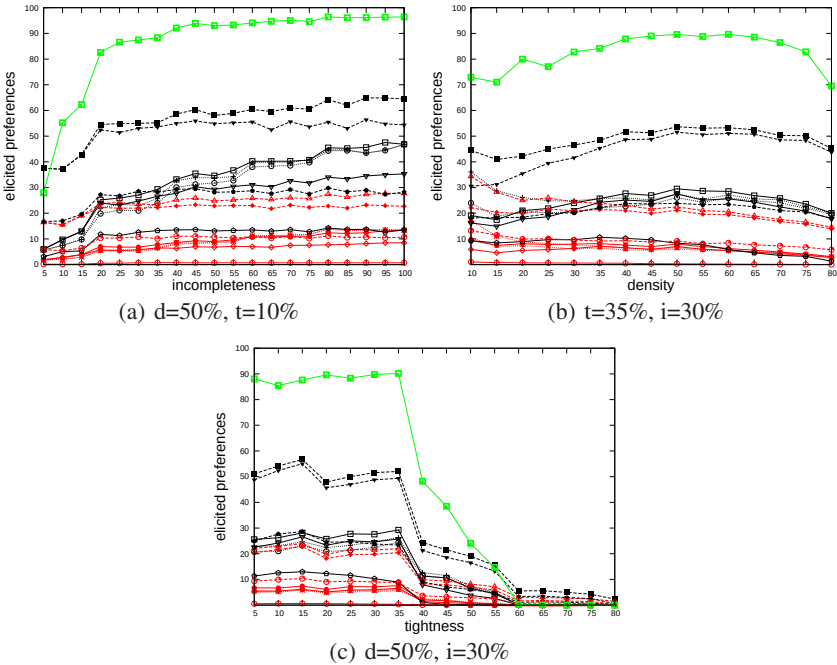


Fig. 4. Percentage of elicited preferences in incomplete fuzzy CSPs

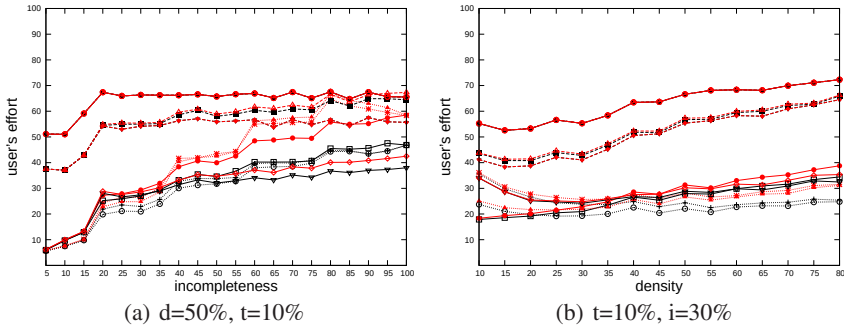


Fig. 5. Incomplete fuzzy CSPs: user's effort

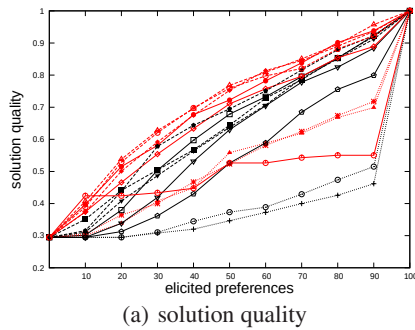


Fig. 6. Incomplete fuzzy CSPs: solution quality

and also independently of the density and the tightness. This algorithm outperforms all others, but relies on help from the user. The best algorithm that does not need such help is DPI.WORST.BRANCH. This never elicits more than about 10% of the missing preferences. Notice that the baseline algorithm is always the worst one, and needs nearly all the missing preferences before it finds a necessarily optimal solution. Notice also that the algorithms with What=worst are almost always better than those with What=all, and that When=branch is almost always better than When=node or When=tree.

Figure 5 (a) shows the user's effort as incompleteness varies. As could be predicted, the effort grows slightly with the incompleteness level, and it is equal to the percentage of elicited preferences only when What=all and Who=dp or dpi. For example, when What=worst, even if Who=dp or dpi, the user has to consider more preferences than those elicited, since to identify the worst preference value the user needs to check all of them (that is, those involved in a partial or complete assignment). DPI.WORST.BRANCH requires the user to look at 60% of the missing preferences at most, even when incompleteness is 100%.

Figure 5 (b) shows the user's effort as density varies. Also in this case, as expected, the effort grows slightly with the density level. In this case DPI.WORST.BRANCH requires the user to look at most 40% of the missing preferences, even when the density is 80%.

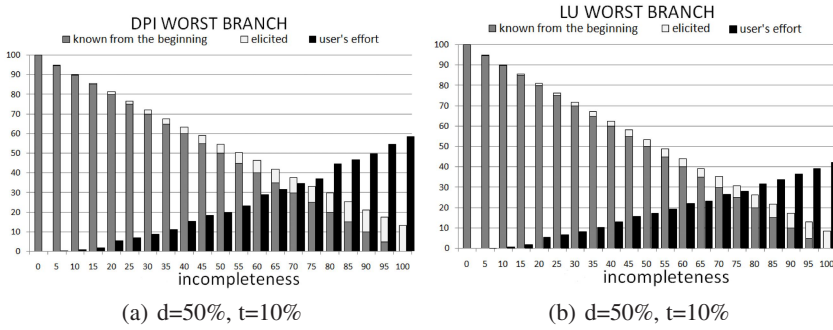


Fig. 7. Incomplete fuzzy CSPs: best algorithms

All these algorithms have a useful anytime property, since they can be stopped even before their termination obtaining a possibly optimal solution with preference value equal to the best solution considered up to that point. Figure 6 shows how fast the various algorithms reach optimality. The y axis represents the solution quality during execution, normalized to allow for comparison among different problems. The algorithms that perform best in terms of elicited preferences, such as DPI.WORST.BRANCH, are also those that approach optimality fastest. We can therefore stop such algorithms early and still obtain a solution of good quality in all completions.

Figure 7 (a) shows the percentage of elicited preferences over all the preferences (white bars) and the user's effort (black bars), as well as the percentage of preferences present at the beginning (grey bars) for DPI.WORST.BRANCH. Even with high levels of incompleteness, this algorithm elicits only a very small fraction of the preferences, while asking the user to consider at most half of the missing preferences.

Figure 7 (b) shows results for LU.WORST.BRANCH, where the user is involved in the choice of the value for the next variable. Compared to DPI.WORST.BRANCH, this algorithm is better both in terms of elicited preferences and user's effort (while SU.WORST.BRANCH is better only for the elicited preferences). We conjecture that the help the user gives in choosing the next value guides the search towards better solutions, thus resulting in an overall decrease of the number of elicited preferences.

Although we are mainly interested in the amount of elicitation, we also computed the time to run the 16 algorithms. Ignoring the time taken to ask the user for missing preferences, the best algorithms need about 200 ms to find the necessarily optimal solution for problems with 10 variables and 5 elements in the domains, no matter the amount of incompleteness. Most of the algorithms need less than 500 ms.

5.2 Incomplete Hard CSPs

We also tested these algorithms on hard CSPs. In this case, preferences are only 0 and 1, and necessarily optimal solutions are complete assignments which are feasible in all completions. The problem generator is adapted accordingly. The parameter What now has a specific meaning: What=worst means asking if there is a 0 in the missing preferences. If there is no 0, we can infer that all the missing preferences are 1s.

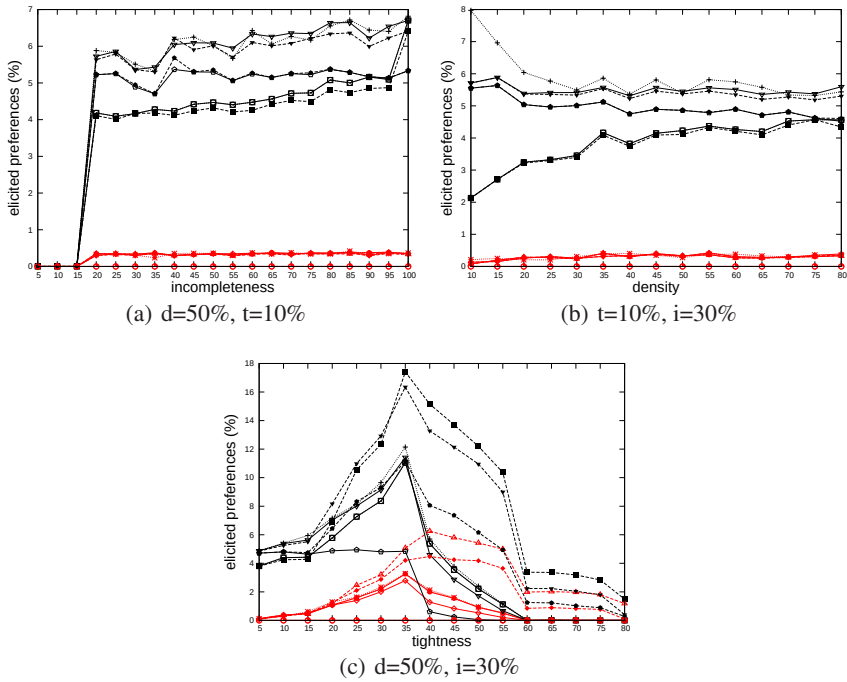


Fig. 8. Elicited preferences in incomplete CSPs

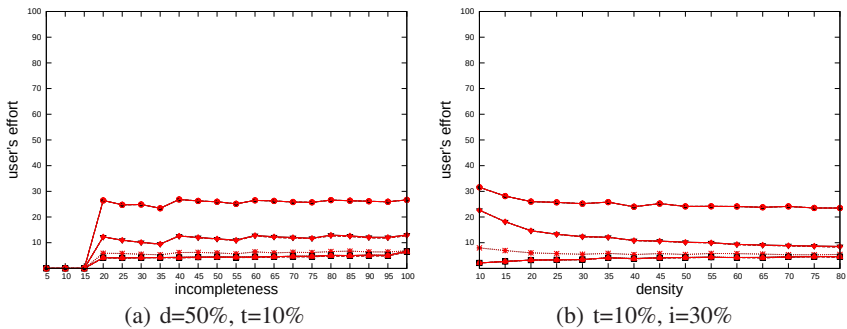


Fig. 9. Incomplete CSPs: user's effort

Figure 8 shows the percentage of elicited preferences for hard CSPs in terms of amount of incompleteness, density, and tightness. Notice that the scale on the y axis varies to include only the highest values. The best algorithms are those with What=worst, where the inference explained above about missing preferences can be performed. It is easy to see a phase transition at about 35% tightness, which is when problems pass from

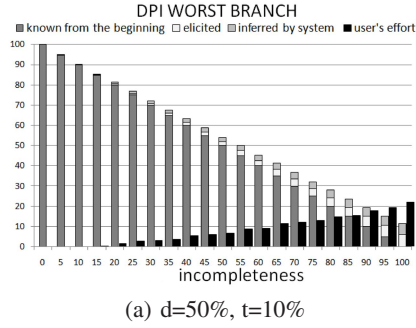


Fig. 10. Incomplete CSPs: best algorithm

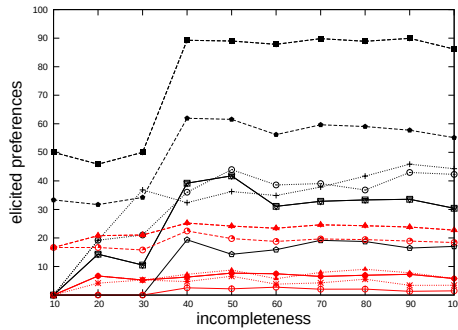


Fig. 11. Percentage of elicited preferences in incomplete fuzzy temporal CSPs

being solvable to having no solutions. However, the percentage of elicited preferences is below 20% for all algorithms even at the peak.

Figure 9 (a) shows the user's effort in terms of amount of incompleteness and Figure 9 (b) shows the user's effort in terms of density for the case of hard CSPs. Overall, the best algorithm is again DPI.WORST.BRANCH. Figure 10 gives the elicited preferences and user effort for this algorithm.

5.3 Incomplete Temporal Fuzzy CSPs

We also performed some experiments on fuzzy simple temporal problems [8]. These problems have constraints of the form $a \leq x - y \leq b$ modelling allowed time intervals for durations and distances of events, and fuzzy preferences associated to each element of an interval. We have generated classes of such problems following the approach in [8], adapted to consider incompleteness. While the class of problems generated in [8] is tractable, the presence of incompleteness makes them intractable in general. Figure 11 shows that in this specialized domain it is also possible to find a necessarily optimal solution by asking about 10% of the missing preferences, for example via algorithm DPI.WORST.BRANCH.

6 Future Work

In the problems considered in this papers, we have no information about the missing preferences. We are currently considering settings in which each missing preference is associated to a range of possible values, that may be smaller than the whole range of preference values. For such problems, we intend to define several notions of optimality, among which necessarily and possibly optimal solutions are just two examples, and to develop specific elicitation strategies for each of them. We are also studying soft constraint problems when no preference is missing, but some of them are unstable, and have associated a range of possible alternative values.

To model fuzzy CSPs, we have not used traditional fuzzy set theory [3], but soft CSPs [1], since we intend to apply our work also to non-fuzzy CSPs. In fact, we plan to consider incomplete weighted constraint problems as well as different heuristics for choosing the next variable during the search. All algorithms with What=all are not tied to fuzzy CSPs and are reasonably efficient. Moreover, we intend to build solvers based on local search and variable elimination methods. Finally, we want to add elicitation costs and to use them also to guide the search, as done in [10] for hard CSPs.

Acknowledgements

This work has been partially supported by Italian MIUR PRIN project “Constraints and Preferences” (n. 2005015491). The last author is funded by the Department of Broadband, Communications and the Digital Economy, and the Australian Research Council.

References

1. Bistarelli, S., Montanari, U., Rossi, F.: Semiring-based constraint solving and optimization. *JACM* 44(2), 201–236 (1997)
2. Dechter, R., Dechter, A.: Belief maintenance in dynamic constraint networks. In: *AAAI*, pp. 37–42 (1988)
3. Dubois, D., Prade, H.: *Fuzzy sets and Systems - Theory and Applications*. Academic Press, London (1980)
4. Faltings, B., Macho-Gonzalez, S.: Open constraint satisfaction. In: Van Hentenryck, P. (ed.) *CP 2002*. LNCS, vol. 2470, pp. 356–370. Springer, Heidelberg (2002)
5. Faltings, B., Macho-Gonzalez, S.: Open constraint optimization. In: Rossi, F. (ed.) *CP 2003*. LNCS, vol. 2833, pp. 303–317. Springer, Heidelberg (2003)
6. Faltings, B., Macho-Gonzalez, S.: Open constraint programming. *AI Journal* 161(1-2), 181–208 (2005)
7. Gelain, M., Pini, M.S., Rossi, F., Venable, K.B.: Dealing with incomplete preferences in soft constraint problems. In: Bessière, C. (ed.) *CP 2007*. LNCS, vol. 4741, pp. 286–300. Springer, Heidelberg (2007)
8. Khatib, L., Morris, P., Morris, R., Rossi, F., Sperduti, A., Brent Venable, K.: Solving and learning a tractable class of soft temporal problems: theoretical and experimental results. *AI Communications* 20(3) (2007)

9. Lamma, E., Mello, P., Milano, M., Cucchiara, R., Gavanelli, M., Piccardi, M.: Constraint propagation and value acquisition: Why we should do it interactively. In: IJCAI, pp. 468–477 (1999)
10. Wilson, N., Grimes, D., Freuder, E.C.: A cost-based model and algorithms for interleaving solving and elicitation of csps. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 666–680. Springer, Heidelberg (2007)

Reformulating Positive Table Constraints Using Functional Dependencies

Hadrien Cambazard and Barry O’Sullivan

Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{h.cambazard, b.osullivan}@4c.ucc.ie

Abstract. Constraints that are defined by tables of allowed tuples of assignments are common in constraint programming. In this paper we present an approach to reformulating table constraints of large arity into a conjunction of lower arity constraints. Our approach exploits functional dependencies. We study the complexity of finding reformulations that either minimize the memory size or arity of a constraint using a set of its functional dependencies. We also present an algorithm to compute such reformulations. We show that our technique is complementary to existing approaches for compressing extensional constraints.

1 Introduction

Constraint satisfaction techniques are ubiquitous in many practical problem-solving contexts [20]. Constraints can either be defined intensionally, as expressions or global constraints, or extensionally as tables of allowed or forbidden tuples of values. In this paper we focus on table constraints that are defined in terms of *allowed* tuples, which are often referred to as *positive* table constraints. Most constraint toolkits, e.g. ILOG Solver and Choco, provide support for specifying such constraints. Table constraints occur “naturally” in many application domains, such as product configuration where data is available from databases, spreadsheets or catalogues [13]. Also, naive users of constraint toolkits often find it convenient to use table constraints rather than more appropriate models using global constraints and other advanced features.

While table constraints might be easy to specify, they suffer from three disadvantages from a computational point of view. Firstly, table constraints are propagated with algorithms such as GAC-SCHEMA [4] with running times that are proportional to the number of tuples allowed by the constraint, which is exponential in its arity. Secondly, when a set of table constraints is inconsistent, we may be interested in characterising the inconsistency by generating an explanation [5]. A typical form of explanation is a minimal set of conflicting constraints. Large arity table constraints do not lend themselves to giving compact explanations because they might involve too large a subset of the variables of the problem. Thirdly, the amount of space required to store a table constraint might be excessive since there can be redundant information copied many times in the tuples of the constraint. Therefore, there is considerable motivation for looking at techniques to automatically reformulate table constraints by either reducing their arity, the number of tuples that define them, or the amount of space they require.

In this paper we study such a technique that exploits functional dependencies in the constraint. In particular, we reformulate by eliminating functional dependencies and computing equivalent conjunctions of lower arity constraints. We study the complexity of the problem of finding a reformulation that either minimizes memory size or arity based on a set of functional dependencies that hold on the constraint. We also propose an algorithm for computing such a reformulation. We show that this reformulation technique is complementary to previous approaches to compactly representing table constraints [6,8,10,12,19] by capturing quite a different structure in their tuples.

2 Background

A constraint satisfaction problem $\mathcal{P}$ is a triple $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where: $\mathcal{X}$ is a finite set of variables; $\mathcal{D}$ is a set of domains corresponding to the possible values of each variable; and $\mathcal{C}$ is a set of constraints. Each constraint $c_i \in \mathcal{C}$ is defined by a scope and a relation. The scope denoted $\text{scope}(c_i)$ is an ordered subset of $\mathcal{X}$ and the relation, $\text{rel}(c_i)$, is a set of tuples over $\text{scope}(c_i)$ that satisfy the constraint. The number of variables constrained by c_i , i.e. $|\text{scope}(c_i)|$, is known as the *arity* of the constraint c_i . A solution to $\mathcal{P}$ is an assignment of all variables to a value of their domain that satisfy all the constraints of the problem. We will moreover denote by $\text{sol}(\mathcal{P})$ the set of all solutions of $\mathcal{P}$.

In order to avoid confusion between the notion of a database relation, and the relation of a constraint, we define the projection operator σ in the following. Let r be a relation over a set of variables X and t , a tuple of r . The projection onto $Z \subseteq X$ of t , denoted $t[Z]$, is the restriction of t to Z . The projection of r onto Z , denoted $\sigma_Z(r)$, is the set $\{t[Z] \mid t \in r\}$. $\sigma_Z(c)$ denotes the corresponding constraint with a scope $Z \subseteq \text{scope}(c)$ and $\text{rel}(\sigma_Z(c)) = \sigma_Z(\text{rel}(c))$. $\sigma_Z(c)$ is a relaxation of c , i.e. its set of allowed tuples is obtained by projecting the set of allowed tuples of c onto Z .

Definition 1 (Constraint Reformulation). A reformulation $\Delta(c)$ of a positive table constraint c is a set of relaxations $\mathcal{R} \stackrel{\text{def}}{=} \{c^1, \dots, c^k\}$ of c such that $\forall c^a, c^b \in \mathcal{R}, a \neq b, \text{scope}(c^a) \not\subseteq \text{scope}(c^b)$.

Reformulations that give rise to conjunctions of constraints that are equivalent, i.e. have the same set of solutions, are very important. We refer to such reformulations as *lossless*.

Definition 2 (Lossless Reformulation). Given a CSP $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{X}, \mathcal{D}, \{c\} \rangle$ involving a single constraint c , $\Delta(c)$ is a lossless reformulation of c if the CSP $\mathcal{P}' \stackrel{\text{def}}{=} \langle \mathcal{X}, \mathcal{D}, \Delta(c) \rangle$ is such that $\text{sol}(\mathcal{P}) = \text{sol}(\mathcal{P}')$.

In contrast with earlier work, e.g. [9], our approach to computing lossless reformulations of positive table constraints exploits the concept of functional dependencies in a relation [11]. A *functional dependency* in a relation $\text{rel}(c)$ is written as $\mathcal{F} : X \rightarrow y$, where $X \cup \{y\} \subseteq \text{scope}(c)$. A functional dependency states that if a pair of tuples in the relation take the same values for the variables in X , they must also take the same value for variable y . $\mathcal{F} : X \rightarrow y$ is minimal if y is not functionally dependent on any subset of X . It is said to be *trivial* if $y \in X$. Algorithms for finding the set of all minimal and non-trivial dependencies that hold in a given relation are known [11].

Example 1 (Functional Dependencies in Constraints). Consider the following constraint:

$$c_a(x_1, x_2, x_3, x_4) \stackrel{\text{def}}{=} \{(0, 0, 0, 4), (1, 0, 0, 2), (2, 4, 1, 3), (0, 4, 2, 4), (2, 2, 3, 2)\}.$$

The following minimal functional dependencies (among the seven that exist) hold in c_a : $\mathcal{F}_1 : \{x_3\} \rightarrow x_2$, $\mathcal{F}_2 : \{x_1, x_2\} \rightarrow x_3$, and $\mathcal{F}_3 : \{x_1, x_2\} \rightarrow x_4$. The values of x_2 are uniquely determined by the value of x_3 and the values of x_4 and x_3 depend, similarly, only on the values taken by x_1 and x_2 . The dependency $\{x_2\} \rightarrow x_3$ does not hold because value 4 for x_2 does not determine the value of x_3 (2 or 1). $\blacktriangle$

For the sake of simplicity, we will denote by S_i the scope $X_i \cup \{y_i\}$ of a dependency $\mathcal{F}_i : X_i \rightarrow y_i$. Moreover, we will say that $\mathcal{F}_i \subset \mathcal{F}_j$ if and only if $S_i \subset S_j$. Finally for a set of dependencies δ , we define $\delta(S) = \{\mathcal{F}_i \in \delta \mid S_i \subset S\}$, the restriction of δ to a scope S , i.e all the dependencies of δ that hold on this scope. A dependency can be used to reformulate a constraint in the following way.

Definition 3 (Constraint Reformulation using a Functional Dependency). Let c be a positive table constraint, $\mathcal{F} : X \rightarrow y$ a functional dependency that holds on $\text{rel}(c)$ such that $X \cup \{y\} \subset \text{scope}(c)$. The reformulation of c , denoted $\Delta(c, \mathcal{F})$, using $\mathcal{F}$ is defined by: $\Delta(c, \mathcal{F}) = \{\sigma_{\text{scope}(c) - \{y\}}(c), \sigma_{X \cup \{y\}}(c)\}$.

Informally, a functional dependency is used to split the scope of a constraint into two scopes by eliminating the functionally dependant variable y . For the sake of simplicity, the notion of reformulation is extended to scopes with $\Delta(\text{scope}(c), \mathcal{F}) = \{\text{scope}(c) - \{y\}, X \cup \{y\}\}$ and relations, $\Delta(\text{rel}(c), \mathcal{F}) = \{\sigma_{\text{scope}(c) - \{y\}}(\text{rel}(c)), \sigma_{X \cup \{y\}}(\text{rel}(c))\}$.

Example 2 (Constraint Reformulation using a Functional Dependency). Consider again the constraint c_a presented in Example 1. The original scope of c_a is (x_1, x_2, x_3, x_4) . If we apply $\mathcal{F}_3 : \{x_1, x_2\} \rightarrow x_4$, this scope is split into (x_1, x_2, x_3) and (x_1, x_2, x_4) , in accordance with the procedure above. If we apply $\mathcal{F}_1 : \{x_3\} \rightarrow x_2$ on the scope (x_1, x_2, x_3) we can split it into (x_1, x_3) , (x_2, x_3) , giving the lossless reformulation: $\Delta(c_a, \langle \mathcal{F}_3, \mathcal{F}_1 \rangle) = \{\{x_3, x_2\}, \{x_1, x_3\}, \{x_1, x_2, x_4\}\}$. $\blacktriangle$

Our reformulation strategy is related to decomposing database relations into Boyce-Codd Normal Form (BCNF). However, we can often decompose relations further. Specifically, we focus on *minimal* reformulations. Given a constraint c , the reformulation $\Delta(c, \delta)$ obtained from a set of dependencies δ holding on $\text{rel}(c)$ is said to be minimal if each element of $\Delta(c, \delta)$ cannot be decomposed further and there does not exist a reformulation $\Delta'(c, \delta) \subset \Delta(c, \delta)$ such that $\Delta'(c, \delta)$ is lossless.

Example 3 (Minimal Reformulation). Consider the constraint scope $c(x_1, x_2, x_3, x_4)$ on which $\mathcal{F}_1 : \{x_1, x_4\} \rightarrow x_3$, $\mathcal{F}_2 : \{x_1\} \rightarrow x_2$ and $\mathcal{F}_3 : \{x_3\} \rightarrow x_1$ hold. The following sequence of reformulations is produced by $\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3$:

- $\Delta_1(c, \mathcal{F}_1) = \{\{x_1, x_4, x_3\}, \{x_1, x_2, x_4\}\};$
- $\Delta_2(c, \langle \mathcal{F}_1, \mathcal{F}_2 \rangle) = \{\{x_1, x_4, x_3\}, \{x_1, x_2\}, \{x_1, x_4\}\};$
- $\Delta_3(c, \langle \mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3 \rangle) = \{\{x_1, x_3\}, \{x_3, x_4\}, \{x_1, x_2\}, \{x_1, x_4\}\}.$

The final reformulation, $\Delta_3(c, \langle \mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3 \rangle)$ is not minimal. In $\Delta_2(c, \langle \mathcal{F}_1, \mathcal{F}_2 \rangle)$, the scope $\{x_1, x_4, x_3\}$ contains the scope $\{x_1, x_4\}$ (the first constraint implies the other), therefore the latter can be removed while still ensuring that the resultant reformulation is lossless. In fact, the decomposition of $\{x_1, x_4, x_3\}$ that follows is itself lossless. A minimal decomposition would, therefore, be $\{\{x_1, x_3\}, \{x_3, x_4\}, \{x_1, x_2\}\}$. ▲

It can be shown [5] that reformulations obtained using the process described in Definition 3 are lossless and preserve arc-consistency, *i.e.* that achieving generalised arc consistency (GAC) [4,15,16] on the reformulation is equivalent to achieving GAC on the original constraint. A reformulation is obtained by applying a sequence of functional dependencies. However, as dependencies are applied, others may no longer be applicable to the reformulation we obtain and this implies compatibilities between dependencies and valid orderings to compute the reformulation.

Theorem 1 (Valid Ordering of Functional Dependencies [5]). *Given a constraint c , let $\mathcal{F}_i : X_i \rightarrow y_i$ and $\mathcal{F}_j : X_j \rightarrow y_j$ be two minimal functional dependencies that hold in $\text{rel}(c)$ such that $y_j \in S_i$ and $\mathcal{F}_i \not\subseteq \mathcal{F}_j$. Then, when $\mathcal{F}_i$ and $\mathcal{F}_j$ are applied on the same scope, $\mathcal{F}_i$ can only be applied before $\mathcal{F}_j$, which we denote as $\mathcal{F}_i \prec \mathcal{F}_j$.*

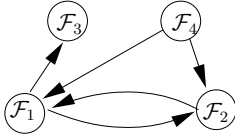
Example 4 (Valid Ordering Dependencies). Consider $F_1 : \{x_1, x_2\} \rightarrow x_3$ and $F_2 : \{x_1, x_3\} \rightarrow x_4$. F_2 can only be applied before F_1 ($\mathcal{F}_2 \prec \mathcal{F}_1$) because the use of $\mathcal{F}_1$ would remove x_3 from the scope and thus prevents $\mathcal{F}_2$ from applying on this scope. ▲

Given a set of dependencies δ , we denote by G_δ the directed graph of precedences between the dependencies in δ . The nodes of G_δ are the dependencies in δ . An edge $(\mathcal{F}_i, \mathcal{F}_j)$ is added if $\mathcal{F}_i \prec \mathcal{F}_j$, *i.e.* $\mathcal{F}_i$ can only be applied before $\mathcal{F}_j$. To fully characterize a set of dependencies that can be used to give rise to a reformulation, we define the *root* of a set δ and a scope S as $\text{root}(\delta, S) = \{\mathcal{F}_i \in \delta(S) \mid \forall \mathcal{F}_j \in \delta(S) : \mathcal{F}_i \not\subseteq \mathcal{F}_j\}$, *i.e.* the subset of δ that applies to scope S and where no dependency is included in another. A root set on S corresponds to the dependencies that will be used to remove variables from S . The precedence graph of such a set, therefore, needs to be acyclic and valid sets of dependencies are defined as follows:

Definition 4 (Valid Set of Dependencies). *A set of functional dependencies $\delta^* \subseteq \delta$ holding on a scope S is said to be valid if and only if $G_{\text{root}(\delta^*, S)}$ is acyclic and $\forall \mathcal{F}_i \in \delta^*, G_{\text{root}(\delta^*, S_i)}$ is acyclic.*

Example 5 (Valid Set of Dependencies). Consider $\delta = \{\mathcal{F}_1, \mathcal{F}_2, \mathcal{F}_3, \mathcal{F}_4\}$, and their corresponding precedence graph presented in Figure 1. This is a valid set, even if G_δ is cyclic, since it can be verified that all roots correspond to acyclic subgraphs of G_δ : $\text{root}(\delta, S) = \{\mathcal{F}_1, \mathcal{F}_4\}$, $\text{root}(\delta, \{x_1, x_2, x_3\}) = \{\mathcal{F}_3\}$, $\text{root}(\delta, \{x_1, x_3, x_5, x_4\}) = \{\mathcal{F}_2\}$, $\text{root}(\delta, \{x_3, x_2\}) = \emptyset$ and $\text{root}(\delta, \{x_3, x_4, x_1\}) = \emptyset$. So, first of all x_3, x_4 will both be removed from S using $\mathcal{F}_1, \mathcal{F}_4$. Secondly, $\mathcal{F}_3$ and $\mathcal{F}_2$ will be used on the two independent subsopes produced by $\mathcal{F}_1, \mathcal{F}_4$. ▲

All dependencies in a valid set δ can be used to decompose the original scope by applying them root by root in an order compatible with the precedences of each root. Moreover, it is shown in [5] that the reformulation associated with the valid sequence of δ , using all elements of δ , is unique. This result relies on the following observation that we will use later.



$$\begin{aligned}\mathcal{F}_1 &: \{x_1 x_2\} \rightarrow x_3 \\ \mathcal{F}_2 &: \{x_3 x_4\} \rightarrow x_1 \\ \mathcal{F}_3 &: \{x_3\} \rightarrow x_2 \\ \mathcal{F}_4 &: \{x_1 x_3 x_5\} \rightarrow x_4\end{aligned}$$

Fig. 1. An example of a graph of precedences

Lemma 1. *Let $\mathcal{F}_i, \mathcal{F}_j$ be two minimal dependencies s.t $\mathcal{F}_i \subseteq \mathcal{F}_j$, then $y_j \in X_i \cup \{y_i\}$.*

Proof. If $y_j \notin X_i \cup \{y_i\}$ it means that $X_i \cup \{y_i\} \subseteq X_j$ and, therefore, that $\mathcal{F}_j$ is not minimal because another dependency, namely $\mathcal{F}_i$, is contained in its left-hand side. ■

Lemma 1 can be used to show that the reformulation using a valid set is unique and the dependencies can only be used once in the process because they are partitioned between the scopes as the scopes are broken. Thus, the reformulation of a valid set can be quickly computed because any order of the dependencies that respects the precedence graph of each root produces the same result. Algorithm 1 computes the reformulation (as a set of scopes) obtained from a valid set δ applying on S . The uniqueness of the reformulation for a valid set underpins our claim that when finding an optimal reformulation one can search for valid sets amongst the subsets of the original dependencies ($\mathcal{O}(2^n)$ complexity) rather than consider all possible sequences, which would give an $\mathcal{O}(n!)$ complexity.

Example 6 (Uniqueness of Reformulation). Consider the set of four dependencies δ of Figure 1. After applying $\mathcal{F}_4$ on S , $\mathcal{F}_2$ can only be applied on the scope x_1, x_3, x_5, x_4 produced by $\mathcal{F}_4$ because, as we have $\mathcal{F}_2 \subset \mathcal{F}_4$, it necessarily involves x_4 (Lemma 1) in its left hand side and x_4 does not appear in the remainder of S . ▲

3 Characterizing a Good Reformulation

For a relation r over a set X of variables, the size of r is computed as the number of values in the table (the memory size) i.e. $|r| * |\text{scope}(r)|$. A reformulation $\Delta(r, \delta)$ is a conjunction of constraints that can be characterized by: its maximum arity $a_{\Delta(r, \delta)} = \max_{r_i \in \Delta(r, \delta)} |\text{scope}(r_i)|$; its maximum number of tuples $l_{\Delta(r, \delta)} = \max_{r_i \in \Delta(r, \delta)} |r_i|$; its overall memory size $s_{\Delta(r, \delta)} = \sum_{r_i \in \Delta(r, \delta)} |r_i| * |\text{scope}(r_i)|$. The complexity of GAC is determined by the maximum number of tuples involved in any constraint relation, $l_{\Delta(r, \delta)}$, or the maximum arity, $a_{\Delta(r, \delta)}$, depending on the GAC scheme used [4, 16]. However it is easy to see that $l_{\Delta(r, \delta)}$ is always equal to $|\text{rel}(c)|$ in what remains of the initial relation. The maximal number of tuples in the reformulation cannot be decreased. By using a dependency, a small table can be extracted but the number of tuples of the relation from which the variable is removed remains unchanged. Consider $X_i \rightarrow y_i$ holding on S , once y_i has been removed from S , no pair of tuples can be identical in the resulting table, otherwise they would be identical on X_i and, therefore, would have been identical on y_i as well, by definition of a functional dependency. This does not mean that the reformulation cannot be more compact. Specifically $s_{\Delta(r, \delta)}$ can be smaller than

Algorithm 1. REFORMULATE($\delta = \{\mathcal{F}_1, \dots, \mathcal{F}_n\}, S = \{x_1, \dots, x_r\}$)

```

1.  $DS \leftarrow \{S\};$ 
2. While  $\delta \neq \emptyset$  Do
3.   For each  $Scope_k \in DS$  Do
4.      $\delta^* \leftarrow \text{root}(\delta, Scope_k);$ 
5.     If  $\delta^* \neq \emptyset$ 
6.        $DS \leftarrow DS - Scope_k; \ c \leftarrow |\delta^*|;$ 
7.       For each  $\mathcal{F}_i \in \delta^*$  Do
8.          $DS \leftarrow DS \cup S_i;$ 
9.         If  $\nexists P \in DS, \{Scope_k - \{y_1, \dots, y_c\}\} \subseteq P; \ // \text{ ensures minimality}$ 
10.         $DS \leftarrow DS \cup \{Scope_k - \{y_1, \dots, y_c\}\};$ 
11.       $\delta \leftarrow \delta - \delta^*;$ 
12. Return  $DS;$ 

```

$|rel(c)| \times |scope(c)|$; arity and memory size, therefore, provide a basis to characterise a *good* reformulation and we will focus our study on those two parameters.

4 Complexity of Size-Bounded Reformulation

Given a set of functional dependencies holding on a scope, we consider the complexity of finding a reformulation of minimum size or a reformulation in which the maximum arity of the constraints is minimized. Both problems can be shown to be NP-Hard, since their corresponding decision problems can be used to solve the Weighted Feedback Vertex Set (WFVS) problem, which is known to be NP-Complete. We have shown in [5] that the problem of minimizing the maximum arity is NP-Hard and now extend this result to minimizing the size to show the nature of the problem and derive an algorithm.

We start first by rewriting the size of a reformulation in $\Delta(r, \delta)$ only in terms of $|\sigma_{S_j}(r)|$ of each dependency $\mathcal{F}_j$ of δ . Notice that each relation r_i of $\Delta(r, \delta)$ is initially derived from a dependency $\mathcal{F}_j$ that produces a relation of scope S_j (with the exception of r_0 denoting here the remainder of the initial scope S). Then p_i variables might have been eventually removed from S_j by other dependencies to reach $scope(r_i)$ (p_i is null for at least one relation of $\Delta(r, \delta)$). The pair (S_j, p_i) is known for each $r_i \neq r_0$ and r_0 can be associated with the pair (S, p_0) . The size of a reformulation can be written as:

$$s_{\Delta(r, \delta)} = \sum_{r_i: (S_j, p_i) \in \Delta(r, \delta)} (|S_j| - p_i) \times |\sigma_{S_j}(r)|.$$

The size can, therefore, be computed from the details of each dependency $\mathcal{F}_i$, in particular the cardinality of its projection onto scope S_i , i.e. $|\sigma_{S_i}(r)|$. We define the basic decision problem as follows:

BOUNDED SIZE REFORMULATION (BSR)

Instance: A set δ of minimal functional dependencies holding on a scope $S = \{x_1, \dots, x_m\}$. A set $H = \{h_0\} \cup \{h_i | \mathcal{F}_i \in \delta\}$ of positive integers denoting the number of tuples, h_0 , of a relation on S and the number of tuples of the relations obtained from S with each dependency of δ . A positive integer b .

Question: Does there exist a subset $\delta_b \subseteq \delta$ with $s_{\Delta(S, \delta_b)} \leq b$, where $s_{\Delta(S, \delta_b)} = \sum_{r_i: (S_j, p_i) \in \Delta(S, \delta_b)} (|S_j| - p_i) \times h_j$ is the size of the reformulation obtained using δ_b ?

Notice that the BSR problem assumes that the set of functional dependencies is given explicitly in advance. The reason is that finding the dependencies from the relation is itself an NP-Complete problem. However, given a set of functional dependencies δ , it can be shown that there always exists a relation on which only those functional dependencies hold; such relations are known as Armstrong relations [3]. While constructing an Armstrong relation is exponential in the size of δ , upper and lower bounds on the minimal number of tuples in the relation are known [3]. The BSR problem, however, refers to a set of dependencies with the specific number of tuples in each projection of the relation on the scope of each dependency of δ . We show here that such a relation always exists for some specific set of dependencies and use it to prove theorem 2.

Lemma 2. *Let δ be a minimal set of dependencies defined on scope S such that one variable $x \in S$ does not appear in the scope of any dependency of δ . Assume that each $\mathcal{F}_i \in \delta$ has at least one element $e_i \in X_i$ that does not appear in the scope of any other dependency. Let h_α be an upper bound on the minimum number of tuples in an Armstrong relation for δ . Consider also a set of integers $h_i \geq h_\alpha$ for all $1 \leq i \leq |\delta|$. There is an Armstrong relation r for δ such that $|r| \geq \sum_i h_i + h_\alpha$ and $\forall i, h_i = |\sigma_{S_i}(r)|$.*

Proof. Consider the minimum Armstrong relation Λ for δ with at most h_α tuples and v_α different values (in the range $[1, v_\alpha]$). We show here how to add tuples to Λ without breaking any dependency in δ (no dependencies can be introduced by adding tuples) to achieve the proper size of each projection. For each dependency $\mathcal{F}_i$ of δ we add a set $T = \{t_j | 1 \leq j \leq h_i - |\sigma_{S_i}(\Lambda)|\}$ of tuples to Λ . Let t be an arbitrary tuple of Λ . Each t_j of T is identical to t except that $t_j[e_i] = v_\alpha + i + j$. It can be easily verified that T appears only in σ_{S_i} and in none of the σ_{S_j} for $i \neq j$. In addition, the only violated dependencies by the addition of T are of the form $V \rightarrow e_i$ that cannot be in δ . Finally $|r|$ can be made greater than $\sum_i h_i + h_\alpha$ by adding tuples equal to a tuple of Λ except on x where $t[x]$ is constructed using a new value for each tuple. ■

Our complexity proof is based on a reduction from the WEIGHTED FEEDBACK VERTEX SET, which is known to be NP-Complete [7].

WEIGHTED FEEDBACK VERTEX SET (WFVS)

Instance: A positive integer k and a weighted directed graph $G = (V, W, E)$ where $w_v \in W$ denotes a positive integer associated with each node $v \in V$.

Question: Does there exist an $X \subseteq V$ with $\sum_{x \in X} w_x \leq k$ such that G with the vertices from X removed is acyclic?

Theorem 2. *The BOUNDED SIZE REFORMULATION problem is NP-Complete.*

Proof. Clearly, this problem is in NP. Given a valid set of functional dependencies, the size of the resultant unique reformulation can be computed in polynomial time using Algorithm 1. To prove completeness we show a reduction from the WFVS problem. Consider an instance of the WFVS on a graph $G = (V, W, E)$ with $|V| = n$. We construct an instance of the BSR problem in the following way. A functional dependency $\mathcal{F}_k : \{\dots\} \rightarrow v_k$ is associated with each node of V . Then each v_k is instantiated to x_k and added to the left-hand side of all dependencies corresponding to the predecessors of v_k in G . Moreover, we add n different new variables to each left side of all functional

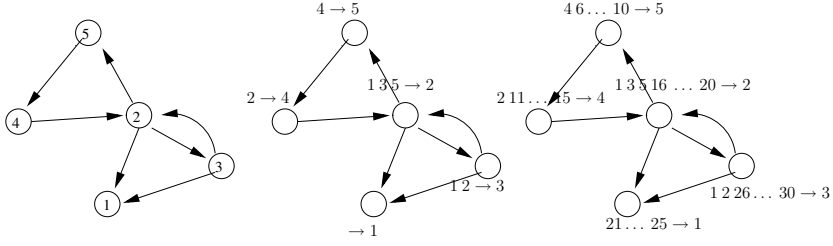


Fig. 2. An example set of dependencies built from a given graph

dependencies and one more variable that does not appear in any dependency. The resulting scope S is equal to $\{x_1, \dots, x_{n^2+n+1}\}$. Figure 2 shows the construction resulting from the two previous steps on an example graph G . These dependencies respect the precedences of G , they are minimal and can only apply on the main scope, i.e. they are not included in one another.

Let δ be such a set of dependencies and $m = n^2 + n + 1$. We denote by L , the least common multiple of all $|S_i|$ and $w'_i = (\frac{h_0}{h_\alpha L} - w_i)$. We set h_i with $0 < i \leq n$, b and h_0 to the following values: $h_0 = L \times \max_i(w_i) \times h_\alpha$, $h_i = w'_i \times \frac{h_\alpha L}{|S_i|}$ and $b = h_\alpha L(\sum_{i \in V} w'_i + k) + h_0 \times (n^2 + 1)$, where h_α is an upper bound on the size of the minimum Armstrong relation for δ (computable in polynomial time). It can be verified that they are all positive integers, that $h_i \geq h_\alpha$ and that $h_0 \geq \sum_{i=1}^n h_i + h_\alpha$ because $\sum_{i=1}^n h_i + h_\alpha = \sum_{i=1}^n \frac{h_0 - h_\alpha L w_i}{|S_i|} + h_\alpha = h_0 \sum_{i=1}^n \frac{1}{|S_i|} - h_\alpha (\sum_{i=1}^n \frac{L w_i}{|S_i|} - 1)$ and that $\sum_{i=1}^n \frac{1}{|S_i|} \leq 1$ and $\sum_{i=1}^n \frac{L w_i}{|S_i|} > 1$. Indeed, all w_i are positive and for all i , $|S_i| > n$. This transformation is polynomial and such an Armstrong relation with the corresponding properties exists according to Lemma 2.

We show that G has a feedback vertex set of weight at most k if and only if S has a reformulation of size at most b . Assume that we have a solution to BSR with $s_{\Delta(S, \delta_b)} \leq b$ and denote by X the set of nodes corresponding to the complement of δ_b in V so that $\delta_b = V - X$. There is a one-to-one correspondence between the nodes of G and dependencies of δ by construction. It can be verified that $\sum_{i \in X} w_i \leq k$, in the following way starting from $s_{\Delta(S, \delta_b)} \leq b$; notice that all the p_i are null except p_0 because all dependencies apply on the main scope and thus:

$$\begin{aligned} \sum_{i \in \delta_b} |S_i| \times h_i + h_0(|S| - |\delta_b|) &\leq b \\ h_\alpha L \sum_{i \in \delta_b} w'_i + h_0(n^2 + n + 1 - (n - |X|)) &\leq b \\ \sum_{i \in \delta_b} w'_i + \frac{h_0}{h_\alpha L} |X| &\leq \sum_{i \in V} w'_i + k \Rightarrow - \sum_{i \in X} w'_i + \frac{h_0}{h_\alpha L} |X| \leq k \Rightarrow \sum_{i \in X} w_i \leq k. \end{aligned}$$

Conversely, it can be easily seen, using the same computation, that any solution X to the WFVS having a weight at most k gives a set of dependencies δ_b (the complement of X in V) that provides a reformulation of size at most b . ■

5 A Dynamic Programming Algorithm for Reformulation

We propose a complete algorithm to find optimal reformulations that improves the one presented in [5]. We observe that many independent subproblems occur when trying to compute the optimal reformulation in terms of arity or size. This can be easily seen when writing the computation in a recursive way :

$$a^*(S, \delta) = \begin{cases} |S| & \text{if } \delta = \emptyset \\ \min_{\mathcal{F}_i \in \delta} (\max(a^*(S_i, \delta(S_i)), a^*(S - \{y_i\}, \delta - \{\mathcal{F}_i\} - \delta(S_i))), & \\ a^*(S, \delta - \{\mathcal{F}_i\})) & \text{otherwise} \end{cases}$$

$$s^*(S, \delta) = \begin{cases} |S| \times |\sigma_S(\text{rel}(c))| & \text{if } \delta = \emptyset \\ \min_{\mathcal{F}_i \in \delta} (s^*(S_i, \delta(S_i)) + s^*(S - \{y_i\}, \delta - \{\mathcal{F}_i\} - \delta(S_i)), & \\ s^*(S, \delta - \{\mathcal{F}_i\})) & \text{otherwise} \end{cases}$$

The rationale behind these formulations is that once a dependency $\mathcal{F}_i$ is chosen, $\delta(S_i)$ can only apply on S_i and, therefore, two independent subproblems appear: the optimal reformulation of S_i using $\delta(S_i)$ on one side and the optimal reformulation of $S - \{y_i\}$ on the other. $\delta(S_i)$ can only apply on scope S_i because all the dependencies of $\delta(S_i)$ involve y_i according to Lemma 1, and y_i only appears in S_i .

Consider, for example, the minimal dependency $\mathcal{F} : \{x_1, x_2, x_3\} \rightarrow y$, the only way to further decompose the scope obtained after applying this dependency is with a functional dependency of the form $\{y\} \rightarrow x_1$ or $\{x_2, y\} \rightarrow x_3$, but y is mandatory on the left-hand side. The reformulation algorithm is presented as Algorithm 2.

Algorithm 2. OPTIMALREFORMULATION($\delta = \{\mathcal{F}_1, \dots, \mathcal{F}_n\}, S$)

1. sort δ by increasing size of scopes;
 2. **For each** $\mathcal{F}_i \in \delta$ **do**
 3. $(a^*/s^*_{\mathcal{F}_i}, \delta^*_{\mathcal{F}_i}) \leftarrow \text{FINDREDUCEDOR}(\delta(S_i), \emptyset, S_i);$
 4. $\text{FINDREDUCEDOR}(\delta, \emptyset, S);$
-

The arity/size of the optimal reformulation obtained by each $\mathcal{F}_i$ of $\delta(S_i)$ is known when calling Algorithm 3, i.e. FINDREDUCEDOR (line 3), as all such subproblems have been solved independently before due to the sorting of line 1. Their optimal value is denoted $a^*_{\mathcal{F}_i}$ or $s^*_{\mathcal{F}_i}$ and the set of corresponding dependencies by $\delta^*_{\mathcal{F}_i}$.

Example 7 (An Optimal Reformulation Problem). Figure 3 gives an example of a reformulation problem where the dependencies have been organized into independent subproblems. The arity of the initial scope is eight and the circled dependencies denote an optimal solution to get a reformulation of arity three. The optimal value of the subproblem associated to each dependency is indicated in parentheses. $1\ 4\ 6\ 8 \rightarrow 7$ leads, for example, to a subscope that can be reformulated with a maximum arity of three by using $7\ 1\ 4 \rightarrow 6$, $7\ 6 \rightarrow 4$ and $7\ 1 \rightarrow 8$. It can also be seen that $7\ 6 \rightarrow 4$, which is included in $1\ 4\ 6\ 8 \rightarrow 7$ and $7\ 1\ 4 \rightarrow 6$, therefore, uses 7 and 6 in its left-hand side as stated by Lemma 1. Algorithm 2 will solve the subproblems from the leaves to the root of this tree by calling Algorithm 3 for each subproblem. ▲

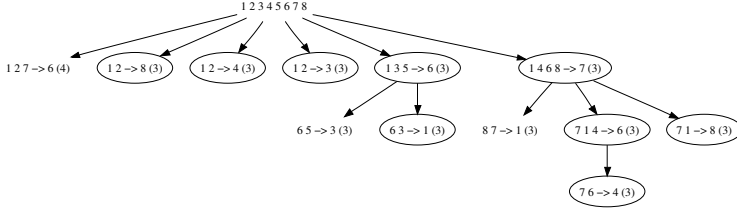


Fig. 3. An example of a hierarchy of subproblems

Algorithm 3. FINDREDUCEDOR($CD = \{\mathcal{F}_i, \dots, \mathcal{F}_n\}, PD, S$)

```

//  $a_{\mathcal{F}_i}^*/s_{\mathcal{F}_i}^*$  and  $\delta_{\mathcal{F}_i}^*$  are assumed to be known for all  $\mathcal{F}_i \in CD$ 
1. If  $CD = \emptyset$ 
2.   REFORMULATE( $PD \cup \{\delta_{\mathcal{F}_i}^* | \mathcal{F}_i \in PD\}, S$ );
3.   If an improving solution has been found
4.     store it and update the upper bound
5. Else
6.    $CD \leftarrow CD - \{\mathcal{F}_i\}$ ; //  $\mathcal{F}_i$  is chosen for branching
7.    $FD \leftarrow \text{PRUNING}(CD, PD \cup \{\mathcal{F}_i\}) \cup \delta(S_i) \cup \{\mathcal{F}_k \in CD | S_i \subseteq S_k\}$ ;
8.   If  $\text{BOUND}(CD - FD, PD \cup \{\mathcal{F}_i\}, S) < \text{upperBound}$ 
9.     FINDREDUCEDOR( $CD - FD, PD \cup \{\mathcal{F}_i\}, S$ );
10.  If  $\text{BOUND}(CD, PD, S) < \text{upperBound}$ 
11.    FINDREDUCEDOR( $CD, PD, S$ );
12.  FINDREDUCEDOR( $CD, PD, S$ );

```

Algorithm 3 assumes that each dependency is labeled with its corresponding $a^*/s_{\mathcal{F}_i}^*$ and $\delta_{\mathcal{F}_i}^*$. It computes the optimal reformulation of S using such dependencies. It is, basically, a branch and bound over the subsets of subproblems defined by each dependency of δ . At each node it considers the current set of subproblems (called PD) chosen to reformulate S and the set of current candidate subproblems (called CD) to be included (or not) in PD . PD and CD are maintained as sets of dependencies and PD can be seen as the root set of S : $\text{root}(\delta, S)$. The algorithm proceeds as follows:

- If CD is empty, the reformulation is computed with Algorithm 1 and the best known solution is updated if needed. Notice that $PD \cup \{\delta_{\mathcal{F}_i}^* | \mathcal{F}_i \in PD\}$ is a valid set.
- Otherwise, the algorithm branches by selecting which subproblem will be used to remove a variable from the scope S . A dependency $\mathcal{F}_i$ is chosen and the algorithm branches on the corresponding subproblem (line 7). FD represents the forbidden dependencies and the method $\text{PRUNING}(CD, PD \cup \{\mathcal{F}_i\})$ computes all the functional dependencies of CD that would create a cycle in the initial root set represented by PD ; As a subproblem is selected for branching, all dependencies from $\delta(S_i)$ and the one containing S_i are pruned (line 8). Branching on $7\ 1\ 4 \rightarrow 6$ on the example of Figure 3 would prune $1\ 4\ 6\ 8 \rightarrow 7$, $7\ 6 \rightarrow 4$ and all dependencies creating a cycle i.e. $1\ 3\ 5 \rightarrow 6$, $6\ 5 \rightarrow 3$, $6\ 3 \rightarrow 1$ and $1\ 2\ 7 \rightarrow 6$.
- If the bound obtained for the new pair $(CD - FD, PD \cup \{\mathcal{F}_i\})$ is compatible with the best known solution, the algorithm branches.

The algorithm relies on the partitioning of functional dependencies amongst subproblems, and specifically that no dependency is added twice (when completing PD

Algorithm 4. BOUNDARITY(CD, PD, S)

-
1. $lb \leftarrow \max_{\mathcal{F}_i \in PD} a_{\mathcal{F}_i}^*$;
 2. $lbs \leftarrow |S| - |PD| - (|CD| - \text{lower bound on the min. vertex feedback set in } G_{CD})$;
 3. **Return** $\max(lb, lbs)$;
-

with the dependencies of each subproblem – line 2), Lemma 1 can be extended to highlight that $\delta_{\mathcal{F}_i}^* \cap \delta_{\mathcal{F}_j}^* = \emptyset$ for all pairs of dependencies added to PD .

Property 1. For any two minimal compatible dependencies $\mathcal{F}_i$ and $\mathcal{F}_j$, *i.e.* such that $\mathcal{F}_i \prec \mathcal{F}_j$ and $\mathcal{F}_j \prec \mathcal{F}_i$ do not hold, then $\delta_{\mathcal{F}_i}^* \cap \delta_{\mathcal{F}_j}^* = \emptyset$.

The algorithm is sound because it computes only valid sets: adding $\delta_{\mathcal{F}_i}^*$ (line 2) cannot introduce any cycle in any root sets because each $\delta_{\mathcal{F}_i}^*$ is known to be valid already. The only root that needs to be checked is the initial one, PD , which is ensured by the pruning of the corresponding cycles (line 9). A heuristic can be applied (at line 7). A preprocessing step can also be applied when minimizing the size by removing all $\mathcal{F}_i$ such that $s_{\mathcal{F}_i}^* > h_0$. Finally simple bounds are used to prune the search. The proof of NP-Completeness showed the strong relationship between this problem and the WFVS. The bounding procedures relate to simple bounds for the WFVS.

The minimum arity expected from a current set of functional dependencies PD and the remaining candidates CD is simply computed here as the maximum over the minimal arity known for each dependency of PD and a lower bound on the arity expected from the remain of the original scope S (see Algorithm 4). The latter is computed by looking at the maximum number of variables that can still be removed from S without creating a cycle. This is the quantity: $(|CD| - \text{lower bound on the min. vertex feedback set in } G_{CD})$.

All bounds known for the FVS can be used. We use a simple lower bound that involves partitioning the graph into cliques $P = \{C_1 \dots C_k\}$. In each clique, all nodes except one must be removed to break all the cycles. Any partition P , therefore, gives a lower bound as $\sum_{C_i \in P} (|C_i| - 1)$. Consider the case of minimizing the size of the reformulation (Algorithm 5). Firstly, the optimal size of the reformulation associated with each dependency of PD is taken into account into the bound. Secondly, we consider the graph G_{CD} where a weight equal to $s_{\mathcal{F}_i}^* - h_0$ is associated with each node $\mathcal{F}_i$. The weight corresponds to the reduction in size (the gain) obtained by the use $\mathcal{F}_i$. A lower bound on the minimum weighted vertex feedback set gives an upper bound on the gain in size that we can expect due to the remaining dependencies.

Algorithm 5. BOUNDSIZE(CD, PD, S)

-
1. $lb \leftarrow \sum_{\mathcal{F}_i \in PD} s_{\mathcal{F}_i}^*$;
 2. $lb \leftarrow lb + h_0 \times (|S| - |PD|) - (\sum_{\mathcal{F}_i \in CD} (s_{\mathcal{F}_i}^* - h_0) - \text{lower bound on the min. weighted vertex feedback set in } G_{CD})$;
 3. **Return** lb ;
-

A lower bound on the minimum WFVS can be based on the partition into cliques as well, by keeping in each clique the node of minimum weight $\sum_{C_i \in P} \min_{\mathcal{F}_j \in C_i} s_{\mathcal{F}_j}^*$.

6 A Comparison with Other Compression Techniques

Several techniques have been introduced to improve the efficiency of reasoning over table constraints [6,8,12,19]. A recent approach based on table compression [12] relies on a *cross-product* representation the tuples [10], e.g. the tuples $\{\langle 1, 1, 1 \rangle, \langle 1, 2, 1 \rangle\}$ can be represented as a single tuple $\langle (1)(1, 2)(1) \rangle$. The authors of [12] state that “*The applicability of the representation is also reduced for tables where some of the variables are functionally dependent on some others*”. We believe that both techniques can strictly benefit from each other as breaking functional dependencies can only reduce the Hamming distance between the tuples by projecting them on sub-scopes. We show here that the technique in [12] is complementary to our reformulation approach.

Proposition 1. *The gain in size achieved by the approach presented in [12] and our functional dependency-based approach to reformulation are incomparable.*

Proof. Consider constraint c of Table 1 with $4kn$ tuples. For any pair of tuples in c , the Hamming distance is at least 2, thus preventing any compression using the representation from [12]. Notice that this table exhibits several dependencies and especially $x_2 \rightarrow x_3$ and $x_1 \rightarrow x_4$ corresponding, in this example, to two equality constraints. The reformulation $\Delta(c)$ is obtained from those two dependencies and its size, $2k + 2n + 2kn$ shows the following gain: $1 < \frac{s_c}{s_{\Delta(c)}} < 2$ if $k, n > 2$ (the gain would increase with the arity). Another observation is that c_3 can now be represented very efficiently by a cross-product $\langle (1, \dots, n) \times (1, \dots, k) \rangle$ whereas no dependencies hold¹, showing that both techniques are complementary. ■

Other representations of the tuples have been proposed such as [8], which relies on a “trie” data structure. A trie aims a factoring the shared prefixes of the tuples so it, essentially, captures the same kind of structure as the cross-product, i.e. the information stored in a redundant way in many tuples. The approach of [19] tries to achieve compression of the tuples by computing a minimal automaton whereas the CASE constraint corresponding to the table constraint in Sicstus [6] uses a DAG to get a representation

Table 1. An example of a table constraint and its reformulation

c				$\Delta(c)$					
x_1	x_2	x_3	x_4	c_1		c_2		c_3	
				x_2	x_3	x_1	x_4	x_1	x_2
1	1	1	1	1	1	1	1	1	1
2	1	1	2	2	2	2	2	2	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
n	1	1	n	k	k	n	n	n	1
1	2	2	1					1	2
⋮	⋮	⋮	⋮					⋮	⋮
n	2	2	n					n	2
⋮	⋮	⋮	⋮					⋮	⋮
1	k	k	1					1	k
⋮	⋮	⋮	⋮					⋮	⋮
n	k	k	n					n	k

¹ This is, in fact, a multi-valued dependency [11] as x_1 and x_2 are independent of each other and could also be detected as such.

similar to the cross-product. All these approaches rely on the idea that the tuples of the table share some information which is stored redundantly and can be compressed by using the appropriate data structure. Dependent variables only hinder their efficiency. Our reformulation approach is orthogonal to those techniques by capturing a very different kind of structure. It cannot, however, achieve by itself an exponential reduction in size, which is possible with the previous compression approaches. The bottleneck lies in the fact that the maximum number of tuples cannot be reduced using functional dependencies alone. Typically, for a constraint of arity a with n tuples of original size an , the reformulation always contains, in the best case, a constraint of size $2n$.

7 Experimental Results

The objective of our experimental evaluation was to study how effectively our functional dependency-based approach to reformulation could reduce the worst-case (maximum) arity of constraints in our reformulation as well as its total memory size². We considered two cases with respect to size: measuring the sum of the sizes of each table constraint in the reformulation, as well as the sum of the sizes of a REGULAR-based compilation of each table. We considered positive table constraints from the following five datasets: the Renault Megane Car Configuration Problem (we used the two largest table constraints, R80 and R140) [2]; a dataset of digital cameras [18]; a dataset of laptops [18]; the AI-CBR travel case-base [14]; and a dataset based on the crossword puzzle CSP benchmark [1]. We used a well-known library, called TANE [11], to compute the set of minimal functional dependencies for each table.

Table 2 presents the results and is divided into four parts. Firstly, we present some information on the original tables including their number of tuples, arity, size, number of minimal functional dependencies and the time needed by Tane to extract them. Secondly, we show the results associated with finding the optimal reformulation that minimizes the maximum arity. Thirdly, we present similar results focused on minimizing memory size. Finally, we show results associated with minimizing the size of the automata used by a REGULAR-based reformulation. A time limit was put at 120 seconds and a ‘-’ indicates that the time-out was reached while bold indicates when the result

Table 2. Results on minimizing arity and memory size (time given in seconds)

Instance Details						Minimize Maximum Arity						Minimize Memory Size						Minimize DFA			
name		nbt	arity	size	ndep	time	Complete			Greedy			Complete			Greedy			Complete		
							max	gain	size	time	arity	time	size	gain	time	size	time	orig	refor	gain	
camera	112	8	896	41	0.21	5	1.6	2220	0.07	5	0.05	896	1.0	0.02	896	0.01	421	421	1.0		
laptop	403	10	4030	54	0.12	6	1.67	14393	0.03	7	0.0	4030	1.0	0.05	4030	0.05	1452	1423	1.02		
rn R80	342	10	3420	2	0.25	8	1.25	2836	0.0	8	0.0	2836	1.21	0.0	2836	0.0	209	137	1.53		
rn R104	164	9	1476	11	0.06	4	2.25	1026	0.0	5	0.0	836	1.77	0.01	1140	0.0	183	100	1.83		
travel	1470	9	13230	7	0.14	6	1.5	27535	0.0	6	0.0	13230	1.0	0.01	13230	0.01	3693	3021	1.22		
cw R10	1881	12	22572	26	0.63	10	1.2	60114	0.0	10	0.0	22572	1.0	0.06	22572	0.06	5138	5138	1.0		
cw R11	1136	13	14768	128	0.65	9	1.44	45690	0.84	9	0.0	14768	1.0	0.13	14768	0.13	5426	4773	1.14		
cw R12	545	14	7630	1211	0.62	7	2.0	33546	109.1	8	0.02	7630	1.0	0.56	7630	0.51	3585	3503	1.02		
cw R13	278	15	4170	2243	0.41	6	2.5	14645	-	7	0.15	4170	1.0	0.83	4170	0.51	2095	2046	1.02		
cw R14	103	16	1648	4268	0.45	5	3.2	5061	-	6	0.74	1648	1.0	1.65	1648	0.4	970	942	1.03		
cw R15	57	17	969	5057	0.37	4	4.25	2611	10.78	5	1.13	969	1.0	2.04	969	0.29	674	627	1.07		
cw R16	23	18	414	3514	0.4	3	6.0	945	1.77	4	0.32	409	1.01	1.58	414	0.12	287	284	1.01		

² These experiments were run on a MacBook 2 GHz Intel Core Duo, 2 GB 667 Mhz DDR2.

has been proven optimal. The columns “gain” present the ratios of the original measure divided by the measure from the reformulation. We also present the results for a simple greedy algorithm selecting first the dependency of smallest scope. We observe that we can always find a reformulation in which the maximum arity is reduced: the gains range from 1.2 to 6. When we focus on minimizing memory size, the results are less successful and only the two Renault configuration benchmark tables are reduced. The optimal reformulation algorithm is very efficient when there are fewer than 1000 dependencies, but the more practical greedy algorithm achieves excellent performance even if the optimal solution is not always found.

Compression techniques can also be used to further reduce the size of our reformulation. In the final three columns of Table 2 we present the size of the automaton of a REGULAR [17,19] constraint for the original table, the sum of the sizes of each automata for the reformulation, and the corresponding gain in space we achieve through reformulation. The heuristic used to order the variables in the automaton is simply to put first the variable of minimum domain. We find that using a compilation of our reformulation, we reduce space in almost every case, and particularly so for the laptop, Renault, and travel datasets confirming the complementarity of the techniques.

8 Conclusion

Constraints that are defined by tables of allowed tuples of assignments are common in constraint programming. They are a very natural modeling tool for beginners in CP who often tend to enumerate the allowed tuples of a logical relation that does not fit perfectly into any of the intentional constraints provided by a constraint toolkit. In this paper we extend the results of [5] and present an approach to reformulating table constraints of large arity into a conjunction of lower arity constraints. Our approach exploits functional dependencies that might hold on the relation. We summarized many issues on dependencies in the context of reformulation, presented the complexity of the reformulation problem, a dynamic programming algorithm for optimal reformulation, and evaluated it on real-world and academic datasets. The experiments show that the gain of size is not large enough for an improvement in performance during search on those benchmark but open many opportunities when combined with compression techniques which deserve further studies. The experiments stand here as a proof of concept, as this technique is intending as an automatic way to deal with naive models made of large arity constraints, thus making CP easier to use.

Acknowledgement

This work was supported by Science Foundation Ireland (Grant number 05/IN/5806).

References

1. Sillito, J., Beacham, A., Chen, X., van Beek, P.: Constraint programming lessons learned from crossword puzzles. In: 14th Canadian Conference on Artificial Intelligence, pp. 78–87 (2001)
2. Amilhastre, J., Fargier, H., Marguis, P.: Consistency restoration and explanations in dynamic CSPs – application to configuration. *Artif. Intell.* 135, 199–234 (2002)

3. Beeri, C., Dowd, M., Fagin, R., Statman, R.: On the structure of armstrong relations for functional dependencies. *J. ACM* 31(1), 30–46 (1984)
4. Bessière, C., Régin, J.-C.: Arc consistency for general constraint networks: Preliminary results. In: *IJCAI*, vol. (1), pp. 398–404 (1997)
5. Cambazard, H., O'Sullivan, B.: Reformulating table constraints using functional dependencies - an application to explanation generation. *Constraints* 13 (2008)
6. Carlsson, M.: Filtering for the case constraint. In: Talk given at the Advanced School on Global Constraints, Samos, Greece (2006)
7. Garey, M.R., Johnson, D.S.: *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Series of Books in the Mathematical Sciences. W.H.Freeman, New York (1979)
8. Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalised arc consistency for extensional constraints. In: *AAAI*, pp. 191–197 (2007)
9. Gyssens, M., Jeavons, P., Cohen, D.A.: Decomposing constraint satisfaction problems using database techniques. *Artif. Intell.* 66(1), 57–89 (1994)
10. Hubbe, P.D., Freuder, E.C.: An efficient cross product representation of the constraint satisfaction problem search space. In: *AAAI*, pp. 421–427 (1992)
11. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: Tane: An efficient algorithm for discovering functional and approximate dependencies. *Comput. J.* 42(2), 100–111 (1999)
12. Katsirelos, G., Walsh, T.: A compression algorithm for large arity extensional constraints. In: Bessière, C. (ed.) *CP 2007*. LNCS, vol. 4741, pp. 379–393. Springer, Heidelberg (2007)
13. Laburthe, F., Caseau, Y.: Using constraints for exploring catalogs. In: Rossi, F. (ed.) *CP 2003*. LNCS, vol. 2833, pp. 883–888. Springer, Heidelberg (2003)
14. Leake, D.B., Sooriamurthi, R.: When two case bases are better than one: Exploiting multiple case bases. In: Aha, D.W., Watson, I. (eds.) *ICCBR 2001*. LNCS (LNAI), vol. 2080, pp. 321–335. Springer, Heidelberg (2001)
15. Lecoutre, C., Szymanek, R.: Generalized arc consistency for positive table constraints. In: Benhamou, F. (ed.) *CP 2006*. LNCS, vol. 4204, pp. 284–298. Springer, Heidelberg (2006)
16. Lhomme, O., Régin, J.-C.: A fast arc consistency algorithm for n-ary constraints. In: Veloso, M.M., Kambhampati, S. (eds.) *AAAI*, pp. 405–410. AAAI Press / The MIT Press, Menlo Park (2005)
17. Pesant, G.: A Regular Language Membership Constraint for Finite Sequences of Variables. In: Wallace, M. (ed.) *CP 2004*. LNCS, vol. 3258, pp. 482–495. Springer, Heidelberg (2004)
18. Reilly, J., Zhang, J., McGinty, L., Pu, P., Smyth, B.: Evaluating compound critiquing recommenders: a real-user study. In: *ACM Conference on Electronic Commerce*, pp. 114–123 (2007)
19. Richaud, G., Cambazard, H., O'Sullivan, B., Jussien, N.: Automata for nogood recording in constraint satisfaction problems. In: Benhamou, F. (ed.) *CP 2006*. LNCS, vol. 4204. Springer, Heidelberg (2006)
20. Wallace, M.: Practical applications of constraint programming. *Constraints* 1(1/2), 139–168 (1996)

Relaxations for Compiled Over-Constrained Problems^{*}

Alexandre Papadopoulos and Barry O'Sullivan

Cork Constraint Computation Centre

Department of Computer Science, University College Cork, Ireland
{a.papadopoulos,b.osullivan}@4c.ucc.ie

Abstract. In many real-world settings, e.g. product configuration, constraint satisfaction problems are compiled into automata or binary decision diagrams, which can be seen as instances of Darwiche's negation normal form. In this paper we consider settings in which a foreground set of constraints can be added to a set of consistent background constraints, that are compactly represented in a compiled form. When the set of foreground constraints introduces inconsistencies with the background constraints we wish to find relaxations of the problem by identifying the subset of the foreground constraints that do not introduce inconsistency; such a subset is called a relaxation. This paper is organised in two parts. First, two novel algorithms for finding relaxations based on automata are presented. They find the relaxation that is consistent with the largest (or smallest) number of solutions from amongst the longest ones (first algorithm), or from amongst the set-wise maximal ones (second algorithm). Then, we generalise our results by identifying the properties that the target compilation language must have for our approach to apply. Finally, we show empirically that on average our algorithms can be more than 500 times faster than a current state-of-the-art algorithm.

1 Introduction

We consider a configuration tool with which a user can specify preferences for options. These preferences are expressed as constraints. When preferences conflict, we want to help the user find which preferences to relax. In an iterative process, the user might relax constraints until at least one solution is found. Alternatively, the user might wish to be told which particular subsets of his constraints can be satisfied. Most current approaches to explanation generation in constraint-based settings are based on the notion of a minimal (with respect to inclusion) set of unsatisfiable constraints, known as a minimal conflict set of constraints. To demonstrate the concepts, we provide an example.

Example 1 (Car Configuration). Consider a simple car configuration problem, based on an example in [10], with the following set of options; the Boolean variable $x_i \in \{0, 1\}$ indicates whether constraint c_i is in the current set of active constraints or not:

Constraint	Option	Selector Cost
c_1	Budget	$x_1 = 1 \quad \sum_{i \in \{2, \dots, 5\}} (k_i \cdot x_i) \leq 3000$
c_2	Roof Rack	$x_2 = 1 \quad k_2 = 500$
c_3	Convertible	$x_3 = 1 \quad k_3 = 500$
c_4	CD Player	$x_4 = 1 \quad k_4 = 500$
c_5	Leather Seats	$x_5 = 1 \quad k_5 = 2600$

^{*} This work was supported by Science Foundation Ireland (Grant No. 05/IN/I886).

Table 1. The maximal relaxations and minimal exclusion sets for the over-constrained problem presented in Example 1. We show both the subset of the constraints in the relaxation (marked with a ✓) and those in the exclusion set, i.e. those that must be removed (marked with a ×).

Exp.	Constraints					Relaxation	Exclusion Set
	c_1	c_2	c_3	c_4	c_5		
I	×	×	✓	✓	✓	$\{c_3, c_4, c_5\}$	$\{c_1, c_2\}$
II	×	✓	×	✓	✓	$\{c_2, c_4, c_5\}$	$\{c_1, c_3\}$
III	✓	×	✓	✓	×	$\{c_1, c_3, c_4\}$	$\{c_2, c_5\}$
IV	✓	✓	×	✓	×	$\{c_1, c_2, c_4\}$	$\{c_3, c_5\}$
V	✓	×	×	×	✓	$\{c_1, c_5\}$	$\{c_2, c_3, c_4\}$

Assume that the technical constraints of the configuration problem forbid convertible cars having roof racks, therefore, constraints c_2 and c_3 form a conflict. Note that, given the budget constraint, if the user selects option c_5 , it is not possible to have any of the options c_2, c_3, c_4 . The set of all minimal conflicts for this example are: $\{c_2, c_3\}$, $\{c_1, c_2, c_5\}$, $\{c_1, c_3, c_5\}$, and $\{c_1, c_4, c_5\}$. ▲

As explanations, these conflicts are sufficient to explain why all constraints cannot be satisfied simultaneously. Based on the set of minimal conflicts we can compute the set of set-wise maximal relaxations showing which of the user’s constraints can be satisfied. Table 1 presents the set of all maximal relaxations, each showing how the user can satisfy at least some of his constraints. For example, consider Explanation I: we can simultaneously satisfy the constraints in $\{c_3, c_4, c_5\}$, but we must exclude c_1 and c_2 .

Sometimes we may have to choose a single relaxation to present to the user. The question is, which one should we select? The obvious response would be to select the relaxation that cannot be extended using any of the user’s choices without eliminating all solutions – this is the standard notion of a maximal relaxation. Amongst the set of maximal relaxations we might prefer to select the one that is longest on the basis that it includes the largest number of user constraints. However, an alternative is to present the relaxation that is consistent with either the fewest or largest number of solutions to the problem, while remaining maximal. This is the question we address in this paper.

In Section 2 we summarise the preliminary concepts required in this paper and present results from a motivating experiment showing that the number of solutions consistent with a maximal relaxation is not correlated with its length. In Section 3 we summarise the basics of automaton-based configuration. In Section 4 we present two novel algorithms. The first algorithm finds from amongst the *longest relaxations* to a set of inconsistent user constraints, the one that is consistent with either the fewest/largest number of solutions; this algorithm is *linear in the size of the automaton*. The second algorithm considers the same objectives in the context of set-wise *maximal relaxations*. In Section 5 we generalise our approach to other compiled representations by studying the properties that these must have in order for our results to carry over. We show that basically our algorithms do not need all the power of automata and that they work with more general, *i.e.* more compact, representations. In Section 6, we show that, while it is not polynomial in the size of the automaton, the second algorithm is in the average more than *500 times faster than a state-of-the-art algorithm*.

2 Preliminaries

We focus on constraint satisfaction problems in this paper, but the results hold for many other settings in which consistency is monotonic. This property holds whenever the set of solutions to a set of constraints $\mathcal{C}$ is a subset of the solutions to any set of constraints that is a subset of $\mathcal{C}$. In addition, we focus on constraint satisfaction problems that are solved in an interactive manner, e.g. product configuration problems. It is useful to distinguish between a background set of constraints, $\mathcal{B}$, that cannot be relaxed, and a set of constraints, $\mathcal{U}$, that are added by the user as he finds a preferred solution to $\mathcal{B}$ by finding a solution to $\mathcal{B} \cup \mathcal{U}$, the constraint problem we denote as $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$.

A set of constraints is consistent if it admits a solution. We will assume that the set of background constraints, $\mathcal{B}$, admits at least one solution. If a set of constraints does not admit a solution, at least one constraint must be excluded in order to recover consistency. Specifically, we are interested in finding *maximal relaxations* of $\mathcal{P}$.

Definition 1 (Maximal Relaxation). *Given a constraint problem $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$ that is inconsistent, a subset R of $\mathcal{U}$ is a relaxation of $\mathcal{P}$ if $\mathcal{B} \cup R$ admits a solution. The relaxation R is a maximal relaxation if $\forall R' \supset R$, $\mathcal{B} \cup R'$ is inconsistent. The maximal relaxation R is a longest relaxation if for other each maximal relaxation R' , $|R'| \leq |R|$.*

In some contexts, it might be more convenient to consider the following dual concept.

Definition 2 (Minimal Exclusion Set). *Given a constraint problem $\mathcal{P} \stackrel{\text{def}}{=} \langle \mathcal{B}, \mathcal{U} \rangle$ that is inconsistent, a subset E of $\mathcal{U}$ is an exclusion set (resp. minimal exclusion, shortest exclusion set) of $\mathcal{P}$ if $\mathcal{U} \setminus E$ is a relaxation (resp. maximal relaxation, longest relaxation).*

While intuitively we might believe that longer relaxations have fewer solutions, the story is not so simple. Theoretically, there is no reason why the number of solutions of two maximal relaxations should be similar. We illustrate this with an example. Consider variables $x_0, \dots, x_n$ with respective domains $D(x_0) = \{0, 1\}$ and $D(x_i) = \{0, \dots, d\}$, $i > 0$, and the constraints $x_0 < x_i, \forall i > 0$, $x_0 > x_i, \forall i > 0$ and $x_0 = 0$. This problem is inconsistent, and $R_1 = \{x_0 < x_i, \forall i > 0\} \cup \{x_0 = 0\}$ and $R_2 = \{x_0 > x_i, \forall i > 0\}$ are two maximal relaxations of the constraints. The number of solutions to R_1 is d^n , while there is only one solution to R_2 . The problem of selecting the maximal relaxation consistent with the largest set of solutions is intractable, in general.

More concretely, in Figure 1 we show the results of a simple experiment on the Renault Mégane configuration problem [1], which has been compiled in an automaton. This problem has 99 variables and about 2.8×10^{12} solutions. We built inconsistent user queries that instantiated 40 randomly chosen variables with a random value. We ran 20 such queries. For each query, we generated the complete set of maximal relaxations using the Dualize & Advance algorithm [2]. Using the automaton we could efficiently count the number of solutions consistent with each relaxation. In Figure 1, we plot, for each maximal relaxation, its length and the number of solutions of the problem consistent with it. It is clear from this figure that the number of solutions of a maximal relaxation is not necessarily correlated with its length.

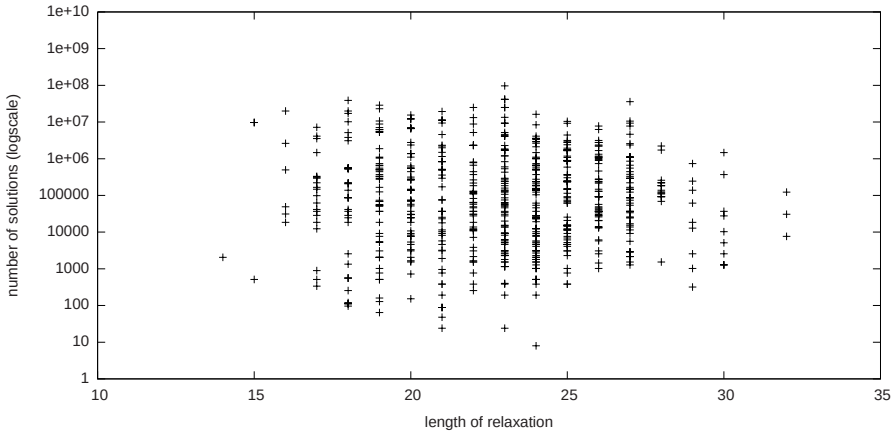


Fig. 1. Results from a simple experiment showing that number of solutions of a maximal relaxation is not necessarily correlated with its length

3 The Basics of Automaton-Based Configuration

In a configuration context, a typical approach is to compile the problem into an automaton in order to facilitate interactive solving [1, 17]. In this case many operations become tractable in practice. Let us focus, therefore, on the case where we have a compiled form of a problem, and the user chooses only unary constraints, i.e. assignments or disjunctions of assignments to the variables. A user query is composed of a set of constraints, each constraint c_i of which holds on a variable x_i .

An automaton gives a compact way of representing the set of all solutions. Informally, an automaton can be seen as a representation of the search tree on which minimisation allows us to reduce its size. This automaton only recognises words of the same length, and each recognised word corresponds to a solution of the problem, a particular ordering on the variables having been fixed in advance. The incoming and outgoing transitions of a state q are denoted by $in(q)$ and $out(q)$, respectively. The origin and destination state of a transition t are denoted by $in(t)$ and $out(t)$, respectively. The initial and the final states (or the source and the sink) are denoted by I and F , respectively. The level of a state q is the length of the words from I to q . The set of all states of level i is denoted $Q(i)$. The level of a transition t is the level of $out(t)$. Each level greater than 0 corresponds to a variable of the problem. Thus, each transition t provides a support

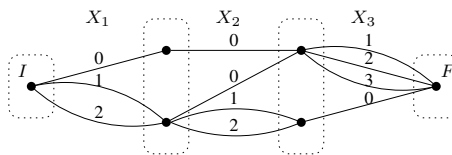


Fig. 2. An example automaton defined on three variables

for the instantiation of the variable of its level with the value labelling t . Figure 2, for example, shows the automaton for a problem on three variables X_0 , X_1 and X_2 . This problem has 13 solutions, some of which are 001, 002, 102, etc.

4 Algorithms

We present two novel algorithms for finding relaxations that are consistent with either the largest or fewest number of solutions, based on an automaton representation of the configuration problem. For a particular user query, comprising a set of unary constraints that restrict the domain of each variable, a valuation $\phi(t)$ is associated with each transition t of the automaton: $\phi(t) = 0$ meaning that this transition supports a valid instantiation (i.e. is labelled by an allowed value) and $\phi(t) > 0$ meaning it does not. Thus, to each complete path from I to F there corresponds a relaxation of the user's constraints, composed of the user's constraints supported by the transitions of the path with a valuation of 0.

If we restrict the valuation of the transitions only to 1 in case of a violation, the cost of a path from the source to the sink, which is the sum of the valuations of the transitions it is composed of, corresponds to the number of user constraints violated. If no such path of cost 0 exists, then the set of user constraints is inconsistent. A procedure is described in [1] that associates with each transition t of an automaton a cost $cost(t)$ of the best path (i.e. of minimal cost) of the automaton that uses t . This allows us to explore only the shortest paths in the automaton and, thus, only the longest relaxations of the user's constraints. Therefore, this can give our first exact algorithm (Algorithm 1) that finds, amongst all the longest relaxations, the one that is consistent with the largest or the smallest number of solutions, in time linear in the size of the automaton.

Obtaining the most, or least soluble, longest relaxation is simply a matter of selecting the appropriate relaxation at line 14. Let us consider the max case; a similar presentation can be used for the min case. Algorithm 1 associates with each state q' of the automaton, the most soluble longest relaxation restricted from I to q' (the automaton “to the left of” q'), stored in $relax(q')$, with $nsols(q')$ storing the corresponding number of solutions. This is valid because of the following property.

Property 1. After the end of the for loop starting at line 6, the value of $nsols(R)$ is the number of solutions of the part of the automaton from I to q' that supports R .

Proof. In the implementation of the algorithm, at some state q' , equal relaxations must be uniquely identified, and thus, for the same relaxation R , $nsols(R)$ takes into account all the ways to reach R , i.e. it is the sum of the numbers of solutions of all the known occurrences of $R \cap \{c_1, \dots, c_{i-1}\}$, which, by induction, are assumed to admit a maximal number of solutions. ■

The set *candidates*, after the iteration (starting at line 6), will contain all the longest relaxations ending at q' . At this point, we can choose the most soluble relaxation, as a less soluble one could not result in a longer relaxation globally more soluble. As the size of *candidates* is bounded by the number of states of the previous level, this algorithm runs in time linear in the size of the automaton.

Algorithm 1. Finding a most or least soluble longest relaxation**Data:** An automaton updated for a user query.**Result:** An optimally soluble longest relaxation.

```

1   $relax(I) \leftarrow \emptyset$ 
2   $nsols(I) \leftarrow 1$ 
3  for  $i \leftarrow 1$  to  $n$  do
4      forall  $q' \in Q(i)$  do
5           $candidates \leftarrow \emptyset$ 
6          forall  $t \in in(q')$  s.t.  $t$  is optimal do
7               $q \leftarrow in(t)$ 
8              if  $\phi(t) = 0$  then  $R \leftarrow relax(q) \cup \{c_i\}$ 
9              else  $R \leftarrow relax(q)$ 
10             if  $R \notin candidates$  then
11                  $candidates \leftarrow candidates \cup \{R\}$ 
12                  $nsols(R) \leftarrow 0$ 
13              $nsols(R) \leftarrow nsols(R) + nsols(q)$ 
14          $nsols(q') \leftarrow \text{opt}_{R \in candidates} nsols(R) // \text{opt} \in \{\max, \min\}$ 
15          $relax(q') \leftarrow R \in candidates \text{ s.t. } nsols(R) = nsols(q')$ 
16 return  $relax(F)$ 

```

However, restricting to the longest relaxation can prove to be too strong. For example, the plot in Section 2 (Figure 1) suggests that there is quite a concentration of long maximal relaxations, but very few of maximum length. Focusing on candidates amongst the maximal (by inclusion) relaxations seems to be a good trade-off between solubility and maximality. Therefore, we can adapt the previous algorithm to explore the whole automaton so as to consider all relaxations. The difference is that we cannot now greedily keep partial optimal solutions, because what is locally a maximal relaxation may not eventually be maximal. For example, in the automaton of Figure 2, suppose we post three unary constraints c_1, c_2, c_3 forcing every variable to be 0. Two maximal relaxations start in the second state of level 1: c_2 and c_3 . The first has three solutions while the second has only two. But the first will be, at the next step, included in the relaxation c_1c_2 , which has three solutions, while c_3 will still be a maximal relaxation, but with four solutions. Therefore, we need to maintain for each state the list of all the maximal relaxations. This procedure has, therefore, a complexity linear in the size of the automaton times the number of maximal relaxations. The corresponding modification is presented in Algorithm 2.

This is an ad-hoc procedure that lists all maximal relaxations of a query. However, being specifically designed for our context, it can be more efficient than generic algorithms, such as Dualize & Advance [2] (which can be theoretically exponential in the number of maximal relaxations), and gives, at the same time, the number of solutions of each relaxation. In this algorithm $relax(q)$ is a set of relaxations, and for each of them, say R , $nsols(q, R)$ stores its number of solutions. At lines 12 and 13, any relaxation that is a subset of another is removed, so as to keep only the maximal elements. As the size of the list $relax(q)$ is bounded by the total number of relaxations, the complexity is linear in the size of the automaton times the number of relaxations.

Algorithm 2. Finding all maximal relaxations**Data:** An automaton updated for a user query.**Result:** All maximal relaxations and their number of solutions.

```

1   $relax(I) \leftarrow \{\emptyset\}$ 
2   $nsols(I, \emptyset) \leftarrow 1$ 
3  for  $i \leftarrow 1$  to  $n$  do
4      forall  $q' \in Q(i)$  do
5          forall  $t \in in(q')$  do
6               $q \leftarrow in(t)$ 
7              forall  $R \in relax(q)$  do
8                  if  $\phi(t) = 0$  then  $R' \leftarrow R \cup \{c_i\}$ 
9                  else  $R' \leftarrow R$ 
10                  $relax(q') \leftarrow relax(q') \cup \{R'\}$ 
11                  $nsols(q', R') \leftarrow nsols(q', R') + nsols(q, R)$ 
12             Sort  $relax(q')$  by decreasing cardinality
13             forall  $R \in relax(q')$  do Remove in  $relax(q')$  subsets of  $R$ 
14 return  $relax(F)$ 

```

5 Generalisation to Other Compiled Representations

Although automata, and their boolean counterparts (reduced ordered) BDDs, are becoming widely used in real-world applications, other ways of compiling and representing a problem exist. An extensive survey of such representations for propositional formulas is provided in [6], which classifies the different representations from the perspective of their compactness, while showing which operations each representation efficiently supports. In this section we study how to generalise our algorithms beyond automata so that more succinct representations can be used. We present some sufficient conditions that the compiled representations must satisfy in order to guarantee that efficient algorithms exist for finding the kinds of relaxations we study in this paper.

The Compilation Map [6] presents a set of compiled representations of a problem, referred to as *target compilation languages*. Only boolean problems are considered. The most general language is **NNF** (for Negation Normal Form). A *sentence* Σ in **NNF**, holding on the set of variables $vars(\Sigma)$, can be represented as a directed acyclic graph (DAG), where the leaf nodes are labelled with $\top$ (true) or $\perp$ (false), X , $\neg X$ (with $X \in vars(\Sigma)$), and the internal nodes are labelled with $\wedge$ or $\vee$, and can have arbitrarily many children. Figure 3 shows an example of an NNF (taken from [5])¹.

Additionally, some key properties are defined. Firstly, an NNF satisfies *decomposability* if the children of every and-node hold on disjoint sets of variables, *i.e.* if for every and-node, $\wedge_i \alpha_i$, $vars(\alpha_i) \cap vars(\alpha_j) = \emptyset$, for $i \neq j$. Every and-node of Figure 3 satisfies this property and therefore this sentence satisfies decomposability. Secondly, an NNF satisfies *determinism* if the children of every or-node are mutually inconsistent, *i.e.* if for every or-node, $\vee_i \alpha_i$, $\alpha_i \wedge \alpha_j \models \perp$, for $i \neq j$. Every or-node in Figure 3

¹ We use a bold-face font to refer to a language, *e.g.* **NNF**, while a regular font is used to denote a formula in a specific language, *e.g.* NNF.

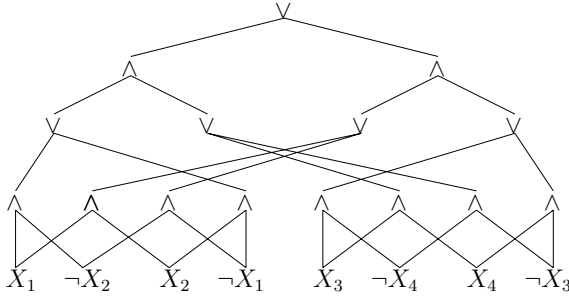


Fig. 3. An example NNF for the odd-cardinality function

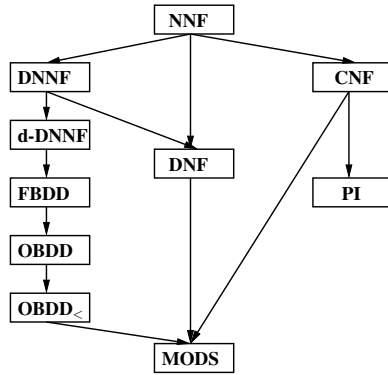


Fig. 4. The succinctness of the different languages; more succinct languages are higher up

satisfies this property and therefore this sentence also satisfies determinism. Thirdly, an NNF satisfies *smoothness* if the children of every **Or**-node hold on the same set of variables, *i.e.* if for every **Or**-node $\vee_i \alpha_i$, $\text{vars}(\alpha_i) = \text{vars}(\alpha_j)$, for $i \neq j$. The sentence of Figure 3 also satisfies smoothness. It is important to mention that smoothness can be enforced on any NNF. In fact, any child α_i of an **Or**-node $C = \vee_i \alpha_i$ such that $\text{vars}(\alpha_i) \subset \text{vars}(C)$ can be replaced by $\alpha_i \wedge (\bigwedge_{X \in \text{vars}(C) \setminus \text{vars}(\alpha_i)} (X \vee \neg X))$. This preserves equivalence, decomposability, determinism, and results in a new sentence whose size is $\mathcal{O}(|\Sigma| \cdot |\text{vars}(\Sigma)|)$. From a practical perspective, this means that any language is equivalent to its smooth version as far as the succinctness and the satisfiability of queries are concerned.

The languages **DNNF**, **d-NNF** and **s-NNF** are the subset of **NNF** that satisfy decomposability, determinism, smoothness, respectively. The language **d-DNNF** is the subset of **DNNF** that also satisfies determinism. The languages **s-DNNF**, **sd-DNNF**, **sd-NNF** are corresponding subsets that also satisfy smoothness. Without entering into the details, the language **OBDD_<** is the language corresponding to classic (reduced order) BDDs. Some other languages are worth mentioning here. **DNF** is the subset of **DNNF** containing DNF sentences, **CNF** is the set of CNF sentences, and **PI** (for prime implicates) is the subset of **CNF** of the sentences represented as the conjunction of their

Table 2. The queries supported by the introduced languages

L	CO	CT	ME	CD	SES	MR
NNF	×	×	×	✓	×	×
DNNF	✓	×	✓	✓	✓	✓
d-DNNF	✓	✓	✓	✓	✓	✓
FBDD	✓	✓	✓	✓	✓	✓
OBDD	✓	✓	✓	✓	✓	✓
OBDD_{<}	✓	✓	✓	✓	✓	✓
DNF	✓	×	✓	✓	✓	✓
CNF	×	×	×	✓	×	×
PI	✓	×	✓	✓	?	?
MODS	✓	✓	✓	✓	✓	✓

prime implicants. Finally, **MODS** is the language where sentences are represented by their set of models.

Informally, a language is more succinct than another if any sentence in the latter is equivalent to a smaller sentence in the former. This relation between languages is fundamental, as one typically wants to find the most succinct language that satisfies some desired properties. The relation between all the aforementioned languages in terms of succinctness is given in Figure 4.

A query allows us to efficiently retrieve information about a sentence. A language satisfies the query **CO** (Consistency) if there exists a polynomial algorithm that decides the consistency of any sentence in it; it satisfies **CT** (Model counting) if there exists a polynomial algorithm that counts the number of solutions of any sentence in it; it satisfies **ME** (Model enumeration) if there exists an algorithm that enumerates the models of any sentence in time polynomial in its size and the number of models. A transformation transforms a sentence into another one. We will only recall one such transformation. A language satisfies the transformation **CD** (Conditioning) if there exists a polynomial algorithm that maps a sentence Σ of that language and a term γ to a sentence denoted $\Sigma|\gamma$ in the same language, equivalent to $\Sigma \wedge \gamma$. Queries are not supported when no polynomial algorithm exists unless $P=NP$.

We define two new queries: **SES** (Shortest exclusion set, in Section 5.2) and **MR** (Maximal Relaxation, in Section 5.3). Table 2 summarises the results known for the queries previously introduced and our contribution of two new queries.

5.1 Counting Solutions

Counting the number of solutions can be efficiently performed on **d-DNNF** [5]. We recall here the function that counts the number of solutions of an sd-DNNF, adapted to take into account $\perp$ and $\top$ nodes. Note that because of smoothness, a $\top$ node can appear only as a child of an and-node, making it therefore neutral with respect to the conjunction, hence its value. Decomposability is required for case (5): the set of models of every child of the and-node hold on disjoint sets of variables, so they combine by simple product. Determinism and smoothness are required for case (6): the set of

models of every child of the **OR**-node hold on the same set of variables and are mutually disjoint, so the overall number of solutions is simply the sum.

$$\text{count}(\top) = 1 \quad (1)$$

$$\text{count}(\perp) = 0 \quad (2)$$

$$\text{count}(l) = 1, \text{ if } \neg l \notin S \quad (3)$$

$$\text{count}(l) = 0, \text{ if } \neg l \in S \quad (4)$$

$$\text{count}(\wedge_i \alpha_i) = \prod_i \text{count}(\alpha_i) \quad (5)$$

$$\text{count}(\vee_i \alpha_i) = \sum_i \text{count}(\alpha_i) \quad (6)$$

This methodology is the skeleton to define other functions on DNNF.

5.2 Longest Relaxations

Let Σ be a sentence. A set of user choices on it can be defined as a term S , *i.e.* an assignment of a subset of $\text{vars}(\Sigma)$. If $\Sigma \wedge S$ is inconsistent, then we can find a longest relaxation of S , *i.e.* a way to satisfy the largest number of assignments, or equivalently a shortest relaxation set of S , *i.e.* a way to give up on the fewest possible assignments. Formally, for a sentence Σ and a query S , $\text{ses}(\Sigma, S)$ is the size of a shortest exclusion set. In particular, $\Sigma \wedge S$ is consistent iff $\text{ses}(\Sigma, S) = 0$. Also, if Σ is inconsistent, $\text{ses}(\Sigma, S)$ is undefined. We must consequently assume that only consistent nodes are considered. Practically, because of decomposability, inconsistent nodes are only $\perp$ -nodes, **and**-nodes that have at least one inconsistent child, and **or**-nodes that have only inconsistent children.

We define a new query **SES**: a language **L** satisfies **SES** iff there exists a polynomial algorithm that computes $\text{ses}(\Sigma, S)$ for every formula Σ from **L** and every term S .

$$\text{ses}(\top, S) = 0 \quad (1)$$

$$\text{ses}(l, S) = 0, \text{ if } \neg l \notin S \quad (2)$$

$$\text{ses}(l, S) = 1, \text{ if } \neg l \in S \quad (3)$$

$$\text{ses}(\wedge_i \alpha_i, S) = \sum_i \text{ses}(\alpha_i, S) \quad (4)$$

$$\text{ses}(\vee_i \alpha_i, S) = \min_i \text{ses}(\alpha_i, S) \quad (5)$$

Case (4) only works with decomposable languages. In fact, if the exclusion sets of two children of the **and**-node hold on variables that do not overlap, they can be combined in a new one whose size is the sum. However, if the variables overlap, no greedy choice can be made, and different possibilities must be tested. Case (5) needs only smoothness: the exclusion sets of the children of the **or**-node hold on the same set of variables, it is just a matter of choosing the optimal one. Any language satisfying decomposability supports **SES**, as for example **DNNF**, **d-DNNF**, **OBDD**_<. On the other hand, this definition does not hold on non-decomposable languages.

Obviously, any language that does not support **CO** cannot support **SES**. Indeed, every language supports **CD**, and $ses(\Delta, \gamma) = 0$ iff $\Delta|\gamma$ is consistent. Notably, **CNF** does not support **CO**. However, **PI** is the only non decomposable language that supports consistency. It remains therefore a question to know whether it supports **SES** or not. That question is left for further work.

Once $ses(\Sigma, \gamma)$ is known, one or all the shortest exclusion sets can be found. The only point of choice is at an **OR**-node: children that have a minimal value correspond to the shortest exclusion sets, and thus, we only need to explore those. If we rely on an oracle for counting the number of solutions of a given exclusion set (like a complete enumeration or an incomplete estimation), enumerating the shortest exclusion sets to pick a most soluble longest relaxation is enough. On the other hand, for languages satisfying determinism, we can also combine the computation of one shortest exclusion set with the solution counter to find a most soluble longest relaxation.

Let $mse(\Sigma, S)$ be the function that returns the number of solutions of the most soluble shortest exclusion set. For the sake of clarity, we do not keep track of the corresponding exclusion set at each level, but of course that can be easily done. $mse(\Sigma, S)$ is defined as follows:

$$mse(\top, S) = 1 \quad (1)$$

$$mse(l, S) = 1, \text{ if } \neg l \notin S \quad (2)$$

$$mse(l, S) = 0, \text{ if } \neg l \in S \quad (3)$$

$$mse(\wedge_i \alpha_i, S) = \prod_i mse(\alpha_i, S) \quad (4)$$

$$mse(\vee_i \alpha_i, S) = \max_{i \text{ s.t. } ses(\alpha_i, S) = ses(\vee_i \alpha_i, S)} mse(\alpha_i, S) \quad (5)$$

Algorithm 1 (Section 4) is effectively a special application of this procedure for automata (compare lines 6 with the condition in Case (5) and 14 with the selection of the maximal child in Case (5)). Of course, we achieve, in the same way, a minimally soluble longest relaxation by changing the max of Case (5) to a min.

5.3 Maximal Relaxations

We now consider the enumeration of all maximal relaxations. A language **L** satisfies **MR** if, for each Σ of **L** and each query S , there exists an algorithm that enumerates all the maximal relaxations of $\Sigma \wedge S$ in time polynomial in the size of Σ and the number of maximal relaxations. Similar to the previous case, the set of all maximal relaxations can be found in time linear in their number in all languages satisfying decomposability. Moreover, we can observe again that any language that does not satisfy **CO** does not satisfy **MR**, and thus there only remains the question about **PI**, which is not decomposable but supports **CO**.

For any decomposable sentence Σ , the function $mr(\Sigma, S)$ defined as follows, returns the set of all maximal relaxations of $\Sigma \wedge S$:

$$mr(\top) = 1 \quad (1)$$

$$mr(l) = \{\{l\}\}, \text{ if } \neg l \notin S \quad (2)$$

$$mr(l) = \{\emptyset\}, \text{ if } \neg l \in S \quad (3)$$

$$mr(\wedge_i \alpha_i) = \otimes_i mr(\alpha_i) \quad (4)$$

$$mr(\vee_i \alpha_i) = \text{simplify}(\cup_i mr(\alpha_i)) \quad (5)$$

where $\mathcal{R} \otimes \mathcal{R}' = \{R \cup R' \mid R \in \mathcal{R} \wedge R' \in \mathcal{R}'\}$ and $\text{simplify}(\mathcal{R}) = \{R \in \mathcal{R} \mid \forall R' \in \mathcal{R} \ R \not\subseteq R'\}$, *i.e.*, we retain only set-wise maximal elements. Assuming we have determinism, we can associate with each maximal relaxation its number of solutions, exactly in the same way we did for *mse* (which does not depend at all on the nature of the relaxations). This way, we obtain a generalisation of Algorithm 2.

It is very interesting to note that basically this shows that, under some assumptions, we have a procedure that lists all maximal relaxations of an over-constrained problem that is theoretically more efficient than the best known one [2] (which is not linear in the number of maximal relaxations). These assumptions are that we post unary constraints, which is relevant in a configuration context, and that we have a problem compiled at least in a DNNF. Again, compilation is common in configuration, and DNNF is one of the most general, and succinct, forms to which a problem can be compiled.

6 Empirical Evaluation

The objective of our evaluation was to demonstrate the effectiveness of Algorithm 2 against a state-of-the-art algorithm for enumerating all maximal relaxations. We also considered two generic heuristic methods. We did not evaluate Algorithm 1 since it is an exact algorithm, linear in the size of the automaton. We based our experiments on the Renault Mégane Problem, also introduced in [1], which was compiled to an automaton. This problem has 99 variables and over 2.8×10^{12} solutions. We built inconsistent sets of user choices that instantiated 40 randomly chosen variables with a random value. We ran 20 such queries. For each of them, we generated the complete set of relaxations using the state-of-the-art Dualize & Advance algorithm [2], for finding all maximal relaxations in a constraint satisfaction context, and compared its performance against that of Algorithm 2 from this paper. For both algorithms, we recorded the time for each to find the most satisfiable maximal relaxation.

In addition to comparing Algorithm 2 against Dualize & Advance, we compared two heuristic techniques in terms of the number of solutions of the best relaxation they found. The goal is to find efficient heuristics for deciding which user constraints to add first when building a maximal relaxation. Each heuristic chooses as its next assignment the one that would reduce the number of solutions of the remaining problem by the least (respectively, largest) amount (*minimise/maximise solution loss*). The key measurements taken in each case were the number of solutions of the relaxation found as well as the time taken to find that relaxation.

In Figure 5 we compare each method in terms of the solubility of the best relaxation each found based on each query; note that we sorted the queries by the solubility of the most satisfiable relaxation for the purposes of clarity. We observe that the heuristics find very good relaxations, and very often an optimal one.

Concerning running time, the heuristics *minimise/maximise solution loss* were prohibitively slow (*i.e.* not much better than a complete method). These heuristics

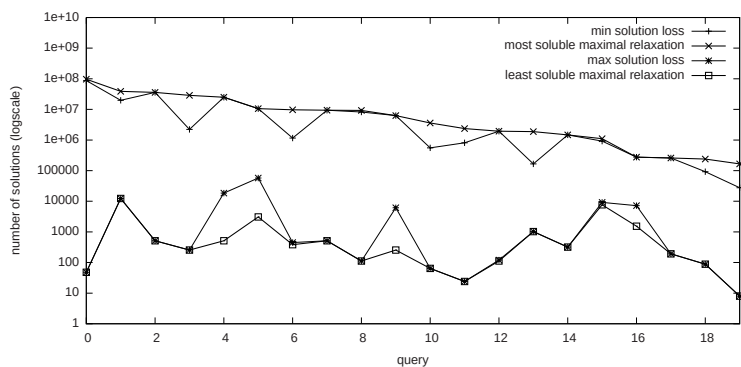


Fig. 5. Solubility of the best relaxation found by each method

Table 3. Running times in seconds for both algorithms

Algorithm	times (seconds)		
	minimum	maximum	average
Dualize and Advance [2]	255	726	416
Most soluble maximal relaxation (Algorithm 2)	0.4	1.3	0.8

are thus interesting more as an indication for future work; we want to look for heuristics that achieve the same purpose but in a less brutal way. We, therefore, do not discuss time results for these. The most interesting comparison regarding time is between the Dualize & Advance algorithm and our exact algorithm (Algorithm 2) for finding the most soluble maximal relaxation. Our results, summarised in Table 3, show the obvious advantage of our algorithm. Not only does our algorithm guarantee that it will find the maximal relaxation consistent with the most solutions of the problem, it is over 500 times faster than a current-state-of-the-art algorithm. Also, the times required by Algorithm 2, is of the order of one second, ideal for interactive applications.

7 Related Work

There have been many technical papers about explanation in the context of constraints [1, 8, 10, 3, 7, 13, 16, 15]. The dominant approach to explanation in configuration is based on computing minimal conflicting sets of constraints, which is related to the problem of finding all maximal relaxations. Approaches to finding the most preferred relaxations are well-known [10]. However, very little has been said about how to choose from amongst the set of all explanations, or how to select amongst equally preferred explanations. We address this problem by showing that it can be practical to prefer explanations based on their solubility.

Approaches have been proposed that attempt to be more “helpful” by presenting users with partial consistent solutions [15], or advise on how to relax constraints in order

to achieve consistency [13, 14]. Our approach is complementary to these by providing a basis for selecting from amongst the set of alternative explanations.

Other recent work has focused on finding minimal unsatisfiable subproblems in temporal problems [11], satisfiability [9, 12] and type error debugging [2]. These techniques find all minimal unsatisfiable sets of constraints, which can be exponential in the number of constraints. Our work can be seen as a generalisation of these algorithms to the case where consistency is determined using an automaton. Furthermore, the concept of minimum cardinality (*i.e.* the minimum number of literals that are set to false in the models of a sentence), although not conceptually identical to the one of relaxations, involves similar procedures on DNNF [4]. Our work establishes a link between this work and the work on automata [1], and extends it.

8 Conclusions

We considered the problem of generating maximal relaxations by reasoning about their solubility, in the context of product configuration, where the constraint model of the problem has been compiled into an automaton. Two novel algorithms were presented. The first finds from amongst the longest relaxations to a set of inconsistent user constraints, the one that is consistent with the most/fewest solutions; while the second considers the problem for maximal relaxations. Based on a large real-world configuration problem we demonstrated the value of our approach. Finally, we generalised our results by identifying the properties that the target compilation language must have for our approach to apply.

References

1. Amilhastre, J., Fargier, H., Marguis, P.: Consistency restoration and explanations in dynamic CSPs – application to configuration. *Artif. Intell.* 135, 199–234 (2002)
2. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: Hermenegildo, M.V., Cabeza, D. (eds.) *PADL 2004*. LNCS, vol. 3350, pp. 174–186. Springer, Heidelberg (2005)
3. Bowen, J.: Using dependency records to generate design coordination advice in a constraint-based approach to concurrent engineering. *Computers in Industry* 22(1), 191–199 (1997)
4. Darwiche, A.: Decomposable negation normal form. *J. ACM* 48(4), 608–647 (2001)
5. Darwiche, A.: On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics* 11(1-2), 11–34 (2001)
6. Darwiche, A., Marquis, P.: A knowledge compilation map. *J. Artif. Intell. Res (JAIR)* 17, 229–264 (2002)
7. Freuder, E.C., Likitvivatanavong, C., Moretti, M., Rossi, F., Wallace, R.J.: Computing explanations and implications in preference-based configurators. In: O'Sullivan, B. (ed.) *CologNet 2002*. LNCS (LNAI), vol. 2627, pp. 76–92. Springer, Heidelberg (2003)
8. Friedrich, G.: Elimination of spurious explanations. In: *Proceedings of ECAI*, pp. 813–817 (2004)
9. Gregoire, E., Mazure, B., Piette, C.: Boosting a complete technique to find mss and mus thanks for a local search oracle. In: *Proceedings of IJCAI*, pp. 2300–2305 (2007)
10. Junker, U.: QuickXplain: preferred explanations and relaxations for over-constrained problems. In: *Proceedings of AAI*, pp. 167–172 (2004)

11. Liffiton, M.H., Moffitt, M.D., Pollack, M.E., Sakallah, K.A.: Identifying conflicts in over-constrained temporal problems. In: Kaelbling, L.P., Saffiotti, A. (eds.) *IJCAI*, pp. 205–211. Professional Book Center (2005)
12. Liffiton, M.H., Sakallah, K.A.: On finding all minimally unsatisfiable subformulas. In: Bacchus, F., Walsh, T. (eds.) *SAT 2005*. LNCS, vol. 3569, pp. 173–186. Springer, Heidelberg (2005)
13. O’Callaghan, B., O’Sullivan, B., Freuder, E.C.: Generating corrective explanations for interactive constraint satisfaction. In: van Beek, P. (ed.) *CP 2005*. LNCS, vol. 3709, pp. 445–459. Springer, Heidelberg (2005)
14. O’Sullivan, B., Papadopoulos, A., Faltings, B., Pu, P.: Representative explanations for over-constrained problems. In: *AAAI*, pp. 323–328 (2007)
15. Pu, P., Faltings, B., Torrens, M.: Effective interaction principles for online product search environments. *Web Intelligence*, 724–727 (2004)
16. Sqalli, M.H., Freuder, E.C.: Inference-based constraint satisfaction supports explanation. In: *Proceedings of AAAI*, pp. 318–325 (1996)
17. Vempaty, N.R.: Solving constraint satisfaction problems using finite state automata. In: *AAAI*, pp. 453–458 (1992)

Approximate Compilation of Constraints into Multivalued Decision Diagrams

Tarik Hadzic¹, John N. Hooker², Barry O’Sullivan¹, and Peter Tiedemann³

¹ Cork Constraint Computation Centre
{thadzic,b.osullivan}@4c.ucc.ie

² Carnegie Mellon University
john@hooker.tepper.cmu.edu

³ IT University of Copenhagen
peter.t@itu.dk

Abstract. We present an incremental refinement algorithm for approximate compilation of constraint satisfaction models into multivalued decision diagrams (MDDs). The algorithm uses a vertex splitting operation that relies on the detection of equivalent paths in the MDD. Although the algorithm is quite general, it can be adapted to exploit constraint structure by specializing the equivalence tests for partial assignments to particular constraints. We show how to modify the algorithm in a principled way to obtain an approximate MDD when the exact MDD is too large for practical purposes. This is done by replacing the equivalence test with a constraint-specific measure of distance. We demonstrate the value of the approach for approximate and exact MDD compilation and evaluate its benefits in one of the main MDD application domains, interactive configuration.

1 Introduction

Compiling a constraint satisfaction model into a tractable representation is useful for a number of tasks related to model analysis and decision support. Various forms of tractable structures have been suggested as target compilation languages, including automata [1], binary decision diagrams [2], and/or decision diagrams [3], and deterministic decomposable negation normal form (d-DNNF) [4].

In this paper we focus on compiling CSP models into *multivalued decision diagrams* (MDDs), as they are well suited for a number of decision support tasks [2,5]. We identify the tests of *infeasibility*, *entailment* and *equivalence* as critical for reasoning about the properties of various compilation schemes. We recognize that the *semantics* of global constraints can be utilized to enhance the compilation, and suggest using *incremental refinement* as a way of dealing with the weaknesses of semantic tests when compiling multiple constraints. Our incremental scheme generalizes both the search based BDD compilation of [9] and standard bottom-up compilation of [10]. We represent two different algorithms for achieving this, one that constructs a new MDD and one that refines the input MDD. The later algorithm makes use of the concept of *vertex splitting* which was first introduced in [8].

Because the full MDD can grow too large for practical use, we are particularly concerned with generating *approximate* MDDs that are limited in size but useful in applications. Additionally we are also concerned with generating approximate MDDs under

tight time requirements, since some of the constraints might be known only during user interaction. We show how equivalence checking offers a principled way to create approximate MDDs (approximate in the sense that they represent a superset of the feasible solutions). Rather than check for equivalence, we measure the “distance” between two partial assignments and view them as equivalent for algorithmic purposes when the distance is below a threshold. The distance measure is specialized to each constraint type, thus again allowing us to exploit special structure in the problem. The refinement process is an iterative one in which the threshold is gradually reduced. This injects a learning element, because the algorithm refines equivalence detection as it refines the MDD, thus allowing the next MDD to be more accurate. An exact MDD can be obtained by reducing the threshold to zero, or an approximate MDD by reducing the threshold to a positive number or terminating when the MDD exceeds a size limit. Terminating before obtaining the exact MDD still provides bounds on the degree of violation of each individual constraint.

We are not aware of related work utilizing explicitly the semantics of highly structured constraints for the purpose of compilation. The related compilation techniques enhance compilation by exploiting independencies among variables [3,9]. The idea is to recognize two partial assignments p_1, p_2 as equivalent when they assign same values to *critical* variables. In [9] the critical variables are determined by a *cutset* and in [3] by a *context* with respect to a *pseudo-tree* extracted from a constraint graph. We note however, that neither technique can enhance equivalence detection when presented with individual global constraints span all variables.

Some work has already been done on generic techniques for approximate compilation [6,7], but these techniques have two major drawbacks in relation to constraint models. Firstly, they conjoin individual constraints precisely until a threshold is reached, and only then start approximating. Therefore, they do not take all constraints into consideration. Secondly, since they are not relying on semantic information captured by highly structured constraints, they provide no guarantees regarding the degree of violation of individual constraints.

2 Preliminaries

A *multivalued decision diagram* (MDD) can be viewed as a branching tree in which isomorphic subtrees have been merged. The tree is constructed to find feasible solutions of a constraint set containing finite-domain variables $x_1, \dots, x_n$. The tree branches on the variables in a fixed order $x_1, \dots, x_n$. The branches at each node correspond to possible values of some variable x_j , or more generally, to disjoint subsets of possible values. To form the MDD, subtrees containing no feasible solutions are first deleted, and subtrees having the same shape are then merged to remove redundancy from the tree. Additional edges connect each vertex in the bottom layer to a single terminal vertex 1.

Thus an MDD for a constraint set S is a directed acyclic graph whose vertices are arranged in layers corresponding to the variables $x_1, \dots, x_n$ in S . If vertex u lies in layer j (corresponding to x_j), we say $var(u) = x_j$, and each edge (u, v) leaving u corresponds to a subset D_{uv} of the domain D_j of x_j . The top layer consists only of the root vertex r , with $var(r) = x_1$. Each path $p = (u_1, \dots, u_{n+1})$ from r to 1 is identified with the cartesian product $\prod_{j=1}^n D_{j,j+1}$, where $u_1 = r$ and $u_{n+1} = 1$. Every path p

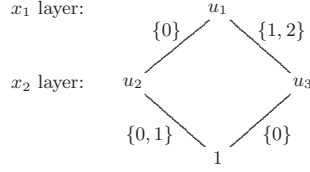


Fig. 1. MDD for $2x_1 + 3x_2 \leq 4$ with domains $x_i \in \{0, 1, 2\}$

from r to 1 in the MDD must *satisfy* S , meaning that every tuple $(x_1, \dots, x_n)$ in p is a feasible solution of S . Conversely, every feasible solution of S belongs to some path from r to 1. For example, the MDD for the constraint $2x_1 + 3x_2 \leq 4$ (with domains $x_i \in \{0, 1, 2\}$) appears in Fig. 1.

We assume that the MDD is *reduced*, meaning that all isomorphic subtrees have been merged. To make this precise, let each vertex u in layer j of the MDD correspond to the function $f_u : D_j \times \dots \times D_n \rightarrow \{0, 1\}$ defined by $f_u(x_j, \dots, x_n) = 1$ when $(x_1, \dots, x_n)$ belongs to a path from u to 1. Then $f_r(x_1, \dots, x_n) = 1$ if and only if $(x_1, \dots, x_n)$ satisfies S . Two vertices u, v in a given layer are *equivalent* if $f_u = f_v$, and the MDD is *reduced* if no two vertices in any layer are equivalent. When the variable ordering is fixed, there is a unique reduced MDD representing a given constraint set. It is common in the literature to remove vertex u in layer j (and join the two edges incident to u) when there is a single outgoing edge (u, v) , and it has the property that $D_{uv} = D_j$. This results in “long edges” that skip one or more layers, but to simplify notation we do not remove any vertices in this fashion.

For convenience in describing the algorithms we further assume that the operation of choosing the edge corresponding to value α returns the special vertex *False* if no such edge exists. When this vertex is included as a child in constructing a node, the semantics is to simply ignore this child. Furthermore we use $True(i)$ to indicate the MDD corresponding to the set of solutions $D_i \times \dots \times D_n$. Given an MDD M and a partial assignment p to variables $x_1, \dots, x_k$ we use M_p to denote the vertex reached in M when following the path corresponding to p . Given a constraint C , we also use C to denote the set of solutions to C . For a partial assignment p to variables $x_1, \dots, x_k$ we use $C(p)$ to represent the solution space of $C(p)$ restricted to the assignments in p .

3 Top-Down Compilation of MDDs

An MDD is a compact representation of a branching tree for a given constraint set. The naive MDD construction based on first constructing the tree and then reducing it can be significantly improved by performing reductions *during search*. This has already been recognized in a more specific context, where a CNF formula is compiled into a binary decision diagram (BDD) using DPLL search with caching [9]. As a starting point in this paper, we suggest a generalized approach for compiling CSPs into MDDs. It is important to realize that the general compilation algorithm is based on three fundamental tests: recognizing when partial assignments encountered during search lead to *infeasible*, *(domain) entailed* or *equivalent* subbranches. Algorithm 1 emphasizes these tests during a depth first search (DFS) traversal of the branching tree.

Algorithm 1. `CompileDFS`(path p , int i , constraints S): A generic backtracking algorithm constructing an MDD for a set of constraints in a cached top-down manner. It is initiated with the call `Compile`(S) which just executes `CompileDFS`($\emptyset, 0, S$).

```

if  $p \equiv_S 0$  then
   $\perp$  return False;
if  $p \equiv_S 1$  then
   $\perp$  return True( $i$ );
 $key = id_S(p)$ ;
 $result = \text{cache-lookup}(key)$ ;
if  $result \neq \text{null}$  then
   $\perp$  return  $result$ ;
Let  $v_1, \dots, v_k$  be the values in  $D_i$ ;
 $result =$ 
   $\text{get-vertex}(i, \text{CompileDFS}(p \times \{v_1\}, i + 1), \dots, \text{CompileDFS}(p \times \{v_k\}, i + 1))$ ;
   $\text{cache-insert}(key, result)$ ;
return  $result$ ;

```

We define a partial assignment p to be *infeasible* for a constraint set S ($p \equiv_S 0$) if it cannot be completed to an assignment in $p \times \prod_{i=k}^n D_i$ satisfying all constraints in S . In that case, the above algorithm returns *False* indicating infeasibility. We say that p is *domain entailed* ($p \equiv_S 1$) if every completion of p satisfies S . In this case, we return *True*(i) representing an MDD for the entire set of solutions $D_i \times \dots \times D_n$.

Finally, p_1 and p_2 are said to be *equivalent* ($p_1 \equiv_S p_2$) if they have the same completions satisfying S . The equivalence test induces a set of equivalence classes among all partial assignments, and we use $id_S(p)$ to denote a unique identifier *key* for the equivalence class to which p belongs. An MDD is stored as a *cache* of keys that is maintained during search. A new node is created (using `cache-insert` and `get-vertex`) only if the current *key* cannot be found (using `cache-lookup`). While the equivalence class identifiers might be prohibitively large in general, in practice they are usually compact.

We say that tests for infeasibility, entailment, and equivalence are *sound* if every “yes” answer is correct, *complete* if every “yes” answer is recognized, and *efficient* if the test can be computed in polynomial time (with respect to the size of the MDD). The performance of Algorithm 1 critically depends on these three properties. Unsound tests lead to MDDs not representing the desired solution space. Incomplete tests make the algorithm traverse equivalent or infeasible parts of the search space. Inefficient tests increase the running time. Ideally, if we have sound and complete tests requiring constant time, and it is possible to represent equivalence classes efficiently then Algorithm 1 builds an MDD in output-optimal time and space. In the remainder of the paper we will use these tests as a basis for discussing the efficiency of various MDD compilation schemes.

4 Semantic Caching

Previous approaches that enhance equivalence tests are based on identifying variable interdependencies in the underlying model. Two partial assignments to variables $x_1, \dots, x_{i-1}$

are equivalent if they assign the same values to variables on which x_i *critically* depends. The set of such variables might be much smaller than $\{x_1, \dots, x_{i-1}\}$ and therefore, equivalence detection could be enhanced [3,9]. However, these approaches cannot be applied if *all variables* depend on each other. It suffices to introduce just a single global constraint, spanning over all variables, to get to this situation.

We argue that in addition to looking at the variable independencies, we should also consider the *semantics* of well-structured constraints. Namely, the CSP modeling vocabulary is full of constraints with rich structure, which is normally exploited during search through efficient filtering algorithms.

We illustrate how the same can be exploited for designing better compilation tests for *inequality*, *equality*, and *Alldiff* constraints. We will then discuss how to extend this to multiple constraints.

Inequality. An inequality constraint C has the form $\sum_i f_i(x_i) \leq b$, where each x_i is a finite domain integer variable and f_i is some cost function. For a given partial assignment $p = (v_1, \dots, v_{k-1})$ to variables $(x_1, \dots, x_{k-1})$, we denote the cost of p with respect to C as $a(p) = \sum_{i=1}^{k-1} f_i(v_i)$. A simple equivalence test for an inequality constraint C is

$$p_1 \equiv_C p_2 \Leftrightarrow a(p_1) = a(p_2).$$

The test is efficient but incomplete, because two equivalent partial assignments can be identified as nonequivalent. For example, $p_1 = (0)$ and $p_2 = (1)$ are equivalent for $x_1 + 2x_2 \leq 3$ (where $x_1, x_2 \in \{0, 1\}$), but they fail the above test for equivalence. We can formulate a complete equivalence test that requires pseudo-polynomial time. Assuming without loss of generality that $a(p_1) < a(p_2)$, the test is

$$p_1 \equiv_C p_2 \Leftrightarrow a(p_1) \leq b - a(p) < a(p_2) \text{ for no } p \in D_k \times \dots \times D_n.$$

The following infeasibility test is both complete and efficient:

$$p \equiv_C 0 \Leftrightarrow a(p) + SP(p) > b. \quad (1)$$

where $SP(p) = \sum_{i=k}^n \min\{f_i(v) \mid v \in D_i\}$ is the *shortest path* in $D_k \times \dots \times D_n$.

An analogous entailment test is also complete and efficient:

$$p \equiv_C 1 \Leftrightarrow a(p) + LP(p) \leq b. \quad (2)$$

where $LP(p) = \sum_{i=k}^n \max\{f_i(v) \mid v \in D_i\}$ is the *longest path* in $D_k \times \dots \times D_n$.

Equality. The equivalence test

$$p_1 \equiv_C p_2 \Leftrightarrow a(p_1) = a(p_2). \quad (3)$$

for an equality constraint C (defined as $\sum_i f_i(x_i) = b$) is complete and efficient. The infeasibility test is essentially a subset sum problem:

$$p \equiv_C 0 \Leftrightarrow a(p) + a(p') \neq b \text{ for all } p' \in D_k \times \dots \times D_n. \quad (4)$$

which is complete but inefficient (pseudo-polynomial). A complete and efficient domain entailment test checks whether all completions of the path have the same cost:

$$p \equiv_C 1 \Leftrightarrow SP(p) = LP(p). \quad (5)$$

Alldiff. Given a partial assignment $p = (v_1, \dots, v_{k-1})$ we define $D(p) = \bigcup_{i=1}^{k-1} v_i$. We can now define a complete and efficient equivalence test for an Alldiff constraint C :

$$p_1 \equiv_C p_2 \Leftrightarrow D(p_1) = D(p_2).$$

Additionally we have the following complete and efficient infeasibility test.

$$p \equiv_C 0 \Leftrightarrow \left| \bigcup_{i=k}^n D_i \right| < n - k + 1.$$

Finally, a complete and efficient entailment test is given by

$$p \equiv_C 1 \Leftrightarrow D(p), D_k, \dots, D_n \text{ are disjoint and nonempty.} \quad (6)$$

The above equivalence detection rules directly indicate how to compute $id_C(p)$ for a constraint C . In case of inequality or equality constraints, $id(p) = a(p)$, and for an Alldiff constraint it is $D(p)$.

Multiple Constraint Caching. The semantic tests described above can be directly generalized to a set of constraints $S = \{C_1, \dots, C_m\}$:

$$p \equiv_S 0 \Leftrightarrow \bigvee_{i=1}^m (p \equiv_{C_i} 0), \quad p \equiv_S 1 \Leftrightarrow \bigwedge_{i=1}^m (p \equiv_{C_i} 1), \quad p_1 \equiv_S p_2 \Leftrightarrow \bigwedge_{i=1}^m (p_1 \equiv_{C_i} p_2).$$

The equivalence class identifier, $id_S(p)$, can be generically constructed as a tuple of individual keys, $id_S^\times(p) = (id_{C_1}(p), \dots, id_{C_m}(p))$. In this case, Algorithm 1 detects the equivalence of two paths p_1, p_2 as soon as $id_{C_i}(p_1) = id_{C_i}(p_2)$ for each $C_i \in S$. This way of combining the individual tests ensures soundness but not completeness of the generic test even if individual tests are complete. Namely, the test allows for a number of "fake" equivalence classes that appear to be different even though they are the same. The potential number of fake classes explodes exponentially if individual tests are incomplete or as we add more constraints to S . For example, if id_S allows for K_f fake equivalence classes and K_e exact equivalence classes, and if we add to S a constraint C' with K' equivalence classes (all exact), then the resulting number of fake equivalence classes is at least $K_f \cdot K'$. Even among remaining $K_e \cdot K'$ pairs, there could be many fake classes.

We could partially remedy this if we could uncover interactions amongst the set of constraints rather than treating them independently. For example, infeasibility detection $p \equiv_S 0$ of a set of integer inequalities could be enhanced by checking for feasibility of their linear relaxation. In addition, if we can detect that some constraints become *entailed* by the remaining ones, we could ignore them when denoting the equivalence classes. In the following section we will show how this can be done efficiently in a very interesting special case.

5 Incremental Refinement

In the previous section we saw that the performance of Algorithm 1 critically depends on completeness of semantic tests, and that these tests become significantly weaker

when dealing with multiple constraints. In order to avoid the explosion of “fake” equivalence classes, we can make the compilation process *incremental*. We compile only a subset S' of S into an MDD M and insert this intermediate MDD into S instead of S' . Each vertex in the MDD represents an exact equivalence class for S' and we can take $id_{S'}(p) = id_M(p) = M_p$, allowing us to compute $id_S(p)$ as $id_M(p) \times id_{S \setminus S'}^\times(p)$ which can provide a reduction in the number of fake equivalence classes that is exponential in $|S'|$. This approach is illustrated in Algorithm 2. In each step it compiles a subset of constraints S' in the manner discussed above. We effectively have a number of different compilation approaches, ranging from compiling all constraints in one pass ($S' = S$), similar to [9], to pairwise conjunctions ($|S'| = 1$), which resembles the standard bottom-up compilation approach to building BDDs [10].

Algorithm 2. IncrementalRefine(M, S)

Data: Constraint set S

Result: MDD Representation of the Solution Space of S

$M \leftarrow True(1);$

while $S \neq \emptyset$ **do**

$S' \leftarrow$ some subset of S ;

$M \leftarrow \text{Compile}(S' \cup \{M\});$

$S \leftarrow S \setminus S';$

return M ;

In the remainder of the paper we will focus on pair-wise operations, where one constraint C is combined with one MDD M in each step. This case is especially interesting as it allows us to create some very efficient tests for $S = \{M, C\}$, while in many cases retaining completeness. In particular, all individual constraint tests described previously relying on shortest or longest path computations of C can easily be generalized efficiently for $S = \{M, C\}$ in such a way to preserve completeness. This is achievable because it is easy to compute shortest and longest paths through an MDD as long as the cost function is separable [2]. For the Alldiff the domain entailment test remains complete, but the infeasibility test of Alldiff is no longer complete since it is an NP-complete problem to determine if an MDD contains a solution satisfying an external Alldiff constraint [8]. In addition, it is efficient to detect whether the MDD entails an inequality or equality since we can compute longest and shortest paths efficiently in the MDD, thereby providing a further reduction in the fake equivalence classes. The same is possible for the Alldiff. For each MDD vertex u in layer $l(u)$ we can efficiently compute the set $D(u)$ of values occurring on *all* paths to u . The MDD then entails the Alldiff constraint iff $|D(u)| = l(u) - 1$ for all nodes u , that is iff a distinct set of values leads to each node.

As previously mentioned the above approach of pair-wise compilation closely resembles standard bottom-up compilation. We do however have two significant advantages. Firstly, the standard approach requires each constraint to be represented as an MDD. To see why this is a problem in itself, consider an Alldiff constraint combined with a lexical ordering constraint. The conjunction of these only allows a single solution, but if we build the Alldiff separately, we will require exponential time and space. If

we on the other hand, build the MDD for the lexical ordering constraint first (yielding a polynomial size MDD), we can efficiently compute the conjunction as most of the equivalence classes from the Alldiff need never be considered since they are disallowed by the lexical ordering constraint. Secondly, the semantic information allows us to detect domain- and general-entailment of some interesting constraint types more efficiently. For example, detecting that an inequality is entailed by an MDD is more efficient if it is represented symbolically rather than as an MDD.

5.1 Vertex Splitting

The algorithm described above operates by always constructing an entirely new MDD, instead of updating the input MDD, even when differences between them are only minor. We can try to minimize redundant work by modifying the input MDD rather than creating a new one. We do this by identifying non-equivalent paths ending in the same vertex, and then *splitting* it.

Figure 2 illustrates a vertex-split and the separation of nonequivalent paths for an Alldiff constraint. The edge (u_4, u_5) is, for algorithmic purposes, regarded as two edges that correspond respectively to values 1, 2. If two or more paths coming into u_5 are nonequivalent, we will split u_5 into two vertices in order to refine the MDD. In this case, the paths (u_1, u_2, u_5) and (u_1, u_3, u_5) are equivalent, but other pairs of paths are nonequivalent. We therefore split u_5 into three vertices and distribute the incoming edges between these two vertices in such a way that no two edges coming into a vertex are nonequivalent. No fewer than three vertices will accomplish this.

This algorithm is shown in Algorithm 3 and replaces **Compile**. It traverses the MDD in a breadth-first manner (BFS) manner. Instead of considering the equivalence classes of partial assignments (correspond to paths), it considers equivalence classes of edges (considering an edge (u, v) where $|D_{uv}| > 1$ as $|D_{uv}|$ separate edges). Since the algorithm always splits the previous layer completely before splitting nodes in the next layer, it is guaranteed that all partial assignments ending in a given edge in the next layer belong to the same equivalence class. Therefore the edge can be considered to be identical to any of these paths for the purpose of equivalence, entailment and

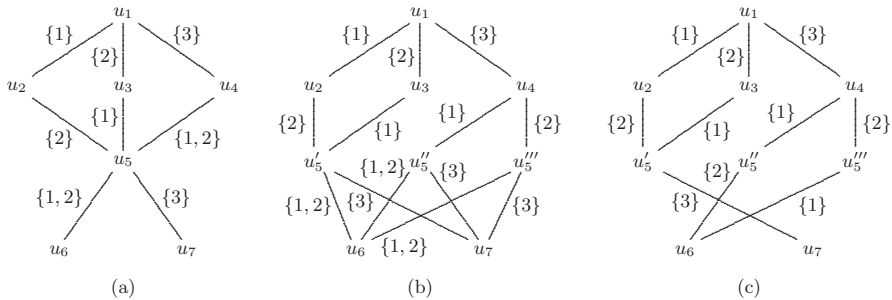


Fig. 2. (a) Part of an MDD just before splitting vertex u_5 with respect to an Alldiff constraint. (b) The edges coming into vertex u_5 have been partitioned into three equivalence classes, and u_5 split into three vertices to receive them. (c) After the split we can prune some infeasible values.

Algorithm 3. SplitCompile(M, C)

Data: MDD M , constraint C
Result: Refined MDD

```

if  $M \Rightarrow C$  then
   $\perp$  return  $M$ ;
foreach vertex  $u \in M$  in layer-by-layer top-down order do
  foreach  $e \in In(u)$  do
    if  $e \equiv_C 0$  then
       $\perp$  delete  $e$  from  $M$  and from  $In(u)$ ;
  if  $In(u) = \emptyset$  then
     $\perp$  delete  $u$  from  $M$ ;
  else if  $In(u) \not\equiv_C 1$  then
    Partition  $In(u)$  into sets  $E_1, \dots, E_m$  such that for each  $e, e' \in E_i, e \equiv_C e'$ ;
    Split( $M, u, E_1, \dots, E_m$ );
   $\perp$ 
return Reduce( $M$ );

```

Algorithm 4. Split($M, u, E_1, \dots, E_m$)

Data: MDD M , vertex u

```

for  $i = 1 \dots m$  do
  Create a new vertex  $u_i$  in  $u$ 's layer of  $M$ ;
  for edges  $(u, u')$  of  $M$  do
     $\perp$  Add edge  $(u_i, u')$  to  $M$  with  $D_{u_i u'} = D_{uu'}$ ;
  for  $(u', u) \in E_i$  do
     $\perp$  Remove edge  $(u', u)$  from  $M$ ;
     $\perp$  Add edge  $(u', u_i)$  to  $M$  with  $D_{u' u_i} = D_{E_i}$ ;
for edges  $(u, u')$  of  $M$  do
   $\perp$  Remove edge  $(u, u')$  from  $M$ ;

```

infeasibility detection. The previously described entailment detection is now done prior to the vertex splitting. Since reduction is not done during splitting, this is performed just before returning the MDD. In our experiments we will rely on this vertex splitting based algorithm to implement the pair-wise conjunction of Algorithm 2.

6 Approximate Compilation

Even when we can compile an MDD for a constraint set using iterative refinement, the MDD may be too large or too hard to compute for practical purposes. This may occur, for example, in an online setting where there is insufficient time or memory to compute an exact MDD. We therefore propose to modify iterative refinement for *approximate semantic compilation*. For given memory and time restrictions we compile an MDD that represents a superset of the solution space. In particular, we produce a sequence of approximate MDDs, each a refinement of the last in the sense that it represents a smaller

superset of the solution space. Each approximate MDD is created by taking all constraints into consideration, thus taking advantage of interactions among the constraints. We also provide approximation guarantees with respect to each constraint.

The basic idea is to regard two partial assignments p_1, p_2 as equivalent for algorithmic purposes when their *distance* is below a threshold $d_{\max}^C$. Thus the equivalence test becomes

$$p_1 \equiv_C p_2 \Leftrightarrow \text{distance}_C(p_1, p_2) \leq d_{\max}^C.$$

A definition of edge equivalence is induced from equivalence of partial assignments in the same way as before. Distance measures are specialized to each type of constraint. For an inequality constraint $\sum_{i=1}^n f(x_i) \leq b$, the distance will be

$$\text{distance}_{\leq}(p_1, p_2) = |a(p_1) - a(p_2)|$$

and similarly for an equality constraint. For Alldiff constraints we can use symmetric difference as a measure of distance:

$$\text{distance}_A(p_1, p_2) = |D(p_1) \triangle D(p_2)|.$$

where $S_1 \triangle S_2 = (S_1 \setminus S_2) \cup (S_2 \setminus S_1)$. Other distance measures could be used as well.

When equivalence is detected in this fashion, Algorithm 1,2 and 3 becomes approximate MDD compilers. The resulting MDD guarantees that any two paths entering the same vertex differ by at most $d_{\max}^C$ with respect to constraint C . If the infeasibility test is complete, then we create no infeasible vertices, and the number of redundant equivalence classes can be limited as desired by adjusting the bound $d_{\max}^C$.

The overall procedure for approximate compilation begins with a trivial MDD M (consisting of the single vertex $\text{True}(1)$) and some initial large distance. It then refines M using `SplitCompile` for each of the constraints in S using the distance based equivalence tests. The process is then repeated with lower distance thresholds, obtained from the previous thresholds by, for example, a multiplicative or additive factor.

The advantage of this approach, regardless of whether the goal is exact or approximate compilation, is that after one distance is processed, the resulting MDD takes all constraints into consideration. This means that we at any time have a bound on the degree of violation on each constraint in the current MDD. In addition, it allows obvious inconsistencies to be removed from the solution space at an earlier time, preventing the corresponding equivalence classes from taking up computation time in subsequent steps. Therefore it can in fact be advantageous to compute an exact MDD through a sequence of approximations in which the distance thresholds are gradually reduced to zero.

7 Experiments

In this section we will show how the presented techniques perform in practice for a selection of applications. Implementation of techniques discussed in this paper is using our own MDD compiler and generic MDD-manipulation package, written in Java. Comparisons to standard compilation techniques makes use of CLab [11].

7.1 Approximation Quality Tradeoff

In the first set of experiments we evaluate the overall quality of our approximation scheme. For an MDD M , and a constraint C we create an approximate MDD M_{apx} with increasing precision (decreasing maximal distance threshold d_{max}^C) and without size limit ($T_{max} = \infty$). For each d_{max}^C we generate the approximate MDD and report its number of edges and solutions. The results are shown in Figure 3, and we can for example see that the solution count decreases super-linearly as a function of MDD size.

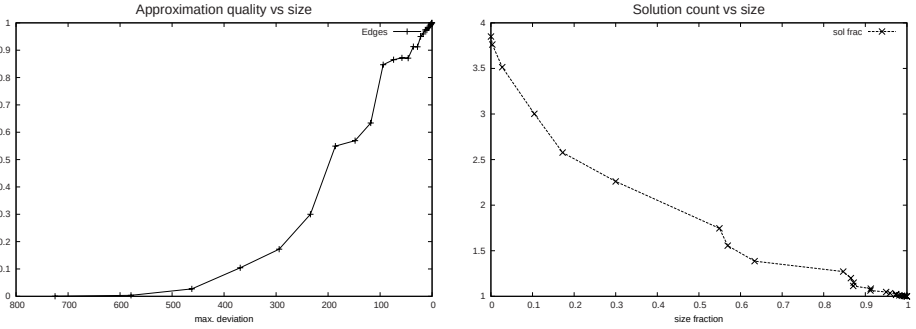


Fig. 3. The two plots above tracks the progress of the approximate compilation process of 5 random separable inequalities, with 15 variables over domains of size 3 and matrix elements from the range -100 to 100 . The leftmost plot shows the approximated distance achieved on the horizontal axis and the size of the MDD on the vertical axis. Since the instance consists of inequalities, the distance corresponds to an upper bound on how much the longest path can violate the bound. The rightmost plot shows the trade-off achieved between solution count and MDD size.

7.2 Approximate Refining for Exact Compilation

In the second set of experiments we illustrate how approximative refining can be a competitive method for exact compilation. We postulated previously that the use of approximate refinement steps with distance thresholds gradually reducing to 0 might be beneficial for exact compilation. We therefore compared the CLab compilation approach, precise refining, and approximate refining for a single randomly generated linear inequality, as well as for a set of linear inequalities over binary variables. Compiling a single inequality might be relevant for assisting a standard compiler (such as CLab) in compiling individual rules, while the set of linear inequalities illustrates behavior when we have weak equivalence detection due to a lack of strong semantics. The results are shown in Figures 4(a) and 4(b).

For a single inequality, we can observe that the number of vertices created by CLab is nearly unaffected by tightness. This is due to the mechanism used in CLab for constructing the BDD for an inequality, which compiles a BDD for each bit of the left-hand side and then builds the BDD by comparing these with the bit representation of the right-hand side. We can also see that both precise and approximate compilation outperform CLab in the number of vertices generated. With regard to time (which is not shown), the approximate compiler outperforms CLab on tightness less than 0.2 and greater than 0.8.

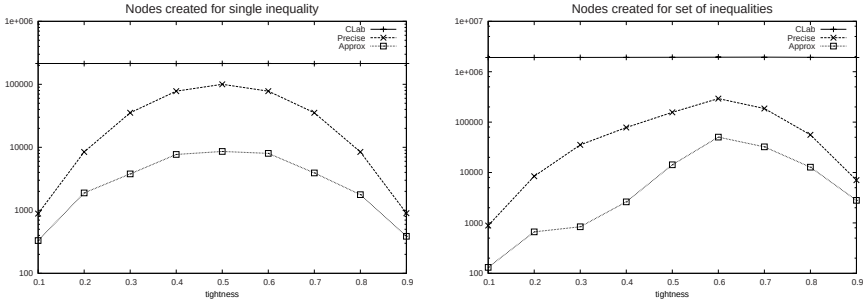


Fig. 4. (a) Total number of vertices created during compilation of a single random linear inequality over 18 variables with binary domains. The coefficients range from 0 to 100000. (b) Total number of vertices created during compilation of 10 random linear inequalities over 18 variables with binary domains. The coefficients range from 0 to 100000. Instances with tightness up to and including 0.4 are unsatisfiable.

The second experiment considers a set of linear inequalities. Again we observe that CLab is almost unaffected by the tightness of the constraints. This is again due to the construction mechanism mentioned before. In fact more than 99% of the vertices created by CLab are generated during the construction of BDDs for individual inequalities, and most of these vertices are created before considering the right-hand side of each constraint. The precise vertex-splitting based compiler produces far fewer vertices, while approximate compilation reduces this number even further, clearly outperforming precise compilation. With regards to time (not shown), the (approximate) vertex-splitting compiler is fastest up to and including tightness 0.5 and again for tightness greater than 0.8. CLab is fastest between 0.5 and 0.8. Note, however, that CLab is based on highly optimized C code, while our Java implementation is far from optimized.

7.3 Interactive Configuration

In our final set of experiments we assess the usefulness of approximative MDD compilation for one of its main application areas: interactive configuration. We consider a scenario where an MDD M^{init} is given for an initially compiled configuration instance along with a set S of external (resource) constraints, which have not been compiled either because the resulting MDD is too large, or the constraints are not known at the time of compilation.

In the presence of external constraints, it is NP-hard to prune all *non-GAC* values; that is, values that are not *generalized arc consistent* with respect to the conjunction $M^{init} \wedge S$ of all constraints. A user is therefore exposed to backtracking, because he is presented with non-GAC values as valid options due to incomplete (but time efficient) pruning algorithms. This often occurs in practice¹ and is regarded negatively. We therefore explore whether approximate compilation can be used to remove non-GAC values while still observing strict time and memory limitations.

¹ Think of buying an airplane ticket online and getting the message, “There is no flight on selected dates. Please go back and try again.”

After each user assignment, we compute initial valid domains, and while the user is assessing available options we refine the existing MDD with respect to S to get refinement M^{apx} . This MDD is then additionally cost-pruned with respect to each constraint $C \in S$, in the sense of cost-bounded configuration [2], and the domains displayed to the user are updated. As an alternative to approximate compilation, we consider computing valid domains only with respect to the initial MDD M^{init} , or with respect to M^{init} after cost pruning. We abbreviate the first scenario as *ApxP* (approximation + cost pruning) and denote the other two as *Init* and *InitP*, respectively. The approach of the last scenario in itself leads to strictly more pruning than in the case of standard CSP propagation, in which the MDD and the constraints in S are posted individually as global constraints.

For the initial MDD M^{init} we loaded an MDD representing the real-world configuration instance “PC” (a personal computer configuration problem), available in the CLib benchmark suite [12]. It has 45 finite-domain variables of up to 33 domain values and 4875 vertices. We then generated a set S of external constraints. For each $m \in \{2, 3, \dots, 13\}$ we generated 10 models of m random separable inequalities, each with a tightness $t = 0.5$. For a separable cost expression $\sum_i c_i(x_i)$ we set the right-hand side bound to $b = \min_c + (\max_c - \min_c) \cdot t$, where $\min_c$ and $\max_c$ are the minimal and the maximal value of the cost function c . We set the maximal vertex size threshold T_{max} to 5000. For each set of separable inequalities we measured a number of parameters averaged over 100 interaction simulations (where in each simulation we randomly simulated user assignments until there was only one solution left). In Table 1 we report, for each number of constraints m , the median of these values over the 10 generated models.

Table 1. Effect of approximate compilation on reducing the non-GAC values in user interaction. Column m indicates the number of external constraints C . M^{apx} is the maximal size of an approximate MDD encountered. M^e is the size of the MDD representing entire conjunction exactly $M^{init} \wedge C$. Columns *Init*, *InitP* and *ApxP* denote the probability of selecting non-GAC value for the three scenarios previously described. Column *Subsume* indicates the average subsumption depth, i.e. after how many assignments does approximate MDD become exact. Finally, columns *Refine* and *Reduce* indicate the number of seconds spent for generating approximate M^{apx} and subsequent elimination of redundant equivalence classes.

m	M^{apx}	M^e	<i>Init</i> (%)	<i>InitP</i> (%)	<i>ApxP</i> (%)	<i>Subsume</i>	<i>Refine</i> (s)	<i>Reduce</i> (s)
13	7894	732	18	7	0.5	1.17	1.27	0.48
12	5838	253	19	6.9	0	0	1.07	0.48
11	5616	872	18	6.3	0	0	0.75	0.43
10	6081	2471	18	6.8	0.1	1	0.89	0.39
9	5031	258	19	5.1	0	0	0.86	0.36
8	7474	3676	16	4.8	0.02	1.45	0.94	0.40
7	6925	2849	16	4.7	0.02	1.43	0.70	0.31
6	6827	7797	14	2.5	0.02	1.62	0.64	0.28
5	7112	17965	11	2.0	0.01	2.17	0.56	0.24
4	7336	25030	11	1.8	0.02	2.42	0.48	0.21
3	7957	35092	9.8	0.82	0.006	2.56	0.42	0.18
2	7231	43108	6.2	0.22	0.0002	3.08	0.29	0.13

The probability of selecting a non-GAC value was assessed by comparing, for every unassigned variable, the size of the domains represented to the user (D^{init} , D^{initP} , and D^{apxP}) against the actual number of non-GAC values D^e . More precisely, if domain D_i is shown to the user, but only a subset D_i^e of values are GAC, then we compute the probability $\frac{|D_i| - |D_i^e|}{|D_i|}$ of selecting a non-GAC value with respect to a single variable. We then average the probability over all unassigned variables and repeat this for every assignment in a simulation. If U_j was the set of unassigned variables at interaction step j , and there were a total of k assignments when the solution was completely specified, we compute:

$$\frac{1}{k} \sum_{j=1}^k \frac{1}{|U_j|} \cdot \sum_{i \in U_j} \frac{|D_i| - |D_i^e|}{|D_i|}$$

as the probability of selecting a non-GAC value in an interaction simulation.

Table 1 indicates that approximate compilation almost entirely eliminates the probability of backtracking. On average, scenario *ApxP* using approximate compilation reduces by several orders of magnitude the probability of selecting a non-GAC value, compared to the *InitP* and especially the *Init* scenario. While *InitP* performs well for a smaller number of constraints (below 1% for two constraints), the probability of backtracking increases with the number of constraints (7% for 13 constraints). Computing domains over initial MDD in *Init* scenario leads to a significant backtracking probability that increases with the number of constraints, up to 19%. Subsumption depth for approximate compilation is very shallow. After an average of 1-3 assignments, the MDD becomes exact. Since we fixed the tightness of individual constraints, the overall tightness of the solution space increases with the number of constraints. As a result, exact MDDs get increasingly smaller, while approximate MDDs are relatively stable. The combined running time for refinement and reduction phase is usually below 1.5 seconds, which is more than acceptable in our interaction setting: we first show initial domains, and while the user is investigating those, we further refine based on an approximate MDD.

8 Conclusions

We presented an incremental refinement algorithm based on vertex splitting, for approximate compilation of constraint satisfaction models into MDDs. The presented approach utilizes the semantics of constraints and a notion of distance to obtain approximate MDDs. Our empirical evaluation demonstrated that approximate refinement can be a competitive compilation method and that significant reductions in backtracking can be made by approximately compiling external constraints during interactive configuration.

Acknowledgments. Tarik Hadzic is supported by an IRCSET/Embark Initiative Postdoctoral Fellowship Scheme. Barry O’Sullivan is supported by Science Foundation Ireland (Grant Number 05/IN/I886).

References

1. Vempaty, N.R.: Solving constraint satisfaction problems using finite state automata. In: Proceedings of the Tenth National Conference on Artificial Intelligence, pp. 453–458 (1992)
2. Hadzic, T., Andersen, H.R.: A BDD-based Polytime Algorithm for Cost-Bounded Interactive Configuration. In: Proceedings of AAAI 2006 (2006)
3. Mateescu, R., Dechter, R.: Compiling constraint networks into AND/OR multi-valued decision diagrams (AOMDDs). In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 329–343. Springer, Heidelberg (2006)
4. Darwiche, A., Marquis, P.: A knowledge compilation map. *Journal of Artificial Intelligence Research* 17, 229–264 (2002)
5. Hadzic, T., Hooker, J.N.: Cost-bounded binary decision diagrams for 0-1 programming. In: Hentenryck, P.V., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 84–98. Springer, Heidelberg (2007)
6. Ravi, K., Somenzi, F.: High-density reachability analysis. In: ICCAD 1995: Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design, Washington, DC, USA, pp. 154–158. IEEE Computer Society, Los Alamitos (1995)
7. Ravi, K., McMillan, K.L., Shiple, T.R., Somenzi, F.: Approximation and decomposition of binary decision diagrams. In: Design Automation Conference, pp. 445–450 (1998)
8. Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemman, P.: A Constraint Store Based on Multivalued Decision Diagrams. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 118–132. Springer, Heidelberg (2007)
9. Huang, J., Darwiche, A.: DPLL with a trace: From SAT to knowledge compilation. In: Kaelbling, L.P., Saffioti, A. (eds.) IJCAI, pp. 156–162. Professional Book Center (2005)
10. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* (1986)
11. Jensen, R.M.: CLab: A C++ library for fast backtrack-free interactive product configuration (2007), <http://www.itu.dk/people/rmj/clab/>
12. CLib: Configuration benchmarks library (2007), <http://www.itu.dk/research/cla/externals/clib/>

Quantified Constraint Optimization^{*}

Marco Benedetti, Arnaud Lallouet, and Jérémie Vautard

Université d'Orléans — LIFO

BP 6759 — F-45067 Orléans cedex 2

{Marco.Benedetti,Arnaud.Lallouet,Jeremie.Vautard}@univ-orleans.fr

Abstract. Solutions to valid Quantified Constraint Satisfaction Problems (QCSPs) are called winning strategies and represent possible ways in which the existential player can react to the moves of the universal one to “win the game”. However, different winning strategies are not necessarily equivalent: some may be preferred to others. We define Quantified Constraint Optimization Problems (QCOP) as a framework which allows both to formally express preferences over QCSP strategies, and to solve the related optimization problem. We present examples and some experimental results. We also discuss how this framework relates to other formalisms for hierarchical decision modeling known as von Stackelberg games and bilevel (and multilevel) programming.

1 Introduction

QCSP (Quantified Constraint Satisfaction Problems) is a constraint-based framework used to model several problems that go beyond classical CSP, such as those involving some degree of uncertainty in the state of the modeled reality, and those structured as game playing [1,2,3] or adversary problems [4], such as conformant planning, model checking, testing, and robust scheduling [5], to name a few.

In QCSP variables may be universally quantified over their domains. Such universal quantification is crucial while modeling, for example, the behavior of a hostile adversary or some potentially harmful uncertainty about the state of the environment. This expressive power comes at a cost: While CSP is solved by just exhibiting values for its (existentially quantified) variables such that all the constraints are satisfied, a QCSP is solved by exhibiting *winning strategies*. A strategy is a set of functions that compute the values of each (existentially quantified) variable in the problem as a function of all the relevant (universally quantified) variables. A *winning strategy* is a strategy that, given whatever assignment to the universal variables, manages to satisfy all the constraints by the values of the existential variables: then, a QCSP is true if it has at least one winning strategy. Strategies, unlike satisfying CSP assignments, are not always docile objects: their worst-case size is exponential in the number of variables.

Given a true QCSP instance, are all its winning strategies equally desirable? It turns out that some strategies should be preferred over others, despite being

^{*} This work is supported by the project ANR-06-BLAN-0383.

equally “winning”. The contribution of this paper is to present a framework, called Quantified Constraint Optimization (QCOP⁺), to express preferences over strategies, and a reasoning engine that solves the resulting optimization problem.

In CSP we express preferences over the set of solutions by means of an objective function, and solve the related optimization problem by determining the satisfying assignment(s) that maximize(s)/minimize(s) such objective. Similarly, in a game-like scenario modeled in QCSP we could be interested, for example, in playing the strategy that gives the earliest win, or in selecting strategies with features that cannot be enforced at the level of the constraint language. As an example, suppose that we face a game-like situation in which strategies to force the opponent to a tie exist, though we cannot defeat him if he plays perfectly. Out of all the strategies that prevent the opponent from winning, we prefer those leading to our win in case the opponent plays less than perfectly. This induces a preference over the space of winning strategies which cannot be modeled in plain QCSP (any attempt to require a strict defeat of the opponent by additional constraints would result in a false instance, as the opponent cannot be overcome in general). QCOP⁺ is the perfect framework for modeling similar situations.

The QCOP⁺ language we introduce is based on QCSP⁺ [3] and extends it by providing means to define *compositional objective functions* built along the quantification structure of the problem. With each universal quantification we associate some *aggregate function*, while *optimization functions* (e.g., minimization, maximization) are associated with existential levels. Let us consider, for example, the QCOP⁺ in Figure 1, in which the domains are numerical. We associate the components of the optimization function with a specific scope by indenting them at the same level as the quantifier they refer to. If we momentarily disregard

- (1) $\exists X \in D_X \cdot [C_1(X)]$
- (2) $\forall Y \in D_Y \cdot [C_2(X, Y)]$
- (3) $\exists Z \in D_Z \cdot [C_3(X, Y, Z)]$
- (4) $C(X, Y, Z)$
- (5) $\min(Z)$
- (6) $k : \text{sum}(Z)$
- (7) $\max(k)$

Fig. 1. Example of QCOP⁺

our sample QCOP⁺ instance asks to identify the subset $W' \subseteq W$ of winning strategies which, beyond satisfying P , optimize the objective function:

$$\max_{s_X \in W_X} \left[\sum_{Y \in D_Y} \left(\min_{\langle s_X, s_Z \rangle \in W} s_Z(Y) \right) \right]$$

Any QCOP⁺ instance is thus composed of two parts: an initial QCSP⁺ portion which identifies candidate winning strategies, followed by a quantifier-by-quantifier specification of an objective function meant to describe optimal candidates. These two parts belong to conceptually different languages and manipulate

lines (5 – 7), we recognize a standard QCSP⁺ instance $P = \exists X \in D_X [C_1] \forall Y \in D_Y [C_2] \exists Z \in D_Z [C_3]$. C , where the conditions C_1 to C_3 are used to restrict dynamically the possible values a variable may assume. Let $W \subseteq S_X \times S_Z$ be the set of winning strategies for such problem, where S_X is the space of (constant) functions onto D_X and S_Z is the space of functions $s_Z : D_Y \mapsto D_Z$, and let W_X denote the set $\{s_X \in S_X : \exists s_Z \langle s_X, s_Z \rangle \in W\}$. Then,

different domains: the first part deals with truth assignments to the problem variables, the second one is concerned with strategies and sub-strategies.

By mixing them in a single specification we obtain several benefits: we make as easy as possible the task of declaring (complex) preferences over (complex) strategies; we keep all the information about the quantified optimization problem in a compact specification; most importantly, as we shall see, we give the solver the possibility to exclude on-the-fly partially-formed sub-optimal candidates, in the spirit of branch and bound algorithms.

We have found no previous account for a general notion of optimization in the QCSP literature. However, this kind of problems has been studied since the 70's in mathematical programming under the name of *bilevel* (or *multi-level*) programming [6], a.k.a. “mathematical programs with optimization problems in the constraints”. Bilevel programs are used for solving decision problems of the form of Stackelberg games, which is a model of oligopoly in game theory [7]. In this kind of problems, there are two actors who perform decisions sequentially but have no control on each other. The first one to act is called “leader”; the second one, called “follower”, uses the leader decisions to adapt her own ones towards her objective. The key issue here is that the leader and the follower have different objective functions, that may be conflicting. For example, the leader can be a government agency which divides an amount of money among several entities which are free to use their amount for their own purpose. The following example, taken from [8], shows that conflicting objectives can lead to a non-optimal equilibrium. In the situation depicted in Figure 2, the choice of (x_1, y_1) which would be optimal without the follower becomes considerably sub-optimal with her response (x_2, y_1) . The equilibrium x^* is depicted in Figure 3, where it can be noticed that dominating solutions exist, for both the leader and the follower, which can never be reached without consensus. We refer to [9] for an extensive survey of bilevel programming. The name multi-level programming applies when more than two levels of hierarchical decisions are involved.

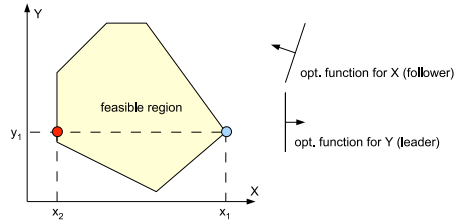


Fig. 2. Optimum for the leader alone and response of the follower

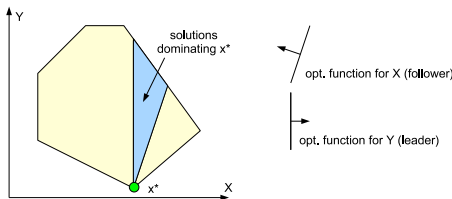


Fig. 3. Optimal equilibrium

The QCOP⁺ framework has been prototyped in the solver QeCode [10] based on Gecode [11]. In addition to optimization, it includes strategy extraction (useful also for classical QCSP⁺, where a simple *true/false* answer is often insufficient). The extraction of strategies has been introduced in [12] in the context of QBF.

The paper presents QCSP, QCSP⁺ and then QCOP⁺ and their evaluation. The search algorithm is presented and its branch and bound variant studied. Finally some examples of bilevel problems are presented.

2 QCSP

Notations. Let V be a set of variables and $D = (D_X)_{X \in V}$ be the family of their domains. We recall that a family is a function from an index set to a set. For $W \subseteq V$, we denote by D^W the set of tuples on W , namely $\prod_{X \in W} D_X$. Projection of a tuple (or a set of tuples) on a variable (or a set of variables) is denoted by $|$. For example, for $t \in D^V$, $t|_W = (t_X)_{X \in W}$ and for $E \subseteq D^V$, $E|_W = \{t|_W \mid t \in E\}$. For $W, U \subseteq V$, the join of $A \subseteq D^W$ and $B \subseteq D^U$ is $A \bowtie B = \{t \in D^{A \cup B} \mid t|_W \in A \wedge t|_U \in B\}$. A sequence is a family indexed by a prefix of $\mathbb{N}$. We denote by $|$ the sequence constructor and by $[]$ the empty sequence. We use $a?b:c$ to denote *if a then b else c* .

Constraints and CSPs. A constraint $c = (W, T)$ is a couple composed of a subset $W \subseteq V$ of variables and a relation $T \subseteq D^W$ (W and T are also respectively noted $var(c)$ and $sol(c)$). An empty constraint such that $sol(c) = \emptyset$ is *false* and a full constraint (which does not constrain the variables) is such that $sol(c) = D^W$. When $W = \emptyset$, only these two constraints exist: $(\emptyset, \emptyset)$ which has value *false* and $(\emptyset, ())$ which has value *true*.

A *Constraint Satisfaction Problem* (or CSP) is a set of constraints. We denote by $var(C) = \bigcup_{c \in C} var(c)$ its set of variables and by $sol(C) = \bigcap_{c \in C} sol(c)$ its set of solutions. The empty CSP which contains no constraint is *true* and will be denoted by $\top$ while any CSP which contains a false constraint is *false* and denoted by $\perp$.

Prefix and QCSP. A *quantified set of variables*, or *qset* is a couple (q, W) where $q \in \{\exists, \forall\}$ is a quantifier and $W \subseteq V$.

Definition 1 (Prefix). A prefix P is a sequence of qsets $[(q_0, W_0), \dots, (q_{n-1}, W_{n-1})]$ such that $i \neq j \Rightarrow W_i \cap W_j = \emptyset$.

We denote by $P|_W$ the prefix P restricted to the variables of a set W . A variable X is *declared* in a qset W_i if $X \in W_i$. A QCSP is defined by adding a CSP to a prefix:

Definition 2 (QCSP). A Quantified CSP, or QCSP is a couple (P, G) where P is a prefix and G is a CSP called goal.

Let $P = [(q_0, W_0), \dots, (q_{n-1}, W_{n-1})]$ be a prefix. We define the following notations. First, let $range(P) = [0..n]$. For all i in $range(P)$, let $var_i(P) = W_i$ be the set of variables at index i , let $before_i(P) = \bigcup_{j \leq i} var_j(P)$ (resp. $after_i(P) = before_n(P) \setminus before_i(P)$) be the set of all variables defined before (resp. after) the index i . We also need to access to the index of the next universal block $nu_i(P)$ located after an index i . We define $nu_i(P) = \min_{j > i} \{j \mid q_j = \forall\}$ if such an index exists, and n otherwise. These notions are naturally extended for QCSP

$Q = (P, G)$ in a straightforward way. Moreover, we have $\text{prefix}(Q) = P$ and $\text{goal}(Q) = G$. The QCSP is *closed* if $\text{var}(G) = \text{before}_n(Q)$, i.e. all variables mentioned in the goal are explicitly quantified. In the sequel, we only consider closed QCSP.

Example 3 (QCSP). The formula:

$$\exists X \in \{0, 1\}, \forall Y \in \{0, 1\}, \exists Z \in \{1, 2\} . X + Y = Z$$

is represented by the following QCSP, in which the domains attached to the variables are not mentionned:

$$Q = ((\exists, X), (\forall, Y), (\exists, Z)), \{X + Y = Z\}$$

Thus, $\text{prefix}(Q) = [(\exists, X), (\forall, Y), (\exists, Z)]$, $\text{goal}(Q) = \{X + Y = Z\}$, $\text{range}(Q) = [1..3]$, $\text{var}_1(Q) = \{X\}$, $\text{before}_2(Q) = \{X, Y\}$, $\text{after}_2(Q) = \{Z\}$. $\square$

Strategy and scenario. A solution, called *strategy*, is intuitively the way the existential player react to every possible move of the the universal player. It is interesting to note that a strategy is a syntactic object that does not depend on a notion of validity. It is just a possible way to play the game as if there is no rule. As a consequence, it can be defined for a prefix only. In [13], a strategy was defined as a family of (Skolem) functions that give a value to an existential variable as a function of its preceding universal ones. For the purpose of this set-theoretic exposition, we rather define it in extension, as a set of tuples. Each of these tuples is a *scenario*, i.e. a possible way the game is played. Here follows the inductive definition of the set of strategies for a given prefix:

Definition 4 (Set of strategies). *The set of strategies $\text{Strat}(P)$ for a prefix $P = [(q_0, W_0), \dots, (q_{n-1}, W_{n-1})]$ is defined inductively as follows:*

- $\text{Strat}([]) = \emptyset$
- $\text{Strat}([(\exists, W) \mid P']) = \{ t \bowtie s' \mid t \in D^W \wedge s' \in \text{Strat}(P') \}$
- $\text{Strat}([(\forall, W, C) \mid P']) = \{ \bigcup \alpha(D^W) \mid \alpha \in \Pi_{t \in D^W} (\{ t \bowtie s' \mid s' \in \text{Strat}(P') \}) \}$

The set of strategies for a prefix beginning with an universal variable is defined as follows: we build, for a tuple $t \in D^W$, the set $\{ t \bowtie s' \mid s' \in \text{Strat}(P') \}$ of all strategies beginning by t . Then we take the Cartesian product $\Pi_{t \in D^W} (\{ t \bowtie s' \mid s' \in \text{Strat}(P') \})$ of all these sets. Each tuple α of this Cartesian product has as value a strategy beginning by a tuple t for every $t \in D^W$. Such tuple α is also a function that associates to each tuple of D^W a strategy, which is a set of tuples. The union of the strategies of the image set $\alpha(D^W)$ of this function is a new strategy which contains a sub-strategy for each $t \in D^W$. The set of strategies for the prefix is the set of all strategies constructed by all tuples α .

Semantics of QCSP. A strategy is a *winning strategy* if all of its scenarios satisfy the goal:

Definition 5 (Winning Strategy for QCSP). *A strategy s is a winning strategy iff $s|_{\text{var}(G)} \subseteq \text{sol}(G)$.*

We denote by $\text{WIN}(Q)$ the set of all winning strategies of Q .

Definition 6 (Semantics of QCSP). *The semantics $\llbracket Q \rrbracket$ of a QCSP Q is:*

$$\llbracket Q \rrbracket = \text{Win}(Q)$$

This notion of solution generalizes exactly the classical notion of solution of CSP: a QCSP is true if it has a winning strategy. Other weaker notions have been proposed. The notion of *outcome*, which is the set of scenarios of all winning strategies, has been used as a notion of solution for QCSP in [13] to model filtering.

Example 7. Consider the following QCSPs:

$$\begin{aligned} Q_1 &: \forall x \in \{0, 1\}, \exists y \in \{1\}, \exists z \in \{0, 1\}. x \vee y = z \\ Q_2 &: \exists x \in \{0, 1\}, \forall y \in \{1\}, \forall z \in \{1\}. x \vee y = z \\ Q_3 &: \exists x \in \{0, 1\}, \forall y \in \{0, 1\}, \exists z \in \{1\}. x \vee y = z \end{aligned}$$

With the tuples defined for the values of x , y and z respectively, we have:

$$\begin{aligned} \llbracket Q_1 \rrbracket &= \{ \{(0, 1, 1), (1, 1, 1)\} \} \\ \llbracket Q_2 \rrbracket &= \{ \{(0, 1, 1)\}, \{(1, 1, 1)\} \} \\ \llbracket Q_3 \rrbracket &= \{ \{(1, 0, 1), (1, 1, 1)\} \} \end{aligned}$$

QCSP⁺. Introducing restricted quantification in QCSPs means changing the nature of the prefix. In addition to a quantifier and a set of variables, each scope includes a CSP whose solutions define the allowed values for the variables of the current qset. QCSP⁺ have been introduced in [3] mainly for modeling purposes. A *restricted quantified set of variables*, or *rqset* is a triple (q, W, C) where $q \in \{\exists, \forall\}$ is a quantifier, $W \subseteq V$ and C is a CSP. The intended meaning is to restrict the possible values of the variables of W to those which satisfy the CSP C . We extend the notion of prefix to rqsets. In particular, it is still required that $i \neq j \Rightarrow W_i \cap W_j = \emptyset$.

Definition 8 (QCSP⁺). *A QCSP⁺ is a couple $Q = (P, G)$ where P is a prefix of rqsets such that $\text{var}(C_i) \cap \text{after}_i(Q) = \emptyset$ and G is a goal CSP.*

A QCSP⁺ $Q = (P, G)$ is closed if $\forall i \in \text{range}(P), \text{var}(C_i) \subseteq \text{before}_i(Q)$ and $\text{var}(G) \subseteq \text{before}_n(Q)$. It is easy to notice that a standard QCSP is a QCSP⁺ for which $\forall i \in \text{range}(P), C_i = \emptyset$. The definition of strategy for a QCSP⁺ is the same as for a QCSP. But being a winning strategy is different. A winning strategy is a strategy for which all possible moves for the universal player end in a winning scenario. This can happen, like in a classical QCSP, when all constraints of the goal and of all restrictions are satisfied because all implications and conjunctions are valid. But, it can also happen when one left-hand side of an implication is contradicted. Then this scenario is valid whatever happens in the remaining assignments of variables after the contradicted rqset's CSP. The set of winning strategies of a QCSP⁺ can be defined recursively as follows:

Definition 9 (Set of winning strategies for a QCSP⁺). Let Q be a QCSP⁺. The set of winning strategies $\text{WIN}(Q)$ is defined by:

- $\text{WIN}([\Box], G) = \text{sol}(G)$
- $\text{WIN}([\Box](\exists, W, C)|P', G) = \{t \bowtie s \mid t \in D^W \wedge t|_{\text{var}(C)} \in \text{sol}(C) \wedge s \in \text{WIN}(P', G)\}$
- $\text{WIN}([\Box](\forall, W, C)|P', G) = \{ \bigcup \alpha(D^W) \mid \alpha \in \prod_{t \in D^W} (\{t \bowtie s \mid t|_{\text{var}(C)} \in \text{sol}(C) \text{ ? } s \in \text{WIN}(P', G) : s \in \text{STRAT}(P', G)\}) \}$

This definition is similar to the definition of the set of strategies for a prefix. However, classical winning sub-strategies are not the only ones to take into account: A strategy can be winning at a universal level if it contradicts the related restriction. It follows that any sub-strategy can be freely glued, whatever its winning status. To reason on all the CSP restrictions at once, a kind of propagation called cascade propagation is introduced in [3].

3 Optimization

QCOP⁺ is formed after QCSP⁺ by adding preferences and aggregates to the rqsets. Let $\mathcal{A}$ be a set of aggregate names and $\mathcal{F}$ be a set of aggregate functions. An aggregate function is defined by an associative function on a multiset of values and a neutral element 0_f which indicates the value of $f(\{\!\!\{\}\!\!\})$. Possible functions are *sum*, *product*, *average*, *standard deviation*, *median*, *count*, etc. If the function is associative and commutative, it can be evaluated using an accumulator initialized to 0_f and the evaluation could be parallelized. An *aggregate* is an atom of the form $a : f(X)$ where $a \in \mathcal{A}$, $f \in \mathcal{F}$ and $X \in V \cup \mathcal{A}$. We call $\text{names}(A)$ the set of aggregate names of a set of aggregates A . An aggregate name has the same status as a variable, except that it cannot be part of a constraint. An *optimization condition* is an atom of the form $\min(X)$, $\max(X)$ where $X \in V \cup \mathcal{A}$ or the atom *any*. An atom $\min(X)$ indicates that the user is interested in strategies that minimize this value and not in the other ones, while *any* simply indicates she does not care about the returned strategy. It is only needed to define minimization since $\max(X)$ is a syntactic sugar for $\min(-X)$.

Definition 10 (Orqset). An $\exists$ -orqset is a 4-uple $(\exists, W, C, o)$ where $(\exists, W, C)$ is a rqset and o is an optimization condition. A $\forall$ -orqset is a 4-uple $(\forall, W, C, A)$ where $(\exists, W, C)$ is a rqset and A is a set of aggregates. An orqset is either an $\exists$ -orqset or a $\forall$ -orqset.

The notion of prefix and all adjoin notations defined in notation 2 are extended to a sequence of orqsets. There is a restriction on the variables which can appear in an optimization condition or an aggregate. Actually, it should be ensured that the variable to be optimized will have an unique value in the current strategy. This is the case if this variable is not in the scope of an universal quantifier located after the optimization condition. However, it can be an aggregate of the next universal block since it will also have an unique value. The same holds for the variable of an aggregate: it can belong to the set of variables of any existential scope between the aggregate declaration and the next universal block's aggregates.

Definition 11 (QCOP and QCOP⁺). A QCOP⁺ is a couple (P, G) where G is a CSP and $P = [orq_0, \dots, orq_{n-1}]$ is a prefix of orqsets such that $\forall i \in \text{range}(P)$, with $k = \text{nu}_i(P)$:

- if $orq_i = (\exists, W, C, o)$ with $o = \min(X)$ or $o = \max(X)$, then we must have $X \in \text{before}_{k-1}(P) \cup (k < n ? \text{names}(A_k) : \emptyset)$
- if $orq_i = (\forall, W, C, A)$, then for all $a : f(X)$ in A , we must have $X \in \text{before}_{k-1}(P) \cup (k < n ? \text{names}(A_k) : \emptyset)$

A QCOP is a QCOP⁺ in which no orqset has restrictions.

The semantics of a QCOP⁺ is defined as a set of strategies which include the computation of the aggregates and which respect the optimization conditions. We first define the function val which computes the value of an aggregate $a:f(X)$ for a strategy s . We have $\text{val}(a:f(X), s) = f(\{\{t|_X \mid t \in s\}\})$.

Definition 12 (Semantics of QCOP⁺). The semantics of a QCOP⁺ is a set of strategies defined as follows:

- $\text{WIN}([\], G) = \text{sol}(G)$
- $\text{WIN}([\![\exists, W, C, \text{any}]\!]P', G) = \text{WIN}([\![\exists, W, C]\!]P', G)$
- $\text{WIN}([\![\exists, W, C, \min(X)]\!]P', G) =$
 $\{s \in \text{WIN}([\![\exists, W, C]\!]P', G) \mid s|_X = \min_{s' \in \text{WIN}([\![\exists, W, C]\!]P', G)}(s'|_X)\}$
- $\text{WIN}([\![\forall, W, C, A]\!]P', G) =$
 $\{(val(a:f(X), s))_{a \in \text{names}(A)} \bowtie s \mid s \in \text{WIN}([\![\forall, W, C, A]\!]P', G)\}$

After the aggregates are evaluated, their value are glued to the scenarios of the strategy and they appear as if they were existential variables of the preceding level. As for CSP which can have multiple optimal solutions, a QCOP⁺ may have multiple optimal strategies. It can happen with the use of *any*, but also when multiple strategies have the same optimal value. They may differ a lot on subsequent optimal values found in sub-strategies. However, the search algorithm described in the next section returns one of these optimal strategies only.

4 Algorithms

This section presents a search algorithm to evaluate QCOP⁺ and its version based on branch and bound. It is implemented in the solver QeCode [10] based on Gecode [11]. The solving technique is based on the QCSP⁺ search procedure that recursively explores the quantified structure. A mechanism for strategy extraction and its recording in a tree is implemented. This feature also benefits to QCSP⁺ because in many cases the user is interested not only in the decision problem but also in the way the game can be played. An explicit representation of strategies—called certificate in [12]—has numerous applications, the first one being to be able to verify the solution in a solver-independent way. In the current


```

Procedure Solve ( $[o|P'], G$ )
  if  $o = \text{existential orqset}$  then
    return Solve_e ( $[o|P'], G$ )
  else
    return Solve_u ( $[o|P'], G$ )
  end if

Procedure
Solve_e ( $[(\exists, W, C, \min(X))|P'], G$ )
  BEST_STR := null
  BEST_Xvalue :=  $+\infty$ 
  for all  $t \in D^W$  s.t.  $t$  is a solution of  $C$ 
  do
    CUR_STR := Solve(  $(P', G)[W \leftarrow t]$  )
    if CUR_STR  $\neq$  null then
      CUR_Xvalue := CUR_STR $|_X$ 
      if CUR_Xvalue < BEST_Xvalue
      then
        BEST_STR := CUR_STR
        BEST_Xvalue := CUR_Xvalue
      end if
    end if
  end if
  return tree( $t, \{ \text{BEST\_STR} \}$ )

Procedure Solve_u ( $[(\forall, W, C, A)|P'], G$ )
  for all  $\forall a: f(X) \in A$  do
    VAL_a :=  $\emptyset$ 
  end for
  STR :=  $\emptyset$ 
  for all  $t \in D^W$  s.t.  $t$  is a solution of  $C$ 
  do
    CUR_STR := Solve(  $(P', G)[W \leftarrow t]$  )
    if CUR_STR = null then
      return null
    else
      for all  $a: f(X) \in A$  do
        VAL_a := VAL_a  $\sqcup$ 
           $\{ \text{CUR\_STR}|_X \}$ 
      end for
      STR := STR  $\cup$  CUR_STR
    end if
  end for
  return tree( $((f(\text{VAL}_a))_{a: f(X) \in A}, \text{STR})$ )

```

Fig. 4. Search procedure

prototype implementation, the tree is recorded without compression, and this could eventually put limits to the size of the examples that can be handled.

The main search procedure is composed of two mutually recursive evaluation functions, one for an $\exists$ -orqset and one for a $\forall$ -orqset. They return a strategy described by a tree which can either be the empty tree *null* or *tree(a, B)* where a is a tuple and B a set of trees. The general algorithm, the algorithm for a minimization condition in an $\exists$ -orqset and the one for a $\forall$ -orqset are given in Figure 4. For an $\exists$ -orqset, the function maintains the best strategy found so far *BEST_STR* and returns it, or null if the orqset is failed. All strategies are successively explored and compared on their X value. The max and any aggregates are defined similarly. Adding branch and bound to this procedure can be done simply by adding the constraint $X < \text{BEST\_Xvalue}$ (resp. $>$) to the rest of the minimization (resp. maximization) problem. This can be seen as an adaptation of the algorithm of [14] for which the lower/upper bounds are directed by the optimization condition and associated to their own optimization variable instead of to the whole problem. Once a solution has been found, the algorithm for a $\forall$ -orqset first evaluates the sub-strategies for every universal tuple. For each of them it computes the set of aggregates. Then it collects all of them in a set *STR* and returns it at the end.

Branch and Bound. Interestingly, the branch and bound algorithm may be incorrect in the case of *overlapping* optimization conditions. This happens if there

exists two orqsets $orq_i = (\exists, W_i, C_i, \min(X))$ with $X \in W_k$ and $orq_j = (\exists, W_j, C_j, \min(Y))$ with $Y \in W_l$ such that $i < j < k$. Any number of condition *any* may appear in between.

Example 13. A sample problem incorrect for branch and bound is in Figure 5.

Suppose there exists three strategies $s_0 = \{(X_0, Y_0, A_0, B_0)\}$, $s_1 = \{(X_1, Y_1, A_1, B_1)\}$ and $s_2 = \{(X_1, Y_2, A_2, B_2)\}$ such that $A_1 > A_0$, $A_2 < A_0$ and $B_1 < B_2$. Having found the strategy s_0 , the constraint $A < A_0$ is added to the search of subsequent strategies. Thus, s_1 is cut. We find s_2 which has a better value A_2 for A and the optimal strategy is s_2 . Without branch and bound, optimization at the level of Y would have preferred strategy s_1 because of its better value on B and would have returned the value A_1 . The best strategy at the upper level would have been s_0 .

$$\begin{array}{l} \exists X \in D_X \\ \exists Y \in D_Y \\ \exists A \in D_A \\ \exists B \in D_B \\ \dots \\ \text{any} \\ \text{any} \\ \min(B) \\ \min(A) \end{array}$$

Fig. 5. Incorrect B&B

Proposition 14. *Branch and bound is correct if optimization conditions are non-overlapping.*

Proof. With the same notations as above, conditions are non-overlapping if $k \leq j$. Then it is ensured that any branch cut by B&B will be cut before the level of Y will be reached, hence only strategies worse for X will be cut.

5 Examples

In this section, we give several examples of use of quantified constraint optimization, ranging from toy examples to real-world problems taken from the bilevel programming literature. In order to give a readable presentation of the examples, we use a pseudo-code syntax in which the aggregates/optimization part is placed at the end. In a similar way, we allow the use of constants and arrays. For example, the following QCOP⁺ which returns a strategy in which $X = 0$ if the sum of odd indices of the array A is less than the sum of its even indices and $X = 1$ otherwise is depicted on the right as pseudo-code:

$\begin{array}{l} ([(\exists, \{X\}, \emptyset, \min(s)), \\ (\forall, \{i\}, \{i \bmod 2 = X\}, \{s : \text{sum}(Z)\}), \\ (\exists, \{Z\}, \emptyset, \text{any})], \\ \{Z = A[i]\}) \end{array}$	<pre> const A[0..9] \exists X in {0..1} \forall i in {0..9} [i mod 2 = X] \exists Z in {0..+oo} Z = A[i] any s:sum(Z) min(s) </pre>

Minimax in adversary scheduling. Often, the objectives of the existential and universal players conflict completely. This situation can be dealt with by a classical minimax algorithm (and in this case the branch and bound implements alpha-beta pruning). We illustrate this case by an extension of the adversary scheduling example introduced in [4]. In this problem, two opponents are involved: the *scheduler* tries to build a schedule that satisfies all the (temporal and resource) constraints, while the *adversary* tries to prevent the formation of a valid schedule by inflicting some (limited) deterioration on the problem setting. In the QCSP⁺ version, the scheduler was trying to build a schedule such that the ending date was below a given threshold. In QCOP⁺ it is possible to capture the more realistic variant in which the scheduler aims to minimize the ending time of the robust schedule, while the adversary tries to maximize the same ending time. Let us consider an example with three activities a_1, a_2, a_3 and a resource r . An activity a_i has a starting date of s_i , a duration of d_i and requires c_i units of the resource r of maximal capacity 5. The precedence is $a_1 \prec a_2$ and the data are $d_1 = 1, d_2 = 2, d_3 = 3, c_1 = 3, c_2 = 2$ and $c_3 = 1$. The adversary is able to add one unit to the resource consumption of at most two activities. We add a fictitious activity *end* whose purpose is to minimize the length of the schedule. The model in QCOP⁺ is as follows:

```

const d[1..3], c[1..n]
\exists k1 in {0,1}, k2 in {0,1}, k3 in {0,1}
| [k1+k2+k3 =< 2]
| \exists S1 in D1, S2 in D2, S3 in D3, Send in Dend,
| | c'1 in Dc1, c'2 in Dc2, c'3 in Dc3
| | [S1+d1 =< Send, S2+d2 =< Send, S3+d3 =< Send, S1+D1 =< S2,
| | c'1=c1+k1, c'2=c2+k2, c'3=c3+k3]
| | cumulative([S1,S2,S3], [d1,d2,d3], [c'1,c'2,c'3], 5)
| minimize(Send)
maximize(Send)

```

Since there are only existential variables, the strategy is reduced to a single branch which gives the best attack and the corresponding scheduler response.

Network links pricing. Here is an example from the telecom industry, taken from [15]. The problem is to set a tariff on some network links in a way that maximizes the profit of the owner of the links (the leader). The network, as depicted in Figure 6, is composed of N *Customer* customers (the followers) that route their data independently at the smallest possible cost. Customer i wishes to transmit d_i amount of data from source x_i to target y_i . Each path from a source to a target has to cross a tolled arc a_j . On the way from x_i to a_j , the cost of the links (owned by other providers) is c_{ij} . It is assumed that each customer i wishes to minimize the cost to route her data and that he can always choose another provider at a cost u_i . The purpose of the problem is to determine the cost t_j to cross a tolled arc a_j in order to maximize the revenue of the telecom operator. In Figure 6, there are 2 customers and 3 tolled arcs. This problem can be expressed as a QCOP as follows:

```

const NCustomer
const NArc
const c[NCustomer,NArc] // c[i,j] = fixed cost for Ci to reach Aj

```

```

const d[NCustomer]      // d[i] = demand for customer i
const u[NCustomer]      // u[i] = maximal price for customer i
\exists t[1], ..., t[NArc] in [0,max]
| \forall k in [1,NCustomer]
| | \exists a in [1,NArc],
| | | cost in [1,max],
| | | income in [0,max]
| | | cost = (c[k,a]+t[a])*d[k]
| | | income = t[a]*d[k]
| | | cost <= u[k]
| | minimize(cost)
| s:sum(income)
maximize(s)

```

We generated sets of random instances of this problem. These sets differ from each other as a consequence of different values assigned to two parameters: (1) the number of links the network operator owns, and (2) the number of clients who want to use these links. The network operator can choose between five prices for each link. For each instance, the maximum price each customer is willing to pay and the initial costs (to go from home to the starting point of a given link) are randomly chosen. Each set contains 100 instances. These tests were run on machines equipped with two dual-core opteron and 4 GB of RAM. QeCode being mono-threaded, each core was assigned one instance. No timeout has been set. The branch and bound is not activated because the condition is not met.

Figure 7 shows the average and median resolution times of these tests. Instances with a number of links smaller than 7 are not shown, as most of them were solved in less than one second. It is noteworthy that the number of clients has a very small impact on the resolution time, contrary to the number of links. This effect can be explained considering that adding clients simply requires to choose the link in which their data will transit, while adding links adds one possible choice to every client, and multiplies the pricing alternatives of the network operator.

Virtual network pricing. Telecom infrastructure requires very large investments, supported by one or a few *network operators* (NO). To increase competition,

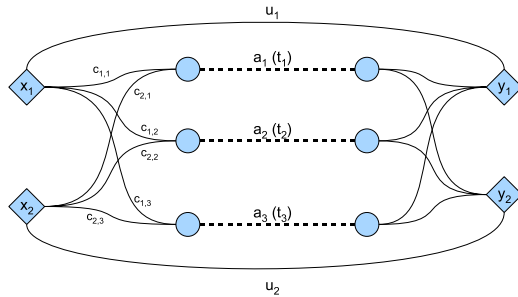


Fig. 6. A network pricing problem

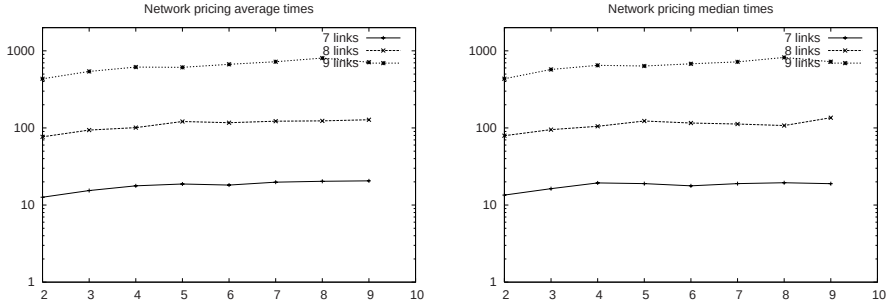


Fig. 7. Average and median resolution times (Y-axis, in seconds) on the *Network Link Pricing* problem for 7, 8 and 9 links, and for between 2 and 9 clients (X-axis)

governments have fostered the introduction of *virtual network operators* (VNO), who provide the same services except that they do not own their network. They rent capacity on the network of NO instead. To some extent, fixed by a regulating authority, network operators are simultaneously competing and cooperating. Taking good decisions in such an environment requires a model of oligopoly which is complex and far from Walras' pure and perfect competition. The following example is taken from [16].

Figure 8 depicts the relations between the NO, the VNO and the customers, each actor being modeled as in [16]. Let us assume the point of view of NO. Our main purpose is to determine the decisions $y = (y_1, y_2)$, y_1 being the service provision to our own customers and y_2 the price for capacity leased to the VNO. The decisions taken by the VNO are $z = (z_1, z_2)$, z_1 being the price for service provision to VNO's customers and z_2 the capacity leased from the NO. The customer are modeled by $n = (n_1, n_2)$ (total number of customers of NO and VNO respectively) according to the prices set for service provision:

$n_i = k_i + r_{i,1}y_1 + r_{i,2}y_2$, the parameters $k_i, r_{i,1}$ and $r_{i,2}$ being determined by analysis of market data. The profit of VNO is given by the revenue of the customers minus the cost of leasing, i.e. $(q - e_2z_1)n_2 - y_2z_2 - g_2$, where q, e_2 and g_2 are respectively the fixed and variable costs by customer and the fixed service provision cost. The profit of the NO is given by the revenue from service provision to the customers and by the capacity allocated to the VNO, i.e. $g_1 + (q + y_1 + e_1)n_1 + y_2z_2$ where e_1 and g_1 are respectively the variable costs by customer and the fixed service provision cost. Note that the revenue of NO depends on decisions taken by VNO. In addition, the price for service to customers is comprised between the limits d and D , the price of leasing has an upper limit of U_1 fixed by the authority, and the maximal

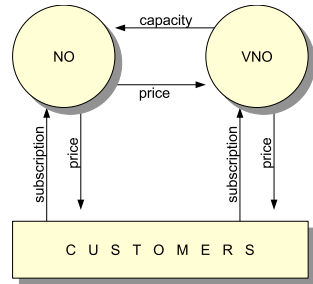


Fig. 8. Virtual network pricing

capacity available for VNO is less than a limit U_2 . We construct the following QCOP⁺ model (in which the universal quantifier is not used since there is only one virtual operator). Note that the non-overlapping condition is not met.

```

const d, D, U1, U2, k1, r11, r12, r21, r22, g1, q
\exists y1 in Dy1, y2 in Dy2
| [d =< y1, y1 =< D, y2 =< U1]
| \exists z1 in Dz1, z2 in Dz2
| | [d =< z1, z1 =< D, z2 =< U2]
| | \exists n1 in Dn1, n2 in Dn2, rno in Drno, rvno in Drvno
| | | n1 = k1 - r11 * y1 + r12 * z1
| | | n2 = k2 + r21 * y1 - r22 * z1
| | | rvno = (q - e2 * z1) * n2 - y2 * z2 - g2
| | | rno = g1 + (q + y1 + e1) * n1 + y2 * z2
| maximize(rvno)
maximize(rno)

```

6 Related Work and Conclusion

Closely related works are the framework of *Plausibility-Feasibility-Utility* (PFU) [17], a work on *iterated expressions* [18], and the language of *Stochastic CSPs* [14]. The PFU framework aims at providing an algebraic unification of QBF, QCSP, Stochastic CSP, Bayesian networks, and Markov Decision Processes, while the purpose of iterated expressions is to model resource allocation in workflows. They both introduce expressions of the form $\bigoplus_{x_1 \in D_1} \dots \bigoplus_{x_n \in D_n} \text{expr}(x_1, \dots, x_n)$ where $\bigoplus \in \{\min, \max, \sum, \Pi\}$. It is not possible to express bilevel models in these frameworks because the optimization condition has to apply on an expression involving the result of an immediate subexpression. However, some constructions like $\min(\sum_x e_1(x) + \sum_y e_2(y))$ are expressible by a PFU or iterated expression and not by a QCOP⁺. Moreover, the branch and bound condition is always verified by construction. In [19] it is proposed to find a boolean QCSP strategy which maximizes the weighted sum of its existential variables set to 1. A dichotomy theorem is also proved (to identify tractable and intractable language classes).

All these works, along with QCOP⁺, pose the problem of finding an adequate language for expressing preferences over strategies. In this respect, the QCOP⁺ framework is general enough to model bi and multi-level problems whose importance is confirmed by several studies of the game theory and operations research.

Several unanswered questions remains: Should the objective function be compositionally defined or not? What is the analog of weighted CSP in this context? When several strategies are optimal at the first level, it can happen that they differ considerably as to sub-strategies. The language of QCOP⁺ does not allow to capture such subtle differences. Enabling constraints on aggregate values is another issue. It is for example impossible to require that strategies should take different values of an existential variable for all values of an universal variable: It would require an “all-different” aggregate that may fail. A last open question concerns the evaluation of partial and/or non-winning strategies, which could open the way to both relaxation and local search in quantified problems.

References

1. Nightingale, P.: Consistency for quantified constraint satisfaction problems. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 792–796. Springer, Heidelberg (2005)
2. Bessière, C., Verger, G.: Strategic constraint satisfaction problems. In: Miguel, I., Prestwich, S. (eds.) Workshop on Constraint Modelling and Reformulation, Nantes, France, pp. 17–29 (2006)
3. Benedetti, M., Lallouet, A., Vautard, J.: QCSP Made Practical by Virtue of Restricted Quantification. In: Veloso, M. (ed.) International Joint Conference on Artificial Intelligence, Hyderabad, India, pp. 38–43. AAAI Press, Menlo Park (2007)
4. Benedetti, M., Lallouet, A., Vautard, J.: Modeling adversary scheduling with QCSP+. In: ACM Symposium on Applied Computing, Fortaleza, Brazil. ACM Press, New York (2008)
5. Nightingale, P.: Consistency and the Quantified Constraint Satisfaction Problem. PhD thesis, University of St Andrews (2007)
6. Bracken, J., McGill, J.: Mathematical programs with optimization problems in the constraints. *Operations Research* 21, 37–44 (1973)
7. Stackelberg, H.: The theory of market economy. Oxford University Press, Oxford (1952)
8. Bialas, W.F.: Multilevel mathematical programming, an introduction. Slides (2002)
9. Colson, B., Marcotte, P., Savard, G.: An overview of bilevel optimization. *Annals of Operations Research* 153, 235–256 (2007)
10. QeCode Team: QeCode: An open QCSP+ solver (2008), <http://www.univ-orleans.fr/lifo/software/qecode/>
11. Gecode Team: Gecode: Generic constraint development environment (2006), <http://www.gecode.org>
12. Benedetti, M.: Extracting certificates from quantified boolean formulas. In: Kaelbling, L.P., Saffioti, A. (eds.) International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, pp. 47–53. Professional Book Center (2005)
13. Bordeaux, L., Cadoli, M., Mancini, T.: CSP properties for quantified constraints: Definitions and complexity. In: Veloso, M.M., Kambhampati, S. (eds.) National Conference on Artificial Intelligence, pp. 360–365. AAAI Press, Menlo Park (2005)
14. Walsh, T.: Stochastic constraint programming. In: ECAI, pp. 111–115 (2002)
15. Bouhtou, M., Grigoriev, A., van Hoesel, S., van der Kraaij, A.F., Spieksma, F.C., Uetz, M.: Pricing bridges to cross a river. *Naval Research Logistics* 54(4), 411–420 (2007)
16. Audestad, J.A., Gaivoronski, A.A., Werner, A.: Extending the stochastic programming framework for the modeling of several decision makers: pricing and competition in the telecommunication sector. *Annals of Operations Research* 142(1), 19–39 (2006)
17. Pralet, C., Verfaillie, G., Schiex, T.: An algebraic graphical model for decision with uncertainties, feasibilities, and utilities. *Journal of Artificial Intelligence Research* 29, 421–489 (2007)
18. Bordeaux, L., Hamadi, Y., Quimper, C.G., Samulowitz, H.: Expressions Itérées en Programmation par Contraintes. In: Fages, F. (ed.) Journées Francophones de Programmation par Contraintes, pp. 98–107 (2007)
19. Chen, H., Pál, M.: Optimization, games, and quantified constraint satisfaction. In: Fiala, J., Koubek, V., Kratochvíl, J. (eds.) MFCS 2004. LNCS, vol. 3153, pp. 239–250. Springer, Heidelberg (2004)

Exploiting Decomposition in Constraint Optimization Problems

Matthew Kitching and Fahiem Bacchus

Department of Computer Science, University of Toronto, Canada
{kitching,fbacchus}@cs.toronto.edu

Abstract. Decomposition is a powerful technique for reducing the size of a backtracking search tree. However, when solving constraint optimization problems (COP's) the standard technique of invoking a separate recursion to solve each independent component can significantly reduce the strength of the bounds that can be applied when using branch and bound techniques. In this paper we present a new search algorithm that can obtain many of the computational benefits of decomposition without having to resort to separate recursions. That is, the algorithm explores a standard OR tree not an AND-OR tree. In this way incremental information gathered from any component can be immediately applied to improve the bounding information for all of the other components. We also discuss how caching and local propagation can be combined with our approach and finally test our methods empirically to verify their potential.

1 Introduction

In this paper we investigate the use of decomposition during search to aid in solving **Constraint Optimization Problems (COPs)**. In particular, we are interested in COPs whose objective function is decomposed into a sum of sub-objectives. This means that when variables are instantiated during search the COP can split into independent components that can be solved separately.

Since backtracking's worst case time complexity is exponential in the number of variables, decomposition into independent components can yield an exponential speedup when applied recursively. More precisely, decomposition can reduce the worst case time complexity from $2^{O(n)}$ to $n^{O(1)}2^{O(w)}$ where n is the number of variables and w is the tree-width of the constraint graph, see, e.g., [1,7].

The downside to using decomposition in COPs is that it can reduce the effectiveness of the bounding techniques that are essential for solving COPs. In particular, the standard method for exploiting decomposition during search is to invoke a separate recursion for each independent component generated during the search, yielding an AND/OR search tree [7,8,9,10,12,14]. Unfortunately, the bounds that can be employed in these separate recursions can be quite weak causing inefficiencies in the search.

In this paper we present an algorithm that is able to exploit decomposition in a standard backtracking search tree (an OR tree). Our algorithm has complete freedom in its variable ordering and is able to switch between working on different components while retaining the space efficiency of depth-first search. In this way incremental information

gathered from any component can be used to improve the bounding information for all of the other components. This can often mean that an entire collection of components can be rejected without ever having to solve any of them to optimality. In addition we discuss how local propagation (soft-arc consistency) [6] can be employed in conjunction with our algorithm, and demonstrate how a fixed tree-decomposition can be flexibly utilized to improve decomposition without having it impose excessive restrictions on the variable ordering.

In the sequel we first present some background on branch and bound and AND/OR search with bounding. Our new algorithm is then presented and some of its properties illustrated. After a discussion of local propagation and our technique for flexibly exploiting a fixed tree-decomposition, we present empirical results demonstrating the potential of our approach.

2 Background

A COP, $\mathcal{C}$, is specified by a tuple $\langle Vars, Dom, Cons, Obj \rangle$, where $Vars$ is a set of variables, for each $V \in Vars$, $Dom[V]$ is a domain for V , $Cons$ is a set of constraints, and Obj is an objective function mapping every complete assignment to $Vars$ to a real value. A *solution* of $\mathcal{C}$ is a complete set of assignments to $Vars$ that *minimizes* Obj and satisfies the constraints in $Cons$.

The techniques we discuss in this paper are effective on COPs whose constraints and objective function are *decomposable*. In particular, we require that Obj be decomposed into a sum of sub-objectives o_i such that: (1) each o_i (and each constraint in $Cons$) depends on only a proper subset of the variables in $Vars$, denoted by $scope(o_i)$; (2) each o_i maps assignments to the variables in $scope(o_i)$ to a real value; and (3) on any complete assignment $\mathcal{A}$ the total objective is the sum of the sub-objectives, $Obj(\mathcal{A}) = \sum_i o_i(\mathcal{A})$.¹

The constraints in $Cons$ can be treated as additional sub-objectives that map satisfying assignments to 0 and violating assignments to ∞ . Thus the problem can be reformulated with a single unified objective $Obj = \sum_i o_i + \sum_{c_j \in Cons} c_j$, and we simply need to minimize this unified objective. Hence, we regard a COP as being defined by a tuple $\langle Vars, Dom, Objs \rangle$ where $Objs$ includes both the original objective sub-functions and the hard constraints. We use the term **objectives** to denote the sub-objectives in $Objs$.

Let $\mathcal{A}$ be any set of assignments to some of the variables of $Vars$: we use $varsOf(\mathcal{A})$ to denote the set of variables assigned by $\mathcal{A}$; $cost(\mathcal{A}, \mathcal{C})$ to denote the sum of the costs of all objectives in the COP $\mathcal{C}$ that are fully instantiated by $\mathcal{A}$; and $mincost(\mathcal{C})$ to denote $cost(\mathcal{A}, \mathcal{C})$ of any solution $\mathcal{A}$ to $\mathcal{C}$ (i.e., the optimal objective value achievable in $\mathcal{C}$).

A set of assignments $\mathcal{A}$ reduces the original COP $\mathcal{C}$ to a smaller COP $\mathcal{C}|_{\mathcal{A}}$ whose variables are the variables of $\mathcal{C}$ not assigned in $\mathcal{A}$ ($\mathcal{C}.Vars - varsOf(\mathcal{A})$), and whose objectives are those that contain at least one unassigned variable and are obtained by restricting the original objectives of $\mathcal{C}$ by $\mathcal{A}$. That is, for any objective $o_i \in Objs$ if $scope(o_i) \not\subseteq varsOf(\mathcal{A})$, then the reduction of o_i by $\mathcal{A}$, $o_i|_{\mathcal{A}}$, is the new objective

¹ If the objective function or constraints are not decomposable (and they can often be reformulated in a decomposed form), our techniques will still correct solve the COP, but no computational advantage will be gained from decomposition.

Algorithm 1. Branch and Bound

```

1  BB( $\mathcal{C}$ ,  $\text{UB}$ )
   /* Return bounds ( $\mathcal{C}.lb, \mathcal{C}.ub$ ) on  $\text{mincost}(\mathcal{C})$ . If  $\text{mincost}(\mathcal{C}) < \text{UB}$ , then return exact bounds
    $\mathcal{C}.lb = \text{mincost}(\mathcal{C}) = \mathcal{C}.ub$ . Else return bounds such that  $\text{UB} \leq \mathcal{C}.lb \leq \text{mincost}(\mathcal{C}) \leq \mathcal{C}.ub$ . */
2  begin
3      ( $\mathcal{C}.lb, \mathcal{C}.ub$ ) =  $\text{getBounds}(\mathcal{C})$ 
4      if ( $\mathcal{C}.lb < \text{UB} \wedge \mathcal{C}.lb \neq \mathcal{C}.ub$ ) then
5          choose (a variable  $V \in \mathcal{C}.Vars$ )
6          foreach  $d \in \text{Dom}[V]$  do
7               $\text{UB} = \min(\text{UB}, \mathcal{C}.ub)$ 
8               $\Delta^d = \text{cost}(V = d, \mathcal{C})$ 
9              ( $lb^d, ub^d$ ) = BB( $\mathcal{C}|_{V=d}, \text{UB} - \Delta^d$ )
10              $\mathcal{C}.ub = \min(\mathcal{C}.ub, ub^d + \Delta^d)$ 
11          $\mathcal{C}.lb = \max(\mathcal{C}.lb, \min_{d \in \text{Dom}[V]} lb^d + \Delta^d)$ 
12     return ( $\mathcal{C}.lb, \mathcal{C}.ub$ )
13 end

```

function with $\text{scope}(o_i|_{\mathcal{A}}) = \text{scope}(o_i) - \text{varsOf}(\mathcal{A})$ and on any set of assignments α to the variables in $\text{scope}(o_i|_{\mathcal{A}})$ we have that $o_i|_{\mathcal{A}}(\alpha) = o_i(\mathcal{A} \cup \alpha)$.

Branch and Bound: (Alg. 1) is a standard technique for solving COPs using backtracking search. It works by building up partial variable assignments in a depth-first manner using bounding to prune the search space. Each recursion is passed a COP $\mathcal{C}$ (a reduction of the original COP by the current set of assignments) and an upper bound UB . It tries to compute $\text{mincost}(\mathcal{C})$, subject to the condition that it can abort its computation as soon as it can conclude that $\text{mincost}(\mathcal{C}) \geq \text{UB}$.

The computation starts with obtaining valid bounds on $\text{mincost}(\mathcal{C})$ (various approximations can be used). If it is possible that $\text{mincost}(\mathcal{C}) < \text{UB}$ (i.e., $\mathcal{C}.lb < \text{UB}$) and $\text{mincost}(\mathcal{C})$ is not already known (i.e., $\mathcal{C}.lb \neq \mathcal{C}.ub$), then the computation of $\text{mincost}(\mathcal{C})$ can proceed. For any variable $V \in \mathcal{C}.Vars$ we know that $\text{mincost}(\mathcal{C})$ must be achieved by assigning V one of its values, and we can try each of these values in turn. The minimal cost for $\mathcal{C}$ under the assignment $V = d$ is the sum of Δ^d , the cost contributed by any objectives of $\mathcal{C}$ that are fully instantiated by $V = d$ (line 8), and $\text{mincost}(\mathcal{C}|_{V=d})$. Hence, to achieve a total cost for $\mathcal{C}$ of less than UB under $V = d$, we must achieve a cost less than $\text{UB} - \Delta^d$ for $\mathcal{C}|_{V=d}$ (line 9). After the recursive call we know that $\text{mincost}(\text{prob})$ can be no greater than the returned ub^d plus Δ^d , so we can reset $\mathcal{C}.ub$ to this value if it provides a tighter bound. We can also update the desired bound for the next value to be the minimum of what was already required, UB , and the current upper bound for $\mathcal{C}$, $\mathcal{C}.ub$ (line 7). That is, we force the rest of the search to achieve an even better value for $\mathcal{C}$. After trying all values, we know that both the initially estimated lower bound, $\mathcal{C}.lb$, and the minimum of the lower bounds, $lb^d + \Delta^d$, achieved under the different values of V are valid lower bounds for $\mathcal{C}$. Hence, we can use the tightest (maximum) of these as a new lower bound $\mathcal{C}.lb$. Note that when all variables have been assigned the recursion must stop. In particular, the passed $\mathcal{C}$ will be the empty COP and will have exact bounds $\mathcal{C}.lb = 0 = \mathcal{C}.ub$.

Branch and Bound with Decomposition: (Alg. 2) A more recent technique used in solving COPs (and other types of constraint problems) is *search with decomposition* (or

Algorithm 2. AND-OR Decomposition with Branch And Bound

```

1 AND-OR-Decomp ( $\kappa, \text{UB}$ )
  /* On entry ( $\kappa.lb, \kappa.ub$ ) must be valid bounds on  $\text{mincost}(\kappa)$ . If  $\text{mincost}(\kappa) < \text{UB}$ , then compute
  exact bounds  $\kappa.lb = \text{mincost}(\kappa) = \kappa.ub$ . Else compute bounds such that
   $\text{UB} \leq \kappa.lb \leq \text{mincost}(\kappa) \leq \kappa.ub$ . */
2 begin
3   if ( $\kappa.lb < \text{UB} \wedge \kappa.lb \neq \kappa.ub$ ) then
4     choose (a variable  $V \in \kappa.Vars$ )
5     foreach  $d \in \text{Dom}[V]$  do
6        $\text{UB} = \min(\text{UB}, \kappa.ub)$ 
7        $\Delta^d = \text{cost}(V = d, \kappa)$ 
8       // Start new Processing for decomposition.
9        $\mathcal{K}^d = \text{toComponents}(\kappa|_{V=d})$ 
10      foreach  $\kappa^d \in \mathcal{K}^d$  do
11         $(\kappa^d.lb, \kappa^d.ub) = \text{getBounds}(\kappa^d)$ 
12        foreach  $\kappa^d \in \mathcal{K}^d$  while  $(\sum_{\kappa^d \in \mathcal{K}^d} \kappa^d.lb < \text{UB} - \Delta^d)$  do
13           $\text{UB}_{\kappa^d} = \text{UB} - \Delta^d - \sum_{\kappa' \in \mathcal{K} \wedge \kappa' \neq \kappa^d} \kappa'.lb$ 
14          AND-OR-Decomp ( $\kappa^d, \text{UB}_{\kappa^d}$ )
15           $(lb^d, ub^d) = \sum_{\kappa^d \in \mathcal{K}} (\kappa^d.lb, \kappa^d.ub)$ 
16          // End new Processing for decomposition.
17           $\kappa.ub = \min(\kappa.ub, ub^d + \Delta^d)$ 
18         $\kappa.lb = \max(\kappa.lb, \min_{d \in \text{Dom}[V]} lb^d + \Delta^d)$ 
19 end

```

AND/OR search) [7,8,10,14]. As variable assignments are made during backtracking search, the COP can become separated into smaller independent COPs called **components**. These components are COPs that share no variables and hence they can be solved independently of each other. For example, if $\mathcal{C}$ has the objectives $o_1(V_1, V_2, V_3)$ and $o_3(V_3, V_4, V_5)$ then the assignment $V_3 = d$ will split $\mathcal{C}$ into two components, the first containing the variables V_1 and V_2 , and the objective $o_1|_{V_3=d}$ while the second contains the variables V_4 and V_5 , and the objective $o_2|_{V_3=d}$. Setting the variables of one component has no effect on the other. AND/OR search works by invoking a separate recursion for each component κ generated during search.

In Alg. 2 components are represented by a data structure κ that is created and destroyed as Alg. 2 performs its search. κ is defined by some subset of the objectives of the original input problem, $\kappa.Objs$, that have been reduced by some set of assignments $\kappa.\mathcal{A}$ sufficient to disconnect these objectives from the rest of the problem. The variables of κ , $\kappa.Var$, are all of the unassigned variables of these objectives (thus $\text{varsOf}(\kappa.\mathcal{A}) \cup \kappa.Vars = \bigcup_{o \in \kappa.Objs} \text{scope}(o)$). Also note that κ is a COP so we can evaluate $\text{cost}(\mathcal{A}', \kappa)$ for any set of assignments $\mathcal{A}'$. κ also contains fields $\kappa.lb$ and $\kappa.ub$ used to store bounds on $\text{mincost}(\kappa)$.

The search starts with the call $\kappa = \mathcal{C}$ and with $\kappa.lb \leq \text{mincost}(\kappa) \leq \kappa.ub$, i.e., a request to solve the original problem with valid bounds on the optimal cost. Each recursion solves a single component κ created by the current set of assignments. To solve the component κ we try all values of one of its variables V , reducing κ by each possible assignment. The reduced component, $\kappa|_{V=d}$, is first separated into a set of sub-components $\mathcal{K}^d$ and a data-structure κ_d is created for each of these sub-components

(line 8). Each sub-component is solved independently (line 13). We know that in order to achieve a total cost of less than UB for κ under the assignment $V = d$, the sum of the lower bounds over all components in $\mathcal{K}$ must be less than $UB - \Delta^d$ (where Δ^d is the immediate cost of making the assignment $V = d$). Thus each sub-component $\kappa^d \in \mathcal{K}$ must achieve a value of no greater than $UB - \Delta^d$ minus the sum of the lower bounds of all of the *other* sub-components in $\mathcal{K}$ (line 12). Since each recursive call updates the lower bound of a sub-component in $\mathcal{K}$, we can abort the solving of these sub-components whenever the sum of their lower bounds exceeds $UB - \Delta^d$ (line 11).

Once we have finished with the value $V = d$ all of the data structures in $\mathcal{K}^d$ can be deleted—so that the space requirements of the algorithm remain polynomial. On the other hand, during search with decomposition the same component can be encountered many times. Thus it is natural to cache the computed bounds for these components so that when they are encountered again we can use these better bounds to optimize the next attempt at solving the component. Cache look up can occur inside of the function *getBounds* (line 10) where the better bounds (perhaps exact bounds) stored in the cache can be retrieved.

3 Decomposition without Separate Recursions

Although the above use of decomposition with bounding gains computational advantage from breaking the problem into independent sub-problems, it suffers from a weakening of the bounding information it can utilize.

Consider solving a component κ under $UB = 100$. Say that we branch on variable V making the assignment $V = d$, and that this adds zero to the cost ($\Delta^d = 0$) while breaking κ into five components $\kappa_1, \dots, \kappa_5$. If $mincost(\kappa_i) = 25$ this value for V must eventually be rejected as under $V = d$, κ can only achieve a minimal cost of 125. Say further that for each of the κ_i , our estimated lower bounds, $\kappa_i.lb$, is 10. We can see that the upper bound applied when solving κ_1 will be $100 - 4 * 10 = 60$, and the search will be forced to solve κ_1 to optimality. This will update $\kappa_1.lb$ to 25, and yield an upper bound of $100 - (3 * 10 + 25) = 45$ for solving κ_2 . Thus the search will also be forced to solve κ_2 to optimality. The bound for κ_3 will then be 30, and κ_3 must be solved to optimality. The bound for κ_4 will be 15 and now the search can terminate before solving κ_4 , after which $\sum_i \kappa_i.lb > 100$ and we can reject $V = d$. Computing the optimal value for κ_1 – κ_3 can be very expensive, and it could be that some much shallower search of all of the components could have served to move their lower bounds to 20 or higher so that $V = d$ could be rejected without having to solve any of them to optimality. Thus we see that although we are solving simpler problems, the bounds we can exploit in these problems are weaker.

The key contribution of this paper is to demonstrate how the computational benefits of decomposition can be obtained without having to perform separate recursions for each component. Instead our method exploits the ideas originally presented in [1] for counting problems where the benefits of decomposition are obtained in a regular backtracking search tree (OR-tree). We extend the ideas of [1] in a non-trivial way so that bounding can be exploited.

Algorithm 3. Decomposition and Bounding in a Standard Backtracking Tree

```

1 OR-Decomp ( $\mathcal{K}$ ,  $UB$ )
   /* On entry each  $\kappa \in \mathcal{K}$  must have valid bounds  $(\kappa.lb, \kappa.ub)$ . If  $\sum_{\kappa \in \mathcal{K}} \text{mincost}(\kappa) < UB$ 
   then compute exact bounds for every  $\kappa \in \mathcal{K}$ ,  $\kappa.lb = \text{mincost}(\kappa) = \kappa.ub$ . Else compute
   valid bounds on the components in  $\mathcal{K}$  such that  $UB \leq \sum_{\kappa \in \mathcal{K}} \kappa.lb$ . */
2 begin
3   if ( $\sum_{\kappa \in \mathcal{K}} \kappa.lb < UB \wedge \sum_{\kappa \in \mathcal{K}} \kappa.lb \neq \sum_{\kappa \in \mathcal{K}} \kappa.ub$ ) then
4     choose (a component  $\tau \in \mathcal{K}$  with  $\tau.lb \neq \tau.ub$  and a variable  $V \in \tau.Vars$ )
5     AddConstraint( $\tau$ ,  $\tau.ub$ )
6     foreach  $d \in \text{Dom}[V]$  while  $\sum_{\kappa \in \mathcal{K}} \kappa.lb < UB$  do
7        $UB = \min(UB, \sum_{\kappa \in \mathcal{K}} \kappa.ub)$ 
8        $\Delta^d = \text{cost}(V = d, \tau)$ 
9        $\mathcal{K}^d = \text{toComponents}(\tau|_{V=d})$ 
10      foreach  $\kappa^d \in \mathcal{K}^d$  do
11         $(\kappa^d.lb, \kappa^d.ub) = \text{getBounds}(\kappa^d)$ 
12       $\mathcal{K}' = (\mathcal{K} - \tau) \cup \mathcal{K}^d$ 
13      OR-Decomp ( $\mathcal{K}'$ ,  $UB - \Delta^d$ )
14       $(lb^d, ub^d) = \sum_{\kappa^d \in \mathcal{K}^d} (\kappa^d.lb, \kappa^d.ub)$ 
15       $\tau.ub = \min(\tau.ub, ub^d + \Delta^d)$ 
16      RemoveConstraint( $\tau$ ,  $\tau.ub$ )
17       $\tau.lb = \max(\tau.lb, \min_{d \in \text{Dom}[V]} lb^d + \Delta^d)$ 
18 end

```

Our new algorithm is shown in Alg. 3. Like Alg. 1 the algorithm takes as input the entire remaining problem. However, instead of regarding the input as being a single reduced COP ($\mathcal{C}$), the input has been broken up into a set of components $\mathcal{K}$. The aim is to solve all of the components in $\mathcal{K}$ (compute bounds such that $\forall \kappa \in \mathcal{K} : \kappa.lb = \kappa.ub$ or equivalently $\sum_{\kappa \in \mathcal{K}} \kappa.lb = \sum_{\kappa \in \mathcal{K}} \kappa.ub$), subject to the condition that we can give up on the computation once we have concluded that the combined cost of these components is greater than or equal to the passed upper bound UB (i.e., when $\sum_{\kappa \in \mathcal{K}} \kappa.lb \geq UB$).

If neither of these conditions have been met, some unassigned variable is chosen, and all of its values are tried. On each instantiation the component, τ , containing V might be split up into a new collection of components. These replace τ in the recursive call (line 12), and as in Alg. 1 the upper bound is updated to account for the cost of making the assignment $V = d$. Note that unlike Alg. 2 we pass all of the remaining components to the recursive call (line 13)—we do not recurse just on a single component. Thus the sub-tree search below this invocation can choose to branch on variables from any component in any order—it is not constrained to branch only on the remaining variables of a single passed component as in Alg. 2.

On return from the search below the newly generated components in $\mathcal{K}^d$ will potentially have had their bounds updated, and we can update the upper bound of τ (line 15). After trying all of the values for V we can update the lower bound of τ (line 17).

Note that in the search below the other components in $\mathcal{K}$ can be branched on, and can have their bounds updated. Thus we may obtain sufficient information to abort the for loop before trying all of the values of V . This motivates the **while** test during the for

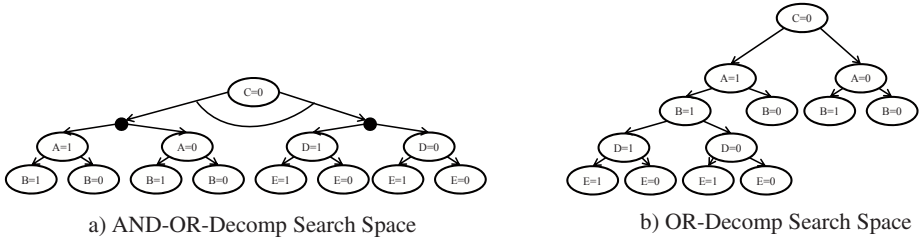


Fig. 1. Search Space of AND/OR decomposition and OR decomposition

loop (line 6). Note also that if any of these components are solved, i.e., if exact bounds on their value are computed, we will never branch on them again: the branch variable must be from an unsolved component (line 4). That is, if under V 's first value we solve some component $\kappa' \in \mathcal{K}$, then in the sub-trees generated by all of V 's other values we will never branch on any of the variables of κ' again. This is where decomposition is exploited. If all of the components of $\mathcal{K}$ beside τ are solved (which will occur if $UB - \Delta^d > \sum_{\kappa \in \mathcal{K} \wedge \kappa \neq \tau} \text{mincost}(\kappa)$) then Alg. 3 will obtain all of the computational benefits of decomposition.

Example 1. Consider a COP $\mathcal{C}$ with two objectives $o_1(A, B, C) = A + B + C$ and $o_2(C, D, E) = C + D + E$ and where all of the variables have domain $\{0, 1\}$. Thus $\text{mincost}(\mathcal{C}) = 0$ is obtained when all variables have been set to zero. Also suppose that *getBounds* always returns the minimum and maximum values for the remaining reduced sub-COP. Say that we first branch on $C = 0$ which splits the problem into two components $\kappa_1 = \{o_1(A, B, C = 0)\}$ and $\kappa_2 = \{o_2(C = 0, D, E)\}$.

If Alg. 1 is used always trying the value 1 before 0, it can be demonstrated that the search below $C = 0$ will attempt 20 variable assignments. In contrast the search by Alg. 2, shown in Fig 1.a is smaller attempting only 12 assignments. It is able to detect that the problem consists of two independent components and solve them independently. Alg. 3, shown in Fig 1.b also searches only 12 nodes. It also exploits decomposition but in a different search tree. In particular, under the left most instantiation of the variables A and B , κ_2 is solved exactly. Hence, the search need never branch on D and E again until it tries a different value for C .

Example 2. However, sometimes bounding in Alg. 3 can interfere with independently solving the components in $\mathcal{K}$. Say that $\mathcal{K}$ contains two components κ_1 and κ_2 , where κ_1 contains only a single unassigned variable V that has values $\{a, b, c\}$. If Alg. 3 first branches on V , then after each value for V it will attempt to solve $\mathcal{K}' = \{\kappa_2\}$ ($\mathcal{K}^d$ will be a empty set of components since V is κ_1 's final value). Dependent on Δ^d we will be trying to solve κ_2 under different, perhaps too stringent bounds. For example, if $\text{mincost}(\kappa_2) = 10$, $UB = 15$, and $\Delta^a = 7$, $\Delta^b = 6$, and $\Delta^c = 3$, the attempts to solve κ_2 under both $V = a$ and $V = b$ will fail (although we will increase $\kappa_2.lb$). Only when $V = c$ will we try to solve κ_2 under a bound that is greater than $\text{mincost}(\kappa_2)$. As the number of variables in κ_1 increases, these repeated attempts to solve κ_2 can multiply. Some savings can, however, occur since each solution attempt can tighten the bounds on κ_2 . Nevertheless, a multiplicative effect can occur destroying independence.

Thus on the positive side Alg. 3 can interleave the solving of the current components by branching on variables from different components at each recursion. This can produce refined bounding information that can be sufficient to refute a whole collection of components without ever having to solve any component to optimality. This can be accomplished while still obtaining many of the benefits of decomposition. On the negative side however, bounding can sometimes interfere with the benefits of decomposition, as illustrated in the example above.

Fortunately, there are a couple of simple ideas that can remove the worst of the negative effects of bounding on decomposition. The first idea is to force a component to be solved once and for all, if the search continues to return to it. The second idea is to force the complete solution of a component if that component is sufficient small. In the example above, the first method would solve κ_2 (i.e., find $\text{mincost}(\kappa_2)$) after having returned to it some number of times; while the second method would solve κ_1 (finding that $V = c$ is the correct assignment to make) before advancing to κ_2 since κ_1 is small. In our implementation we did not find an effective way of utilizing the first idea: any fixed count of how often the search can return to a component before forcing it to be solved sometimes degraded performance. The second idea of forcing the solution of a component when it is small was effective, and in our implementation we forced the solution of any component whose variables had a product domain size of 20 or less.

Local Bounding: There is one further aspect of Alg. 3: the two lines **AddConstraint** and **RemoveConstraint** that bracket the for loop over V 's values. The intuition for these lines is that the current component τ has an upper bound $\tau.\text{ub}$ that is initialized when τ was first added to $\mathcal{K}$ and is updated after each value for V has been attempted (line 15). Thus in the search below it is never effective to instantiate the variables of $\tau.\text{Vars}$ to values that will cause τ to achieve a value greater than $\tau.\text{ub}$. Note that this can happen even though the global bound of $\sum_{\kappa \in \mathcal{K}} \text{lb} < \text{UB}$ remains valid. We have found that the easiest way to enforce this local bound on the settings of $\tau.\text{Vars}$ is to post a constraint on the search below. Note that the strength of this constraint increases as we obtain tighter bounds on $\tau.\text{ub}$.

For example, say that τ contains the objectives $o_1(A, B, C)$, $o_2(A, E, F)$, and $o_3(F, G)$ and that $\tau.\text{ub} = 10$. If we branch on $A = a$ with $\Delta^a = 3$ then sometime later on $F = f$ with $\Delta^f = 3$, we will have broken τ into three sub-components: $\kappa_1 = \{o_1(A = a, B, C)\}$, $\kappa_2 = \{o_2(A = a, E, F = f)\}$, and $\kappa_3 = \{o_3(F = f, G)\}$, and accumulated an immediate cost of $\Delta^a + \Delta^f = 6$. This means that if $\kappa_1.\text{lb} + \kappa_2.\text{lb} + \kappa_3.\text{lb} > (10 - 6)$ we can immediately backtrack to the deepest point a variable of τ has been instantiated (in this case to undo the assignment $F = f$). That is, under the assignments $A = a$ and $F = f$, τ cannot achieve its optimal value—it is already exceeding a known upper bound on its optimal value. More formally, we require that in the subtree below if S is the set of components that have been generated from τ , and A is the set of assignments that have been made to variables of τ , then $\sum_{\kappa \in S} \kappa.\text{lb} + \text{cost}(A, \tau) \leq \tau.\text{ub}$.

There are other ways of implementing this local bound condition, but utilizing a hard constraint is a simple method. There are also potentially other ways the local bounds could be used, including, e.g., using more sophisticated propagation of the added constraint. In our current implementation we are only checking this constraint and backtracking when it is violated.

Formal Results: It can be proven that Alg. 3 is correct.

Theorem 1. *If on entry to Alg. 3 $\forall \kappa \in \mathcal{K}. (\kappa.lb \leq \text{mincost}(\kappa) \leq \kappa.ub)$, then if $UB > \sum_{\kappa \in \mathcal{K}} \text{mincost}(\kappa)$ on return $\forall \kappa \in \mathcal{K}. (\kappa.lb = \text{mincost}(\kappa) = \kappa.ub)$. On the other hand under the same entry conditions, if $UB \leq \sum_{\kappa \in \mathcal{K}} \text{mincost}(\kappa)$, then on return $UB \leq \sum_{\kappa \in \mathcal{K}} lb(\kappa)$.*

This theorem can be proved by induction on the total number of variables in the set of components in $\mathcal{K}$. The base case is when there are no variables in $\mathcal{K}$, i.e., $\mathcal{K}$ is empty. The inductive case is straight-forward. This theorem means that Alg.3 correctly solves the initial input COP, $\mathcal{C}$, as long as the initial bounds on $\text{mincost}(\mathcal{C})$ are valid—any valid bounds will work, but tighter bounds can yield smaller search trees.

The space requirements of Alg. 3 are also worth looking at. It can be noted that after the value $V = d$ has been tried, all newly generated components ($\mathcal{K}^d$) can be discarded. If the input COP $\mathcal{C}$ has n variables, then there can be at most n components in $\mathcal{K}$ (each component must contain at least one variable), and we can descend a path of at most length n . Thus at most $O(n^2)$ space is ever needed to store the active components during the algorithm's operation, above and beyond the space initially needed to represent the input problem $\mathcal{C}$.

The algorithm's performance can be considerably enhanced by remembering previously encountered components in a cache. Thus after new bounds on τ have been computed, at line 17, these bounds (perhaps exact) can be stored in the cache and reused whenever τ is encountered again in the search. Caching is an important part of our implementation and we have utilized the template techniques described in [10] to make its use more efficient. Note, that the cache serves only to improve the algorithm's performance, it is not required for the algorithm's correctness.

4 Local Propagation

An important technique when solving COPs is local propagation or soft-arc consistency, developed in a number of previous works, e.g., [6,8,13]. This technique works by “sweeping” values from the sub-objectives to a zero-arity sub-objective. The value of the zero-arity objective can then be used as a lower-bound on the COP's value, and to prune the variable domains.

The technique works by adding to the original COP unary sub-objectives $o_i(V_i)$, one for each variable, and a zero-arity objective $0()$ (none of these added objectives affect decomposition). Sweeping (enforcing soft-arc consistency) moves value into the zero-arity objective.

Two sweeping transformations are employed. First, values can be swept between a unary objective $o_1(V)$ and any binary objective involving V , e.g., $o_2(V, X)$. In particular, if for $a \in \text{Dom}[V]$ we have that $\min_{b \in \text{Dom}[X]} o_2(a, b) = \alpha > 0$, then we can sweep α from o_2 into o_1 : for all $b \in \text{Dom}[X]$ we reset $o_2(a, b) = o_2(a, b) - \alpha$, and $o_1(a) = o_1(a) + \alpha$. Intuitively, if o_2 yields a value of at least α when $V = a$, then we can move α into the unary objective over V adding it to the unary cost of $V = a$. Similarly, we can sweep a value from o_1 into o_2 . If $o_1(a) = \alpha > 0$: we reset $o_2(a, b) = o_2(a, b) + \alpha$ for all $b \in \text{Dom}[X]$, and $o_1(a) = 0$. Second,

values can be swept from a unary objective $o_1(V)$ into the zero-ary objective $0()$: if $\min_{a \in \text{Dom}[V]} o_1(a) = \alpha > 0$ we can reset $0() = 0() + \alpha$ and $o_1(a) = o_1(a) - \alpha$ for all $a \in \text{Dom}[V]$. These two types of transformations are equivalence preserving in the sense that the updated COP has an unchanged minimum cost.

Local propagation can be added to the three previously specified algorithms as follows. For Alg. 1 we set Δ^d on line 8 so that it is equal to all of the costs that have been swept to $0()$ as a result of applying local propagation after the assignment $V = d$, and we invoke the algorithm recursively (line 9) on $C|_{V=d}$ after local propagation has been applied to the reduced problem (thus the bounds computed at line 3 are with respect to a problem that has already been modified by local propagation). Similarly to add local propagation to Alg. 2, we set Δ^d on line 7 to be the total value swept to $0()$ from variables of the component κ , and break $\kappa|_{V=d}$ into components (line 8) after local propagation has been performed.

Finally, to add local propagation to our new algorithm (Alg. 3), we again set Δ^d (line 8) to be the total value swept to $0()$ as a result of applying local propagation after the assignment $V = d$, and break $\kappa|_{V=d}$ into components (line 9) after local propagation has been performed. In addition, to accommodate local bounding we enforce the constraints added at line 5 by ensuring that in the search below the sum of the lower bounds of all of the components generated from τ plus the total value swept to $0()$ from variables of τ always remains $\leq \tau.ub$.

Caching: Local propagation can also interfere with caching. With caching, we store the bounds computed on components that arise during search and reuse these bounds if the component reappears in the search. However, the next time the component appears local propagation might have moved a different amount of value into or out of the component as compared to the previous time the component was encountered. This can invalidate the cached bounds.

To exploit caching in the presence of local propagation we must make the bounds independent of the current propagation before we store them in the cache, and adjust these bounds to account for the current propagation when we retrieve them from the cache. In [8] a technique was developed for accomplishing this when a fixed tree-decomposition is used to guide the search. With a fixed decomposition the components that will arise during search can be predicted ahead of time. In Alg. 3, however, the order in which the variables are instantiated is unconstrained—i.e., a fixed tree-decomposition is not used. Rather, components are detected dynamically whenever they are created by the instantiated variables. Nevertheless, we were able to generalize the techniques of [8] so that we can compute the value that has flowed into and out of the components as they are generated during search. Using these flows the cached bounds can be adjusted so that they are made independent of the context when they are to be stored in the cache, and made compatible with the current context when retrieved from the cache.

5 Tree Decompositions

A commonly used technique for exploiting decomposition during search is to compute a tree-decomposition $\mathcal{T}$ for the constraint graph prior to search. $\mathcal{T}$ is a tree where each

node is labeled by a set of variables of the COP (these labels satisfy certain conditions, see, e.g., [1]). We then force the variable ordering of the backtracking search to follow $\mathcal{T}$ by requiring that it always branch on an unassigned variable from an active node of $\mathcal{T}$. Initially only the root of $\mathcal{T}$ is active, and once all variables in an active node have been assigned all of its children become active nodes. By then forcing the variable ordering to follow the tree-decomposition the components that will appear during search can be determined before the search commences. This reduces the overhead of detecting and caching components. The other advantage of computing a tree-decomposition prior to search is that more expensive algorithms can be used that can better analyze how to effectively decompose the COP.

We have specified our algorithms as using arbitrary variable orderings. When these orderings are not following a fixed tree-decomposition detecting and caching components during search is more expensive. However, such fully dynamic variable orderings can yield small search trees. In [1] it was proved that for some problem instances fully dynamic variables orderings can yield a super-polynomial speedup over variable orderings forced to follow *any* fixed tree-decomposition. Empirical evidence has also been given that despite the higher overheads, search with decomposition can perform better with dynamic variable orderings in COPs [15].

With the additional flexibility for variable ordering provided by our algorithm we have found that a hybrid approach can be very effective. In this hybrid we compute a tree-decomposition and try to follow it. However, we allow the algorithm to deviate from the ordering dictated by the tree-decomposition if an alternative variable looks particularly promising. In particular, using a heuristic to score the variables, we impose an additional penalty on any variable that would violate the ordering imposed by the tree-decomposition. However, if that variable's heuristic score (measuring the merit of branching on it next) is high enough it can overcome the penalty and cause the search to make a different decision than that dictated by the tree-decomposition.

6 Experimental Results

We implemented the three algorithm described above. We have additionally added local propagation to these algorithms, and for Alg. 2 and Alg. 3 we include caching of previously solved components. We have tested these algorithms on both weighted-CSP (wCSP) problems and Most Probable Explanation (MPE) problems from Bayesian networks.

The following specific versions of these algorithms were tested: (1) **BB** which is Alg. 1 with FDAC local propagation implemented in the state-of-the-art solver Toolbar [3]; (2) **AND/OR** which is Alg. 2 with FDAC local propagation using a variable ordering that follows a fixed tree-decomposition; (3) **OR-Decomp+T** which is our Alg. 3 with FDAC local propagation using a variable ordering that follows a fixed tree-decomposition; (4) **OR-Decomp+D** which is Alg. 3 with FDAC local propagation using a heuristically guided dynamic variable ordering; and (5) **OR-Decomp+G** which is Alg. 3 with FDAC local propagation using a hybrid variable ordering that follows a fixed tree-decomposition but can opportunistically branch on other variables if they have high enough heuristic score. It should be noted that although OR-Decomp+T

follows a fixed tree-decomposition it still has more flexibility in its variable ordering than AND/OR following a fixed tree-decomposition. This added flexibility arises from the fact that OR-Decomp+T can branch on a variable from any active node of the tree decomposition, AND/OR search on the other hand must commit to a particular node n of the tree-decomposition and branch on all variables in the subtree below n before being able to branch on any variable in the labels of n 's siblings.

The heuristic score used for in the variable ordering decisions² is the an adaptation of the Jerslow heuristic that has previously been used for solving COPs [3]. Previous work has found that this heuristic tends to be more effective than simpler heuristics based on domain size or variable degree. Intuitively, in COPs this heuristic considers both the variable's domain size and the average cost of the objective functions the variable appears in.

All algorithms utilized a value ordering determined by the unary objectives used during local propagation. That is, the values for variable V_i were ordered by lowest unary cost $o_i(V_i)$. For those algorithms that utilized a tree decomposition, these decompositions were computed using a min-fill algorithm [11]. The AND/OR search ordered its components so as to solve the largest component first. All experiments were run with 600 second timeouts, and were conducted on 2.66GHz machines with 8GB of memory. In our experiments we found that the space used in caching never exceeded available memory, so we did not have to prune the cache during search. The following four benchmarks were tested.

The **Radio Link Frequency Assignment Problem** (RLFAP) assigns frequencies to a set of radio links in such a way that all the links may operate together without noticeable interference. The RLFAP instances were cast as binary wCSP's [5]. The benchmark family includes 6 problems.

The **Earth Observing Satellites** (SPOT5) problems select from a set of candidate photographs a subset such that some imperative constraints are satisfied and the total importance of the selected photographs is maximized. The problems have been formulated as wCSP's with binary and ternary constraints in the SPOT5 benchmark [2]. The benchmark family consists of 20 problems.

The **GridNetworks** (Grid) problems involve computing the setting of the variables in a Bayes Net that have maximum probability (an MPE problem). The net is a $N \times N$ grid with CPT's that have been filled with values that were either randomly (uniformly) chosen from the interval (0,1) or were randomly assigned 0 or 1. The problem instances have N ranging between 10 and 38, with 90% of the CPTs entries were 0 or 1 [17]. The benchmark family consists of 13 problems.

ISCAS-89 circuits are a benchmark used in formal verification and diagnosis. The problem set has been converted into n-ary wCSPs [4]. The benchmark family consists of 10 problems.

Table 1 summarizes the number of problems solved by the various algorithms form the various benchmarks. Of the 49 total problem instances **BB** solved 22 problems across the four benchmarks; **AND/OR** solved 22 problems; **OR-BBDecomp+T** solved

² Even in those algorithms where the variable ordering must follow a fixed tree-decomposition there are typically a range of variables in the active nodes that can be branched on; the choice of which of these variable to branch on next is determined by the heuristic score.

Table 1. Number of Problems Solved (600 second timeout)

Benchmark	RLFAP (6)	Spot5 (20)	Grids (13)	Iscas89 (10)	Total (49)
BB	5	4	4	9	22
AND/OR	5	5	3	9	22
OR-Decomp+T	5	5	3	10	23
OR-Decomp+D	5	4	4	9	22
OR-Decomp+G	5	5	4	10	24

Table 2. RLFAP Instances - Time in Seconds (600 second timeout)

Instance	BB	AND/OR	OR-Decomp+T	OR-Decomp+D	OR-Decomp+G
CELAR6-SUB0	0.16	0.03	0.21	0.33	0.21
CELAR6-SUB1-24	2.64	13.2	8.27	2.95	3.35
CELAR6-SUB1	41.3	94.96	57.14	43.42	55.16
CELAR6-SUB2	15.42	101.21	135.19	16.15	16.2
CELAR6-SUB3	239.65	446.28	430.98	222.67	228.61

Table 3. Spot5 Instances - Time in Seconds (600 second timeout)

Instance	BB	AND/OR	OR-Decomp+T	OR-Decomp+D	OR-Decomp+G
1502	0.05	0.02	0.06	0.04	0.06
29	1.66	0.01	0.04	0.03	0.03
404	89.18	0.7	1.2	1.49	1.83
54	0.18	0.01	0.02	0.04	0.03
503	Timeout	0.23	1.19	Timeout	40.19

23 problems, showing an improvement by dtrack over standard AND/OR search; and **OR-BBDecomp+G** solved the most problems at 24.

The RLFAP instances have small tree-width that can be calculated quickly. Hence decomposition techniques offer a much lower theoretical time bounds than standard branch and bound search. However, we found that many of the instances benefit more from added flexibility in variable ordering than from decomposition.

In particular, as shown in Table 2 **BB**, **OR-Decomp+D**, and **OR-Decomp+G** all perform well on these benchmarks, since they allow for the largest variable ordering freedom. **AND/OR** and **OR-Decomp+T** both perform poorly on the benchmarks, although **OR-Decomp+T** took less total time to solve all instances since it allows more freedom of variable selection than **AND/OR**.

The Spot5 instances shown in Table 3 are solved efficiently by algorithms exploiting decomposition. For example, **BB** and **OR-Decomp+D** could not solve one of the instances that was solved by all other approaches. Although **OR-Decomp+D** can decompose problems, the problem does not decompose quickly unless a tree decomposition is used to guide search toward decompositions. **AND/OR**, **OR-Decomp+T**, and **OR-Decomp+G** all used decomposition effectively on the Spot5 instances.

Table 4. Grid Instances - Time in Seconds (600 second timeout)

<i>Instance</i>	BB	AND/OR	OR-Decomp+T	OR-Decomp+D	OR-Decomp+G
90-10-1	0	1.01	0.85	0.03	0.06
90-14-1	0.02	17.74	1.74	0.12	0.26
90-16-1	0.23	364.07	89.93	1.08	4.45
90-24-1	455.11	Timeout	Timeout	3.26	10.82

Table 5. ISCAS'89 Instances - Time in Seconds (600 second timeout)

<i>Instance</i>	BB	AND/OR	OR-Decomp+T	OR-Decomp+D	OR-Decomp+G
c432	0.23	0.13	129.76	0.58	5.17
c499	0.09	0.09	0.23	0.39	0.43
c880	0.3	0.28	0.82	1.33	1.54
s1196	0.13	0.13	256.03	0.57	0.65
s1238	0.11	0.12	0.5	0.55	0.63
s1423	1.64	1	3.23	0.89	1.04
s1488	0.17	0.16	0.53	0.88	1.01
s1494	Timeout	0.16	0.51	Timeout	1.0
s386	3.38	0.01	0.36	12.39	0.06
s953	0.09	Timeout	0.05	3.38	1.07

Table 6. Number of instances solved (600 second timeout)

<i>Benchmark</i>	RLFAP	Spot5	Grids	Iscas89	Total
OR-Decomp+G	5	5	4	10	24
BF	0	5	8	10	23

The Grid problems also benefit greatly from flexibility in variable ordering. **OR-Decomp+D** and **OR-Decomp+G** are both extremely effective since they exploit decomposition in the problem while still allowing complete dynamic variable ordering. Neither **AND/OR** nor **OR-Decomp+T** could solve instance 90-24-1.

ISCAS'89 instance also benefit from more flexible variable ordering, but decomposition is also effective. The two algorithms that solved the most ISCAS'89 instances are **OR-Decomp+T** and **OR-Decomp+G**. **OR-Decomp+T** could solve one instance not solved by **AND/OR**.

Finally, we compared **OR-Decomp+G** with Best First search using static mini buckets (**BF**) [16]. This algorithm explores an **AND/OR** tree, but does so in a best first manner rather than in a depth-first manner. Thus it can need considerably more space, which unlike caching is required for correctness. So we see, e.g., that it could not solve any of the RLFAP problems due to its space requirements. However, it was also able to solve some problems not solvable by **OR-Decomp**. It is difficult to assess the cause of this difference however, since the bounding technique of mini-buckets is quite distinct from the bounding technique of local propagation. Thus it is hard to say if these results (except for the RFLAP results) are due to better bounding or due to the differences in the search used by the algorithms.

7 Conclusions and Future Work

Constraint Optimization Problems can benefit greatly from both dynamic variable ordering and decomposition. Unfortunately the recursive nature of current decomposition techniques forces search to solve only one active component at a time. In this paper, we have introduced a novel search method that is able to exploit decomposition while at the same time allowing complete freedom to branch on any unassigned variable of any active component. We also introduced a new variable ordering algorithm which guides search toward decomposition, but still allows for the flexibility to choose any variable.

References

1. Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and Complexity Results for #SAT and Bayesian Inference. In: 44th Symposium on Foundations of Computer Science (FOCS), pp. 340–351 (2003)
2. Bensana, E., Lemaitre, M., Verfaillie, G.: Earth observation satellite management. *Constraints* 4(3), 293–299 (1999)
3. Bouveret, S., de Givry, S., Heras, F., Larrosa, J., Rollon, E., Sanchez, M., Schiex, T., Verfaillie, G., Zytnicki, M.M.: Max-csp competition 2007. In: *Proceedings of the Second International CSP Solver Competition*, pp. 19–21 (2008)
4. Brglez, F., Bryan, D., Kozminski, K.: Combinatorial Profiles of Sequential Benchmark Circuits. In: *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, pp. 1229–1234. IEEE, Los Alamitos (1989)
5. Cabon, B., de Givry, S., Lobjois, L., Schiex, T., Warners, J.: Radio link frequency assignment. *Constraints* 4(1), 79–89 (1999)
6. Cooper, M., de Givry, S., Schiex, T.: Optimal soft arc consistency. In: *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 68–73 (2007)
7. Darwiche, A.: Recursive conditioning. *Artif. Intell.* 126(1-2), 5–41 (2001)
8. de Givry, S., Schiex, T., Verfaillie, G.: Exploiting Tree Decomposition and Soft Local Consistency in Weighted CSP. In: *AAAI*, pp. 22–27 (2006)
9. Jégou, P., Ndiaye, S., Terrioux, C.: Dynamic heuristics for backtrack search on tree-decomposition of csp. In: *IJCAI*, pp. 112–117 (2007)
10. Kitching, M., Bacchus, F.: Symmetric component caching. In: *IJCAI*, pp. 118–124 (2007)
11. Kjaerulff, U.: Triangulation of graphs - algorithms giving small total state space, Technical Report R90-09. Technical report, Department of Computer Science, University of Aalborg (March 1990)
12. Larrosa, J., Meseguer, P., Sánchez, M.: Pseudo-tree search with soft constraints. In: *ECAI*, pp. 131–135 (2002)
13. Larrosa, J., Schiex, T.: Solving weighted csp by maintaining arc consistency. *Artificial Intelligence* 159(1-2), 1–26 (2004)
14. Marinescu, R., Dechter, R.: And/or branch-and-bound for graphical models. In: *IJCAI*, pp. 224–229 (2005)
15. Marinescu, R., Dechter, R.: Dynamic orderings for and/or branch-and-bound search in graphical models. In: *ECAI*, pp. 138–142 (2006)
16. Marinescu, R., Dechter, R.: Best-first and/or search for graphical models. In: *Proceedings of the AAAI National Conference (AAAI)*, pp. 1171–1176 (2007)
17. Sang, T., Beame, P., Kautz, H.: Performing bayesian inference by weighted model counting. In: *AAAI*, pp. 475–482 (2005)

A Coinduction Rule for Entailment of Recursively Defined Properties

Joxan Jaffar, Andrew E. Santosa, and Răzvan Voicu

School of Computing
National University of Singapore
Republic of Singapore 117543
{joxan, andrews, razvan}@comp.nus.edu.sg

Abstract. Recursively defined properties are ubiquitous. We present a proof method for establishing entailment $G \models H$ of such properties G and H over a set of common variables. The main contribution is a particular proof rule based intuitively upon the concept of *coinduction*. This rule allows the inductive step of assuming that an entailment holds during the proof the entailment. In general, the proof method is based on an unfolding (and no folding) algorithm that reduces recursive definitions to a point where only constraint solving is necessary. The constraint-based proof obligation is then discharged with available solvers. The algorithm executes the proof by a search-based method which automatically discovers the opportunity of applying induction instead of the user having to specify some induction schema, and which does not require any base case.

1 Introduction

A large category of formal verification problems can be expressed as proof obligations of the form G entails H , written $G \models H$, where G and H are recursively defined properties. Such problems appear in functional and logic programs, and specification languages such as JML, and they usually represent verification requirements for systems with infinite, or unbounded number of states, such as parameterized, or software systems. For instance, G might represent the semantics of a program, expressed as a formula in a suitable theory, whereas H may express a safety assertion.

Once the proof obligation $G \models H$ is formulated, it may be discharged with the help of a theorem prover such as Coq [1], HOL [6], or PVS [20]. While, in general, the proof process may be very complex, these tools provide a high level of assistance, automating parts of the process, and guaranteeing the correctness of the proof, once it is obtained. While there is, currently, a sustained research effort towards automating the process of discharging proof obligations, this process still requires, in general, a significant level of manual input. In the case of inductive proofs, for instance, the inductive variable, its base case, and the induction hypothesis need to be provided manually.

In this paper we present a proof method that establishes an entailment of the form $G \models H$, where G and H are two recursively defined properties over a set of common variables. The use of a coinduction principle (which does not require a base case), coupled with the standard operation of unfolding recursive definitions, allows the opportunistic discovery of suitable induction hypotheses, and makes our method amenable

to automation. The entire framework is formalized in Constraint Logic Programming (CLP), so that CLP predicates can be used to describe the recursive properties of interest. Our method is, in fact, centered around an algorithm whose main operation is the standard unfolding of a CLP goal. The unfolding operation is applied to both the lhs G and the rhs $\mathcal{H}$ of the entailment. The principle of coinduction allows the discovery of a valid induction hypothesis, thus terminating the unfolding process. Through the application of the coinduction principle, the original proof obligation usually reduces to one that no longer contains recursive predicates. The remaining proof obligation contains only base constraints, and can be relegated to the underlying constraint solver.

Let us illustrate this process on a small example. Consider the definition of the following two recursive predicates

$$\begin{array}{ll} m4(0). & even(0). \\ m4(X+4) :- m4(X). & even(X+2) :- even(X). \end{array}$$

whose domain is the set of non-negative integers. The predicate $m4$ defines the set of multiples of four, whereas the predicate $even$ defines the set of even numbers. We shall attempt to prove that $m4(X) \models even(X)$, which in fact states that every multiple of four is even. We start the proof process by performing a *complete unfolding* on the lhs goal. By “complete,” we mean that we use all the clauses whose head unify with $m4(X)$ ¹. We note that $m4(X)$ has two possible unfoldings, one leading to the empty goal with the answer $X = 0$, and another one leading to the goal $m4(X'), X' = X - 4$. The two unfolding operations, applied to the original proof obligation result in the following two new proof obligations, both of which need to be discharged in order to prove the original one.

$$\begin{array}{l} X = 0 \models even(X) \quad (1) \\ m4(X'), X' = X - 4 \models even(X) \quad (2) \end{array}$$

The proof obligation (1) can be easily discharged. Since unfolding on the lhs is no longer possible, we can only unfold on the rhs. We choose¹ to unfold with clause $even(0)$, which results in a new proof obligation which is trivially true, since its lhs and rhs are identical.

For proof obligation (2), before attempting any further unfolding, we note that the lhs $m4(X')$ of the current proof obligation, and the lhs $m4(X)$ of the original proof obligation, are unifiable (as long as we consider X' a fresh variable), which enables the application of the coinduction principle. First, we “discover” the *induction hypothesis* $m4(X') \models even(X')$, as a variant of the original proof obligation. Then, we use this induction hypothesis to replace $m4(X')$ in (2) by $even(X')$. This yields the new proof obligation

$$even(X'), X' = X - 4 \models even(X) \quad (3)$$

To discharge (3), we unfold twice on the rhs, using the $even(X+2) :- even(X)$ clause. The resulting proof obligation is

$$even(X'), X' = X - 4 \models even(X'''), X''' = X'' - 2, X'' = X - 2 \quad (3)$$

¹ The requirement of a complete unfold on the lhs, and the lack of such requirement on the rhs, is explained in Section 3.

where variables X'' and X''' are existentially quantified². Using constraint simplification, we reduce this proof obligation to $\text{even}(X - 4) \models \text{even}(X - 4)$, which is obviously true.

At this point, all the proof obligations have been discharged and the proof is complete. Informally, we have performed four kinds of operations: (a) left unfolding, (b) right unfolding, (c) application of coinduction, and (d) constraint solving/simplification. While we shall relegate to Section 3 the argument that all these steps are correct, we would like to further emphasize several aspects concerning our proof method.

First, our method is amenable to automation, in the form of a non-deterministic algorithm. The state of the proof is given by a proof tree, whose frontier has the current proof obligations, all of which have to be discharged in order to complete the proof. Each proof step applies non-deterministically one of the four operations given above to one of the current proof obligations. Of these four, the lhs and rhs unfolding operations expand the tree by adding new descendants. In contrast, the coinduction operation searches the ancestors of the current goal for a matching lhs. If one is found, then a suitable induction hypothesis is generated, and applied to the lhs of the current goal, as shown in the small example given above. The fourth kind of operation performs constraint simplification/solving, possibly discharging the current proof obligation. As our examples show, the unfolding process and the application of the coinduction principle require no manual intervention.

Second, our coinductive proof step is inspired from tabled logic programming [24]. The intuition behind the correctness of this step is that, since the unfolding of the lhs is complete, we are already exploring all the possibilities of finding a counterexample, i.e. a substitution θ for which $G\theta$ is true while $H\theta$ is false. Whenever we find an ancestor with lhs G' which is variant of the lhs G (or some subgoal thereof) of the current proof obligation, we can immediately conclude that the current proof obligation would not contribute counterexamples that wouldn't already be visible from its matching ancestor. However, for this statement to be indeed true, we need to establish a similar matching between the rhs of the two proof obligations. This condition is expressed by the proof obligation obtained after the application of the coinductive step.

Finally, we would like to clarify that the use of the term *coinduction* pertains to the way the proof rules are employed for a proof obligation $G \models H$, and has no bearing on the *greatest* fixed point of the underlying logic program P . In fact, our proof method, when applied successfully, proves that G is a subset of H wrt. the *least* fixpoint of (the operator associated with) the program. However, as further clarified in Section 4, the success of the proof method is modeled as a property of a potentially infinite proof tree, and thus *coinduction*, rather than induction, needs to be employed to establish it.

1.1 Related Work

Variants of our proof method have been applied in more restricted settings of timed automata verification [10] and reasoning about structural properties of programs [12]. In the current paper, we focus on the common techniques used as well as hinting towards greater class of applications.

² In Section 3 we handle these variables formally.

Among logic-programming-based proof methods, early works of [13,14] propose *definite clause inference* and *negation as failure inference (NFI)* which are similar to our unfolding rules. These inferences are applied prior to concluding a proof of an implication using a form of computational induction. A form of structural induction in a similar framework is employed in [4]. We note that these proof methods are based on fitting in the allowable inductive proofs into an *induction schema*, which is usually syntax-based. Mesnard et al. [18] propose a CLP proof method for a system of implications, whose consequents contain only constraints. This technique is not completely general. Craciunescu [3] proposed a method to prove the equivalence of CLP programs using either induction or coinduction. The notion of coinduction here is different from ours; they reason about the *greatest* fixpoint of a CLP program, while we reason about the least.

Among the more automated approaches, [21,22] used unfold/fold transformation of logic programs to prove equivalences of goals. [22] presents a proof method for equivalence assertions on parameterized systems. Hsiang and Srivas [7,8] propose an inductive proof method for Prolog programs. The main feature of the proof method is a semi-automatic generation of induction schema (in the sense, this objective is similar to those of Kanamori and Fujita [13] mentioned above). The generation of inductive assertions is by producing the reduct of the goals (unfolding). Termination of the unfolding is implemented by a marking mechanism on the variables. Whenever an input variables is instantiated during an unfold (in other words, we need to make a decision about its value), it is marked. In a sense, this is similar to the use of *bomblist* in the Boyer-Moore prover [2]. As is the case with Boyer-Moore prover, the induction is structural. Here, the method requires the user to distinguish a set of *input* variables to structurally induct on. In comparison, we employ no induction schema. We detect the point where we apply the induction hypothesis automatically using constraint subsumption test. In other words, we discover the induction schema dynamically using indefinite steps of unfolds. This approach is more complete and automatable.

The work of Roychoudhury et al. [23] systematizes induction proofs using tabled resolution of logic programming. It is essentially based on unfolding, delaying upon detection of potential infinite resolution, and finally a folding step to conclude similarity. These serve to extend the tabled resolution engine of XSB tabled logic programming system. Our work generalizes this idea by providing a constraint-based inductive proof rules based on automated detection of cycles using constraints. Our rules are also based on the notion of tabling of assertions, which are later re-used as induction hypothesis.

Another form of tabling is also employed in *Prolog Technology Theorem Prover (PTTP)* [25]. Here the proof process is basically Prolog's search for refutation with several extensions, including a *model elimination reduction (ME reduction)*, which memoizes literals, and whenever a new goal which is contradictory to a stored literal is found, we stop because this constitutes a refutation. The part of PTTP that is akin to our coinduction is the detection when there is an occurrence of the same literal in which case, the system backtracks. Our work departs from PTTP mainly by the use of constraints.

Recursive definitions are also encountered in data structure verification area. [17] presents an algorithm for specification and verification of data structure using equality axioms. In [19] user-defined recursive definitions are allowed to specify "shape"

properties. Proofs are carried out via fold/unfold transformations. As we will exemplify later, our algorithm can be used to automatically perform proofs of assertions containing recursive data structure definitions.

Finally, we mention the work in [5], which uses a coinductive interpretation of logic programming rules to express properties of infinite or circular data structures. The term *coinductive* is used here to refer to the greatest fixed point of the program at hand. We re-emphasize at this point that, in contrast with [5], our use of the term “coinductive” refers to the way our proof rules are employed, and bears no direct relationship to the greatest fixed point of the logic program.

2 Constraint Logic Programs

We use CLP [9] definitions to represent our verification conditions. To keep our paper self-contained, we provide a minimal background on the constraint logic programming framework.

An *atom* is of the form $p(\tilde{t})$ where p is a user-defined predicate symbol and $\tilde{t}$ a tuple of terms, written in the language of an underlying constraint solver. A *clause* is of the form $A : -\Psi, \tilde{B}$ where the atom A is the *head* of the clause, and the sequence of atoms $\tilde{B}$ and constraint Ψ constitute the *body* of the clause. The constraint Ψ is also written in the language of the underlying constraint solver, which is assumed to be able to decide (at least reasonably frequently) whether Ψ is satisfiable or not. In our examples, we assume an integer and array constraint solver, as described below.

A *program* is a finite set of clauses. A *goal* has exactly the same format as the body of a clause. A goal that contains only constraints and no atoms is called *final*.

A *substitution* θ simultaneously replaces each variable in a term or constraint e into some expression, and we write $e\theta$ to denote the result. A *renaming* is a substitution which maps each variable in the expression into a distinct variable. A *grounding* is a substitution which maps each integer or array variable into its intended universe of discourse: an integer or an array. Where Ψ is a constraint, a grounding of Ψ results in *true* or *false* in the usual way.

A *grounding* θ of an atom $p(\tilde{t})$ is an object of the form $p(\tilde{t}\theta)$ having no variables. A grounding of a goal $G \equiv (p(\tilde{t}), \Psi)$ is a grounding θ of $p(\tilde{t})$ where $\Psi\theta$ is *true*. We write $\llbracket G \rrbracket$ to denote the set of groundings of G .

Let $G \equiv (B_1, \dots, B_n, \Psi)$ and P denote a non-final goal and program respectively. Let $R \equiv A : -\Psi_1, C_1, \dots, C_m$ denote a clause in P , written so that none of its variables appear in G . Let the equation $A = B$ be shorthand for the pairwise equation of the corresponding arguments of A and B . A *reduct* of G using a clause R , denoted $\text{reduct}(G, R)$, is of the form

$$(B_1, \dots, B_{i-1}, C_1, \dots, C_m, B_{i+1}, \dots, B_n, B_i = A, \Psi, \Psi_1)$$

provided the constraint $B_i = A \wedge \Psi \wedge \Psi_1$ is satisfiable.

A *derivation sequence* for a goal G_0 is a possibly infinite sequence of goals $G_0, G_1, \dots$ where $G_i, i > 0$ is a reduct of G_{i-1} . If the last goal G_n is a final goal, we say that the derivation is *successful*. A *derivation tree* for a goal is defined in the obvious way.

Definition 1 (Unfold). Given a program P and a goal G , $\text{UNFOLD}(G)$ is $\{G' \mid \exists R \in P : G' = \text{reduct}(G, R)\}$. $\square$

In the formal treatment below, we shall assume, without losing generality, that goals are written so that atoms contain only distinct variables as arguments.

2.1 An Integer and Array Constraint Language

In this section we provide a short description of constraint language allowed by the underlying constraint solver assumed in all our examples. We consider three kinds of terms: integer and array terms. Integer terms are constructed in the usual way, with one addition: the array element. The latter is defined recursively to be of the form $a[i]$ where a is an *array expression* and i an integer term. An array expression is either an array variable or of the form $\langle a, i, j \rangle$ where a is an array expression and i, j are integer terms. A term is either constructed from an array “segment”: $a\{i..j\}$ where a is an array expression and i, j integer variables.

The meaning of an array expression is simply a map from integers into integers, and the meaning of an array expression $a' = \langle a, i, j \rangle$ is a map just like a except that $a'[i] = j$. The meaning of array elements is governed by the classic McCarthy [16] axioms:

$$\begin{aligned} i = k &\rightarrow \langle a, i, j \rangle[k] = j \\ i \neq k &\rightarrow \langle a, i, j \rangle[k] = a[k] \end{aligned}$$

A *constraint* is either an integer equality or inequality, an equation between array expressions. The meaning of a constraint is defined in the obvious way.

In what follows, we use constraint to mean either an atomic constraint or a conjunction of constraints. We shall use the symbol ψ or Ψ , with or without subscripts, to denote a constraint.

3 Proof Method for Recursive Assertions

3.1 Overview

In this key section, we consider proof obligations of the form $G \models \mathcal{H}$ where $\text{var}(\mathcal{H}) \subseteq \text{var}(G)$. The validity of this formula expresses the fact that $\mathcal{H}\theta$ succeeds w.r.t. the CLP program at hand whenever $G\theta$ succeeds, for any grounding θ of G . They are the central concept of our proof system, by being expressive enough to capture interesting properties of data structures, and yet amenable to automatic proof process.

Intuitively, we proceed as follows: unfold G completely a finite number of steps in order to obtain a “frontier” containing the goals $G_1, \dots, G_n$. Then unfold $\mathcal{H}$, but this time not necessarily completely, that is, not necessarily obtaining *all* the reducts each time, obtain goals $\mathcal{H}_1, \dots, \mathcal{H}_m$. This situation is depicted in Figure 1. Then, the proof holds if

$$G_1 \vee \dots \vee G_n \models \mathcal{H}_1 \vee \dots \vee \mathcal{H}_m$$

or alternatively, $G_i \models \mathcal{H}_1 \vee \dots \vee \mathcal{H}_m$ for all $1 \leq i \leq n$. This follows from the fact that $G \models G_1 \vee \dots \vee G_n$, (which is not true in general, but true in the least-model semantics

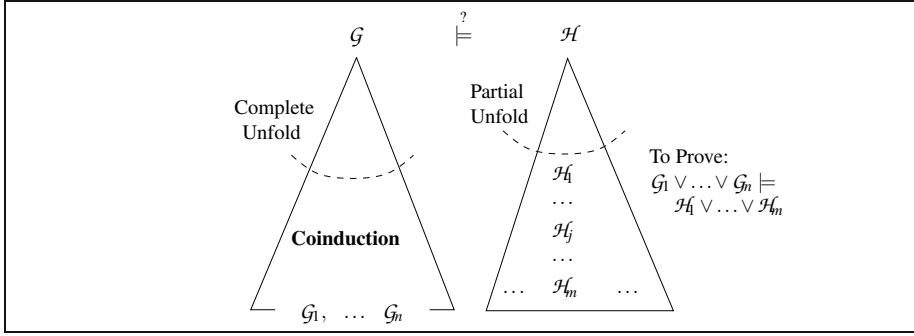


Fig. 1. Informal Structure of Proof Process

of CLP), and the fact $\mathcal{H}_j \models \mathcal{H}$ for all j such that $1 \leq j \leq m$. More specifically, but with some loss of generality, the proof holds if

$$\forall i : 1 \leq i \leq n, \exists j : 1 \leq j \leq m : \mathcal{G}_i \models \mathcal{H}_j$$

and for this reason, our *proof obligation* shall be defined below to be simply a pair of goals, written $\mathcal{G}_i \models \mathcal{H}_j$.

3.2 The Proof Rules

We now present a formal calculus for the proof of $\mathcal{G} \models \mathcal{H}$. To handle the possibly infinite unfoldings of $\mathcal{G}$ and $\mathcal{H}$, we shall depend on the use of a key concept: *coinduction*. Proof by coinduction allows us to assume the truth of a *previous* obligation. The proof proceeds by manipulating a set of *proof obligations* until it finally becomes empty or a counterexample is found. Formally, a *proof obligation* is of the form $\tilde{A} \vdash \mathcal{G} \models \mathcal{H}$ where the $\mathcal{G}$ and $\mathcal{H}$ are goals and $\tilde{A}$ is a set of *assumption* goals. The role of proof obligations is to capture the state of a proof. The set $\tilde{A}$ contains goals whose truth can be assumed coinductively to discharge the proof obligation at hand. This set is implemented in our algorithm using a table as described in the next section.

(LU+I)	$\frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \bigcup_{i=1}^n \{\tilde{A} \cup \{\mathcal{G} \models \mathcal{H}\} \vdash \mathcal{G}_i \models \mathcal{H}\}} \quad \text{UNFOLD}(\mathcal{G}) = \{\mathcal{G}_1, \dots, \mathcal{G}_n\}$
(RU)	$\frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}'\}} \quad \mathcal{H}' \in \text{UNFOLD}(\mathcal{H})$
(CO)	$\frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}}{\Pi \cup \{\emptyset \vdash \mathcal{H}' \theta \models \mathcal{H}\}} \quad \mathcal{G}' \models \mathcal{H}' \in \tilde{A} \text{ and there exists a substitution } \theta \text{ s.t. } \mathcal{G} \models \mathcal{G}' \theta.$
(CP)	$\frac{\Pi \uplus \{\tilde{A} \vdash \mathcal{G} \wedge p(\tilde{x}) \models \mathcal{H} \wedge p(\tilde{y})\}}{\Pi \cup \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H} \wedge \tilde{x} = \tilde{y}\}}$
(DP)	$\frac{\Pi \uplus \{\mathcal{G} \models \mathcal{H}\}}{\Pi} \quad \mathcal{G} \models \mathcal{H} \text{ holds by constraint solving}$

Fig. 2. Proof Rules

Our proof rules are presented in Figure 2. The $\uplus$ symbol represents the disjoint union of two sets, and emphasizes the fact that in an expression of the form $A \uplus B$, we have that $A \cap B = \emptyset$. Each rule operates on the (possibly empty) set of proof obligations Π , by selecting one of its proof obligations and attempting to discharge it. In this process, new proof obligations may be produced.

The *left unfold with new induction hypothesis* (LU+I) (or simply “left unfold”) rule performs a complete unfold on the lhs of a proof obligation, producing a new set of proof obligations. The original assertion, while removed from Π , is added as an assumption to every newly produced proof obligation, opening the door to using coinduction later in the proof.

The rule *right unfold* (RU) performs an unfold operation on the rhs of a proof obligation. In general, the two unfold rules will be systematically interleaved. The resulting proof obligations are then discharged either coinductively or directly, using the (CO) and (CP) rules, respectively.

The rule *coinduction application* (CO) transforms an obligation by using an assumption, and thus opens the door to discharging that obligation via the direct proof (CP) rule. Since assumptions can only be created using the (LU+I) rule, the (CO) rule realizes the coinduction principle. The underlying principle behind the (CO) rule is that a “similar” assertion $\mathcal{G}' \models \mathcal{H}'$ has been previously encountered in the proof process, and assumed as true.

Note that this test for coinduction applicability is itself of the form $\mathcal{G} \models \mathcal{H}$. However, the important point here is that this test can only be carried out using constraints, in the manner prescribed for the CP rule described below. In other words, this test does not use the definitions of assertion predicates.

Finally, the rule *constraint proof* (CP), when used repeatedly, discharges a proof obligation by reducing it to a form which contains no assertion predicates. Note that one application of this removes one occurrence of a predicate $p(\bar{y})$ appearing in the rhs of an obligation. Once a proof obligation has no predicate in the rhs, a *direct proof* (DP) may be attempted by simply removing any predicates in the corresponding lhs.

Given a proof obligation $\mathcal{G} \models \mathcal{H}$, a proof shall start with $\Pi = \{\emptyset \vdash \mathcal{G} \models \mathcal{H}\}$, and proceed by repeatedly applying the rules in Figure 2 to it.

3.3 The Algorithm

We now describe a strategy so as to make the application of the rules automated. Here we propose systematic interleaving of the left-unfold (LU+I) and right-unfold (RU) rules, attempting a constraint proof along the way. As CLP can be execution by resolution, we can also execute our proof rules, based on an algorithm which has some resemblance to tabled resolution.

We present our algorithm in pseudocode in Figure 3. Note that the presentation is in the form of a nondeterministic algorithm, and thus each of the nondeterministic operator **choose** needs to be implemented by some form of systematic search. Note also that when applying coinduction step, we test that some assertion $\mathcal{G}' \models \mathcal{H}'$ is stored in some table.

In Figure 3, by a *constraint proof* of a obligation, we mean to repeatedly apply the CP rule in order to remove all occurrences of assertion predicates in the obligation, in an obvious way. Then the constraint solver is applied to the resulting obligation.

REDUCE($\mathcal{G} \models \mathcal{H}$) **returns boolean**

- **Constraint Proof: (CP) + Constraint Solving (DP)**
 Apply a constraint proof to $\mathcal{G} \models \mathcal{H}$.
 If successful, **return true**, otherwise **return false**
- **Memoize ($\mathcal{G} \models \mathcal{H}$) as an assumption**
- **Coinduction: (CO)**
if there is an assumption $\mathcal{G}' \models \mathcal{H}'$ such that
 REDUCE($\mathcal{G} \models \mathcal{G}'\theta$) = true and REDUCE($\mathcal{H}'\theta \models \mathcal{H}$) = true **then return true**.
- **Unfold:**
choose left or right
case: Left: (LU+I)
choose an atom A in $\mathcal{G}$ to reduce
 for all reducts $\mathcal{G}_L$ of $\mathcal{G}$ using A :
 if REDUCE($\mathcal{G}_L \models \mathcal{H}$) = false **return false**
return true
case: Right: (RU)
choose an atom A in $\mathcal{H}$ to reduce, obtaining $\mathcal{G}_R$
return REDUCE($\mathcal{G} \models \mathcal{G}_R$)

Fig. 3. The Algorithm

3.4 Correctness

Given a proof obligation $\mathcal{G} \models \mathcal{H}$, a proof starts with $\Pi = \{\tilde{A} \vdash \mathcal{G} \models \mathcal{H}\}$, and proceeds by repeatedly applying the rules in Figure 2. The omission of negative literals in the body of the clauses of program P ensures that it has a unique *least model*, denoted $lm(P)$.

Theorem 1 (Soundness). *A proof obligation $\mathcal{G} \models \mathcal{H}$ holds, that is, $lm(P) \rightarrow (\mathcal{G} \models \mathcal{H})$ for the given program P , if, starting with the proof obligation $\emptyset \vdash \mathcal{G} \models \mathcal{H}$, there exists a sequence of applications of proof rules that results in proof obligations $\tilde{A} \vdash \mathcal{G}' \models \mathcal{H}'$ such that (a) $\mathcal{H}'$ contains only constraints, and (b) $\mathcal{G}' \models \mathcal{H}'$ can be discharged by the constraint solver.* $\square$

Proof Outline. The rule (RU) is sound because by the semantics of CLP, when $\mathcal{H}' \in \text{UNFOLD}(\mathcal{H})$ then $\mathcal{H}' \models \mathcal{H}$. Therefore, the proof of the obligation $\tilde{A} \vdash \mathcal{G} \models \mathcal{H}$ can be replaced by the proof of the obligation $\tilde{A} \vdash \mathcal{G} \models \mathcal{H}'$ since $\mathcal{G} \models \mathcal{H}'$ is stronger than $\mathcal{G} \models \mathcal{H}$. Similarly, the rule (CP) is sound because $\mathcal{G} \models \mathcal{H} \wedge \tilde{x} = \tilde{y}$ is stronger than the $\mathcal{G} \wedge p(\tilde{x}) \models \mathcal{H} \wedge p(\tilde{y})$.

The rule (LU+I) is *partially sound* in the sense that when $\text{UNFOLD}(\mathcal{G}) = \{\mathcal{G}_1, \dots, \mathcal{G}_n\}$, then proving $\mathcal{G} \models \mathcal{H}$ can be substituted by proving $\mathcal{G}_1 \models \mathcal{H}, \dots, \mathcal{G}_n \models \mathcal{H}$. This is because in the least-model semantics of CLP, $\mathcal{G}$ is equivalent to $\mathcal{G}_1 \vee \dots \vee \mathcal{G}_n$. However, whether the addition of $\mathcal{G} \models \mathcal{H}$ to the set of assumed assertions $\tilde{A}$ is sound depends on the use of the set of assumed assertions in the application of (CO).

Notice that in the rule (CO) we require the proofs of both $\mathcal{G} \models \mathcal{G}'\theta$ and $\mathcal{H}'\theta \models \mathcal{H}$ for some substitution θ . These proofs establish *subsumption*, that is the implication $(\mathcal{G}' \models \mathcal{H}') \rightarrow (\mathcal{G} \models \mathcal{H})$.

Assume that using our method, given a program P , we managed to conclude $G \models \mathcal{H}$ where G and $\mathcal{H}$ are goals possibly containing atoms and it is not the case that $G \models \mathcal{H}$ can be proved without the application of (LU+I) (since otherwise trivial by soundness of (RU) and (CP)). Assume that in the proof, there are a number of assumed assertions $A_1, \dots, A_n$ used coinductively as induction hypotheses. This means that in the proof of $G \models \mathcal{H}$ the left unfold rule (LU+I) has been applied at least once (possibly interleaved with the applications of (RU) and (CP)) obtaining two kinds of assertions:

1. Assertions C which are directly proved using (RU), (CP), and constraint solving (DP).
2. Assertions B which are proved using (CO) step using some assumed assertion A_j as hypothesis for $1 \leq j \leq n$.

We may conclude that $G \models \mathcal{H}$ holds. We now outline the proof of this.

First, define a *refutation* to an assertion $G \models \mathcal{H}$ as a successful derivation of one or more atoms in G whose answer Ψ has an instance (ground substitution) θ such that $\Psi\theta \wedge \mathcal{H}\theta$ is false. A finite refutation corresponds to a such derivation of finite length. A nonexistence of finite refutation means that $lm(P) \rightarrow (G \models \mathcal{H})$. A derivation of an atom is obtainable by left unfold (LU+I) rule only. Hence a finite refutation of length k implies a corresponding k left unfold (LU+I) applications that results in a contradiction.

Due to:

1. The soundness of other rules (RU) and (CP) and the partial soundness of (LU+I) with the fact that A_i for all $1 \leq i \leq n$ is obtained from $G \models \mathcal{H}$ by applying these rules, and
2. All assertions C are proved by (RU), (CP) and constraint solving (DP) alone,

we have: $G \models \mathcal{H}$ holds if A_i holds for all $1 \leq i \leq n$, and this holds iff for all i such that $1 \leq i \leq n$, and for all $k \geq 0$: A_i has no finite refutation of length k .

We prove inductively:

- **Base case:** When $k = 0$, for all i such that $1 \leq i \leq n$, A_i trivially has no finite refutation of length 0.
- **Inductive case:** Assume that for all i such that $1 \leq i \leq n$, A_i has no finite refutation of length k or less (*), we want to prove that for all i such that $1 \leq i \leq n$, A_i has no finite refutation of length $k + 1$ or less (**).

Notice again in our assumptions above that assertions B are proved by applying (CO) using A_j for some $1 \leq j \leq n$. Because subsumption holds in every application of (CO), this means that for such B , $A_j \rightarrow B$. (**).

The proof is by contradiction. Now suppose that (**) is false, that is, A_i for some i such that $1 \leq i \leq n$ has a finite refutation of length $k + 1$ or less. But due to our hypothesis (*), A_i has no finite refutation of length k or less. Therefore it must be the case that A_i has a finite refutation of length $k + 1$.

Again, note that we have applied (LU) to A_i at least once on the resulting assertions, possibly interleaved with applications of (RU) and (CP) obtaining the following two kinds of assertions:

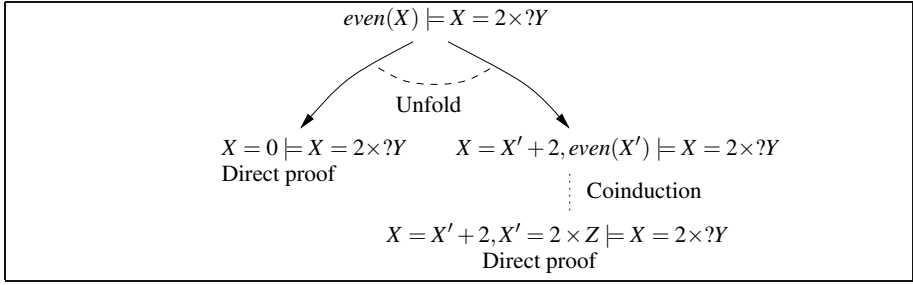


Fig. 4. Proof Tree Example

1. Assertions C which are proved by applications of (RU) and (CP) and constraint solving alone.
2. Assertions B which are proved by (CO) using some A_j for $1 \leq j \leq n$ in the set of assumed assertions as induction hypothesis.

Then in the above set of assertions, either:

1. Some assertion of type C is a refutation to A_i of length $k + 1$. However, regardless of the length, since all such assertions C are already proved by (RU), (CP), and constraint solving, this case is not possible.
2. Since A_i has to have a finite refutation of length $k + 1$, therefore there has to be at least one assertion of type B that is reached in k or less unfolds. Therefore, B has to have a refutation of length k or less. Now since subsumption (***) holds, then it should be the case that some A_j for $1 \leq j \leq n$ such that $A_j \rightarrow B$ also has a finite refutation of length k or less. But this contradicts our hypothesis (*) that A_i for all $1 \leq i \leq n$ has no finite refutation of length k or less. $\square$

We finally mention that the proof rules are not *complete*. For example, when we have a program

$$\begin{aligned} p(X) &:- 0 \leq X \leq 3. \\ q(X) &:- 0 \leq X \leq 2. \\ q(X) &:- 1 \leq X \leq 3. \end{aligned}$$

obviously $p(X) \models q(X)$ holds, but we cannot prove this using our rules. The reason is that $0 \leq X \leq 3$ (obtained from the unfold of $p(X)$) does not imply either $0 \leq X \leq 2$ or $1 \leq X \leq 3$ (both obtained by right unfolding $q(X)$). It is possible, however, to introduce new rules toward achieving completeness. For proving the above assertion, we could introduce a splitting of an assertion. For instance, we may split $\mathcal{G} \models \mathcal{H}$ into $\mathcal{G}, \phi \models \mathcal{H}$ and $\mathcal{G}, \neg\phi \models \mathcal{H}$ (ϕ in our example would be, say, $X \leq 1$). However, this is beyond the scope of this paper.

4 On the Coinduction Rule

Consider again the predicate *even* presented in Section 1. We now demonstrate a simple application of our rules to prove a property on the predicate. Consider proving the

assertion $even(X) \models X = 2 \times ?Y$, call it A (we denote existentially-quantified variables with the query symbol “?”). The proof process starts by applying the (LU+I) rule unfolding the $even(X)$ goal, resulting in two new proof obligations, each with the original goal A as its assumption. On the left branch, after unfolding with the base-case clause, we are left with $X = 0 \models X = 2 \times ?Y$, which can be discharged by direct proof using a constraint solver. On the right branch of the proof, the unfolding rule produces the proof obligation $even(X'), X = X' + 2 \models X = 2 \times ?Y$. Here we apply the coinduction (CO) rule using $even(X) \models X = 2 \times ?Y$ as induction hypothesis, spawning an obligation to prove $X' = 2 \times Z, X = X' + 2 \models X = 2 \times ?Y$. This can then be proved using constraint solving.

Let us now recall our example in Section 1. In Section 1 we have applied (LU+I) to unfold the predicate $m4(X)$ resulting in the two obligations (1) and (2). We apply (RU) to perform right unfold on (1). We apply (CO) to (2) obtaining (3). We then apply (RU) to (3) twice to establish it.

Our system does not require the user to manually specify induction hypothesis and/or construct induction schema. Instead, any induction hypothesis used is obtained dynamically during the proof process. Let us now exemplify this concept by considering the program

$$\begin{aligned} p(X) &:- q(X). \\ q(X) &:- q(X). \\ r(X). \end{aligned}$$

Here we want to prove $p(X) \models r(X)$. Call this A_1 . We first apply (LU+I) to the assertion obtaining $q(X) \models r(X)$. Call this assertion A_2 . At this point, our algorithm tests whether A_1 can be used as a induction hypothesis to establish A_2 . This fails, and we again apply (LU+I) obtaining another assertion A_3 which is equivalent to A_2 . Upon obtaining A_3 , the set of assumed assertions contain both A_1 and A_2 . The algorithm now tests whether any of these can be used in a (CO) application. Indeed, we can use the assertion A_2 which is identical to A_3 . In this way induction hypotheses are chosen dynamically.

In the preceding examples we have demonstrated the use of the rule (CO) to conclude proofs. Moreover, the last example illustrates the fact that, in contrast to most inductive proof methods, our proof process may be successful even in the absence of a base case. While the lack of a base case requirement justifies the qualifier “coinductive” being applied to our proof method, the fact that this term has been somewhat overused in the logic programming community warrants further clarification.

In our view, induction and coinduction are two flavours of one general proof scheme, which is used to prove properties of objects defined by means of recursive rules. This general scheme proves properties of such objects by assuming that the property of interest already (inductively) holds for the “smaller” object on which the definition recurses. Now, recursive definitions may be interpreted in an inductive or coinductive manner, and each of these interpretations would lead to the general proof scheme being construed as either induction or coinduction.

The crux of our proof method is to automatically generate an induction hypothesis for the goal at hand, in an attempt to produce a successful application of the general proof scheme mentioned above. The method works correctly irrespective of whether

Program:	CLP Model:
$F(x) \leftarrow \text{if } p(x) \text{ then } x$	$s(X, X) :- X = \text{error}.$
$\quad \text{else } F(h(x))$	$s(X, X_f) :- X \neq \text{error}, p(X) = 1, X_f = X.$
	$s(X, X_f) :- X \neq \text{error}, p(X) = 0, s(h(X), Y), s(Y, X_f).$

Fig. 5. Idempotent Function

the rules defining the properties of interest are interpreted inductively or coinductively³. Since our proof method does not explicitly look for base cases, and since it can also handle the situation where a recursive definition of a property would be interpreted coinductively, we have chosen to use the qualifier “coinductive.” However, this qualifier bears no direct relationship to the greatest fixed point of the logic program at hand. Throughout this paper, our recursive definitions are meant to be interpreted inductively, and the meaning of the goal $G \models \mathcal{H}$ is that whenever a grounding $G\theta$ lies in the *least* fixed point of the program at hand P , it follows that the grounding $\mathcal{H}\theta$ is also in $\text{lp}(P)$. Our proof method will be successful only when this interpretation of a goal holds.

5 Proof Examples

In our driving examples area of program verification, most of the entailment problems we have encountered can be proved by our algorithm automatically. We believe they cannot be automatically discharged by any existing systematic method. In this section, we present two examples.

5.1 Function Idempotence

Suppose that we have the function in Figure 5 [15] with its CLP representation. Note that *error* represents the return value of the function on divergent termination. Here we want to prove idempotence, that is $F(x) = F(F(x))$, or that both the assertions A) $s(X, Y), s(Y, X_f) \models s(X, X_f)$ and B) $s(X, X_f) \models s(X, ?Y), s(?Y, X_f)$ holds. The mechanical proof of Assertion A requires coinduction and will be exemplified here. The algorithm first applies (LU+I) obtaining the assertions 1) $s(\text{error}, X_f) \models s(\text{error}, X_f)$, 2) $X \neq \text{error}, p(X) = 1, X = Y, s(Y, X_f) \models s(X, X_f)$, and 3) $X \neq \text{error}, p(X) = 0, s(h(X), Z), s(Z, Y), s(Y, X_f) \models s(X, X_f)$. Assertions 1 and 2 are proved by (CP) and (DP), and the algorithm attempts to apply (CO) to Assertion 3 using the ancestor Assertion A as hypothesis.

The application of (CO) obtains the obligation $X \neq \text{error}, p(X) = 0, s(h(X), Y), s(Y, X_f) \models s(X, X_f)$. This assertion cannot be proved by constraint proof nor by coinduction (since the set of assumed assertions are empty), and the algorithm proceeds to proving by unfolding. Here it applies right unfold (RU) rule obtaining $X \neq \text{error}, p(X) = 0, s(h(X), Y), s(Y, X_f) \models X \neq \text{error}, p(X) = 0, s(h(X), ?Z), s(?Z, X_f)$, which can be proved directly. Since the application of (CO) to Assertion 3 has been successful, the proof concludes.

³ Nevertheless, the complete unfold of the left goal ensures that correct base case proofs are generated whenever the current recursive definition provides such base cases.

Program:	Assertion Predicate:
<pre> {h = h₀, p = p₀ > 0} (0) while (p > 0) do [p] := 0 (1) p := [p+1] (2) end (3) {∃y.allz(h₀, h, p₀, y), h[y+1] = 0} </pre>	<pre> allz(H, ⟨H, L, 0⟩, L, L) :- L > 0. allz(H₁, ⟨H₂, L, 0⟩, L, R) :- L > 0, allz(H₁, H₂, H₁[L+1], R). </pre>

Fig. 6. List Reset

5.2 A Pointer Data Structure Example: List Reset

We represent pointers as indices in an array which we call the *heap*. We write $[p]$ to refer to the location referenced by the pointer p . To implement a linked list, we shall assume that a list element is made up of two adjacent heap cells. Thus, for the list element referenced by p , the data field is $[p]$, and the reference to the next element is $[p+1]$. In the CLP program, given an array H , which typically denotes the heap, we denote by $H[I]$ the element referenced by index I in the array. We also denote by $\langle H, I, J \rangle$ the array that is identical to H for all indices, except I , where the original value is replaced by J . The steps for solving constraints containing these constructs are discussed in [11].

Figure 6 shows a program which “zeroes” all elements of a given linked list with head p . We prove that the program produces a nonempty null-terminating list with zero values. Note that in Figure 6, h is a program variable denoting the current heap. The predicate takes into consideration the memory model of the program and expresses the relationship between the heap H before the execution of the program, and the heap H' obtained after the program has completed. Thus, the predicate $allz(H, H', L, R)$ states that the heap H' differs from H only by having zero elements in the non-empty sublist from L to R .

In Figure 6 we provide a *tail-recursive* definition of $allz$ which defines a zeroed list segment (L, R) as one whose head contains zero, and its tail is, recursively, the zeroed list segment $(H[L+1], R)$ ⁴. We could have used a *sublist-recursive* specification: a zeroed list segment (L, R) is defined to be a zeroed list segment (L, T) appended by one extra zero element R . Clearly the program behaves in consistency with the latter definition, and not the former. We show that despite this, our method automatically discharges the proof.

Here we want to prove that $\Psi \equiv allz(h_0, h, p_0, p)$ is a loop invariant. Formally,

$$allz(H_0, H, P_0, P), H[P+1] > 0 \models allz(H_0, \langle H, H[P+1], 0 \rangle, P_0, H[P+1]). \quad (Z.1)$$

For this assertion, constraint proof fails and coinduction (CO) is not applicable due to an empty set of assumed assertions. The algorithm applies left unfold (LU+1) using the definition of $allz$ obtaining two new obligations, of which one is:

$$allz(H_0, H_1, H_0[P_0+1], P), P_0 > 0, H_1[P+1] > 0 \models allz(H_0, \langle \langle H_1, P_0, 0 \rangle, H_1[P+1], 0 \rangle, P_0, H_1[P+1]). \quad (Z.2)$$

Now the algorithm applies (CO) using Z.1 as the hypothesis. As required by (CO), the algorithm spawns two sub-obligations, one of which proves

$$allz(H_0, H_1, H_0[P_0+1], P), P_0 > 0, H_1[P+1] > 0 \models allz(H_0, H_1, H_0[P_0+1], P), H_1[P+1] > 0$$

⁴ Note that we do not require that the list is acyclic ($L \neq R$).

This is established by eliminating the predicates using (CP) and applying constraint solving to the following assertion:

$$P_0 > 0, H_1[P + 1] > 0 \models H_0 = H_0, H_1 = H_1, H_0[P_0 + 1] = H_0[P_0 + 1], P = P, H_1[P + 1] > 0.$$

The second sub-obligation is

$$\begin{aligned} allz(H_0, \langle H_1, H_1[P + 1], 0 \rangle, H_0[P_0 + 1], H_1[P + 1]) &\models \\ allz(H_0, \langle \langle H_1, P_0, 0 \rangle, H_1[P + 1], 0 \rangle, P_0, H_1[P + 1]) & \end{aligned} \quad (Z.3)$$

Here again the application of constraint proof and coinduction fails, and the algorithm performs a right unfold using the second clause of *allz* resulting in

$$\begin{aligned} allz(H_0, \langle H_1, H_1[P + 1], 0 \rangle, H_0[P_0 + 1], H_1[P + 1]) &\models \\ allz(H_0, ?H_2, H_0[P_0 + 1], H_1[P + 1]), \langle \langle H_1, P_0, 0 \rangle, H_1[P + 1], 0 \rangle &= \langle ?H_2, P_0, 0 \rangle \end{aligned} \quad (Z.4)$$

By an application of (CP) proof rule, the algorithm removes the predicates and then solves the following implication by constraint solving (DP):

$$\begin{aligned} true \models H_0 = H_0, H_0[P_0 + 1] = H_0[P_0 + 1], \\ H_1[P + 1] = H_1[P + 1], \langle \langle H_1, P_0, 0 \rangle, H_1[P + 1], 0 \rangle &= \langle \langle H_1, H_1[P + 1], 0 \rangle, P_0, 0 \rangle. \end{aligned} \quad (Z.5)$$

6 Conclusion

We presented an automatic proof method which is based on unfolding recursive CLP definitions of user-specified program properties. The novel aspect is a principle of coinduction which is used in conjunction with a set of unfold rules in order to efficiently dispense recursive definitions into constraints involving integers and arrays. This principle is applied opportunistically and automatically over a dynamically generated set of potential induction hypotheses. As a result, we can now automatically discharge many useful proof obligations which previously could not be discharged without manual intervention. We finally demonstrated our method, assuming the use of a straightforward constraint solver over integers and integer arrays, to automatically prove two illustrative examples.

References

1. Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J., Giménez, E., Herbelin, H., Huet, G., Noz, C.M., Murthy, C., Parent, C., Paulin, C., Saïbi, A., Werner, B.: The Coq proof assistant reference manual—version v6.1. Technical Report 0203, INRIA (1997)
2. Boyer, R.S., Moore, J.S.: Proving theorems about LISP functions. *J. ACM* 22(1), 129–144 (1975)
3. Craciunescu, S.: Proving equivalence of CLP programs. In: Stuckey, P.J. (ed.) *ICLP 2002*. LNCS, vol. 2401. Springer, Heidelberg (2002)
4. Fribourg, L.: Automatic generation of simplification lemmas for inductive proofs. In: *ISLP 1991*, pp. 103–116. MIT Press, Cambridge (1991)
5. Gupta, G., Bansal, A., Min, R., Simon, L., Mallya, A.: Coinductive logic programming and its applications. In: Dahl, V., Niemelä, I. (eds.) *ICLP 2007*. LNCS, vol. 4670, pp. 27–44. Springer, Heidelberg (2007)
6. Harrison, J.: HOL light: A tutorial introduction. In: Srivas, M.K., Camilleri, A.J. (eds.) *FM-CAD 1996*. LNCS, vol. 1166, pp. 265–269. Springer, Heidelberg (1996)

7. Hsiang, J., Srivas, M.: Automatic inductive theorem proving using Prolog. *TCS* 54(1), 3–28 (1987)
8. Hsiang, J., Srivas, M.K.: A PROLOG framework for developing and reasoning about data types. In: Ehrig, H., Floyd, C., Nivat, M., Thatcher, J. (eds.) *TAPSOFT 1985*. LNCS, vol. 186, pp. 276–293. Springer, Heidelberg (1985)
9. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. *J. LP* 19/20, 503–581 (1994)
10. Jaffar, J., Santosa, A., Voicu, R.: A CLP proof method for timed automata. In: *25th RTSS*, pp. 175–186. IEEE Computer Society Press, Los Alamitos (2004)
11. Jaffar, J., Santosa, A.E., Voicu, R.: Recursive assertions for data structures, <http://www.comp.nus.edu.sg/~joxan/papers/rads.pdf>
12. Jaffar, J., Santosa, A.E., Voicu, R.: Relative safety. In: Emerson, E.A., Namjoshi, K.S. (eds.) *VMCAI 2006*. LNCS, vol. 3855, pp. 282–297. Springer, Heidelberg (2005)
13. Kanamori, T., Fujita, H.: Formulation of induction formulas in verification of Prolog programs. In: Siekmann, J.H. (ed.) *CADE 1986*. LNCS, vol. 230, pp. 281–299. Springer, Heidelberg (1986)
14. Kanamori, T., Seki, H.: Verification of Prolog programs using an extension of execution. In: Shapiro, E. (ed.) *ICLP 1986*. LNCS, vol. 225, pp. 475–489. Springer, Heidelberg (1986)
15. Manna, Z., Ness, S., Vuillemin, J.: Inductive methods for proving properties of programs. *Comm. ACM* 16(8), 491–502 (1973)
16. McCarthy, J.: Towards a mathematical science of computation. In: Popplewell, C.M. (ed.) *IFIP Congress 1962*. North-Holland, Amsterdam (1983)
17. McPeak, S., Necula, G.C.: Data structure specifications via local equality axioms. In: Etesami, K., Rajamani, S.K. (eds.) *CAV 2005*. LNCS, vol. 3576, pp. 476–490. Springer, Heidelberg (2005)
18. Mesnard, F., Hoarau, S., Maillard, A.: CLP(X) for automatically proving program properties. In: Baader, F., Schulz, K.U. (eds.) *1st FroCoS*. Applied Logic Series, vol. 3. Kluwer Academic Publishers, Dordrecht (1996)
19. Nguyen, H.H., David, C., Qin, S.C., 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)
20. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.K.: PVS: combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T.A. (eds.) *CAV 1996*. LNCS, vol. 1102, pp. 411–414. Springer, Heidelberg (1996)
21. Pettorossi, A., Proietti, M.: Synthesis and transformation of logic programs using unfold/fold proofs. *J. LP* 41(2–3), 197–230 (1999)
22. Roychoudhury, A., Kumar, K.N., Ramakrishnan, C.R., Ramakrishnan, I.V.: An unfold/fold transformation framework for definite logic programs. *ACM TOPLAS* 26(3), 464–509 (2004)
23. Roychoudhury, A., Ramakrishnan, C.R., Ramakrishnan, I.V., Smolka, S.A.: Tabulation-based induction proofs with application to automated verification. In: *1st TAPD*, April 1998, pp. 83–88 (1998), <http://pauillac.inria.fr/~clerger/tapd.html>
24. Sagonas, K., Swift, T., Warren, D.S., Freire, J., Rao, P., Cui, B., Johnson, E., de Castro, L., Dawson, S., Kifer, M.: *The XSB System Version 2.5 Volume 1: Programmer's Manual* (June 2003)
25. Stickel, M.E.: A Prolog technology theorem prover: A new exposition and implementation in prolog. *TCS* 104(1), 109–128 (1992)

Maintaining Generalized Arc Consistency on Ad Hoc r -Ary Constraints

Kenil C.K. Cheng and Roland H.C. Yap

School of Computing
National University of Singapore
{chengchi, ryap}@comp.nus.edu.sg

Abstract. In many real-life problems, constraints are explicitly defined as a set of solutions. This ad hoc (table) representation uses exponential memory and makes support checking (for enforcing GAC) difficult. In this paper, we address both problems simultaneously by representing an ad hoc constraint with a multi-valued decision diagram (MDD), a memory efficient data structure that supports fast support search. We explain how to convert a table constraint into an MDD constraint and how to maintain GAC on the MDD constraint. Thanks to a sparse set data structure, our MDD-based GAC algorithm, `mddc`, achieves full incrementality in constant time. Our experiments on structured problems, car sequencing and still-life, show that `mddc` is a fast GAC algorithm for ad hoc constraints. It can replace a Boolean sequence constraint [1], and scales up well for structural MDD constraints with 208 variables and 340984 nodes. We also show why it is possible for `mddc` to be faster than the state-of-the-art generic GAC algorithms in [2,3,4]. Its efficiency on non-structural ad hoc constraints is justified empirically.

1 Introduction

An ad hoc constraint is an explicitly defined constraint (arity > 2), usually represented as a set of solutions stored in a *table* (e.g., sequence of tuples). Naturally, a table or list takes exponential space (in the arity) in the worst case; furthermore, support checking (`seekSupport`) is also exponential. Some improvements are indexing [2,4,5] and binary search [3]. The trade-off is that additional data structures inevitably require extra memory and manipulation effort.

Indexing may speed up `seekSupport` because it only has to visit a (small) subset of the solutions. This is done by linking related solutions with pointers. Figs. 1a, 1b and 1c depict three ways to index a constraint with solutions $\theta_1, \dots, \theta_9$. In Fig. 1a, solutions with the same assignment (x_i, a) are chained [5] — each (x_i, a) has its own table that contains all its supports. In this example, `seekSupport` checks at most 4 out of 9 solutions to find a support for $(x_4, 0)$.

In another indexing scheme (called *hologram* here) by Lhomme and Régin [4], for every (x_i, a) in a solution θ_k , there are d pointers, where d is the size of $\text{dom}(x_i)$. Each pointer points to the next solution $\theta_{k'}$ ($k' > k$) where $(x_i, a') \in \theta_{k'}$ and $a' \in \text{dom}(x_i)$. Fig. 1b shows the three pointers from θ_1 to θ_2 , θ_5 and θ_9 ,

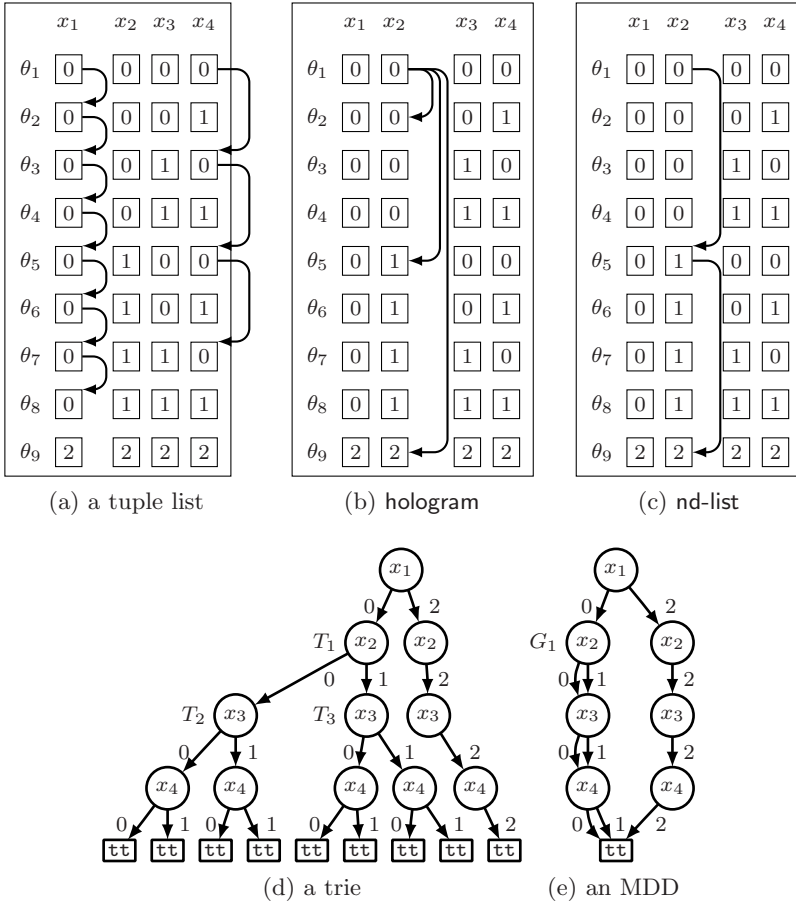


Fig. 1. Different indexing schemes and representations of the same ad hoc constraint. (a) (b) (c) Not all pointers are drawn. (d) A trie and (e) an MDD representing the same constraint. Each non-terminal node v is labeled with a variable x_i . An outgoing edge of v with label a depicts an assignment (x_i, a) . Each path from the root to the t-terminal (**tt**) corresponds to a solution.

corresponding to the assignments $(x_2, 0)$, $(x_2, 1)$ and $(x_2, 2)$ respectively. Now let θ_1 be the support for $(x_1, 0)$ when 2 is assigned to x_2 . When **seekSupport** is finding a new support for $(x_1, 0)$, it jumps to θ_9 directly via the pointers, and concludes $(x_1, 0)$ has no support. Without indexing, the entire table will be scanned.

The last indexing method (called **nd-list** in this paper) was suggested by Gent et al. [2]. Here, every (x_i, a) in a solution θ_k has a pointer to the next solution $\theta_{k'}$ ($k' > k$) where $(x_i, a') \in \theta_{k'}$ and $a' \neq a$. See Fig. 1c for an example. To find a new support for $(x_1, 0)$ after x_2 is assigned with 2, **seekSupport** first jumps from θ_1 to θ_5 . Since $(x_2, 2) \notin \theta_5$, it jumps again and finally stops at θ_9 .

As memory limits can be more important than time in large problems, we want a data structure more compact than a table, but supports efficient support

checking. One approach is to store all solutions in a *trie*, which is often smaller than a table due to the prefix sharing of similar solutions (see Fig. 1d). To find a support for an assignment (x_i, a) , we traverse the trie recursively. For example, suppose we seek a support for $(x_1, 0)$ when $x_2 = 2$. From the root of the trie, we go down to T_1 because of $(x_1, 0)$. Now T_2 which requires $x_2 = 0$ is excluded, so is T_3 . The search then can stop early. This idea was used by [2] in their GAC algorithm (we call *mtrie*). For a r -ary ad hoc constraint, *mtrie* generates r tries, all represent the same solutions but each is rooted at a different variable. To seek a support for (x_i, a) , *mtrie* looks up the trie rooted at x_i . Multiple tries avoid the scenario that, if there is only one trie, *mtrie* may need to traverse the entire trie to find a support for (x_i, a) , if x_i is at the bottom of the trie.

Sometimes multiple tries do not help. Consider the trie constraint in Fig. 1d. When x_4 is assigned with 2, *mtrie* seeks a support for $(x_1, 0)$ in exponential time as it has to visit every $(x_4, 0)$ and $(x_4, 1)$ at the bottom of T_1 . This is because a trie may contain identical sub-tries (T_2 and T_3 in Fig. 1d). Merging these sub-tries, we obtain a directed acyclic graph (DAG), called a *multi-valued decision diagram* (MDD) [6]. Fig. 1e depicts the MDD obtained from the trie in Fig. 1d. Note that an MDD can be exponentially smaller than a trie (e.g., T_1 versus G_1).

GAC using DAGs (similar to MDD) to represent ad hoc constraints were introduced in *case* [7] and *bddc* [8]. To enforce GAC, the DAG is traversed recursively; the worst case time complexity is linear to the number of edges in the DAG. When GAC is maintained during search, we can improve the average runtime by taking *incrementality* into account: an inconsistent constraint remains inconsistent when more variables are assigned, or equivalently, an inconsistent and pruned sub-DAG should remain pruned. A simple and fast implementation for this is not trivial. The *case* algorithm is only partially incremental — all pruned sub-DAGs have to be reset when the solver backtracks — due to the limitation of time-stamping that *case* uses to record the status of a sub-DAG. The *bddc* algorithm is incremental, but its implementation is inefficient: pruned sub-DAGs are kept in a stack of bit vectors, so undoing changes upon backtracking takes linear time, and memory usage is high for large ad hoc constraints.

In Section 3, we present an MDD-based GAC algorithm, *mddc*, that achieves full incrementality in constant time. The key is a sparse set data structure [9], which is more suitable for *mddc* than bit vectors and time-stamping. We show why *mddc* can be faster than the above mentioned GAC algorithms [2,3,4]. Our experiments (Section 4) on structured problems, car sequencing and still-life, show that *mddc* is a fast GAC algorithm for ad hoc constraints. It can replace a Boolean sequence constraint [1], and scales up well for large structural MDD constraints. Our results suggest that ad hoc constraints can be competitive or even outperform problem-specific global constraints, and for this *mddc* can be useful in modeling new global constraints: *mddc* is generic and building MDDs is easier than inventing propagation algorithms. The efficiency of *mddc* on non-structural ad hoc constraints is justified empirically: on our random benchmarks, *mddc* is always faster than *mtrie*, and 2–3 orders of magnitude faster than *hologram*.

2 Background

A *constraint satisfaction problem* (CSP) $\mathcal{P} = (X, \mathcal{C})$ consists of a finite set X of variables and a finite set $\mathcal{C}$ of constraints. Every *variable* $x_i \in X$ can only take values from its *domain* $\text{dom}(x_i)$ which is a finite set of integers. An *assignment* (x_i, a) denotes $x_i = a$. A r -ary *constraint* $C \in \mathcal{C}$ on an ordered set of r distinct variables $x_1, \dots, x_r$ is a subset of the Cartesian product $\prod_{i=1}^r \text{dom}(x_i)$ that restricts the values the variables in C can take simultaneously. The *arity* of C is r and the *scope* is $\text{var}(C)$. Sometimes we write $C(x_1, \dots, x_r)$ to make the scope explicit. A set of assignments $\theta = \{(x_1, a_1), \dots, (x_r, a_r)\}$ *satisfies* C , and is a *solution* of C , iff $\theta \in C$. Denote $|C|$ the number of solutions of C . We say *True* is the trivially true constraint. Solving a CSP requires finding a value for each variable from its domain so that all constraints are satisfied. Two constraints C_1 and C_2 are *equivalent*, written as $C_1 \equiv C_2$, iff $\theta \in C_1 \iff \theta \in C_2$.

Consider a CSP $\mathcal{P} = (X, \mathcal{C})$. An assignment (x_i, a) is *generalized arc consistent* (GAC) [10] iff for every constraint $C \in \mathcal{C}$ such that $x_i \in \text{var}(C)$, there is a solution θ of C where $(x_i, a) \in \theta$ and $a \in \text{dom}(x_i)$ for every $(x_i, a) \in \theta$. This solution is called a *support* for (x_i, a) in C . A variable $x_i \in X$ is GAC iff (x_i, a) is GAC for every $a \in \text{dom}(x_i)$. A constraint is GAC iff every variable in its scope is GAC. A CSP is GAC iff every constraint in $\mathcal{C}$ is GAC.

A *multi-valued decision diagram* (MDD) [6] is either the *t-terminal* (**tt**) that denotes *True*, or a directed acyclic graph G rooted at a non-terminal node of the form $\text{root}(G) = \text{mdd}(x_i, \{a_1/G_1, \dots, a_d/G_d\})$ where $G_1, \dots, G_d$ are MDDs and $a_1, \dots, a_d$ are distinct integers. Here we abuse the notation by writing G instead of $\text{root}(G)$. Semantically G represents the *MDD constraint*¹

$$\Phi(G) \equiv \bigvee_{k=1}^d (x_i = a_k \wedge \Phi(G_k)). \quad (1)$$

A *binary decision diagram* (BDD) [11] is an MDD in which every non-terminal node has at most two branches, $0/G_0$ and $1/G_1$, where G_0 and G_1 are BDDs. A *BDD constraint* is a Boolean MDD constraint.

We can transform a r -ary ad hoc constraint C to an MDD in two steps. First, we build a trie that represents the $|C|$ solutions in $O(r \cdot |C|)$ time. To add a solution, we traverse the trie from its root and make new edges accordingly. The trie will have $O(r \cdot |C|)$ edges. Second, we combine identical sub-tries in a depth-first, bottom-up manner. The resultant MDD is so-called *reduced*. Fig. 2 shows the algorithm `mddReduce`.² By construction, two MDDs G and G' are identical iff they have the same children. As a result, representing each MDD node as an array of integers (indexes), line 1 is a typical dictionary lookup, which takes

¹ When an MDD node has many identical children (G_k 's), it may be advantageous to replace $x_i = a_k$ with $x_i \in J_k$ (an interval), as in the implementation of `case`. We have chosen values (a_k 's) for the sake of simplicity.

² It is different from Bryant's original reduction algorithm on BDD, which applies a *breadth-first*, bottom-up transformation.

```

mddReduce( $T$ ) //  $T$  is a trie (a tree-like MDD)
// the set  $\mathcal{G}$  of created MDDs is initially empty
begin
  if  $T$  is the  $t$ -terminal then return tt
  // let  $T = \text{mdd}(x_i, \{a_1/T_1, \dots, a_d/T_d\})$ 
   $B := \emptyset$ 
  for  $k := 1$  to  $d$  do
     $G_k := \text{mddReduce}(T_k)$ 
     $B := B \cup \{a_k/G_k\}$ 
   $G := \text{mdd}(x_i, B)$ 
  1 if  $\exists G' \in \mathcal{G}$  such that  $G'$  is identical to  $G$  then
    return  $G'$  // sharing: reuse existing MDDs
  else
     $\mathcal{G} := \mathcal{G} \cup \{G\}$ 
    return  $G$ 
end

```

Fig. 2. mddReduce

$O(d)$ time. The time complexity of `mddReduce` is $O(r \cdot |C|)$ as the entire trie is traversed exactly once. This is optimal since it takes the same amount of time to input the $|C|$ solutions from a file. As a remark, a static order on the MDD variables is sufficient because no new MDD is constructed during search.

3 Maintaining GAC on an MDD Constraint

Fig. 3 shows the `mddc` algorithm which enforces GAC on an MDD constraint $\Phi(G)(x_1, \dots, x_r)$. The algorithm is coarse grained: maintaining GAC during search works on one (MDD) constraint at a time. For each x_i , `mddc` keeps a set S_i of values in the domain of x_i which have no support (yet). The function `mddcSeekSupports` traverses G recursively and updates $S_1, \dots, S_r$ on the fly. Line 3 then removes for each variable all values that have no support from its domain.

To simplify the implementation of `mddcSeekSupports`, we assume that for every path from the root of G to the t -terminal, every variable in the scope of $\Phi(G)$ appears exactly once and in the same (natural) order. In other words, each path to the t -terminal corresponds to exactly one solution of the constraint.

The function `mddcSeekSupports` works as follows: If G is the t -terminal `tt` *YES* is returned. Otherwise, let $G = \text{mdd}(x_i, \{a_1/G_1, \dots, a_d/G_d\})$, which represents the MDD constraint (1). Now if `mddcSeekSupports`(G_k) returns *YES* (line 7), $\Phi(G_k)$ is satisfiable and (x_i, a_k) has at least one support. We thus remove a_k from S_i . Line 8 terminates the iteration as soon as $\Phi(G)$ is GAC (the guard is $O(1)$ using an incremental counter [8]). At last the function returns *res*, which is *YES* if $\Phi(G)$ is satisfiable and *NO* otherwise.

Proposition 1. *When `mddcSeekSupports` terminates, the value $a_k \in S_i$ iff the assignment (x_i, a_k) has no support.*

```

mddc( $G$ ) // MDD constraint  $\Phi(G)(x_1, \dots, x_r)$ 
begin
   $\mathcal{G}^{YES} := \emptyset$ 
  2  restore( $\mathcal{G}^{NO}$ )
    for  $i := 1$  to  $r$  do  $S_i := \text{dom}(x_i)$  // values that have no support yet
    mddcSeekSupports( $G$ ) // update  $S_1, \dots, S_r$ 
  3  for  $i := 1$  to  $r$  do  $\text{dom}(x_i) := \text{dom}(x_i) \setminus S_i$ 
end
mddcSeekSupports( $G$ ) // recursive:  $\Phi(G)(x_i, \dots, x_r)$ 
begin
  if  $G = \text{tt}$  then return YES
  4  if  $G \in \mathcal{G}^{YES}$  then return YES // visited and consistent
  5  if  $G \in \mathcal{G}^{NO}$  then return NO // pruned
  // let  $G = \text{mdd}(x_i, \{a_1/G_1, \dots, a_d/G_d\})$ 
  res := NO
  6  for  $k := 1$  to  $d$  do
    if  $a_k \in \text{dom}(x_i)$  then
      7  if mddcSeekSupports( $G_k$ ) = YES then
        res := YES
         $S_i := S_i \setminus \{a_k\}$ 
        8  if  $\forall i' \geq i : S_{i'} = \emptyset$  then break //  $\Phi(G)(x_i, \dots, x_r)$  is GAC
      end if
    end if
  end for
  9   $\mathcal{G}^{res} := \mathcal{G}^{res} \cup \{G\}$ 
  return res
end

```

Fig. 3. mddc and mddcSeekSupports

Lines 4 and 5 guarantee mddcSeekSupports traverses every sub-MDD at most once. A sub-MDD G is in the set $\mathcal{G}^{YES}$ iff $\Phi(G)$ is satisfiable, and in the set $\mathcal{G}^{NO}$ iff $\Phi(G)$ is unsatisfiable. Each visited sub-MDD is added to the appropriate set according to the value of res (line 9). As an unsatisfiable constraint remains unsatisfiable when more variables are assigned, mddc achieves incrementality by maintaining a stack of $\mathcal{G}_1^{NO}, \dots, \mathcal{G}_t^{NO}$ during search, where $\mathcal{G}_j^{NO}$ ($1 \leq j \leq t$) is the accumulated set of pruned MDD nodes at the search state j . Later, when the search backtracks to j , the procedure restore (line 2) will reset $\mathcal{G}^{NO}$ to $\mathcal{G}_j^{NO}$.

To make restore efficient, we wish to do three tasks on $\mathcal{G}^{YES}$ and $\mathcal{G}^{NO}$ in constant time: (1) check whether an MDD is in the set, (2) insert an MDD to the set, and (3) undo all insertions upon backtracking. The sparse set data structure by Briggs and Torczon [9] meets our requirements. To represent a set S of at most e elements (0 to $e - 1$), they use two arrays $S.dense$ and $S.sparse$ of size e , and a counter $S.members$ that keeps the number of elements in S . Initially, $S.members = 0$. Fig. 4 shows how to do tasks 1 and 2 in constant time. For task 3, we stack the values of $S.members$ and reset them accordingly. Fig. 5 demonstrates the operations. Since every MDD can be uniquely identified with its root, we can always map MDD nodes to unique integers.

```

isMember( $S, k$ ) // is  $k$  an element of  $S$ ?
begin
   $a := S.sparse[k]$ 
  if  $0 \leq a < S.members$  and  $S.dense[a] = k$  then
    return YES
  else
    return NO
end

addMember( $S, k$ ) // add  $k$  to  $S$ 
begin
   $a := S.sparse[k]$ 
   $b := S.members$ 
  if  $a \geq b$  or  $S.dense[a] \neq k$  then
     $S.sparse[k] := b$ 
     $S.dense[b] := k$ 
     $S.members := b + 1$ 
end

```

Fig. 4. Sparse set operations [9]

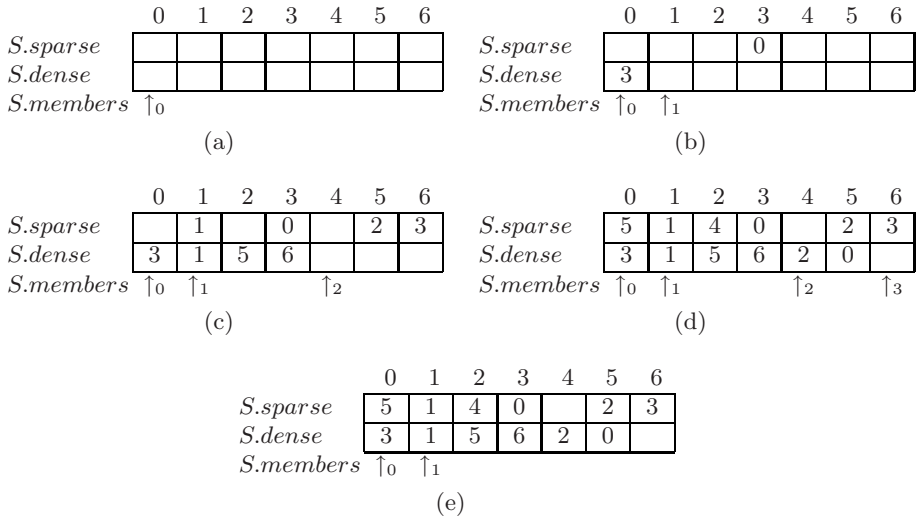


Fig. 5. A demonstration of the sparse set operations. (a) After the initialization of S . Note that the set operations work correctly regardless the initial values (not shown) in the arrays. For incremental trailing, $\uparrow_t$ denotes the number of elements ($S.members$) at time t . (b) After adding 3 at $t = 1$. (c) After adding 1, 5 and 6 at $t = 2$. (d) After adding 2 and 0 at $t = 3$. (e) After backtracking to $t = 1$. Only $S.members$ is restored; the values in the arrays are unchanged. As an example, 5 is no longer an element of S because $S.sparse[5] = 2 > 1 = S.members$. When S is incrementally maintained within depth-first search, $\uparrow_t$'s can be kept in the program heap space so that the undo operation takes constant time during backtracking.

$\mathcal{G}^{NO}$ in `bddc` [8] was implemented with a stack of bit vectors. However, for large MDD constraints, copying and clearing bit vectors (task 3) become expensive (compared with the constant time sparse set operations). The memory requirement for bit vectors is another issue: Given a r -ary MDD constraint with m MDD nodes, there are at most $t = 1 + r(d - 1)$ bit vectors (m bits each) for $\mathcal{G}^{NO}$, where d is the size of the largest domain of the variables in the constraint. This is because `mddc` is executed when it is initialized and every time the domain of a relevant variable is modified. Bit vectors thus require at most $mt/8$ bytes. Meanwhile, a sparse set comprises two integer arrays of size m and a stack of t integer counters, so it needs $4(2m + t)$ bytes (4 bytes per integer). The sparse set uses less memory than a stack of bit vectors when

$$\frac{64}{1 + r(d - 1)} < 1 - \frac{32}{m}.$$

Under the reasonable assumption that $m \gg 32$, the condition becomes $r(d - 1) > 63$. So one should use a sparse set when an MDD constraint has many variables and their domains are large.

Provided that the early-cutoff optimization (line 8) is implemented, time-stamping cannot achieve (in constant time) the same level of incrementality of $\mathcal{G}^{NO}$: when the solver backtracks, one has to reset all pruned MDD nodes. This is because `mddcSeekSupports` may not visit every MDD node and update its time-stamp in a lazy manner. Hence we cannot tell if a previously pruned node is still pruned using a condition such as “time-stamp $\leq$ current time.”

As a result, `case`, which implements both time-stamping and early-cutoff, resets all nodes (visited or pruned) when the solver backtracks [7]. In contrast, `mddc` is able to maintain the pruned nodes *across* backtracking due to the stack-like sparse set. This difference is crucial in terms of runtime efficiency, when the MDD is large and the problem is difficult (many backtracks).

The overall memory requirement for `mddc` (using sparse set) is calculated as follows: Suppose there are h MDDs and k MDD constraints ($h \leq k$ since equivalent MDD constraints can share the same MDD). Let m be the number of nodes in the largest MDD. Let d be the size of the largest domain among all variables in the constraints. We store the nodes in an MDD in an $m \times d$ integer array. If an integer takes 4 bytes, h MDDs require $O(hmd)$ bytes. Since at most one MDD is traversed in each call of `mddc` and we assume there is no parallel execution, we need only one $\mathcal{G}^{YES}$ of size $O(m)$. On the other hand, each MDD constraint has its own copy of $\mathcal{G}^{NO}$ (of size $O(m)$). In total, for k MDD constraints, the memory requirement for $\mathcal{G}^{NO}$ is $O(km)$ bytes. The memory for trailing the number of elements in $\mathcal{G}^{NO}$ at each choice point during search is negligible. The overall memory requirement for `mddc` is therefore $O((hd + k)m)$ bytes.

From the description of `mddc`, it is not difficult to see the following:

Proposition 2. *Given an MDD G with e edges, there are at most e recursive calls of `mddcSeekSupports` in each run of `mddc(G)`.*

Corollary 1. *Given an MDD constraint $\Phi(G)(x_1, \dots, x_r)$ where the MDD G has e edges, mddc enforces GAC on $\Phi(G)$ in $O(e)$ time.*

Finally, we analyze the potential runtime improvement by mddc over the GAC algorithms in [2,3,4]. We consider the ad hoc constraint

$$EG_{d,r}(x_1, \dots, x_r) \equiv \left[x_1 = 0 \wedge \bigwedge_{i=2}^r x_i \in \{0, 1\} \right] \vee \left[\bigvee_{j=2}^{d-1} \bigwedge_{i=1}^r x_i = j \right].$$

The domain of each variable is $\{0, \dots, d-1\}$. Fig. 1 depicts various representations of $EG_{3,4}(x_1, \dots, x_4)$. In general, $EG_{d,r}$ has $2^{r-1} + d - 2$ solutions, and its MDD representation has $1 + (r-1)(d-1)$ nodes and $2r-1 + r(d-2)$ edges. Without loss of generality, we assume when $EG_{d,r}$ is represented as a table, the solutions are sorted lexicographically.

Property 1. Enforcing GAC on $EG_{d,r}$ using mddc takes $O(rd)$ time.

Property 2. To enforce GAC on $EG_{d,r}$, binary search [3] takes $O(r^3 + r^2d)$ time.

Proof. Suppose there is a sub-table of supports for every assignment (x_i, a) (see Fig. 1a). To find a support for any of the $2r-1$ assignments where $a \in \{0, 1\}$, binary search takes $O(r \cdot \log(2^{r-1})) = O(r^2)$ time. But for each of the remaining $r(d-2)$ assignments where $a \geq 2$, only $O(r)$ comparisons are required, because the corresponding sub-table contains only one tuple.

Property 3. To seek a support for (x_i, a) in $EG_{d,r}$, nd-list [2] takes $O(r \cdot 2^r)$ time.

Proof. Let $x_r = 2$. To seek a support for $(x_1, 0)$, one iterates over the $O(2^{r-1})$ solutions with $x_1 = 0$ because the next-different pointer of (x_r, a) in any solution of $EG_{d,r}$ always points to the immediately next solution (for any two adjacent solutions $\theta, \theta' \in EG_{d,r}$: $(x_r, a) \in \theta$, $(x_r, a') \in \theta'$ and $a \neq a'$; see Fig. 1c). Checking if a support is valid takes $O(r)$ time.

Property 4. To seek a support for (x_i, a) in $EG_{d,r}$ using mtrie [2] is $O(2^r)$.

Proof. Let $x_r = 2$. To seek a support for $(x_1, 0)$, one completely traverses the trie representing the $O(2^{r-1})$ solutions with $x_1 = 0$ (x_r is at the bottom of the trie; see Fig. 1d). The trie is a complete binary tree and has $O(2^r)$ edges.

Property 5. To enforce GAC on $EG_{d,r}$, hologram [4] can be d times slower than mddc.

Proof. Let $x_1 = 2$. Enforcing GAC using mddc takes $O(r)$ time because there is exactly one path (no branching) in the MDD such that $x_1 = 2$ holds (x_1 is at the root of the MDD; see Fig. 1e). On the other hand, to seek a support for (x_i, a) , where $a \neq 2$, using hologram, although in constant time³ one can jump to $\theta = \{(x_1, 2), \dots, (x_r, 2)\}$ and confirm there is no support for (x_i, a) , this process is repeated $(r-1)(d-1)$ times for every (x_i, a) where $i \neq 1$ and $a \neq 2$.

³ Here d indexing pointers are used for every (x_i, a) . When only one pointer is used, the memory requirement drops but the jump operation takes $O(d)$ time [12].

Our analysis reveals that the memory compactness and the runtime efficiency of `mddc` go hand and hand — if the MDD is small, `mddc` can be drastically faster than GAC algorithms that do not exploit the semantics of the ad hoc constraint.

4 Experimental Results

We implemented `mddc` in Gecode 2.1.1 (<http://www.gecode.org>). All experiments were run on a 2 GHz Core 2 Duo MacBook with 1 GB RAM. Our benchmarks include structured problems (the still-life and car sequencing problem) and random CSPs. In the two structural problems, the arity of an MDD constraint ranges from 27 to 400 and the number of solutions grows exponentially with the arity. Therefore, the scalability of `mddc` is crucial. We remark that `mtree` and `nd-list` could not be tested on the structured problems which are too large.

There are three implementations of `mddc`: `mddc(sp)` uses sparse sets, `mddc(bv)` uses bit vectors and `mddc(ts)` uses time-stamping. Note that `mddc(ts)` is strictly non-incremental (the time-stamp is incremented at every call of `mddc`) and hence a base-line reference.

We also tested `regular` [13] (available in Gecode), which maintains GAC on a constraint represented as a deterministic finite state automata (DFA), via explicit and incremental modification of the graph. It works for MDD constraints because MDDs are virtually DFAs with one final state (`tt`). Unfortunately, the graph-updating mechanism in `regular` is expensive in memory and computation. As we will see, `regular` scales up poorly and is always 1 to 3 orders of magnitude slower than `mddc`, although both use exactly the same MDDs.

4.1 The Still Life Problem

An $n \times n$ *still-life problem* has n^2 Boolean variables $x_{1,1}, \dots, x_{n,n}$, $4n$ *border constraints* ($x_{i-1,k} + x_{i,k} + x_{i+1,k} < 3$, $x_{k,j-1} + x_{k,j} + x_{k,j+1} < 3$, where $k \in \{1, n\}$, $1 \leq i, j \leq n$), and n *super-row density (SRD) constraints* [14] of the form

$$SRD_i \equiv [f_i = \sum_{d=-1}^1 \sum_{j=1}^n x_{i+d,j}] \wedge \bigwedge_{j=1}^n [(x_{i,j} = 0 \vee s_{i,j} \in \{2, 3\}) \wedge (x_{i,j} = 1 \vee s_{i,j} \neq 3)]$$

where

$$s_{i,j} = -x_{i,j} + \sum_{p=i-1}^{i+1} \sum_{q=j-1}^{j+1} x_{p,q}.$$

Note that the variable f_i has a domain $\{0, \dots, 2n\}$. The objective is to maximize

$$z = \sum_{i=1}^n \sum_{j=1}^n x_{i,j} = \sum_{k=1}^{n/3} f_{3k-1}.$$

In this experiment n is always a multiple of 3. The still-life problem is a good benchmark for ad hoc constraints because, on the one hand, an SRD constraint

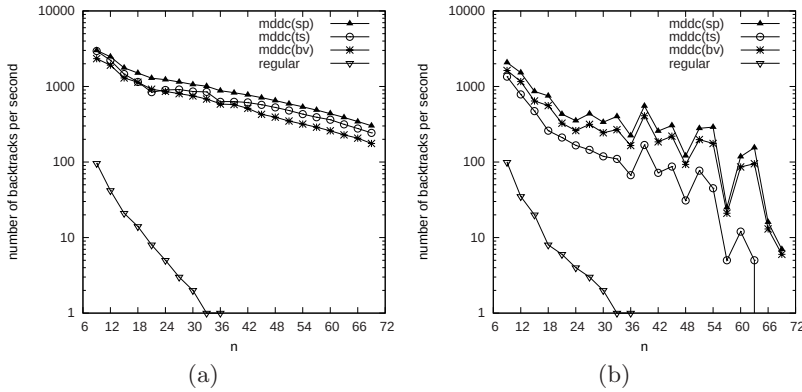


Fig. 6. Results on the still-life problem. The variables are assigned (a) top-down and (b) bottom-up, with respect to the MDD variable order.

can be constructed easily via BDD operations; on the other hand, its arity grows fast ($3n+1$) and its number of solutions grows exponentially. Indeed, even with a good BDD variable order [14], we found empirically that the number of nodes in the MDD (because of f_i) representation grows roughly as $32.73 \cdot 0.9121^n \cdot 6.486\sqrt{n}$.

Here the memory factor becomes significant: when $n = 69$ (340984 nodes in an MDD), `mddc(sp)` uses 162 MB, versus 287 MB by `mddc(bv)`. Gencode returns a memory allocation error when we use `regular` for the instance $n = 39$ (102904 nodes in an MDD, `regular` consumes > 600 MB, `mddc` uses 28 MB).

For each implementation of `mddc` and `regular`, two searches were run with different labeling orders: one corresponds to the top-down MDD variable order and the other to the bottom-up. Both lead to the same search space because of the symmetries in this problem (think flipping the SRD constraints vertically).

Fig. 6 plots the number of backtracks per second (bt/s) against n when various implementations of `mddc`, or `regular`, is used to enforce GAC on the SRD constraints. Bt/s is a valid measure of the speed (the more bt/s the faster) of different implementations of `mddc` because the search space is the same and no instance was solved by either implementation within one minute (time-out).

We can see although `mddc(ts)` is non-incremental, it is faster than `mddc(bv)` when the labeling is top-down. This is because the MDD is reduced implicitly, making incrementality unnecessary and copying bit vectors adds pure overhead. On the contrary, under the bottom-up labeling, the lower part of the MDD is pruned first and incrementality becomes critical. As a result, `mddc(ts)` performs just 24 backtracks in one minute when n reaches 66. Finally, sparse set is the most robust and is always the fastest.

4.2 The Car Sequencing Problem

In a car factory, different classes of cars are assembled along an assembly line. Each class has its own options (e.g., air-conditioning) and each option has its

own demands on resources (e.g., manpower). Since the stations on the assembly line can only handle a limited number of cars within any time interval (e.g., 2 out of 5 cars passing along the line), the cars requiring the same options must not be clustered. Solving the *car sequencing problem* is to arrange the cars in a sequence such that the capacity of each station is not exceeded.

Let o_i be a Boolean variable which is 1 iff the i -th car along the assembly line needs the option o . The capacities of the stations can be modeled as a sequence constraint [1]:

$$Seq(o_1, \dots, o_n) \equiv \bigwedge_{i=0}^{n-q} l \leq \sum_{j=1}^q o_{i+j} \leq u$$

which means, along an assembly line of length n , there are at least l and at most u out of q consecutive cars requiring the option o . Hoeve et al. [15] suggested to transform the constraint into a DFA and use **regular** to enforce GAC on it. We instead use an ad hoc constraint that represents *Seq* as a BDD, so that **mddc** can be used. The construction is straightforward using standard BDD operations.

All problems are from CSPLib (<http://www.csplib.org/prob/prob001>). There are 9 instances with $n = 100$ cars, 70 with 200 cars (excluding the 10 instances that can be solved in one minute), 10 with 300 and 10 with 400 cars. Respectively, the sequence constraints have 1617 (mean), 11949, 10548 and 24726 BDD nodes.

The model is standard and our goal is to evaluate the performances of **mddc** and **regular**, rather than to study how to solve car sequencing quickly. Indeed, none of the 99 instances can be solved in one minute (time-out). During search the classes of the cars are assigned to the stations lexicographically. By symmetry, labeling the stations from 1 to n (top-down BDD variable order) or from n to 1 (bottom-up BDD variable order) results in the same search space. Again this allows us to study the best and the worst-case scenarios for **mddc**.

Table 1 gives the mean bt/s and e/c, the number of calls of **mddcSeekSupports** per **mddc** run. The tiny difference in e/c between **mddc(sp)** and **mddc(bv)** is due to the different amount of **mddc** executions (the search is always timed out).

As expected, incrementality is more useful when the labeling is bottom-up — it cuts 75 to 89 percent of the **mddcSeekSupports** calls by **mddc(ts)** and makes **mddc** 3.5 times faster when sparse set is used. Another observation is, although the BDDs are small, bit vector is about 15–29% slower than sparse set.

Our results confirm the scalability of **mddc** for structural ad hoc constraints. This means **mddc** can readily substitute for global constraints with a compact MDD representation. Since MDD construction is relatively straightforward, **mddc** will be useful in prototyping global constraints. Clearly, table-based GAC algorithms do not have this flexibility.

4.3 Random Problems

In this final experiment, we compare **mddc** with **mtree** and **nd-list** in Minion 0.4.1 (<http://minion.sourceforge.net>), which are not available in Gecode. We do not report results on **hologram** (also available in Minion) because it performs poorly:

Table 1. Results on the car sequencing problem. The column “bt/s” gives the number of backtracks per second (the more the faster). The column “e/c” shows the number of `mddcSeekSupports` calls per `mddc` run. The percentage gives the relative performance of `mddc`(`{sp,bv}`) over `mddc`(`ts`).

n	mddc(ts)		mddc(sp)		mddc(bv)		regular
	bt/s	e/c	bt/s	e/c	bt/s	e/c	bt/s
100	3645	346	3519 (-3.5%)	338 (-2.1%)	3061 (-16.0%)	337 (-2.4%)	190
200	3045	476	2943 (-3.4%)	454 (-4.6%)	2375 (-22.0%)	455 (-4.3%)	26
300	1023	1321	973 (-4.9%)	1317 (-0.3%)	822 (-19.6%)	1320 (-0.1%)	6
400	829	2705	819 (-1.3%)	2026 (-25.1%)	667 (-19.5%)	2034 (-24.8%)	2

(a) Top-down labeling

n	mddc(ts)		mddc(sp)		mddc(bv)		regular
	bt/s	e/c	bt/s	e/c	bt/s	e/c	bt/s
100	1902	2015	3180 (+67%)	492 (-75.6%)	2707 (+42%)	493 (-75.5%)	108
200	1217	4328	2584 (+112%)	787 (-81.8%)	2042 (+68%)	790 (-81.7%)	7
300	288	15951	798 (+178%)	2287 (-85.7%)	646 (+125%)	2300 (-85.6%)	5
400	135	39167	611 (+351%)	4277 (-89.1%)	475 (+251%)	4337 (-88.9%)	1

(b) Bottom-up labeling

for only 6 out of the 80 instances `hologram` finished in 5 minutes (for these 6, `mddc` is 2 to 3 orders of magnitude faster); runs using other algorithms always terminated within the time limit. We did not test the binary-search-based algorithm [3] (no longer supported by Minion) as it was shown almost always slower than `mtree` (see Fig. 10 in [2]). The Minion solvers were built with `make_table_minions.sh` in the Minion distribution.

A problem in the benchmark $\langle n, d, c, r, t \rangle$ has n variables whose domain is $\{1, \dots, d\}$, and c random r -ary constraints with t solutions. The variables in the scope of each constraint are randomly ordered. There are totally 80 instances in $\langle 22, 5, \{5, 6\}, 10, \{10K, 20K\} \rangle$, $\langle 25, 10, \{17, 18\}, 5, \{3K, 6K\} \rangle$, $\langle 25, 20, \{10, 11\}, 5, \{3K, 6K\} \rangle$, and $\langle 25, 5, 3, 20, \{5K, 10K\} \rangle$. These benchmarks, in terms of arity and domain size, are compatible to, and sometimes larger than those in [2,3,4].

During search the variable that appears in the most constraints is selected first, and the smallest value in a domain is tried first. Both orders are static and the backtracks are about the same.⁴

Fig. 7 plots the solving time by `mddc(sp)` against that by `mtree` or `nd-list`. On average, `mddc(sp)` is 2.9 times faster than `mtree` and 8.3 times faster than `nd-list`. The medians are 2.2 and 7.5 respectively. The results are good, provided that the constraints are random and there is little MDD sharing (on average, an MDD has 3 times less nodes than the corresponding trie). Indeed, random constraints are challenging for any GAC algorithm that exploits hidden structure, and our results reveal the worst case performance of `mddc`. Finally, `mddc(sp)` is 20% faster

⁴ The mean absolute difference in the numbers of backtracks by Gecode and Minion is 3 and median is 1, so we assume both solvers traverse the same search space.

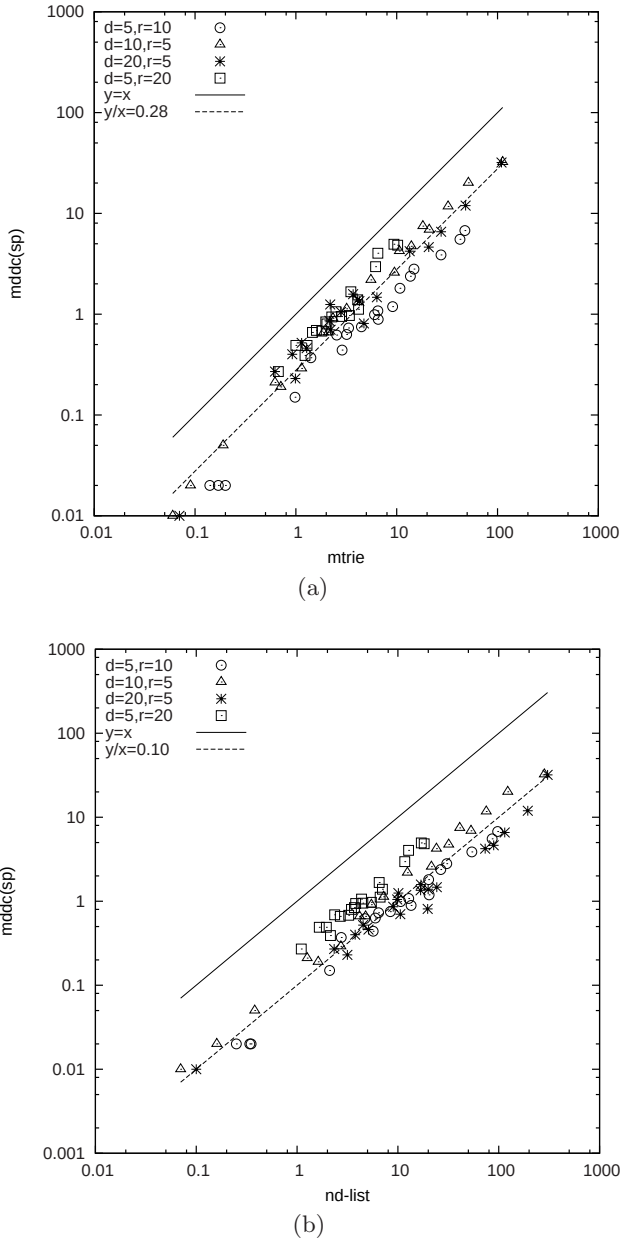


Fig. 7. Results on random CSPs. The X and Y-axes are the solving times (in seconds) by the respective algorithms. The line $y/x = c$ is obtained via curve fitting.

than $mddc(by)$ and 74% faster than $mddc(ts)$. The speedup is moderate, compared with the one in the structural problems, because the MDDs are small (on average, an MDD has 20790 nodes; the median is 4794).

References

1. Beldiceanu, N., Contejean, E.: Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling* 20(12), 97–123 (1994)
2. Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Data structures for generalized arc consistency for extensional constraints. In: *National Conference on Artificial Intelligence* (2007)
3. Lecoutre, C., Szymanek, R.: Generalized arc consistency for positive table constraints. In: *International Conference on Principles and Practice of Constraint Programming*, pp. 284–298 (2006)
4. Lhomme, O., Régin, J.C.: A fast arc consistency algorithm for n -ary constraints. In: *National Conference on Artificial Intelligence*, pp. 405–410 (2005)
5. Bessière, C., Régin, J.C.: Arc consistency for general constraint networks: preliminary results. In: *International Joint Conference on Artificial Intelligence*, pp. 398–404 (1997)
6. Srinivasan, A., Kam, T., Malik, S., Brayton, R.: Algorithms for discrete function manipulation. In: *Computer Aided Design*, pp. 92–95 (1990)
7. Carlsson, M.: Filtering for the case constraint. Talk given at *Advanced School on Global Constraints* (2006)
8. Cheng, K.C.K., Yap, R.H.C.: Maintaining generalized arc consistency on ad-hoc n -ary boolean constraints. In: *European Conference on Artificial Intelligence*, pp. 78–82 (2006)
9. Briggs, P., Torczon, L.: An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems* 2(1–4), 59–69 (1993)
10. Mackworth, A.K.: On reading sketch maps. In: *International Joint Conference on Artificial Intelligence*, pp. 598–606 (1977)
11. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Comp.* 35(8), 667–691 (1986)
12. Lhomme, O.: Arc-consistency filtering algorithm for logical combinations of constraints. In: *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pp. 209–224 (2004)
13. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: *International Conference on Principles and Practice of Constraint Programming*, pp. 482–495 (2004)
14. Cheng, K.C.K., Yap, R.H.C.: Applying ad-hoc global constraints with the case constraint to Still-life. *Constraints* 11(2–3), 91–114 (2006)
15. van Hoeve, W.J., Pesant, G., Rousseau, L.M., Sabharwal, A.: Revisiting the sequence constraint. In: *International Conference on Principles and Practice of Constraint Programming*, pp. 620–634 (2006)

Perfect Constraints Are Tractable

András Z. Salamon^{1,2} and Peter G. Jeavons¹

¹ Computing Laboratory, University of Oxford, UK
{Andras.Salamon,Peter.Jeavons}@comlab.ox.ac.uk

² The Oxford-Man Institute for Quantitative Finance
Andras.Salamon@oxford-man.ox.ac.uk

Abstract. By using recent results from graph theory, including the Strong Perfect Graph Theorem, we obtain a unifying framework for a number of tractable classes of constraint problems. These include problems with chordal microstructure; problems with chordal microstructure complement; problems with tree structure; and the “all-different” constraint. In each of these cases we show that the associated microstructure of the problem is a perfect graph, and hence they are all part of the same larger family of tractable problems.

1 Introduction

Considerable effort has been devoted to identifying tractable subclasses of the constraint satisfaction problem. Most of this work has focused on just *two* general approaches: either identifying forms of constraint which are sufficiently restrictive to ensure tractability no matter how they are combined, or else identifying structural properties of constraint networks which ensure tractability no matter what forms of constraint are imposed (see Chapters 7 and 8 of [10]).

However, some important tractable classes of problems do not fall into either of these categories. A notable example is the class of problems where all the variables must be assigned different values. The tractability of this problem has been exploited to great effect by designing an efficient propagator for the global “all-different” constraint [9], which is widely used in practical constraint solvers.

The binary disequality relation used to specify the “all-different” problem is *not* contained in any tractable language (as it can express the NP-complete GRAPH COLOURING problem). The structure of the “all-different” problem is also *not* a tractable structure (since each variable constrains each other variable, and an arbitrary binary constraint problem can be represented on such a complete structure). It is the *combination* of structure and constraint language that leads to tractability for this problem, and there is currently little theory available to analyse such “hybrid” reasons for tractability.

In this paper we analyse such hybrid properties by examining the properties of a graph associated with every constraint problem instance, known as the *microstructure*. We show that for “all-different” problems, as well as several other known tractable classes, this microstructure is a *perfect* graph (see Figure 1 for a summary). For all such problems the tractability can then be deduced as an immediate consequence of classical results about perfect graphs [5].

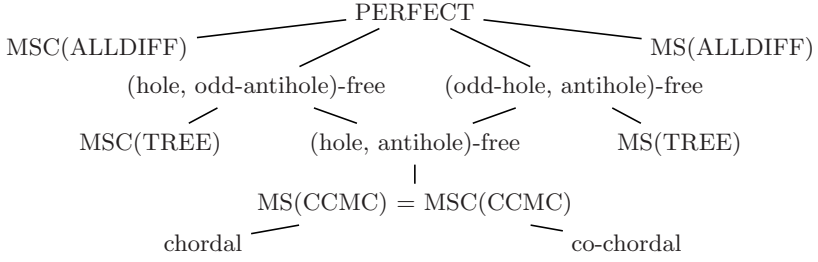


Fig. 1. Inclusions among constraint problems with perfect microstructure

2 Graphs, Perfect Graphs, and Microstructures

A **graph** is a structure $G = (V(G), E(G))$ containing a set $V(G)$ of vertices and a set $E(G) \subseteq \{\{u, v\} \mid u, v \in V(G), u \neq v\}$ of edges. The **order** of a graph is the number of its vertices. The **complement** $\overline{G}$ of graph G contains the same vertices as G , and its edges are the non-edges of G .

A graph G is a **subgraph** of H , written $G \subseteq H$, if $V(G) \subseteq V(H)$ and $E(G) \subseteq E(H)$. The graph H **contains** G if G is a subgraph of H such that $E(G)$ contains all edges in $E(H)$ that have both endpoints in $V(G)$. In this case G is also known as an **induced subgraph** of H . If G does *not* contain a graph that is isomorphic to any graph in class $\mathcal{C}$, then we say that G is **$\mathcal{C}$ -free**.

A **clique** is a subgraph which has edges connecting each pair of vertices. A **cycle** of order k in a graph is a subgraph with vertices $\{v_1, \dots, v_k\}$ and edges $\{v_k, v_1\}$ and $\{v_i, v_{i+1}\}$ for $i = 1, 2, \dots, k-1$. Such a cycle is usually denoted C_k .

A **vertex-colouring** of a graph is an assignment of colours to the vertices of the graph, such that the vertices of every edge are assigned different colours. The **chromatic number** of a graph is the smallest number of colours required for a vertex-colouring. A graph G is **perfect** if for every induced subgraph H of G , the chromatic number of H is equal to the order of the largest clique contained in H [5,6]. The smallest non-perfect graph is C_5 (which has chromatic number 3, but maximum clique size 2). Perfect graphs are important for our purposes because of the following classical result.

Theorem 1 ([5, Sect. 6]). *A maximum clique in a perfect graph can be found in polynomial time.*

A **hole** is a cycle of order $n \geq 5$. An **antihole** is the complement of a hole. The results in this paper rely heavily on the following recent result.

Theorem 2 ([2, 1.2], Strong Perfect Graph Theorem). *A graph is perfect if and only if it is (odd-hole, odd-antihole)-free.*

It has also recently been established that perfect graphs can be recognised in polynomial-time [1]. The class of perfect graphs will be denoted PERFECT.

The **microstructure** (MS) of a constraint problem instance is a graph where the set of vertices corresponds to the set of possible assignments of values to variables: a vertex (x, a) represents the assignment of value a to variable x [7].

The edges of the microstructure connect all pairs of variable-value assignments that are allowed (simultaneously) by the constraints. Note that if there is no explicit constraint between two variables x and y , then the microstructure includes edges between all pairs of variable-value assignments involving x and y .

The **microstructure complement** (MSC) is the complement of the microstructure: its edges represent pairs of variable-value assignments that are *disallowed* by the constraints [3] (including distinct values for the same variable).

We denote the microstructure and microstructure complement of a constraint problem instance P by $MS(P)$ and $MSC(P)$, respectively, and let $MS(\mathcal{C})$ and $MSC(\mathcal{C})$ denote the classes of graphs formed by microstructures and microstructure complements of instances in class $\mathcal{C}$. A **perfect constraint problem** is a class $\mathcal{C}$ of binary constraint problem instances such that $MS(\mathcal{C})$ is perfect.

Theorem 3. *Any perfect constraint problem can be solved in polynomial-time.*

Proof. For binary constraint problems, a solution corresponds to a (maximum) clique in the microstructure of order n , where n is the number of variables [7]. The result follows by Theorem 1. $\square$

3 Examples of Perfect Constraint Problems

A graph is called **chordal** or *triangulated* if it is (C_{n+4}) -free, where C_{n+4} denotes all cycles of order at least 4. Such graphs may have a cycle of order 4 or greater as a subgraph, but not as an *induced* subgraph — in other words, every cycle of order at least 4 must have a “chord” (an edge connecting two of its vertices that are not adjacent in the cycle). A graph is called **co-chordal** if its complement is chordal. All chordal and co-chordal graphs are perfect [6].

Jégou noted that binary constraint problems with chordal MS form a tractable class [7]. Cohen noted that binary constraint problems with chordal MSC also form a tractable class [3]. In fact this latter class has been shown to consist of problems that are “permutably max-closed” [4]. We observe that both of these classes are perfect, and can be combined to obtain a larger tractable class. Let $CCMC$ be the class of binary constraint problem instances P , such that either $MS(P)$ is chordal or $MSC(P)$ is chordal.

Proposition 4. $MS(CCMC) = MSC(CCMC) \subset (hole, antihole)\text{-free}^1$

Proof. The antihole of order 5 is isomorphic to C_5 and all larger antiholes contain C_4 , so chordal graphs are antihole free. The remaining inclusions are easy. $\square$

It is well-known that tree-structured binary constraint problems are tractable [10, Chapter 7]. However, the MS of a tree-structured problem is no longer a tree, and nor is the MSC . For example, the MSC of a single binary disequality constraint contains C_4 , so it is not a tree (and is also not chordal). On the other hand, tree-structured problems are perfect, as we now show. Let $TREE$ be the class of all tree-structured binary constraint problem instances.

¹ Also called *weakly chordal*.

Proposition 5. $MSC(TREE) \subset (hole, odd-antihole)\text{-free} \subset PERFECT$.

Proof. Let $G = MSC(P) \in MSC(TREE)$. If G contains a cycle C_k of order $k \geq 5$, then the vertices of C_k must involve at least 3 different variables (since different values for the same variable are all connected in a microstructure complement), and this implies that the structure of the instance P must contain a cycle, which contradicts the fact that it is tree-structured. Hence G is hole-free.

Since the antihole of order 5 is isomorphic to C_5 we need only consider antiholes of order ≥ 7 . Assume for contradiction that G contains an antihole A of order $k \geq 7$, on the vertices $(v_0, v_1, \dots, v_{k-1})$ (in that order around the cycle). We note that A contains an induced subgraph isomorphic to C_4 on every 4 vertices $(v_i, v_{i+1}, v_{j+1}, v_j)$ such that $0 \leq i < j < k$ and $i < i+2 < j < j+2 < i+k$ (with subscripts taken modulo k). Any induced 4-cycle in the MSC of a tree-structured problem must involve exactly 2 variables, hence every set of 4 vertices of this kind in A involves exactly 2 variables. This implies that the vertices $(v_1, v_2, \dots, v_k)$ involve just two variables, which alternate around the cycle.

If k is odd, these conditions are unsatisfiable, so G is odd-antihole-free. $\square$

We also obtain the following symmetrical result on taking complements.

Corollary 6. $MS(TREE) \subset (odd-hole, antihole)\text{-free} \subset PERFECT$.

An “**all-different**” constraint can be represented by a set of binary constraints of the form $x \neq y$, for each pair of distinct variables x and y .

The microstructure complement of such a constraint problem contains edges between vertices (x, a) and (y, b) (corresponding to assignments $x = a$ and $y = b$ respectively), when either $x = y$ and $a \neq b$, or else $a = b$ and $x \neq y$. For example, the MSC of an “all-different” problem with 3 variables, with domains $\{a_1, a_2\}, \{a_2, a_3\}, \{a_1, a_3\}$, is the even hole, C_6 . Hence “all-different” problems do not lie in any of the tractable classes described so far.

Let ALLDIFF be the class of “all-different” constraint problem instances. A **gridline** graph is one whose vertices can be embedded in the real plane, such that there is an edge between two distinct vertices precisely when they are on the same horizontal line or the same vertical line [8].

Proposition 7. $MSC(ALLDIFF) = gridline \subset (odd-hole, odd-antihole)\text{-free}$.

Proof. Let $G = MSC(P) \in MSC(ALLDIFF)$. We can embed each vertex (x, a) of G in the real plane, using the associated variable, x , to determine the horizontal position, and the associated value, a , to determine the vertical position. Now the edges of G are precisely those required by the definition of gridline.

Conversely, for any gridline graph G we have an associated embedding in the real plane, so we can map each vertex to a pair (x, y) , and consider these to be the (variable, value) pairs of a constraint problem. The graph G can then be considered as the MSC of an all-different problem.

Hence $MSC(ALLDIFF) = gridline$. By the results of [8], gridline graphs are (odd-hole, odd-antihole)-free (and hence perfect, by Theorem 2). $\square$

We also obtain the following symmetrical result on taking complements.

Corollary 8. $MS(ALLDIFF) \subset (odd-hole, odd-antihole)\text{-free} = PERFECT$.

4 Conclusions and Future Work

We have shown that a wide range of constraint problem classes (including hybrid classes which have previously been difficult to analyse) have a perfect microstructure, and hence can be solved efficiently by a single method.

However, the classical algorithm for finding a maximum clique in a perfect graph [5], although polynomial, is currently too slow to be practically useful, although there has been some progress in improving the method [11]. Recent advances such as the Strong Perfect Graph Theorem could provide a new route to developing more practical algorithms. The results presented above provide a strong additional motivation to develop such algorithms, which could then be used to solve a wide variety of constraint problems in a unified way.

Conversely, it may be that some of the algorithmic ideas developed for constraint solvers and propagators could be used in a more general, graph-theoretic context. For example, for gridline graphs, Peterson suggests that matching techniques could be used to obtain $O(n^3)$ algorithms for finding a maximum independent set [8], and that better algorithms should exist. The algorithm developed by Régim for obtaining domain consistency in all-different constraints is based on matching in a bipartite graph [9], which can be achieved in time $O(n^{2.5})$, and it may be that similar ideas could be exploited for arbitrary gridline graphs.

References

1. Chudnovsky, M., Cornuéjols, G., Liu, X., Seymour, P., Vušković, K.: Recognizing Berge graphs. *Combinatorica* 25(2), 143–186 (2005)
2. Chudnovsky, M., Robertson, N., Seymour, P., Thomas, R.: The strong perfect graph theorem. *Annals of Mathematics* 164(1), 51–229 (2006)
3. Cohen, D.A.: A new class of binary CSPs for which arc-consistency is a decision procedure. In: Rossi, F. (ed.) *CP 2003*. LNCS, vol. 2833, pp. 807–811. Springer, Heidelberg (2003)
4. Green, M.J., Cohen, D.A.: Domain permutation reduction for constraint satisfaction problems. *Artificial Intelligence* 172(8–9), 1094–1118 (2008)
5. Grötschel, M., Lovász, L., Schrijver, A.: The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica* 1(2), 169–197 (1981)
6. Hougardy, S.: Classes of perfect graphs. *Discrete Math.* 306, 2529–2571 (2006)
7. Jégou, P.: Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems. In: *Proceedings AAAI 1993*, pp. 731–736 (1993)
8. Peterson, D.: Gridline graphs: a review in two dimensions and an extension to higher dimensions. *Discrete Applied Mathematics* 126, 223–239 (2003)
9. Régim, J.C.: A filtering algorithm for constraints of difference in CSPs. In: *Proceedings AAAI 1994*, pp. 362–367 (1994)
10. Rossi, F., van Beek, P., Walsh, T. (eds.): *Handbook of Constraint Programming*. Elsevier, Amsterdam (2006)
11. Yıldırım, E.A., Fan-Orzechowski, X.: On extracting maximum stable sets in perfect graphs using Lovász’s theta function. *Comp. Optim. and Appl.* 33, 229–247 (2006)

Efficiently Solving Problems Where the Solutions Form a Group

Karen E. Petrie and Christopher Jefferson*

Computing Laboratory, University of Oxford, UK
karen.petrie@comlab.ox.ac.uk, chris.jefferson@comlab.ox.ac.uk

Abstract. Group theory is the mathematical study of symmetry. This paper presents a CP method of efficiently solving group-theoretic problems, where each of the solutions is an element of a group. This method allows us to answer questions in group theory which are computationally unfeasible with traditional CP techniques.

1 Introduction

Many problems arising in group theory can be naturally expressed as constraint problems but current solvers are often unable to solve instances of an interesting size. Our aim is to create a constraint programming based tool for mathematicians, that allows group theorists to search for groups with a specific property. It will allow counter example generation by answering for example: “Does a group exist with a given subgroup, and a given element of a certain order”.

This paper provides the fundamental basis for such a system; by providing a constraint programming method for solving group-theoretic problems, where each of the solutions is an element of a group. This algorithm works by finding only a small subset of solutions which are sufficient to generate every other solution. As we will see our method allows group-theoretic problems to be solved which can not be solved using traditional constraint techniques.

2 Overview of Method

In this section we will briefly define a number of common group-theoretic concepts, for a more complete introduction see [1]. Stabiliser chains provide an algorithmic method of constructing a small generating set [2] for any group and provide the inspiration for our algorithm. The stabiliser chain relies on the concept of the point wise stabiliser. We start by giving the definition of a stabiliser.

Definition 1. *Let G be a permutation group acting on the set of points Ω . Let $\beta \in \Omega$ be any point. The stabiliser of β is the subgroup of G defined by: $\text{Stab}_G(\beta) = \{g \in G \mid \beta^g = \beta\}$, which is the set of elements in G which fixes or*

* Chris Jefferson was supported by EPSRC Grant EP/D032636/1 and Karen Petrie by a Royal Society Fellowship. We would like to thank the referees for their comments.

stabilises the point β . The stabiliser of any point in a group G is a subgroup of G . The stabiliser of a set of points, denoted $\text{Stab}_G(i, j, \dots)$, is the elements of G which move none of the points.

The definition of the stabiliser chain follows.

Definition 2. *Stabiliser chains are built in an recursive fashion. Given a permutation group G and a point p , the first level of the stabiliser chain is built from an element of G which represents each of the places p can be mapped to. The next level of the stabiliser chain is built from applying this same algorithm to $\text{Stab}_G(p)$, again choosing representative elements for all the places some point $q \neq p$ can be mapped to. The stabiliser chain is finished when the stabiliser generated contains only the identity element.*

Stabiliser chains, in general, collapse quickly to the subgroup containing only the identity since the order of each new stabiliser must divide the order of the stabilisers above it. The following example is given to crystallise the stabiliser chain concept.

Example 1. Consider the symmetric group consisting of the 24 permutations of $\{1, 2, 3, 4\}$. We compute a chain of stabilisers of each point, starting arbitrarily with 1 (denoted $\text{Stab}_{S_4}(1)$). 1 can be mapped to 2 by $[2, 1, 3, 4]$, 3 by $[3, 1, 2, 4]$ and 4 by $[4, 1, 2, 3]$. These group elements form the first level of the stabiliser chain.

The second level is generated by looking at the orbit and stabiliser of 2 in $\text{Stab}_{S_4}(1)$. In the stabiliser of 1, 2 can be mapped to both 3 and 4 by the group elements $[1, 3, 2, 4]$ and $[1, 4, 2, 3]$. We now stabilise both 1 and 2, leaving only the group elements $[1, 2, 3, 4]$ and $[1, 2, 4, 3]$. Here 3 can be mapped to 4 by the second group element, and once 1, 2 and 3 are all stabilised the only element left is the identity and the algorithm finishes.

The stabiliser chain shows how a generating set of elements can be generated from a limited number of simple calculations. Our method is based around splitting search into a number of pieces, and finding only the first solution in each of these pieces. Each of these pieces is equivalent to a step in the stabiliser chain. We state without proof due to space restrictions that an arbitrary solution, should one exist, to each member of the split we define, form a stabiliser chain, and therefore a set of generators for the group of solutions. The precise split we use is given in Definition 3.

Definition 3. *Given a CSP P where the projection of the solutions onto some list of variables V of length n forms a permutation group, then the **generator split** of P is the following set of CSPs, each equal to P with a list of additional constraints: $\forall 1 \leq i < j \leq n. P_{i,j} = P \wedge (\forall 1 \leq k \leq i - 1. V_k = k) \wedge V_i = j$.*

Our algorithm is very simple. It requires creating a generator split of a CSP and then finding the first solution, if one exists, to each of the CSPs in the generator split by whichever means we wish. The major strength of this algorithm is that it can be implemented with no changes to the constraint solver. This does however

create a small overhead due to having to start the solver many times. In our Minion implementation, we instead create the CSP once in the solver, and then solve it multiple times. This is possible as the extra conditions imposed by the generator split are only variable assignments. No other changes were necessary to implement the algorithm.

3 Experimental Results

We consider a number of experiments, each of which involves solving a CSP whose solutions form a group. These will show that the gains made from identifying that the solutions to a problem form a group often provides massive advantages, and almost no loss in even the worst case.

3.1 Graph Automorphism

Probably the most famous problem whose solutions form a group is graph automorphism. Our model does not use the propagators given in [3] due to lack of an implementation in the Minion constraint solver.

We will consider finding the symmetries of two families of graphs, randomly generated graphs and the grid graph, given in Definition 4.

Definition 4. *The $l \times w$ **grid** graph is a graph on $l \times w$ vertices arranged in a grid of height l and width w , where two vertices are connected by an edge if they are either in the same row or same column.*

The symmetry group of the grid graph arises frequently in constraint programming, as this is the symmetry group of problems with “row and column” matrix symmetry [4], a commonly occurring group in constraint programming. Therefore being able to quickly identify this group would be an important and useful property for any system which would be used to identify the graph automorphisms which occur in CP.

Table 1 shows a comparison of our algorithm against a traditional complete search for identifying the automorphism group of grid graphs of various sizes. It is clear from these results that using a traditional search quickly becomes unfeasible. Using the generators found by our algorithm, a computational group theory package such as GAP can almost instantly produce the total size of the group, which we fill in for the two largest instances. Clearly no constraint solver could enumerate this many solutions.

Note that while our algorithm takes a non-trivial period of time for large graphs, the size of the search is still very small. Given a more efficient propagator for graph automorphism, we expect these times would drop dramatically.

We also conducted experiments to compare finding the symmetries of a small selection of random graphs. In general we expect such graphs to have no symmetries except for the identity symmetry, and indeed all the graphs we considered did only have this symmetry. As these graphs have no symmetries, we do not expect our algorithm to perform any better than a complete search. The aim therefore of these experiments is to investigate the overhead which is introduced.

Table 1. Comparing algorithms for finding the automorphisms of a grid graph

Size	Traditional			New		
	Solutions	Nodes	Time	Generators	Nodes	Time
3×3	72	143	0.007	12	31	0.07
4×4	1,152	2303	0.26	24	103	0.08
5×5	28,800	57599	9.64	40	238	0.12
6×6	1,036,800	2073599	711.8	60	455	0.31
10×10	2.6×10^{13}	-	-	180	2523	11.61
15×15	3.4×10^{24}	-	-	420	9233	297.6

The results show that there is almost no measurable overhead introduced by our algorithm.

For a static variable ordering along the permutation, we expect the searches produced with and without our algorithm to be almost identical, except for a tiny variance in the number of search nodes introduced from splitting the search into pieces before beginning and this is what we see in practice. We also conducted experiments using a dynamic variable ordering, smallest domain first. While this does introduce some measurable differences into the resulting searches, it is not clear if our algorithm is better or worse, and once again any variance is small. While this by no means proves our algorithm would not interfere with any dynamic variable ordering, it produces promising evidence that it does not effect search even when dynamic heuristics are used.

One important step we have not taken here is comparing our algorithm against specialised graph isomorphism systems, such as those provided in specialise graph isomorphism tools such as NAUTY [5] and SAUCY [6]. We feel for a fair comparison our algorithm must first be combined with a specialised propagator. We note that the experiments in [3] show a specialised propagator can find single automorphisms very competitively, giving hope that combined with our new algorithm the result should be comparable to these systems, while allowing a much greater degree of flexibility.

3.2 Group Intersection

One of the major advantages of designing our algorithm as a modification to search in a traditional CP framework is that allows us to use the flexibility of CP when modelling our problems. As an example of this flexibility, we consider finding the intersection of the grid graph, given in Definition 4, with the alternating group, given in Definition 5. Expressing this as a CSP requires simply imposing the constraints for both in the same problem. Definition 5 does not provide an obvious method of expressing that a permutation is even. A well known alternative method of checking if a permutation V is even is to check if the value of the expression $\sum_{1 \leq i < j \leq n} (V[i] > V[j])$ is even. This is the formulation which we use to express that a permutation is alternating.

Definition 5. *A permutation is even if it can be expressed as an even number of swaps of pairs of values. The alternating group contains even permutations.*

Table 2. Comparing algorithms for finding the intersection of the grid graph and alternating group

Size	Traditional			New		
	Solutions	Nodes	Time	Generators	Nodes	Time
3×3	36	107	0.02	11	30	0.04
4×4	1,152	2,303	0.81	24	103	0.06
5×5	14,400	43,199	7.5	39	237	0.13
6×6	518,400	1,555,199	509.4	55	274,010	109
7×7	25,401,600	76,204,799	40304	83	772	1.46

Table 2 gives results for this experiment. The results show how the power of constraint programming can be used to solve complex problems. It is unclear from where the large number of nodes for the 6×6 grid arise and this shows that it is non-obvious how hard it will be to find the intersection of two groups. Our algorithm still noticeably outperforms complete search in this instance and performs magnitudes better on the largest instance.

4 Conclusion

We have extended the abilities of constraint programming to allow problems in Computational Group Theory to be efficiently solved. We have, moreover, demonstrated experimentally that constraint programming can be a useful tool to solve group theoretic problems. Our method, allows us to solve problems which would not be possible without this constraint based decomposition technique. This result is important, since it shows the scope for constraint programming to be applied to group theoretic research.

References

1. Armstrong, M.A.: Groups and Symmetry. Springer, Heidelberg (2000)
2. Jerrum, M.: A compact presentation for permutation groups. *J. Algorithms* 7, 71–90 (2002)
3. Sorlin, S., Solnon, C.: A parametric filtering algorithm for the graph isomorphism problem. *Journal of constraints* (December 2008)
4. Flener, P., Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetries in matrix models. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 462–476. Springer, Heidelberg (2002)
5. McKay, B.: Practical graph isomorphism. In: Proc. 10th Manitoba Conf. on Numerical mathematics and computing, Winnipeg/Manitoba 1980, Congr. Numerantium, vol. 30, pp. 45–87 (1981), <http://cs.anu.edu.au/people/bdm/nauty>
6. Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting structure in symmetry detection for cnf. In: DAC 2004: Proceedings of the 41st annual conference on Design automation, pp. 530–534. ACM, New York (2004)

Approximate Solution Sampling (and Counting) on AND/OR Spaces

Vibhav Gogate and Rina Dechter

Donald Bren School of Computer Science
University of California, Irvine, CA 92697, USA
{vgogate, dechter}@ics.uci.edu

Abstract. In this paper, we describe a new algorithm for sampling solutions from a uniform distribution over the solutions of a constraint network. Our new algorithm improves upon the Sampling/Importance Resampling (SIR) component of our previous scheme of SampleSearch-SIR by taking advantage of the decomposition implied by the network's *AND/OR search space*. We also describe how our new scheme can approximately count and lower bound the number of solutions of a constraint network. We demonstrate both theoretically and empirically that our new algorithm yields far better performance than competing approaches.

1 Introduction

In this paper, we present a new Sampling/Importance Resampling (SIR) algorithm that exploits AND/OR search spaces for graphical models [1]. Although, our algorithm is quite general, in this paper, we focus on using it for sampling solutions from a constraint network. Our main contributions are: (a) We show that SIR can be understood as a method that learns a probability distribution on an OR tree. (b) We generalize SIR to AND/OR spaces yielding AO-SIR which learns a probability distribution on an AND/OR tree (or graph). (c) We show theoretically and by an experimental evaluation on satisfiability benchmarks that AO-SIR is far more accurate than SIR. (d) We derive a new unbiased estimator from the distribution learnt over the AND/OR tree which can be used to approximately count and lower bound the number of solutions, improving over our previous solution counter presented in [4].

2 Preliminaries and Previous Work

Definition 1 (constraint network, counting and sampling). A *constraint network (CN)* is defined by a 3-tuple, $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$, where $\mathbf{X} = \{X_1, \dots, X_n\}$ is a set of variables associated with a set of discrete-valued domains, $\mathbf{D} = \{\mathbf{D}_1, \dots, \mathbf{D}_n\}$, and $\mathbf{C} = \{C_1, \dots, C_r\}$ is a set of constraints. Each constraint C_i is a relation $\mathbf{R}_{S_i}$ defined on a subset of variables $\mathbf{S}_i \subseteq \mathbf{X}$. The relation denotes all compatible tuples of the cartesian product of the domains of $\mathbf{S}_i$. A solution is an assignment of values to all variables $\mathbf{x} = (X_1 = x_1, \dots, X_n = x_n)$, $x_i \in \mathbf{D}_i$, such that $\mathbf{x}$ belongs to the natural join of all constraints i.e. $\mathbf{x} \in \mathbf{R}_{S_1} \bowtie \dots \bowtie \mathbf{R}_{S_r}$. The **solution counting problem** $\#csp$ is the problem of counting the number of solutions. The **solution sampling problem** $\$csp$ is the problem of sampling solutions from a uniform distribution over the solutions.

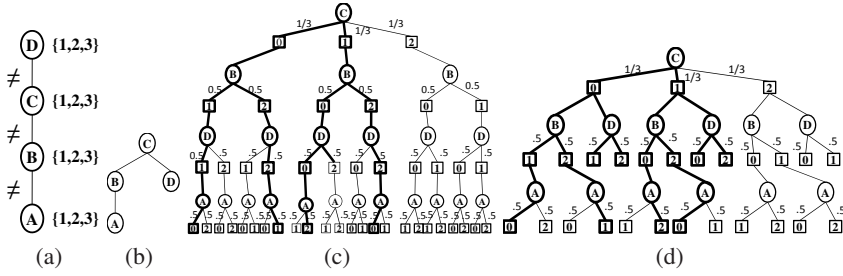


Fig. 1. (a) A 3-coloring problem, (b) Pseudo-tree (c) OR tree (d) AND/OR tree

2.1 AND/OR Search Spaces for Graphical Models

Given a constraint network, its AND/OR search space is guided by the pseudo-tree defined below (for more information see [1]).

Definition 2 (Pseudo Tree). Given a constraint graph $G = (V, E)$, a pseudo-tree $T = (V, E')$ is a directed rooted tree in which any arc not included in E' is a back-arc.

Definition 3 (AND/OR tree). Given a constraint network $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$ and a pseudo tree T , the AND/OR search tree S_{AOT} , has alternating levels of AND and OR nodes. The root of S_{AOT} is an OR node labeled by the root of T . The children of an OR node X_i are AND nodes labeled with assignment $X_i = x_i$ that are consistent with the assignment $(X_1 = x_1, \dots, X_{i-1} = x_{i-1})$ along the path from the root. The children of an AND node $X_i = x_i$ are OR nodes labeled with the children of variable X_i in T . A **solution subtree** of S_{AOT} contains the root node. For every OR node it contains one of its children and for each of its AND nodes it contains all its children. An **OR tree** is an AND/OR tree whose pseudo-tree is a chain.

Example 1. Figure 1(c) and 1(d) show a complete OR tree and an AND/OR tree (guided by the pseudo-tree in Figure 1(b)) respectively for the 3-coloring problem in Figure 1(a).

2.2 Exact Solution to the $\$csp$ Problem

Dechter et al. [2] proposed the following scheme to exactly solve the $\$csp$ problem. We first express the uniform distribution $\mathcal{P}(\mathbf{x})$ in a product factored form: $\mathcal{P}(\mathbf{x} = (x_1, \dots, x_n)) = \prod_{i=1}^n P_i(x_i | x_1, \dots, x_{i-1})$. The probability $P_i(X_i = x_i | x_1, \dots, x_{i-1})$ is equal to the ratio between the number of solutions that $(x_1, \dots, x_i)$ participates in and the number of solutions that $(x_1, \dots, x_{i-1})$ participates in. Second, we generate multiple samples by repeating the following process: for $i = 1$ to n , given a partial assignment $(x_1, \dots, x_{i-1})$ to the previous $i - 1$ variables, we assign a value to variable X_i by sampling it from $P_i(X_i | x_1, \dots, x_{i-1})$. All algorithms described in this paper are devoted to finding an approximation to $P_i(X_i | x_1, \dots, x_{i-1})$ at each branch of the search tree.

Example 2. The labeled OR and AND/OR tree of Figure 1(c) and 1(d) respectively depict the uniform distribution over the solutions expressed in a product factored form.

2.3 Sampling Importance Resampling to Solve the $\$csp$ Problem Approximately

Because constructing $\mathcal{P}(\mathbf{x})$ can be quite hard [2], in [6] we proposed to use Sampling Importance Resampling (SIR) [8] in conjunction with the SampleSearch scheme [3] to approximate it. This scheme operates as follows. First, given a proposal distribution Q , it uses SampleSearch to draw random solution samples $\mathbf{A} = (\mathbf{x}^1, \dots, \mathbf{x}^N)$ from the backtrack-free distribution Q^F of Q . Second, a possibly smaller number of samples $\mathbf{B} = (\mathbf{y}^1, \dots, \mathbf{y}^M)$ are drawn from $\mathbf{A}$ with sample probabilities, proportional to the weights $w(\mathbf{x}^i) = 1/Q(\mathbf{x}^i)$ (this step is referred to as the *re-sampling step*). For $N = 1$, the distribution of solutions is same as Q^F . For a finite N , the distribution of solutions is somewhere between Q^F and $\mathcal{P}$ improving as N increases and equals $\mathcal{P}$ as $N \rightarrow \infty$.

3 Sampling Importance Resampling on AND/OR Search Spaces

We first describe the main intuition involved in defining a new SIR scheme called AO-SIR which operates on the AND/OR search space in the following example.

Example 3. The bold edges and nodes in Figure 1 (c) and (d) show four solution samples arranged on an OR and AND/OR tree respectively. Note that SIR and AO-SIR operate on the OR and AND/OR tree respectively. One can verify that the 4 solution samples correspond to 8 solution samples (solution sub-trees) on the AND/OR tree. Thus, the AND/OR representation yields a larger set of virtual samples. It includes for example the assignment $(C = 0, B = 2, D = 1, A = 0)$ which is not represented in the OR tree. From SIR theory [8], we know that the accuracy of SIR increases with the number of samples and therefore we expect AO-SIR to be more accurate than SIR.

In AO-SIR, the first step of using SampleSearch to generate solution samples remains the same. What changes is the way in which we (a) store samples and (b) define the distribution over the initial set of samples for resampling. We explain each of these modifications below. We first define the notion of an AND/OR sample tree (or graph) which can be used to store the initial set of samples.

Definition 4 (Arc Labeled AND/OR sample tree and graph). Given (1) a constraint network $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$, (2) a pseudo-tree $T(V, E)$, (3) the backtrack-free distribution $Q^F = \prod_{i=1}^n Q^F(X_i | \mathbf{Anc}(X_i))$ such that $\mathbf{Anc}(X_i)$ is a subset of all ancestors of X_i in T , (4) a sequence of samples $\mathbf{S}$ (assignments) generated from Q^F , **an arc-labeled AND/OR sample tree** S_{AOT} is a complete AND/OR tree (see definition 3) from which all assignments not in $\mathbf{S}$ are removed. The arc-label from an OR node X_i to an AND node $X_j = x_j$ in S_{AOT} is a pair $\langle w, \# \rangle$ where: (a) $w = \frac{1}{Q^F(X_j=x_j | \mathbf{anc}(X_i))}$ is called the weight of the arc. $\mathbf{anc}(X_i)$ is the assignment of values to all variables in $\mathbf{Anc}(X_i)$ and (b) $\#$: the frequency of the arc is the number of times $(X_i = x_i, \mathbf{anc}(X_i))$ is seen in $\mathbf{S}$.

As noted earlier, all approximate algorithms for solving the $\$csp$ problem can be thought of as approximating the probability labels on the arc of an AND/OR tree. Because, the probability labels are just ratios of solution counts, we define the notion of value of a node which can be semantically understood as providing an unbiased estimate of the solution counts of the subtree rooted at the node.

Algorithm 1. $AO - SIR(\mathcal{R}, Q^F, N, M)$

```

1: Generate  $N$  i.i.d. samples  $\mathbf{A} = \{\mathbf{x}^1, \dots, \mathbf{x}^N\}$  from  $Q^F$  using SampleSearch.
2: Store the  $N$  solution samples on an AND/OR sample tree  $S_{AOT}$  or a graph and label it using definition 4.
3: FOR all leaf nodes  $i$  of  $S_{AOT}$  do
4:   IF And-node  $v(i) = 1$  ELSE  $v(i) = 0$ 
5: FOR every node  $n$  from leaves to the root do
6:   Let  $C(n)$  denote the child nodes of node  $n$ 
7:   IF  $n = \langle X, x \rangle$  is a AND node, then  $v(n) = \prod_{n' \in C(n)} v(n')$ 
8:   ELSE if  $n = X$  is a OR node then  $v(n) = \frac{\sum_{n' \in C(n)} (\#(n, n') w(n, n') v(n'))}{\sum_{n' \in C(n)} \#(n, n')}$ .
9: Update the weights by updating the arc labels as:  $p(n, n') = \frac{v(n') w(n, n') \#(n, n')}{\sum_{n'' \in C(n)} (\#(n, n'') w(n, n'') v(n''))}$ 
10: Return  $M$  solution samples generated at random from  $S_{AOT}$ 

```

Definition 5 (Value of a node). *The value of a node in a arc-labeled AND/OR sample tree (see Definition 4) is defined recursively as follows. The value of leaf AND nodes is "1" and the value of leaf OR nodes is "0". Let $C(n)$ denote the child nodes of n and $v(n)$ denotes the value of node n . If n is an AND node then: $v(n) = \prod_{n' \in C(n)} v(n')$ and if n is a OR node then $v(n) = \frac{\sum_{n' \in C(n)} (\#(n, n') w(n, n') v(n'))}{\sum_{n' \in C(n)} \#(n, n')}$.*

Lemma 1. *The value of a node n is an unbiased estimate of the number of solutions of the subtree rooted at n .*

We now have the required definitions to formally present algorithm AO-SIR (see Algorithm 1). Here, we first generate samples in the usual way from Q^F . We then store these samples on an arc labeled AND/OR sample tree and compute the value of each node (Steps 3-8). The AND/OR sample tree is then converted to an AND/OR sample probability tree by normalizing the values at each OR node (Step 9). Finally, the required M solution samples are generated from the AND/OR sample probability tree. We can prove that:

Theorem 1. *As $N \rightarrow \infty$, the solutions generated by AO-SIR will consist of independent draws from the uniform distribution over the solutions.*

Finally, we summarize the relationship between AO-SIR and SIR in Theorem 2:

Theorem 2. *When the pseudo-tree is a chain, the solution samples output by AO-SIR will have the same distribution as those output by SIR. Asymptotically, if the pseudo-tree is not a chain then AO-SIR has lower sampling error than SIR.*

3.1 Approximate Counting on AND/OR Search Spaces

From Lemma 1, it is easy to see that:

Proposition 1. *The value of the root node of the AND/OR sample tree is an unbiased estimate of the number of solutions.*

We can utilize this unbiased estimate obtained from the AND/OR sample tree to obtain a lower bound on the solution counts [7,4] in a straight forward way. The main virtue of using the AND/OR space estimator over our previous scheme [4] is that the former may have lower variance (and therefore likely to have better accuracy) than the latter (for more details, see [5]).

4 Experimental Results

For lack of space, we present only a part of our empirical results for solution sampling. Detailed experimental results for both solution sampling and counting are presented in the extended version of this paper [5]. For solution sampling, we experimented with the following schemes (a) SampleSearch [3] (b) SampleSearch-SIR [6] (c) SampleSat [9] and (d) AO-SIR. Table 1 summarizes the results of running each algorithm for exactly 1 hr on various benchmarks. The second and the third column report the number of variables and clauses respectively. The remaining columns report the KL distance between the exact and the approximate distribution for various competing schemes. Note that lower the KL distance the more accurate the sampling algorithm is. We can see that AO-SIR is more accurate than SampleSearch-SIR on most benchmarks. Also the SIR-type methods are more accurate than pure SampleSearch and SampleSat.

Table 1. Results for Solution Sampling

Problem	#Var	#Cl	SampleSearch	SampleSearch-SIR	AO-SIR	SampleSat
			KL	KL	KL	KL
Pebbling						
grid-pbl-25	650	1226	0.13	0.027	0.00600	0.138
grid-pbl-30	930	1771	0.15	0.040	0.00170	0.154
Circuit						
2bitcomp_5	125	310	0.03	0.003	0.00100	0.033
2bitmax_6	252	766	0.11	0.006	0.00100	0.039
Coloring						
Flat-100	300	1117	0.08	0.001	0.00190	0.020
Flat-200	600	2237	0.11	0.016	0.00800	0.030

Acknowledgements. This work was supported in part by the NSF under award numbers IIS-0331707, IIS-0412854 and IIS-0713118 and the NIH grant R01-HG004175-02.

References

1. Dechter, R., Mateescu, R.: AND/OR search spaces for graphical models. *Artificial Intelligence* 171(2-3), 73–106 (2007)
2. Dechter, R., Kask, K., Bin, E., Emek, R.: Generating random solutions for constraint satisfaction problems. In: *AAAI*, pp. 15–21 (2002)
3. Gogate, V., Dechter, R.: A new algorithm for sampling csp solutions uniformly at random. In: Benhamou, F. (ed.) *CP 2006*. LNCS, vol. 4204. Springer, Heidelberg (2006)
4. Gogate, V., Dechter, R.: Approximate counting by sampling the backtrack-free search space. In: *AAAI*, pp. 198–203 (2007)
5. Gogate, V., Dechter, R.: Approximate solution sampling (and counting) on and/or spaces. Technical Report, University of California, Irvine (2008)
6. Gogate, V., Dechter, R.: Studies in solution sampling. In: *AAAI* (2008)
7. Gomes, C., Hoffmann, J., Sabharwal, A., Selman, B.: From sampling to model counting. In: *IJCAI* (2007)
8. Rubin, D.B.: The calculation of posterior distributions by data augmentation. *Journal of the American Statistical Association* 82 (1987)
9. Wei, W., Erenrich, J., Selman, B.: Towards efficient sampling: Exploiting random walk strategies. In: *AAAI* (2004)

Model Restarts for Structural Symmetry Breaking^{*}

Daniel Heller, Aurojit Panda, Meinolf Sellmann, and Justin Yip

Brown University, Department of Computer Science
115 Waterman Street, P.O. Box 1910, Providence, RI 02912

There exist various methods to break symmetries. The two that concern us in this paper are static symmetry breaking where we add static constraints to the problem (see e.g. [1,3]) and symmetry breaking by dominance detection (SBDD) where we filter values based on a symmetric dominance analysis when comparing the current search-node with those that were previously expanded [2,5]. The core task of SBDD is dominance detection. The first provably polynomial-time dominance checkers for value symmetry were devised in [18] and [14]. For problems exhibiting both “piecewise” symmetric values and variables, [15] devised structural symmetry breaking (SSB). SSB is a polynomial-time dominance checker for piecewise symmetries which, used within SBDD, eliminates symmetric subproblems from the search-tree. Piecewise symmetries are of particular interest as they result naturally from symmetry detection based on a static analysis of a given CSP that exploits the knowledge about problem substructures as captured in global constraints [17]. Static SSB was developed in [4] and is based on the structural abstractions that were introduced in [17].

Compared with other symmetry-breaking techniques, the big advantage of dynamic symmetry breaking is that it can accommodate dynamic search orderings without running an increased risk of thrashing. Dynamic orderings have often been shown to vastly outperform static orderings in many different types of constraint satisfaction problems. However, when adding static symmetry-breaking constraints that are not aligned with the variable and value orderings, it is entirely possible that we dismiss perfectly good solutions just because they are not the ones that are favored by the static constraints, which (ideally) leave only one representative solution in each equivalence class of solutions. To address this problem, Puget suggested an elegant semi-static symmetry breaking method that provably does not remove the first solution found by a dynamic search-method [12]. It is not clear how this method can be generalized, though, and for the case of piecewise variable and value symmetry, no method with similar properties is known yet. On the other hand, static methods are generally easy to use, enjoy a low overhead per choice point, and exhibit an anticipatory character that emerges from filtering symmetry-breaking constraints in combination with constraints in the problem.

In this paper, for the first time ever we compare static and dynamic SSB in practice. We will show that static SSB works much faster than dynamic SSB. However, this gain comes at a cost: Static SSB introduces a huge variance in runtime as static symmetry breaking constraints may clash with dynamic search orderings. Using static search orderings, on the other hand, can also cause large variances in runtime as they are not equally well suited for different problem instances. To avoid this core problem of static symmetry breaking, we introduce the idea of “model restarts.” We will show

^{*} This work was supported by the National Science Foundation through the Career: Cornflower Project (award number 0644113).

how they allow us to efficiently combine static symmetry breaking with semi-dynamic search-orderings. The method is very simple to use and we show that model restarts greatly improve the robustness of static symmetry breaking.

1 Static and Dynamic Structural Symmetry Breaking

A *Constraint Satisfaction Problem (CSP)* is a tuple (Z, V, D, C) where Z is a finite set of variables, V is a set of values, $D = \{D_1, \dots, D_n\}$ is a set of finite domains where each $D_i \subseteq V$ is the set of possible instantiations to variable $X_i \in Z$, and $C = \{c_1, \dots, c_p\}$ is a finite set of constraints where each $c_i \in C$ is defined on a subset of the variables in Z and specifying their valid combinations. Given a set S and a set of sets $P = \{P_1, \dots, P_r\}$ such that $\bigcup_i P_i = S$ and the P_i are pairwise non-overlapping, we say that P is a *partition* of S , and we write $S = \sum_i P_i$. Given a set S and a partition $S = \sum_i P_i$, a bijection $\pi : S \mapsto S$ such that $\pi(P_i) = P_i$ (where $\pi(P_i) = \{\pi(s) \mid s \in P_i\}$) is called a *piecewise permutation* over $S = \sum_i P_i$. Given a CSP (Z, V, D, C) , and partitions $Z = \sum_{k \leq r} P_k$, $V = \sum_{l \leq s} Q_l$, we say that the CSP has *piecewise variable and value symmetry* iff all variables within each P_k and all values within each Q_l are considered symmetric.

As mentioned in the introduction, dynamic SSB is a special case of SBDD. Before we expand a new search-node we first check if the partial assignment that led us to the current node is not dominated by any partial assignment that has been fully explored earlier. SBDD ensures that there is only a linear number of dominance checks needed. SSB performs the dominance checks by setting up a bipartite graph and pruning the current node if and only if a perfect matching can be found. In [15] it was shown how dynamic SSB can be used for filtering in time $O(nm^{3.5} + n^2m^2)$, where m is the number of values and n the number of variables in the given CSP.

Static SSB is based on the abstractions used by the dominance checker in dynamic SSB. In [4] a linear set of (global) constraints was devised which provably leaves one and only one solution in each equivalence class of solutions. Using Regin's filtering algorithm [13] for the global cardinality constraints (GCCs) in this set, filtering all static SSB constraints does not take longer than amortized $O(\sum_{k=1}^a |P_k|^2 m) = O(n^2 m)$, where m is the number of values and n the number of variables in the given CSP.

2 Model Restarts

As the runtime comparison shows, static symmetry-breaking imposes much less overhead. However, it suffers from one important drawback, and that is the fact that it is much more sensitive to search-orderings. Both in [10] and [16] it has been noted that static symmetry-breaking constraints can cause great variances in expected runtime. Knowing that static symmetry-breaking constraints work by excluding all but one representative out of each equivalence class of solutions, this is hardly surprising: When the symmetry-breaking constraints are not aligned with the search-orderings, static symmetry-breaking constraints may interrupt the construction of many perfectly good solutions, simply because they are not the representatives we have chosen. On the other hand, when using static search orderings, they themselves are much less robust

and perform with greatly varying performance on different problem instances. The question arises how we can combine the benefits of being able to change the search orderings while using lean static symmetry breaking.

We exploit the idea of randomization and restarts [7,9], which has been shown to greatly improve the robustness of systematic search: When the search takes too long (as determined by exceeding a given fail-limit) we interrupt our search, and try again with an updated fail-limit [9]. To avoid that we conduct the same search over and over again, a random component is added to the selection heuristics for the branching variable and/or the branching value. The method has been proven both experimentally and theoretically to eliminate heavy-tailed run-time distributions [7]. Complemented by non-chronological backtracking and no-good learning, the idea of randomization and restarts marks one of the backbones of modern systematic SAT solvers.

To improve on the robustness of static symmetry breaking, we therefore propose to exploit randomization and restarts. Note that, when posing static symmetry breaking constraints, there is often a lot of freedom in how we determine the representatives that we leave in each equivalence class of solutions (see, e.g., [16]). In our case, with respect to the ordering of variables, we have the freedom to arbitrarily choose the ordering of the variable partitions. We start our search and use the static variable ordering as induced by the ordering of variable partitions used in the static symmetry breaking constraints. This avoids clashes between static SSB and the search ordering used. However, the particular search ordering we choose may not be suited well for the concrete instance we need to solve. Consequently, we interrupt our search when a fail-limit is reached. Now, we would like to choose a new and somewhat randomized search ordering, but if we do, then it is likely to clash with our static constraints. Consequently, rather than updating the search ordering only, *we also change the underlying CP model!* That is, at every restart we do not only change the search-orderings, *but also the corresponding static symmetry-breaking constraints*. This way, we avoid clashes between the search-orderings and static symmetry-breaking constraints, and are still not bound to one static search-ordering which may be bad for the given problem instance. We refer to this simple and easy to use idea as “model restarts.”

3 Experimental Results

We experiment on the benchmark introduced in [11] that consists of graph coloring problems over symmetric graphs. Colors are interchangeable values and nodes that have the same set of neighbors form a partition of piecewise interchangeable variables. Randomized graph coloring problems are generated with either a uniform or a biased distribution of partitions of interchangeable nodes in the graphs, and a parameter q influences the density of the graphs (for details, see [11]). For reasons of comparability, our experimental set-up is the same as in [11]. Each data point we report represents the

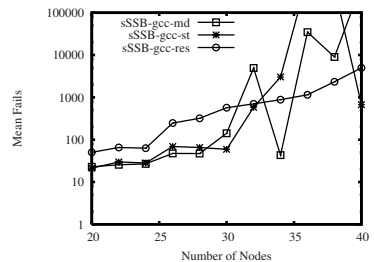


Fig. 1. Mean number of fails (log-scale) for the biased graph-coloring benchmark ($q=1$)

mean of runs on 20 different instances with a cutoff of one hour. For each data point, at least 90% of all runs finish within this time-frame. For the restarted methods, fail-limits grow linearly as multiples of 100. We use the same CSP model as in [11].

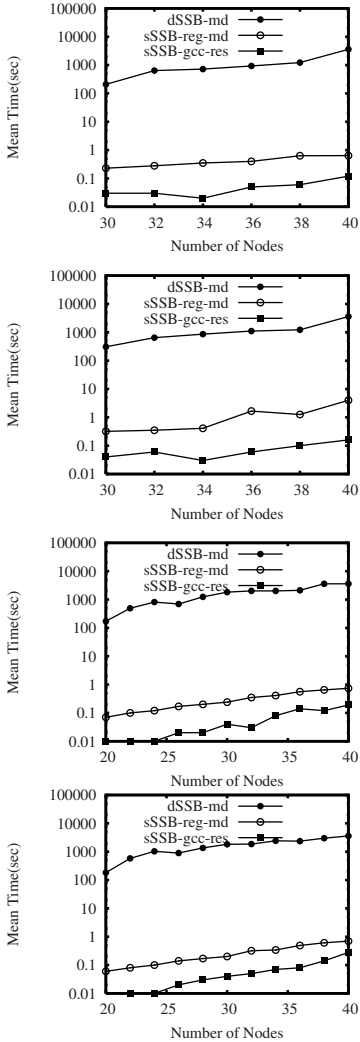


Fig. 2. Mean times (seconds, log-scale) for the uniform (top two) and biased (bottom two) graph-coloring benchmark with $q = 0.5$ (1st and 3rd) and $q = 1$ (2nd and 4th)

In Figure 1, we study three different algorithms on the biased graph-coloring benchmark: static SSB in combination with a min-domain heuristic (sSSB-gcc-md), static SSB in combination with a corresponding static variable ordering (sSSB-gcc-st), and static SSB with model restarts (sSSB-gcc-res), where the ordering of variable partitions is permuted at every model restart, with a bias to place larger partitions earlier in the ordering. The figure shows the average number of failures, but since the time per choice point for all variants is practically the same, we get the exact same picture when comparing running times (for the actual runtime of sSSB-gcc-res, compare with Figure 2). We observe that sSSB-gcc-md and sSSB-gcc-st are both not robust at all. Due to a high variance in runtime the curves are highly erratic. The reason why sSSB-gcc-st shows such a high variance in solution time is that the static ordering that is chosen is good for some instances and bad for others. Dynamic variable orderings like the min-domain heuristic usually lead to much more robust performance. However, in the case of symmetric problems, the dynamic orderings may clash with the static constraints, and sSSB-gcc-md is not performing consistently well either. On the other hand, we can see clearly how model restarts greatly improve the robustness of sSSB.

In Figure 2 we compare restarted static SSB based on GCC constraints (sSSB-gcc-res) with dynamic SSB (with min-domain heuristic, dSSB-md) and a different variant of static SSB based on regular constraints that was introduced in [11] (with min-domain heuristic, sSSB-reg-md). The first two algorithms were run on an AMD-Athlon 64-X2 3800+ (2.0GHz), the latter was run on a Sun Blade 2500 (1.6GHz) and the curve shown is an adaptation of that shown in [11]. Based on the data given in that paper, we can infer that their machine can process about 20K failures per second when

running sSSB-gcc-md, while we measured 30K failures per second on our architecture for the same method. We conclude that our machine works about a factor 1.5 faster and thus divided the data-points underlying the curve shown in [11] by that factor to make the comparison fair.

We see that the restarted method works equally robust as sSSB-reg-md, but roughly one order of magnitude faster. In [11] it is reported that sSSB-reg visits less than 100 choice points on the biased instances (which makes it highly unlikely that restarts will lead to any further improvements), the number of failures of sSSB-gcc-res is shown in Figure 1. Consequently, sSSB-gcc-res visits about two orders of magnitude more choice points than sSSB-reg. This implies that sSSB-gcc works almost three orders of magnitude faster per choice point. This efficiency gives the simple sSSB-gcc-res the advantage over the much more effective filtering of sSSB-reg-md. When comparing with dynamic SSB, finally, as the theoretical runtime comparison of dSSB and sSSB in Section 1 already suggested, we find that the dynamic variant cannot compete with the static methods, despite our great efforts to tune the method as best as possible using the heuristic improvements introduced in [8].

In Figure 3 we compare dSSB with sSSB on a different benchmark where we generate random AllDifferent constraints which each partition a set of 12 values and 15 variables, thus leaving a piecewise symmetric CSP. By varying the number of values per constraint, we achieve a range of more and more restricted piecewise symmetric problems which allows us to compare methods over an entire regime of constrainedness. Again, we see that static symmetry breaking vastly outperforms dynamic symmetry breaking. Although we cannot show the result of those tests here, we would also like to note that restarts do not lead to performance improvements for dynamic SSB.

We conclude that dynamic SSB for piecewise symmetry is inferior to its static counterpart. Moreover, we found that static SSB based on GCC constraints is far less effective than static SSB based on regular constraints as it visits many more choice points. However, its extremely low cost per choice points causes it to run faster, and when used with model restarts it works equally robustly.

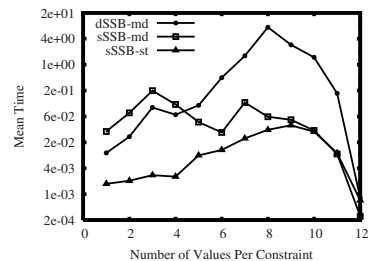


Fig. 3. Mean times (seconds, log-scale, cutoff 600 seconds) on 100 random AllDiff-CSP instances

References

1. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. In: KR, pp. 149–159 (1996)
2. Fahle, T., Schamberger, S., Sellmann, M.: Symmetry Breaking. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 93–107. Springer, Heidelberg (2001)
3. Flener, P., Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetries in matrix models. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 462–476. Springer, Heidelberg (2002)
4. Flener, P., Pearson, J., Sellmann, M., Van Hentenryck, P.: Static and Dynamic Structural Symmetry Breaking. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 695–699. Springer, Heidelberg (2006)

5. Focacci, F., Milano, M.: Global cut framework for removing symmetries. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 77–92. Springer, Heidelberg (2001)
6. Gent, I., Harvey, W., Kelsey, T., Linton, S.: Generic SBDD using computational group theory. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 333–347. Springer, Heidelberg (2003)
7. Gomes, C.P., Selman, B., Crato, N.: Heavy-Tailed Distributions in Combinatorial Search. In: CP, pp. 121–135 (1997)
8. Heller, D., Sellmann, M.: Dynamic Symmetry Breaking Restarted. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 721–725. Springer, Heidelberg (2006)
9. Kautz, H., Horvitz, E., Ruan, Y., Gomes, C., Selman, B.: Dynamic Restart Policies. In: AAAI, pp. 674–682 (2002)
10. Kiziltan, Z.: Symmetry Breaking Ordering Constraints. PhD Thesis (2004)
11. Law, Y.-C., Lee, J., Walsh, T., Yip, J.: Breaking symmetry of interchangeable variables and values. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 423–437. Springer, Heidelberg (2007)
12. Puget, J.F.: Dynamic Lex Constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 453–467. Springer, Heidelberg (2006)
13. Régin, J.-C.: Generalized arc-consistency for global cardinality constraint. In: AAAI, pp. 209–215. AAAI Press, Menlo Park (1996)
14. Roney-Dougal, C., Gent, I., Kelsey, T., Linton, S.: Tractable symmetry breaking using restricted search trees. In: ECAI, pp. 211–215 (2004)
15. Sellmann, M., Van Hentenryck, P.: Structural Symmetry Breaking. In: IJCAI, pp. 298–303 (2005)
16. Smith, B.M.: Sets of Symmetry Breaking Constraints. In: SymCon 2005 (2005)
17. Van Hentenryck, P., Flener, P., Pearson, J., Agren, M.: Compositional derivation of symmetries for constraint satisfaction. In: Zucker, J.-D., Saitta, L. (eds.) SARA 2005. LNCS (LNAI), vol. 3607, pp. 234–247. Springer, Heidelberg (2005)
18. Van Hentenryck, P., Flener, P., Pearson, J., Agren, M.: Tractable symmetry breaking for CSPs with interchangeable values. In: IJCAI, pp. 277–282 (2003)

An Elimination Algorithm for Functional Constraints

Yuanlin Zhang¹, Roland H.C. Yap²,
Chendong Li¹, and Satyanarayana Marisetti¹

¹ Texas Tech University, USA

{y.zhang, chendong.li, satyanarayana.marisetti}@ttu.edu

² National University of Singapore, Singapore

ryap@comp.nus.edu.sg

1 Introduction and Preliminaries

Functional constraints are studied in Constraint Satisfaction Problems (CSP) using consistency concepts (e.g., [1,4]). In this paper, we propose a new method — *variable substitution* — to process functional constraints. The idea is that if a constraint is functional on a variable, this variable in another constraint can be substituted using the functional constraint without losing any solution. We design an efficient algorithm to reduce, in $\mathcal{O}(ed^2)$, a general binary CSP containing functional constraints into a canonical form which simplifies the problem and makes the functional portion trivially solvable. When the functional constraints are also bi-functional, then the algorithm is linear in the size of the CSP.

We use the standard notations in CSP. Two CSPs are *equivalent* if and only if they have the same solution space. Throughout this paper, n represents the number of variables, d the size of the largest domain of the variables, and e the number of constraints in a problem. The composition of two constraints is defined as $c_{jk} \circ c_{ij} = \{(a, c) \mid \exists b \in D_j, \text{ such that } (a, b) \in c_{ij} \wedge (b, c) \in c_{jk}\}$. Composing c_{ij} and c_{jk} gives a new constraint on i and k .

A constraint c_{ij} is *functional* on j if for any $a \in D_i$ there exists at most one $b \in D_j$ such that $(a, b) \in c_{ij}$. c_{ij} is *functional* on i if c_{ji} is functional on i . When a constraint c_{ij} is functional on j , for simplicity, we say c_{ij} is functional by making use of the fact that the subscripts of c_{ij} are an ordered pair. In this paper, the definition of functional constraints is different from the one in [4,5] where constraints are functional on each of its variables, leading to the following notion.

A constraint c_{ij} is *bi-functional* if c_{ij} is functional on both i and j . A bi-functional constraint is called *bijective* in [2] and simply functional in [4].

2 Elimination Algorithm

Definition 1. Consider a CSP (N, D, C) , a constraint $c_{ij} \in C$ functional on j , and a constraint $c_{jk} \in C$. To substitute i for j in c_{jk} , using c_{ij} , is to get a new CSP where c_{jk} is replaced by $c'_{ik} = c_{ik} \cap (c_{jk} \circ c_{ij})$. The variable i is called the substitution variable.

Property 1. Given a CSP (N, D, C) , a constraint $c_{ij} \in C$ functional on j , and a constraint $c_{jk} \in C$, the new problem obtained by substituting i for j in c_{jk} is equivalent to (N, D, C) .

Based on variable substitution, we can eliminate a variable from a problem so that no constraint will be on this variable (except the functional constraint used to substitute it).

Definition 2. Given a CSP (N, D, C) and a constraint $c_{ij} \in C$ functional on j , to eliminate j using c_{ij} is to substitute i for j , using c_{ij} , in every constraint $c_{jk} \in C$ ($k \neq i$).

Given a functional constraint c_{ij} of a CSP (N, D, C) , let C_j be the set of all constraints involving j , except c_{ij} and c_{ji} . The elimination of j using c_{ij} results in a new problem (N, D, C') where $C' = (C - C_j) \cup \{c'_{ik} \mid c'_{ik} = (c_{jk} \circ c_{ij}) \cap c_{ik}, c_{jk}, c_{ik} \in C\} \cup \{c'_{ik} \mid c'_{ik} = c_{jk} \circ c_{ij}, c_{jk} \in C, c_{ik} \notin C\}$.

In the new problem, there is only one constraint c_{ij} on j and thus j can be regarded as being “eliminated”. By Property 1, the variable elimination preserves the solution space of the original problem.

Property 2. Given a CSP (N, D, C) and a functional constraint $c_{ij} \in C$, the new problem (N, D, C') obtained by the elimination of variable j using c_{ij} is equivalent to (N, D, C) .

We now extend variable elimination to general CSPs with functional and non-functional constraints. The idea of variable elimination can be used to reduce a CSP to the following canonical functional form.

Definition 3. A CSP (N, D, C) is in canonical functional form if for any constraint $c_{ij} \in C$ functional on j , the following conditions are satisfied: 1) if c_{ji} is also functional on i (i.e., c_{ij} is bi-functional), either i or j is not constrained by any other constraint in C ; 2) otherwise, j is not constrained by any other constraint in C .

In a canonical functional form CSP, the functional constraints form disjoint star graphs. A star graph is a tree where there exists a node, called the *center*, such that there is an edge between this center node and every other node. We call the variable at the center of a star graph, a *free variable*, and other variables in the star graph *eliminated variables*. The constraint between a free variable i and an eliminated variable j is functional on j , but it may or may not be functional on i . In the special case that the star graph contains only two variables i and j and c_{ij} is bi-functional, any one of the variables can be called a free variable while the other called an eliminated variable.

If a CSP is in canonical functional form, all functional constraints and the eliminated variables can be *ignored* when we try to find a solution for this problem. Thus, to solve a CSP (N, D, C) in canonical functional form whose non-eliminated variables are NE , we only need to solve a smaller problem (NE, D', C') where D' is the set of domains of the variables NE and $C' = \{c_{ij} \mid c_{ij} \in C \text{ and } i, j \in NE\}$.

```

algorithm Variable-Elimination(inout  $(N, D, C)$ , out consistent) {
   $L \leftarrow N$ ;
  while ( There is  $c_{ij} \in C$  functional on  $j$  where  $i, j \in L$  and  $i \neq j$  ){
    // Eliminate variable  $j$ ,
  1.    $C \leftarrow \{c'_{ik} \mid c'_{ik} \leftarrow (c_{jk} \circ c_{ij}) \cap c_{ik}, c_{jk} \in C, k \neq i\} \cup (C - \{c_{jk} \in C \mid k \neq i\})$ ;
  2.    $L \leftarrow L - \{j\}$ ;
      Revise the domain of  $i$  wrt  $c_{ik}$  for every neighbour  $k$  of  $i$ ;
      if ( $D_i$  is empty) then { consistent  $\leftarrow$  false; return }
    }
  consistent  $\leftarrow$  true;
}

```

Fig. 1. A variable elimination algorithm to transform a CSP into a canonical functional form

Any CSP with functional constraints can be transformed into canonical functional form by variable elimination using the algorithm in Fig. 1. Given a constraint c_{ij} functional on j , Line 1 of the algorithm substitutes i for j in all constraints involving j .

Theorem 1. *Given a CSP (N, D, C) , Variable-Elimination transforms the problem into a canonical functional form in $\mathcal{O}(n^2 d^2)$*

A good ordering of the variables to eliminate will result in a faster algorithm. The intuition is that once a variable i is used to substitute for other variables, i itself should not be substituted by any other variable later.

Example. Consider a CSP with functional constraints c_{ij} and c_{jk} . Its constraint graph is shown in Fig. 2 where a functional constraint is represented by an arrow. If we eliminate k and then j , we first get c_{jl} and c_{jm} , and then get c_{il} and c_{im} . Note that j is first used to substitute for k and later is substituted by i . If we eliminate j and then k , we first get c_{ik} , and then get c_{il} and c_{im} . In this way, we reduce the number of compositions of constraints. $\square$

Given a CSP $P = (N, D, C)$, P^F is used to denote its directed graph (V, E) where $V = N$ and $E = \{(i, j) \mid c_{ij} \in C \text{ and } c_{ij} \text{ is functional on } j\}$.

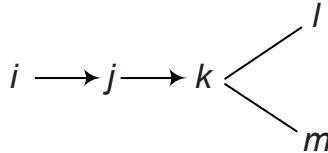


Fig. 2. The constraint graph of a CSP with functional constraints c_{ij} and c_{jk} and non-functional constraints c_{kl} and c_{km}

Definition 4. Given a directed graph (V, E) , a sequence of the nodes of V is a functional elimination ordering if for any two nodes i and j , i before j in the sequence implies that there is a path from i and j . A functional elimination ordering of a CSP problem P is a functional elimination ordering of P^F .

Given a directed graph G , a functional elimination ordering can be found by 1) finding all the strongly connected components of G , 2) modifying G by taking every component as one vertex with edges changed and/or added accordingly, 3) finding a topological ordering of the nodes in the new graph, and 4) replacing any vertex v in the ordering by any sequence of the vertices of the strongly connected component represented by v .

The algorithm **Linear-Elimination** in Fig. 3 first finds a functional elimination ordering (Line 1). Line 4 and 6 are to *process* all the variables in O . Every variable i of O is *processed* as follows: i will be used to substitute for all the variables *reachable* from i through constraints that are functional in C^0 and still exist in the current C . Those constraints are called *qualified* constraints. Specifically, L initially holds the immediate reachable variables through qualified constraints (Line 8). Line 9 is a loop to eliminate all variables reachable from i . The loop at Line 11 is to eliminate j using i from the current C . In this loop, if a constraint c_{jk} is qualified (Line 14), k is reachable from i through qualified constraints. Therefore, it is put into L (Line 15).

```

algorithm Linear-Elimination(inout  $(N, D, C)$ ) {
  1. Find a functional elimination ordering  $O$  of the problem;
  2. Let  $C^0$  be  $C$ ; any  $c_{ij}$  in  $C^0$  is denoted by  $c_{ij}^0$ ;
  3. For each  $i \in N$ , it is marked as not eliminated;
  4. while ( $O$  is not empty) {
    Take and delete the first variable  $i$  from  $O$ ;
  6.   if ( $i$  is eliminated) continue;
  8.    $L \leftarrow \{j \mid (i, j) \in C \text{ and } c_{ij}^0 \text{ is functional}\}$ ;
  9.   while ( $L$  not empty) {
    Take and delete  $j$  from  $L$ ;
  11.    for any  $c_{jk} \in C - \{c_{ji}\}$  { // Substitute  $i$  for  $j$  in  $c_{jk}$ ;
       $c'_{ik} \leftarrow c_{jk} \circ c_{ij} \cap c_{ik}$ ;
       $C \leftarrow C \cup \{c'_{ik}\} - \{c_{jk}\}$ ;
  14.      if ( $c_{jk}^0$  is functional) then
  15.         $L \leftarrow L \cup \{k\}$ ;
    }
  16.   Mark  $j$  as eliminated;
} // loop on  $L$ 
} // loop on  $O$ 
} // end of algorithm

```

Fig. 3. A variable elimination algorithm of complexity $O(ed^2)$

Theorem 2. *Given a CSP problem, the worst case time complexity of Linear-Elimination is $O(ed^2)$ where e is the number of constraints and d the size of maximum domain in the problem.*

For the algorithm **Linear-Elimination**, we have the following nice property.

Theorem 3. *Consider a CSP with both functional and non-functional constraints. If there is a variable of the problem such that every variable of the CSP is reachable from it in P^F , the satisfiability of the problem can be decided in $\mathcal{O}(ed^2)$ using Linear-Elimination.*

For a problem with the property given in the theorem above, its canonical functional form becomes a star graph. So, any value in the domain of the free variable is extensible to a solution if we add (arc) consistency enforcing during **Linear-Elimination**. The problem is not satisfiable if a domain becomes empty during the elimination process. In contrast to our algorithm **Linear-Elimination**, using an arc consistency based (bi-)functional algorithm [4] or view based [3] implementations of propagators for functional constraints may not be able to achieve global consistency in general.

3 Conclusion

We have introduced a variable substitution method to reduce a problem with both functional and non-functional constraints. Compared with the previous work on bi-functional and functional constraints, the new method is not only conceptually simple and intuitive but also reflects the fundamental property of functional constraints. Our experiments (not included here) also show that variable elimination can significantly improve the performance of a general solver in dealing with functional constraints.

References

1. David, P.: When functional and bijective constraints make a CSP polynomial. In: Intl. Joint Conf. on Artificial Intelligence, pp. 224–229 (1993)
2. David, P.: Using pivot consistency to decompose and solve functional CSPs. *Journal of Artificial Intelligence Research* 2, 447–474 (1995)
3. Schulte, C., Tack, G.: Views and iterators for generic constraint implementations. *Constraint Solving and Constraint Logic Programming*, 118–132 (2005)
4. Van Hentenryck, P., Deville, Y., Teng, C.M.: A generic arc-consistency algorithm and its specializations. *Artificial Intelligence* 58, 291–321 (1992)
5. Zhang, Y., Yap, R.H.C., Jaffar, J.: Functional elimination and 0/1/all constraints. In: *Natl. Conf. on Artificial Intelligence*, pp. 275–281 (1999)

Crossword Puzzles as a Constraint Problem

Anbulagan and Adi Botea

NICTA* and Australian National University, Canberra, Australia
{anbulagan,adi.botea}@nicta.com.au

Abstract. We present new results in crossword composition, showing that our program significantly outperforms previous successful techniques in the literature. We emphasize phase transition phenomena, and identify classes of hard problems. Phase transition is shown to occur when varying problem parameters, such as the dictionary size and the number of blocked cells on a grid, of large-size realistic problems.

1 Introduction

In this paper we propose new ideas in solving crossword puzzles presented in a hybrid model with two viewpoints, one containing cell variables and the other containing word slot variables. We discuss an architecture where search and nogood learning exploit the strenghts of each viewpoint. Our program solves more crossword problems than previous successful techniques in the literature. We present the program performance on a collection of realistic puzzles, which has been used in previous studies [1,2,3].

We also analyze the behaviour of the crossword domain in detail. We emphasize phase transition phenomena and identify classes of hard problems. We discuss how the structure of a problem is affected by varying parameters such as the size of a dictionary and the number of blocked cells on a grid. Such structural changes exhibit phase transition phenomena. Unlike previous CSP contributions on phase transition (e.g., [4,5]), which always consider randomly generated problems, we experiment with large-size realistic problems.

The earliest contribution to crossword grid composition reported in the literature belongs to Mazlack [6]. In that work, a grid is filled with a letter-by-letter approach. Ginsberg *et al.* [7] focus on an approach that adds an entire word at a time. The list of matching words for each slot is updated dynamically based on the slot positions already filled with letters. Meehan and Gray [8] compare a letter-by-letter approach against a word-by-word encoding and conclude that the latter is able to scale up to harder puzzles.

Cheesman *et al.* [9] showed that the hard instances in NP-hard problems often exhibit a phase transition phenomenon. The classical phase transition in SAT [10] is observed when the ratio between the number of variables and the number of clauses varies.

* 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.

2 Encoding Crosswords into CSP

Formally, a crossword puzzle consists of a grid size, a fixed configuration of blocked cells, and a dictionary. The problem is to fill the grid with words from the dictionary. No word can be placed more than once on the grid. As in [1,2], we adopt a hybrid encoding where both cells and word slots are used as CSP variables. Consider a slot s and its i -th cell c . A binary *intersection* constraint enforces that the letter assigned to c is the same as the i -th letter of the word assigned to s . Each pair of same-length slots defines a *repetition* constraint, which forbids to place the same word into two distinct slots. The hybrid encoding can be obtained from a basic model with only cell variables by applying the hidden variable transformation. It can also be seen as two combined viewpoints, one with *high-level* (or *dual*) slot variables and one with *low-level* cell variables.

3 COMBUS: A Crossword Composer

The solving engine highlighted in this section exploits the hybrid problem encoding by instantiating only dual variables in search and by using only low-level variables as part of nogood records. An instantiation to a dual variable in search is a macro of low-level instantiations. Macro-actions can reduce the depth of a search at the cost of increasing the branching factor per node (the utility problem). When the non-binary constraints that generated the dual variables are reasonably tight, the utility problem does not appear to be an issue. Since each dual variable contains several low-level variables, the search tree depth is reduced considerably. The branching factor can be kept low due to constraint propagation and to preferring variables with small domains to be instantiated.

In building nogoods we exploit that, in crosswords and other real-life, structured problems, a partial assignment to the dual variables can partition the uninstantiated variables into *clusters* that do not interact via common constraints. In crosswords, clustering is possible if we ignore the repetition constraints, which can connect slots on any two grid areas. A cluster is initialized to a seed slot and extended iteratively up to a fix point by adding new uninstantiated dual variables (i.e., empty or partially filled slots) that intersect the cluster.

To extract a nogood from a deadlocked node n , a *deadlock cluster* is built around the variable selected for instantiation, whose possible assignments have been invalidated either through further search or statically (via arc-consistency propagation). If n is not a leaf node and its subtree has explored instantiating variables in other clusters too, an additional condition is needed to ensure that the deadlock is contained within the current cluster, being independent of the rest of the problem. Specifically, we require that no parts of n 's subtree have been pruned because of deadlocks that involve variables from other clusters. Then the instantiated cells in the deadlock cluster are a superset of a nogood.

Nogoods are stored in a database and used for pruning in the future. As nogood learning ignores repetition constraints, nogoods might be built that are actually part of a correct solution, giving up the method completeness. As repetition constraints are handled in the main search, all found solutions are correct.

4 Empirical Results on Composer Performance

We examine the performance of COMBUS on a suite of problems introduced in [1] and subsequently used in other studies [2,3]. The suite contains ten grids of each of the following sizes: 5x5, 15x15, 19x19, 21x21 and 23x23. There are two dictionaries: the smaller one, called “words”, contains 45,000 words and “UK” contains 220,000 words. Each combination of a grid and a dictionary creates a problem instance. Thus, we obtain a set of 100 instances.

Table 1. Time (T) in seconds and expanded nodes (N) of the 80 instances. The 5x5 problems are too easy and their details are not presented. A dash means that no solution was found in the given time limit.

Inst ance	15x15 grids				19x19 grids				21x21 grids				23x23 grids			
	words		UK		words		UK		words		UK		words		UK	
	T	N	T	N	T	N	T	N	T	N	T	N	T	N	T	N
01	86	83	287	71	56	118	320	123	113	471	851	128	1	0	209	157
02	11	75	216	74	23	96	429	109	134	143	1133	141	81	178	697	172
03	41	71	239	73	34	315	245	108	76	139	624	130	455	12121	1185	160
04	18	304	290	66	75	127	738	116	740	15367	525	139	180	1700	715	162
05	25	65	354	70	13	112	121	115	56	238	288	132	105	178	707	170
06	84	1678	521	64	96	126	313	121	99	140	560	137	—	—	462	227
07	146	118	548	65	34	118	296	126	128	152	571	136	86	241	572	162
08	41	76	196	78	62	122	396	120	68	145	551	142	320	7842	766	156
09	25	77	210	75	23	121	342	121	64	141	479	138	689	18109	366	160
10	114	506	462	65	14	119	120	121	—	—	857	119	—	—	680	135

In Table 1, we present the results obtained by switching on the nogood recording procedure. COMBUS solves 97 instances in less than 20 minutes per instance on a 2.4GHz Intel Duo Core. The results show that problems corresponding to the “UK” dictionary require few node expansions. The number of expanded nodes is close to the depth of the search tree, indicating that, often, solutions are found with no backtracking. The “words” dictionary generates harder problems with respect to the number of nodes. All three unsolved problems are in this category. In general, our results are significantly better than the results presented in [1,2,3].

5 Empirical Results on Crossword Phase Transition

The experiments were run using the COMBUS engine. The incomplete method of nogood learning is switched off, to ensure that an instance reported as UNSAT has no solution indeed. In this study, we vary the dictionary size and the percentage of blocked cells.

5.1 Changing Dictionary Size

Here, we study the problem hardness and the phase transition by varying the dictionary size. The full dictionary has 220,000 words [1]. Subsets of it are

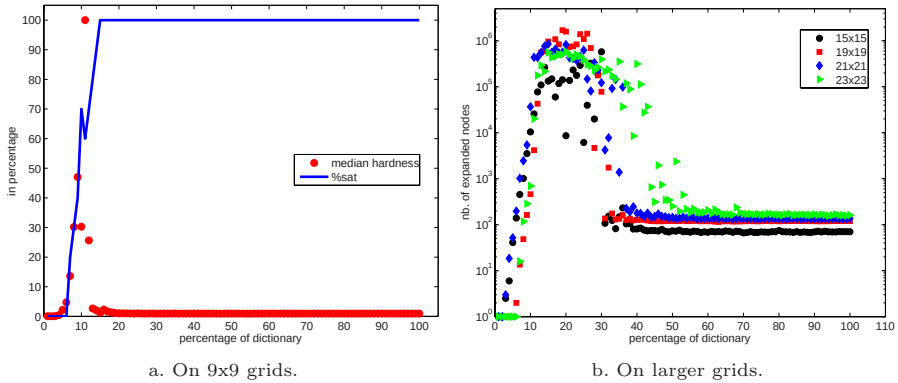


Fig. 1. Phase transition and problem hardness when percentage of dictionary varies

obtained by adding 2,200 words (1%) at a time. We created a set of 10 9x9 grids, having 12 blocked cells each, to be able to solve them all in a reasonable time and compute the percentage of SAT instances. Figure 1a shows the problem hardness (the median expanded nodes) and the percentage of SAT instances on 9x9 grids. The hard instances occur in the phase transition region, where the dictionary ranges from about 10,000 to 35,000 words.

Figure 1b presents the median expanded nodes for larger grids. The data contain ten grids of each of the following sizes: 15x15, 19x19, 21x21 and 23x23 [1]. The percentage of blocked cells is around 15 to 20%. Here, the time limit is set to 5 hours per problem. The results clearly show easy-hard-easy transitions for each grid collection. The hard region size increases with the grid size. The 15x15 data set shows a hard region from around 22,000 to 66,000 words, whereas hard 23x23 problems occur from about 22,000 to 110,000 words. Problems with a small dictionary size have no solution. The search cost to prove this is low. As we progress along the horizontal axis, problems become solvable. Finding a solution is hard at the beginning of the SAT range. The search effort decreases as larger and larger dictionaries are used.

5.2 Changing Number of Blocked Cells

In this experiment we use the 23x23 grids and the 220,000 words dictionary. To vary the number of blocked cells, we started from a configuration with 192 blocked cells (36% of all cells) placed symmetrically on the grid. We gradually removed pairs of symmetrical cells until an entirely blank grid was obtained.

Table 2 presents data showing the occurrence of phase transition phenomena when solving the crossword puzzles on 23x23 grids. In the table, “Time” represents the mean runtime in seconds and “Total” represents the number of instances in the given solution region. There are three distinct experiments, one with the full dictionary (D=100%), one with half of it (D=50%), and one with

Table 2. The phase transition of crossword puzzles on 23x23 grids

Solution Region	D=100%			D=50%		
	blocked cells	Total	Time	blocked cells	Total	Time
UNSAT	0-36	19	60	0-48	25	1835
HARD	38-66	15	>86400	50-72,76	13	>86400
SAT	68-192	60	734	74, 78-192	56	228

Solution Region	D=30%		
	blocked cells	Total	Time
UNSAT	0-48	25	13
HARD	50-80,84,104,110,114,116	21	>86400
SAT	82,86-100,106,108,112,118-192	48	516

D=30%. The data show that there are more hard problems for the D=30% case. As the dictionary size gets smaller, the number of UNSAT instances increases.

The problem set is partitioned into three regions. Problems with few blocked cells have no solutions and are easy to solve. We call this the UNSAT region. Instances with a large number of blocked cells have solutions that are easy to compute (SAT region). Since we work with large problems, the hard instances between the previous two regions cannot be solved in 24 hours (HARD region).

References

1. Beacham, A., Chen, X., Sillito, J., van Beek, P.: Constraint Programming Lessons Learned from Crossword Puzzles. In: Stroulia, E., Matwin, S. (eds.) Canadian AI 2001. LNCS (LNAI), vol. 2056, pp. 78–87. Springer, Heidelberg (2001)

2. Samaras, N., Stergiou, K.: Binary Encodings of Non-binary Constraint Satisfaction Problems: Algorithms and Experimental Results. JAIR, 641–684 (2005)

3. Katsirelos, G., Bacchus, F.: Generalized NoGoods in CSPs. In: Proc. of 20th AAAI, pp. 390–396 (2005)

4. Smith, B.: Phase Transition and the Mushy Region in Constraint Satisfaction Problems. In: Proc. of 11th ECAI, pp. 100–104 (1994)

5. Gent, I.P., MacIntyre, E., Prosser, P., Walsh, T.: Scaling Effects in the CSP Phase Transition. In: Proc. of 1st CP, pp. 70–87 (1995)

6. Mazlack, L.J.: Computer Construction of Crossword Puzzles Using Precedence Relationships. Artificial Intelligence, 1–19 (1976)

7. Ginsberg, M.L., Frank, M., Halpin, M.P., Torrance, M.C.: Search Lessons Learned from Crossword Puzzles. In: Proc. of 8th AAAI, pp. 210–215 (1990)

8. Meehan, G., Gray, P.: Constructing Crossword Grids: Use of Heuristics vs Constraints. In: Proc. of R & D in Expert Systems, pp. 159–174 (1997)

9. Cheeseman, P., Kanefsky, B., Taylor, W.M.: Where the Really Hard Problems Are. In: Proc. of 12th IJCAI, pp. 163–169 (1991)

10. Gent, I.P., Walsh, T.: The SAT Phase Transition. In: Proc. of 11th ECAI, pp. 105–109 (1994)

Recent Hybrid Techniques for the Multi-Knapsack Problem

Carlos Diego Rodrigues^{1,2}, Philippe Michelon¹, and Manoel B. Campêlo²

¹ Université d'Avignon, Laboratoire d'Informatique de Avignon, F84911 Avignon
{carlos-diego.rodrigues, philippe.michelon}@univ-avignon.fr

² Universidade Federal do Ceará, Campus do Pici, Brazil
mcampelo@lia.ufc.br

1 Introduction

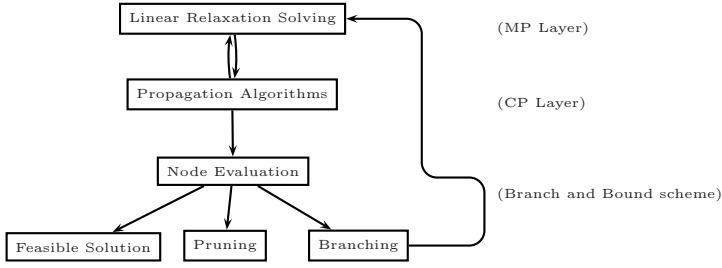
The cooperation between the Mathematical Programming (MP) and Constraint Programming (CP) paradigms has come to focus in the recent years. The impossibility of solving bigger instances alone and the apparently complementarity between the two approaches pushes towards further researches on hybrid methods. We can define a linear binary optimization problem as $(P)\{\min c.x : Ax \leq b, x \in B^n\}$, where x is a binary variable, $A \in R^{m \times n}$ is the constraint matrix, $b \in R^m$ is the right-hand side vector and $c \in R^n$ is the objective function.

Commonly, when authors are studying such a general problem, they start by studying the case where $A \geq 0$, $b \geq 0$ and $c \geq 0$, which is the Multi-Knapsack Problem (*MKP*). For example, Osorio and Glover [7] developed a new paradigm to introduce the logical cuts into linear and integer programming and applied to the (*MKP*). Hooker [4] showed that MP could benefit from the domain reduction, base idea of CP also in the (*MKP*).

In this work, we revisit the cooperative scheme from Oliva, Artigues and Michelon [6] and what has been done over the years to implement it efficiently in Section 2. Then, in Section 3, we define a global constraint for the MKP, as well as its filtering algorithm, and show how it can be inserted into the provided framework. The Section 4 shows some preliminary results comparing the enhanced framework to its previously best version and the to the commercial solver ILOG CPLEX.

2 The Cooperative Scheme

Introduced in [6], this cooperative scheme between the CP and MP is based on adding to a mathematical programming enumeration a new layer when evaluating a node. That means using the constraint programming filtering algorithms to deduce constraints over the domains of the variables, hence possibly changing the enumeration variable state or proving the infeasibility of the node.



2.1 Reduced Costs Implicit Constraint

The first type of propagation algorithm we can add to the CP layer in the cooperative scheme was presented in [6]. Based on the information given by the linear relaxation of our model, the reduced costs, this method assembles a new constraint which can be used to tighten the domains of the variables.

Given a problem (*MKP*) and a lower bound for this problem, L , if we consider N_0 and N_1 as the set of indices of the non-basic variables which have received the value 0 and 1, respectively, in the solution of the linear relaxation of (*MKP*) and U to be the value of this solution, as showed in [10], the reduced cost implicit constraint can be simplified in our case to the following expression:

$$\sum_{j \in N_0} |c_j| x_j + \sum_{j \in N_1} |c_j| (1 - x_j) \geq U - L \quad (1)$$

If a variable x_j is given a different value (by branching or any other fixing process) from what it was given by the linear relaxation at a certain node, then the left hand side of (1) will be decreased by $|c_j|$. In other words, filtering this constraint means checking whether or not the value of the variable x_j can change from the linear relaxation solution without violating (1). It is important to further remark that at each node of a branch and bound enumeration tree, the reduced cost, for each variable, changes, hence resulting in a new reduced costs implicit constraint that should be taken into account as well as the constraints from the parent nodes in the enumeration tree, which can be filtered backwards [10].

2.2 Enumeration Framework

Conceived in [10] this type of branch-and-bound enumeration tries to take the advantage from the reduced cost propagation algorithm with its branching strategy. This enumeration is divided in three distinct phases: lower bound calculation, cardinality bounding and finally the branch-and-bound.

The first phase consists in calculating a lower bound for the problem. Having this value, we proceed, following the remarks in [9], to calculate a lower and an upper bound to the number of items in the optimal solution k_{min} and k_{max} . Then we separate the root node into $\lfloor k_{max} \rfloor - \lceil k_{min} \rceil + 1$ nodes where we add a cardinality constraint (phase 2). Then, the model to be solved at each node of the tree from now on is the following:

$$(|MKP|_k) \max \sum_{j=1}^n c_j x_j \quad (2a)$$

$$\text{s.a: } \sum_{j=1}^n a_{ij} x_j \leq b_i, \quad i = 1, \dots, m \quad (2b)$$

$$\sum_{j=1}^n x_j = k \quad (2c)$$

$$x_j \in \{0, 1\}, \quad j = 1, \dots, n \quad (2d)$$

Having all these presets, we start phase 3, where the cooperative scheme is applied. For the branching strategy, first we order the variables from the biggest to the smallest reduced cost. Then, for the first variable we branch fixing it to the opposite value to the one received by the linear relaxation solution and all other variables are free. Note that this will take the maximum amount of reduced cost from the left hand side of (1). The second branch fixes the first variable to its value in the linear relaxation solution and fixes the second variable in the order to its opposite value. The third branch does the same strategy until we have no more non-basic variables free. The last branch has only basic variables as free variables. Here we use a depth-first search algorithm to enumerate the possibilities for these variables.

3 A New Global Constraint

Once we have a cardinality constraint, we can combine it to each of the original constraints of the problem to define the "knapSum" global constraint, since it seems interesting to take both into account at the same time, considering the integrality of the problem.

$$\text{knapSum}(a_i, b_i, k, x) \Leftrightarrow \left\{ \underline{b}_i \leq \sum_{j=1}^n a_{ij} x_j \leq \bar{b}_i; \quad \sum_{j=1}^n x_j = k \right\} \quad (3)$$

The idea of the filtering is to "force" the limits of a constraint, respecting the number of variables in the solution, determined by k . If the bounds are respected, we can then proceed to a second phase of the process, where we fix the variables, iteratively, leading to a smaller, or possibly infeasible problem (in this case, the node is cut off).

Let's first assume that for a constraint $\text{knapSum}(a_i, b_i, k, x)$ the coefficients a_{ij} are ordered in a non-increasing order. We can make an estimation of the biggest and the smallest value we can have for this constraint by summing up the k biggest/smallest values in it. Then, if

$$\sum_{j=1}^k a_{ij} < \underline{b}_i \quad \text{or} \quad \sum_{j=n-k+1}^n a_{ij} > \bar{b}_i, \quad (4)$$

the problem is infeasible.

If these bounds are respected, then we try to fix the variables corresponding to the maximum and minimum values on 0, separately, and re-evaluate the

expressions (4). If we find an infeasibility, then we can fix that variable to 1 from now on. If the bounds were respected, then we can try to fix them to 1 and check (4) again. If any fixation was found in this process, we can iteratively repeat it with the new maximum/minimum variable.

Algorithm: KnapSum Filtering

1. Calculate $\bar{S} = \sum_{j=1}^k a_{ij}$.
2. Calculate $\underline{S} = \sum_{j=n-k}^n a_{ij}$.
3. If $\bar{S} < \underline{b}_i$ or $\underline{S} > \bar{b}_i$ then the problem is infeasible.
4. Repeat
 - (a) If $\bar{S} - a_{i1} + a_{i,k+1} < \underline{b}_i$ then
 - i. $\underline{S} \leftarrow \underline{S} - a_{i,n-k} + a_{i1}$
 - ii. $k \leftarrow k - 1$
 - iii. $n \leftarrow n - 1$
 - iv. Fix x_1 to 1
 - (b) If $\bar{S} - a_{ik} + a_{in} < \underline{b}_i$ then
 - i. $\underline{S} \leftarrow \underline{S} - a_{in} + a_{i,n-k-1}$
 - ii. $n \leftarrow n - 1$
 - iii. Fix x_n in 0
 - (c) If $\underline{S} - a_{in} + a_{i,n-k-1} > \bar{b}_i$ then
 - i. $\bar{S} \leftarrow \bar{S} - a_{ik} + a_{in}$
 - ii. $k \leftarrow k - 1$
 - iii. $n \leftarrow n - 1$
 - iv. Fix x_n in 1
 - (d) If $\underline{S} - a_{i,n-k} + a_{i1} > \bar{b}_i$ then
 - i. $\bar{S} \leftarrow \bar{S} - a_{i1} + a_{i,k+1}$
 - ii. $n \leftarrow n - 1$
 - iii. Fix x_1 in 0
5. Until there's no fixation.

As it is shown, this global constraint can be directly applied to the CP Layer of our cooperative scheme, just after the reduced costs constraint. Then, if an infeasibility is found, the node is fathomed. Else, if a variable is attributed a different value from what it had on the linear relaxation, we can re-solve the linear problem, hence getting new reduced costs and making all the cycle again.

There's also the possibility of using the "knapSum" filtering algorithm to deduce valid cuts to the problem at a given node. Let's consider the sets N_0 and N_1 as defined in Section 2.1. If the "knapSum" returns an infeasibility for (N_0, N_1) , then the following cut can be added to the problem to forbid that state:

$$\sum_{j \in N_0} x_j + \sum_{j \in N_1} (1 - x_j) \geq 1 \quad (5)$$

Also, if a variable x_l is fixed by this filtering a similar valid cut can be found by adding l to the corresponding opposite set x_l was fixed (eg. add l to N_0 if x_l was fixed to 1), meaning the complementary state of variable is infeasible for the problem.

4 Numerical Results and Conclusion

A table of results is shown, comparing the times (in seconds) of our strategies, adding the knapSum to fix variables or prove infeasibilities and the version that add the cuts derived from the knapSum filtering. We also compare our strategies with a method that implements the same enumeration framework (VBV [10])

and the commercial solver ILOG CPLEX v. 10.1 for the instances found in the OR-Library[1]. Our objective was to see how much one can gain when adding this new form of cooperation between the paradigms. We present the arithmetic average of our results for 3 sets of instances. The name of the instance set tells how many constraints and variables for each one in the form $m.n$. Each set has 30 instances. The tests were realized on a PC Celeron 3GHz with 1GB of RAM.

Table 1. Summary of Results - times expressed in seconds

Set	CPLEX	VBV	KPS	+Cuts
5.250	244.02	99.73	1.96	1.90
5.500	4219.56	926.7	31.43	29.41
10.250	N/A	8536.26	4718.23	4788.56

The insertion of the knapSum global constraint as a fixing algorithm and as a cut-generation algorithm on the CP Layer into this cooperative framework has shown an improvement over the total time of resolution of the best known solutions and may lead to further improvements.

References

1. Beasley, J.E.: OR-Library Instances, <http://people.brunel.ac.uk/%7Emastjjb/jeb/info.html>
2. 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)
3. Fréville, A., Plateau, G.: An efficient preprocessing procedure for the multidimensional 0-1 knapsack problem. *Discret Appl Math.* 49, 189–212 (1994)
4. Hooker, J.N.: Logic, Optimization and Constraint Programming (manuscript, 2001)
5. Milano, M., Wallace, M.: Integrating operations research in constraint programming. *4OR* 4(3), 175–219 (2006)
6. Oliva, C., Michelon, P., Artigues, C.: Constraint and Linear Programming: Using Reduced Costs for solving the Zero/One Multiple Knapsack Problem. In: International Conference on Constraint Programming, Proceedings of the workshop on Cooperative Solvers in Constraint Programming(CoSolv 2001), Paphos, Cyprus, pp. 87–98 (2001)
7. Osorio, M.A., Glover, F.: Logic cuts using surrogate constraint analysis in the multidimensional knapsack problem. In: CPAIOR 2001 (2001)
8. Osorio, M.A., Glover, F., Hammer, P.: Cutting and Surrogate Constraint Analysis for Improved Multidimensional Knapsack Solutions. *Annals of Operations Research* 117, 71–93 (2002)
9. Vasquez, M., Hao, J.K.: A Hybrid Approach for the 01 Multidimensional Knapsack problem. *IJCAI*, 328–333 (2001)
10. Vimont, Y., Boussier, S., Vasquez, M.: Reduced costs propagation in an efficient implicit enumeration for the 01 multidimensional knapsack problem. *Journal of Combinatorial Optimization* 15(2), 165–178 (2008)

Edge Matching Puzzles as Hard SAT/CSP Benchmarks*

Carlos Ansótegui, Ramón Béjar, César Fernández, and Carles Mateu

Dept. of Computer Science, Universitat de Lleida, Spain
{carlos, ramon, cesar, carlesm}@diei.udl.cat

Abstract. Recently, edge matching puzzles, an NP-complete problem, have received, thanks to money-prized contests, considerable attention from wide audiences. This paper studies edge matching puzzles focusing on providing generation models of problem instances of variable hardness and on its resolution through the application of SAT and CSP techniques. From the generation side, we also identify the phase transition phenomena for each model. As solving methods, we employ both; SAT solvers through the translation to a SAT formula, and two ad-hoc CSP solvers we have developed, with different levels of consistency, employing generic and specialized heuristics. Finally, we conducted an extensive experimental investigation to identify the hardest generation models and the best performing solving techniques.

1 Introduction

The purpose of this paper is to introduce a new set of problems, edge matching puzzles, a problem that has been shown to be NP-complete [4], modelling them as SAT/CSP problems. These puzzles have recently received world wide attention with the publication of an edge matching puzzle with a money prize of 2 million dollars if resolved (*Eternity II*). Our contribution is threefold. First, we provide a method for generating edge matching puzzles. The proposed method is simpler and faster than other generators of hard SAT/CSP instances. Second, to our best knowledge, we provide the first detailed analysis of the phase transition phenomenon for edge matching puzzles in order to locate hard/easy puzzles. Third, we provide a collection of solving methods and a wide experimental evaluation, including SAT and CSP solving techniques.

2 Preliminary Definitions

A generic edge matching puzzle (GEMP) is a puzzle where we must place a set of tokens in a board following a simple rule. Tokens have four sides (called also half-edges), in our case for simplicity we assume square tokens, each of a different color or pattern. The rule to follow when placing tokens is that two tokens can be placed side by side iff adjacent half-edges are of the same color (or pattern), such when placed side by side they will form an edge with an unique color. Tokens in GEMP, unlike in easier variants as Tetravex, can be freely rotated. An example of such a puzzle is shown in Figure 1.

* Research partially supported by projects TIN2006-15662-C02-02, TIN2007-68005-C04-02 and José Castillejo 2007 program funded by the *Ministerio de Educación y Ciencia*.

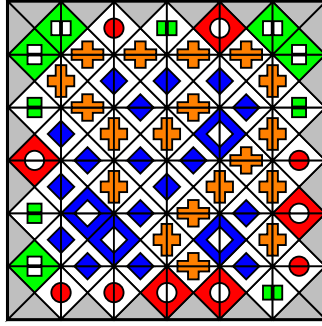


Fig. 1. 6x6 size two-set GEMP-F example with 4 frame colors and 3 inner colors

There are several variants of puzzles attending to the sets of colors employed on distinct areas of the puzzle. In this paper we deal with two types, that have a profound impact on hardness, **one-set GEMP-F** when colors can be used at any edge of the puzzle, and **two-set GEMP-F** when two disjoint sets of colors are used; one set for edges joining frame pieces and another set for any other edge as in Figure 1, where one can observe that colors joining frame pieces are different than the rest. Another variant is when there are no *frame* in the puzzle. As real-world puzzles (as in *Eternity II*¹) are usually framed puzzles and due to the interesting effect that the frame has on hardness this work deals with GEMP-F.

3 Generation Models

The general method for a solvable puzzle generator is as follows; one assigns colors to edges of puzzle pieces (assigning a color to both half-edges), and when all edges are colored, tokens are built from the existing color assignment. Extending this algorithm to generate framed puzzles is easy. First the inner part of the puzzle is generated (tokens without gray color), without taking into account the frame. Then colors are assigned to the half-edges of the frame adjacent to inner tokens, that are already determined by the inner tokens, and then half-edges that join tokens of the frame are filled with colors, randomly choosing either from the same set of colors used for the inner tokens (**one-set GEMP-F**) or from a second set of colors with no colors in common with the first set (**two-set GEMP-F**).

4 Solving Approaches

The following section details the methods used for solving edge matching puzzles used in this paper. We use two different approaches to the problem, solving it as a SAT formula and as a CSP. For both methods, state of the art solvers or ad-hoc solvers have been used, choosing the most efficient ones for our experimental results in the following sections.

¹ In fact, *Eternity II* is a two-set GEMP-F.

The advantage of the SAT approach is the availability of a wide variety of competitive SAT solvers that can be applied to our SAT encoding. The first SAT encoding we introduce is the *primal* encoding. A *primal* Boolean variable denoted as $p_{t,r,i,j}$ is true iff the token t with rotation r is *placed* at cell (i, j) . The primal constraints are of the form: a cell has exactly one token, one token placed exactly at one cell, a token matches its neighbors and token edges located at puzzle frame are gray colored. Similarly, we could think on an encoding just working on a set of dual variables, where the dual variables represent how the edges of the puzzle are colored. A *dual* Boolean variable denoted as $e_{c,d,i,j}$ is true iff the edge located at cell (i, j) at direction d is *colored* with color c . Finally, we sort of merge both encodings in what we call the *primal-dual* encoding, what increases the power of unit propagation. In order to keep the formula in a reasonable size, it is crucial to take special care at encoding the Cardinality Constraints.

Edge matching puzzles are easily modeled as CSP problems, with two basic sets of constraints, one set of constraints for neighboring relations, modelling the relation between half-edges and a set of global constraints modelling the fact that every token must be assigned to one variable. We have used two base algorithms for CSP solving, PLA (Partial Look-ahead) and MAC (Maintaining Arc-Consistency), and we have added specific improvements for increasing constraint propagation. Both algorithms have been tested with two variable selection heuristics, DOM (minimum domain) and CHESS. CHESS is a static variable heuristic that considers all the variables of the problem as if placed in a Chess board, and proceeds by choosing all 'black' variables following a spiral shaped order from the center towards the frame, and then repeats the same procedure with 'white' variables. That causes unitary variables (singletons) appear earlier.

With the MAC algorithm we have considered the inclusion of global (n-ary) constraints with powerful filtering algorithms for maintaining generalized arc-consistency (GAC). The most important n-ary constraint we have identified is the exactly- k constraint between the set of $2k$ half-edges with a same color. That is, in any solution this set of $2k$ half edges must be arranged in a set of k disjoint pairs of half-edges. With this aim, we use the symmetric alldiff constraint (that is formally equivalent to our exactly- k constraint), and its specialized filtering algorithm [7], that achieves GAC over this constraint in polynomial time. So, we define a symmetric alldiff constraint for each exactly- k constraint we have (one for each color). More specifically, we have a color graph that represents either already matched half-edges (that become disconnected from the rest of the graph) or half-edges that could be matched given the current domains of the unassigned variables. Observe that in order to be able to extend the current partial solution to a complete solution, a necessary condition is that any color graph must contain at least one perfect matching. If this is not the case for some color, we may backtrack. Moreover, using the filtering algorithm of Regin we can eliminate any edge that will not appear in any perfect matching of the graph (i.e. to maintain GAC) and discover forced partial matchings.

We have also considered maintaining GAC for the alldiff constraint over the set of position variables using the filtering algorithm of [6].

5 Experimental Results

We present experimental results and an analytical approach for the location of the Phase Transition on one-set and two-set GEMP-F models, as well as a solver performance comparison on the instances on the peak of hardness.

5.1 Model Hardness and Phase Transition

One-set and two-set GEMP-F present a hardness characterization depending on their constituent number of colors. As shown in Figure 2, an accurate selection of the number of colors increases the puzzle hardness by several orders of magnitude. While comparing results for one-set and two-set GEMP-F, it is worth to note that for the same puzzle size, two-set GEMP-F are harder. As detailed in [1], one can link this hard-

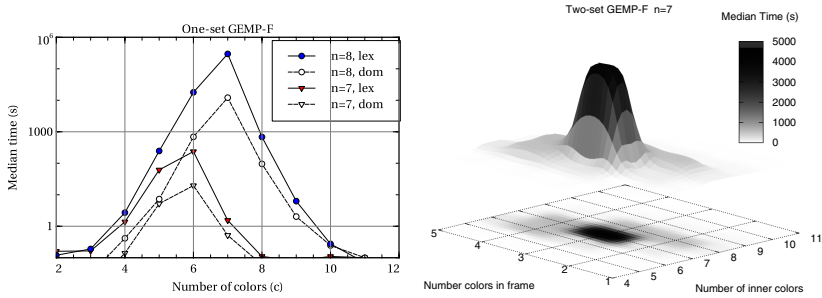


Fig. 2. Hardness characteristic for a one-set GEMP-F as a function of the number of colors (left). Hardness characteristic for a 7x7 two-set GEMP-F as a function of the number of colors. Minimum median time of PLA-CHESS and PLA-DOM (right).

ness characterization, on only satisfiable problems, with a phase transition effect when the backbone is considered, i.e. the number of variables that take the same value on all the solutions [5]. Backbone measurements for two-set GEMP-F problems shows a phase transition of the backbone fraction vs. c_f . From an analytical point of view, we can derive some expressions that predict the phase transition location. For the sake of tractability, we consider tokens generated randomly, unregarding adjacency constraints that give only SAT puzzles. Of course, this is only an approach, but experimental results and numerical evaluations agree for both models (see [2] for further details). It is worth to note that for $n = 16$ and $c_f = 5$, our model predicts a phase transition at $c_m = 17$, that is exactly the number of inner colors of the two-set GEMP-F puzzle used in the Eternity II contest.

5.2 SAT and CSP Solving Methods

We generated the SAT instances according to the two previously described SAT encodings. The complete SAT solvers we experimented with were: Minisat2 (v.061208-simp), siege(v.4.0), picosat(v.535) and satz. Minisat2 was the best performing SAT solver, and required to activate the option *polarity-mode=true*. For the state-of-the-art CSP solvers,

Table 1. Comparison of solving approaches for the one-set and two-set models. 100 instances per point.

Size ($n \times n$)	One-set GEMP-F			Two-set GEMP-F				
	7×7	8×8	6×6	7×7				
Colors inner[:frame]	6	7	6:2	6:4	7:2	7:3	7:4	8:2
PLA-LEX	235	$>2 \cdot 10^5$	-	-	-	-	-	-
PLA-DOM	20	12125	15	520	18193	9464	581	8387
PLA-CHESS	42	52814	0.5	5249	137	4181	6906	510
MAC+GAColor	90	23210	0.94	328	96	646	348	208
MAC+GAColor+CTadiff	92	22442	0.73	377	94	727	395	216
SAT(P)	1341	$>2 \cdot 10^5$	7.45	$>2 \cdot 10^4$	4418	$>2 \cdot 10^4$	7960	6465
SAT(PD)	34	117823	0.55	777	125	1785	682	359
MAC _b dom/deg	154	39742	19	2415	$>2 \cdot 10^4$	$>2 \cdot 10^4$	3307	$>2 \cdot 10^4$
Minion	413	$>2 \cdot 10^5$	125	3463	$>2 \cdot 10^4$	$>2 \cdot 10^4$	4675	$>2 \cdot 10^4$
Solver	Median Time (seconds)							

Minion and MAC_b dom/deg [3], we adapted the primal and primal-dual encodings taking into account variables with a domain greater than or equal to two (we only report results for the best encoding).

Table 1 shows median time results for one-set and two-set GEMP-F with distinct sizes and number of colors, solved with several techniques. These techniques are: (i) PLA CSP solvers with variable selection heuristics LEX, DOM and CHESS, explained above; (ii) MAC with filtering algorithm for Generalized Arc-Consistency for color graphs (GAColor), using CHESS heuristic, with and without GAC filtering for the all-diff over position variables (CTadiff); (iii) the Minisat2 (v.061208-simp) on the primal encoding SAT(P), and on the primal-dual encoding SAT(PD), and (iv) the state-of-the-art CSP solvers Minion and MAC_b dom/deg.

On one hand, the best performer for one-set GEMP-F is PLA-DOM meanwhile for two-set puzzles MAC+GAColor is the best one. It seems that the additional pruning effect of the GAColor filtering is powerful enough to pay off the additional time needed by such filtering in the two-set GEMP-F.

On the other hand, on PLA solvers for two-set GEMP-F, CHESS heuristic performs better than DOM when the number of frame colors is lower, and this could be because CHESS instantiates frame variables at the end of the search, and in those cases, the probability of finding a consistent frame is higher than when the number of frame colors is higher. About SAT solvers, the best performing encoding is the primal-dual encoding being quite competitive with the CSP approaches, but still with a poor scaling behaviour.

References

1. Achlioptas, D., Gomes, C., Kautz, H., Selman, B.: Generating satisfiable problem instances. In: Proceedings of the AAAI 2000, pp. 256–261. AAAI Press / The MIT Press (2000)
2. Ansótegui, C., Béjar, R., Fernández, C., Mateu, C.: Edge matching puzzles as hard SAT/CSP benchmarks (extended version). Technical Report TR-1-08, Dept. of Computer Science, Universitat de Lleida (2008), http://ccia.udl.cat/images/stories/Papers/techrep1_08.pdf

3. Bessière, C., Régin, J.-C.: MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In: *Principles and Practice of Constraint Programming*, pp. 61–75 (1996)
4. Demaine, E.D., Demaine, M.L.: Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity. *Graphs and Combinatorics* 23(s1), 195 (2007)
5. Monasson, R., Zecchinna, R., Kirkpatrick, S., Selman, B., Troyansky, L.: Determining computational complexity from characteristic phase transitions. *Nature* 400, 133–137 (1999)
6. Régin, J.-C.: A filtering algorithm for constraints of difference in CSPs. In: *Proceedings of the AAAI 1994*, pp. 362–367. AAAI Press / The MIT Press (1994)
7. Régin, J.-C.: The symmetric alldiff constraint. In: *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI 1999*, pp. 420–425. Morgan Kaufmann, San Francisco (1999)

Test Strategy Generation Using Quantified CSPs

Martin Sachenbacher and Paul Maier

Technische Universität München, Institut für Informatik
Boltzmannstraße 3, 85748 Garching, Germany
{sachenba,maierpa}@in.tum.de
<http://www9.in.tum.de>

Abstract. Testing is the process of stimulating a system with inputs in order to reveal hidden parts of the system state. We consider a variant of the constraint-based testing problem that was put forward in the model-based diagnosis literature, and consists of finding input patterns that can discriminate between different, possibly non-deterministic models. We show that this problem can be framed as a game played between two opponents, and naturally lends itself towards a formulation in terms of quantified CSPs. This QCSP-based formulation is a starting point to extend testing to the practically relevant class of systems with limited controllability, where tests consist of stimulation *strategies* instead of simple input patterns.

Keywords: Test generation, adversarial planning, quantified CSPs.

1 Introduction

As the complexity of technical devices grows, methods and tools to automatically check such systems for the absence or presence of faults become increasingly important. *Testing* asks whether there exist inputs (test patterns) to stimulate a system, such that a given fault will lead to observable differences at the outputs. For the domain of digital circuits, it has been shown how this question can be framed and solved as a constraint satisfaction problem [3,6].

In this paper, we consider constraint-based testing for a broader class of systems, where the outputs need not be deterministic. There are several sources for non-determinism in model-based testing of technical systems: in order to reduce the size of a model, for example, to fit it into an embedded controller [7,10], it is common to aggregate the domain of continuous variables into discrete, qualitative values such as 'low', 'medium', 'high', etc. A side-effect of this abstraction is that the resulting models can no longer be assumed to be deterministic functions, even if the underlying system behavior was deterministic [9]. Another source is the test situation itself: even in a controlled environment like an automotive test-bed, there are inevitably variables or parameters that cannot be completely controlled.

Struss [8] introduced a theory of testing for such general, constraint-based models: finding so-called *discriminating tests* (DTs) asks whether there exist

inputs that can unambiguously reveal or exclude the presence of a certain fault in a system, even if there might be several possible outputs for a given input. Generating DTs is a problem of considerable practical importance; the framework was applied to real-world scenarios from the domain of railway control and automotive systems [5]. [8] also provided a characterization of this problem in terms of relational (constraint-based) models, together with an ad-hoc algorithm to compute DTs.

In this paper, we build a bridge from this earlier, application-oriented work to newer developments in the area of constraint programming. In particular, we show how the DT problem can be conveniently formulated using quantified CSPs (QCSPs), which can be viewed as an extension of CSPs to multi-agent (adversarial) scenarios. This leads to three contributions: first, we observe that the problem of generating definitely discriminating tests is in a different complexity class than the testing problem described in [6] (Σ_2^P vs. NP). Second, formulating DT generation as an instance of QCSP solving enables to leverage recent progress in QCSP/QBF solvers in order to effectively compute DTs, instead of using ad-hoc algorithms as in [5,8]. Third, we show that our QCSP (adversarial planning) formulation of testing can be straightforwardly extended to systems with limited controllability, which require complex test strategies instead of simple input patterns and thus go beyond the framework in [8].

2 Discriminating Tests

We briefly review the theory of constraint-based testing of physical systems as introduced in [8]. Testing attempts to discriminate between hypotheses about a system – for example, about different kinds of faults – by stimulating the system in such a way that the hypotheses become observationally distinguishable. Formally, let $M = \bigcup_i M_i$ be a set of different models (hypotheses) for a system, where each M_i is a set of constraints over variables V . Let $I = \{i_1, \dots, i_n\} \subseteq V$ be the subset of input (controllable) variables, $O = \{o_1, \dots, o_m\} \subseteq V$ the subset of observable variables, and $U = \{u_1, \dots, u_k\} = V - (I \cup O)$ the remaining (uncontrollable and unobservable) variables. The goal is then to find assignments to I (input patterns) that will cause different assignments to O (output patterns) for the different models M_i .

Definition 1 (Discriminating Tests). *An assignment t_I to I is a possibly discriminating test (PDT), if for all M_i there exists an assignment t_O to O such that $t_I \wedge M_i \wedge t_O$ is consistent and for all M_j , $j \neq i$, $t_I \wedge M_j \wedge t_O$ is inconsistent. The assignment t_I is a definitely discriminating test (DDT), if for all M_i and all assignments t_O to O , if $t_I \wedge M_i \wedge t_O$ is consistent then for all M_j , $j \neq i$, it follows that $t_I \wedge M_j \wedge t_O$ is inconsistent.*

In the following, we restrict ourselves to the case where there are only two possible hypotheses, corresponding to normal and faulty behavior of the system. For example, consider the (simplified) system in Fig. 1. It consists of five variables x, y, z, u, v , where x, y, z are input variables and v is an output variable, and two

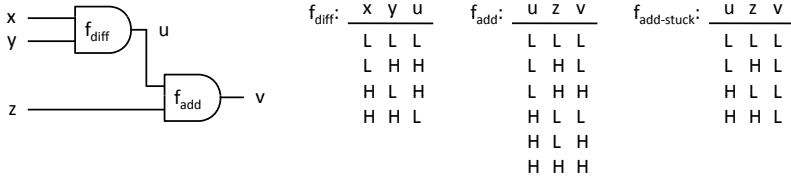


Fig. 1. Circuit with a possibly faulty adder

components that compare signals (x and y) and add signals (u and z). The signals have been abstracted into qualitative values 'low' (L) and 'high' (H); thus, for instance, values L and H can add up to the value L or H, and so on. Assume we have two hypotheses about the system that we want to distinguish from each other: the first hypothesis is that the system is functioning normally, which is modeled by the constraint set $M_1 = \{f_{\text{diff}}, f_{\text{add}}\}$. The second hypothesis is that the adder is stuck-at-L, which is modeled by $M_2 = \{f_{\text{diff}}, f_{\text{addstuck}}\}$. Then for example, the assignment $x \leftarrow L, y \leftarrow H, z \leftarrow L$ is a PDT for M (it leads to the observation $v = L$ or $v = H$ for M_1 , and $v = L$ for M_2), while the assignment $x \leftarrow L, y \leftarrow H, z \leftarrow H$ is a DDT for M (it leads to the observation $v = H$ for M_1 , and $v = L$ for M_2).

3 Test Generation as QCSP Solving

The two forms of testing in Def. 1, finding PDTs and DDTs, can be characterized as a game played between two opponents. The first player ($\exists$ -player) tries to reveal the fault by choosing input values for which the two hypotheses yield disjunct observations. The second player ($\forall$ -player) instead tries to hide the fault by choosing values for outputs or internal variables such that the two hypotheses yield overlapping observations. In the case of PDTs, he can choose values only for internal variables, whereas in the case of DDTs, he can choose values both for internal and observable variables. Both the $\exists$ -player and the $\forall$ -player must adhere to the rules that they can only choose among values that are consistent with the model of the system, as not all values are possible in all situations. The goal of the game is that exactly one hypothesis becomes true. Clearly, a PDT or DDT then exists if and only if the first player has a winning strategy.

Thus, the first form of testing in Def. 1, finding PDTs, is captured by the QCSP formula

$$\exists i_1 \dots i_n \exists o_1 \dots o_m \forall u_1 \dots u_k . M_1 \rightarrow \neg M_2 \quad (1)$$

whereas the second (stronger) form of testing, finding DDTs, is captured by the QCSP formula

$$\exists i_1 \dots i_n \forall o_1 \dots o_m \forall u_1 \dots u_k . M_1 \rightarrow \neg M_2 \quad (2)$$

From the QCSP-based formulation it can be observed that the two problems of finding PDTs and DDTs have the same worst-case complexity (Σ_2^P -complete).

The formulations (1) and (2) allow us to use standard solvers in order to compute discriminating tests, as opposed to using special algorithms as in [8,5].

The QCSP-based formulation of testing is also a starting point to tackle new classes of applications. In Def. 1, tests are assumed to consist of assignments to the controllable variables I ; the underlying assumption is that these variables characterize all relevant causal inputs to the system. As noted in the introduction, this assumption is often too restrictive; in practice, during testing there might be variables or parameters who influence the system’s behavior but whose values cannot be completely controlled. This scenario of *testing under limited controllability* can be captured using a modification of (2). Let I be partitioned into input variables $I_c = \{i_1 \dots i_s\}$ that can be controlled (set during testing), and input variables $I_{nc} = \{i_{s+1} \dots i_n\}$ that can be observed but not controlled. Then a definitely discriminating test exists iff the following formula is satisfiable:

$$\forall i_{s+1} \dots i_n \exists i_1 \dots i_s \forall o_1 \dots o_m \forall u_1 \dots u_k . M_1 \rightarrow \neg M_2 \quad (3)$$

Note that while solutions to (1) and (2) are simply assignments to the values of the input variables, solutions to (3) are in general more complex and correspond to a *strategy* or *policy* that states how the values of the controllable variables I_c must be set depending on the values of the non-controllable variables I_{nc} . To illustrate this, consider again the example in Fig. 1, but assume that variable x can’t be controlled. According to Def. 1, no DDT exists in this case, as the possible observations for v will always overlap for the two hypotheses M_1 and M_2 . However, there exists a test strategy to distinguish M_1 from M_2 , which consists of setting y depending on the value of x : choose input $y \leftarrow H, z \leftarrow H$ if $x = L$, and choose input $y \leftarrow L, z \leftarrow H$ if $x = H$. Generating such strategies goes beyond the theory in [8], which assumed that tests consist of assignments (patterns) for the input variables, but it is possible in the QCSP framework.

We conducted preliminary experiments of QCSP-based DDT generation with the solvers Qecode [1] and sKizzo [2] (since the present version of Qecode does not allow one to extract solutions from satisfiable instances, we transform the instance into QBF and use sKizzo to extract solutions). Figure 2 shows solutions generated from (3) for the example in Fig. 1. The solutions are represented in the form of BDDs with complemented arcs [2], where $\neg x$ stands for $x \leftarrow L$, x stands for $x \leftarrow H$, etc. The left-hand side of the figure shows the strategy (in this case, a simple set of assignments) that is generated if variables x, y, z are specified as controllable (input) variables, whereas the right-hand side of the



Fig. 2. Test strategies generated for the example in Fig. 1

figure shows the strategy when only y, z are controllable (in this case, y must be set depending on the value of x). No solution (definitely discriminating test strategy for the fault) exists if only z is assumed to be controllable.

4 Conclusion and Directions for Future Work

We reviewed an existing theory of testing for physical systems, which defines a weaker (PDTs) and a stronger form (DDTs) of test inputs, and showed how it can be framed as QCSP solving. Assumptions in this theory about the complete controllability of system inputs can be relaxed and lead to a more powerful class of tests, where inputs are intelligently set in reaction to observed values. Such *test strategies* go beyond the test pattern approach of the existing theory, but they can be captured in the QCSP framework. We are currently working on larger, more realistic examples to evaluate our QCSP-based testing approach. We are also extending our framework to systems with dynamic behavior (transition systems), in order to complement *passive* verification tools [4] for embedded autonomous controllers [10] with a capability to generate test strategies that can *actively* reveal faults.

References

1. Benedetti, M., Lallouet, A., Vautard, J.: QCSP Made Practical by Virtue of Restricted Quantification. In: Proceedings IJCAI 2007 (2007)
2. Benedetti, M.: sKizzo: A Suite to Evaluate and Certify QBFs. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632. Springer, Heidelberg (2005)
3. Brand, S.: Sequential Automatic Test Pattern Generation by Constraint Programming. In: Proc. CP 2001 Workshop on Modelling and Problem Formulation (2001)
4. Cimatti, A., Pecheur, C., Cavada, R.: Formal Verification of Diagnosability via Symbolic Model Checking. In: Proceedings IJCAI 2005 (2003)
5. Esser, M., Struss, P.: Fault-Model-Based Test Generation for Embedded Software. In: Proceedings IJCAI 2007 (2007)
6. Larrabee, T.: Test Pattern Generation Using Boolean Satisfiability. IEEE Transactions on Computer-Aided Design 2(1), 4–15 (1992)
7. Sachenbacher, M., Struss, P.: Task-dependent Qualitative Domain Abstraction. Artificial Intelligence 162(1-2), 121–143 (2005)
8. Struss, P.: Testing Physical Systems. In: Proceedings AAAI 1994 (1994)
9. Weld, D., de Kleer, J.: Readings in Qualitative Reasoning About Physical Systems. Morgan Kaufmann Publishers, San Francisco (1990)
10. Williams, B.C., Ingham, M., Chung, S., Elliott, P.: Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. Proceedings IEEE Special Issue on Modeling and Design of Embedded Software 91(1), 212–237 (2003)

Perfect Derived Propagators

Christian Schulte¹ and Guido Tack²

¹ ICT, KTH - Royal Institute of Technology, Sweden
`cschulte@kth.se`

² PS Lab, Saarland University, Saarbrücken, Germany
`tack@ps.uni-sb.de`

Abstract. When implementing a propagator for a constraint, one must decide about variants: When implementing $\min$, should one also implement $\max$? Should one implement linear equations both with and without coefficients? Constraint variants are ubiquitous: implementing them requires considerable effort, but yields better performance.

This paper shows how to use variable views to derive *perfect* propagator variants: derived propagators inherit essential properties such as correctness and domain and bounds completeness.

1 Introduction

When implementing a propagator for a constraint, one typically needs to decide whether to also implement some of its variants. For example, when implementing a propagator for $\max_{i=1}^n x_i = y$, should one also implement $\min_{i=1}^n x_i = y$? When implementing the linear equation $\sum_{i=1}^n a_i x_i = c$ for integer variables x_i and integers a_i and c , should one implement $\sum_{i=1}^n x_i = c$ for better performance?

While resulting in better performance, special implementations for propagator variants inflate code and documentation, and impair maintainability. The approach we take is to derive propagators from already existing propagators using variable views. In [8], we introduced an implementation architecture for variable views to reuse generic propagators without performance penalty. Gecode [4] makes massive use of views: every propagator implementation is reused 3.6 times on average. Without views, Gecode would feature 140 000 rather than 40 000 lines of propagator implementation to be written, tested, and maintained. Due to the extensive use of views, it is vital to develop a model that allows us to prove that derived propagators have the desired properties.

In this paper, we argue that propagators that are derived using variable views are indeed *perfect*: they are not only perfect for performance, we prove that they inherit all essential properties such as correctness and completeness from their original propagator. The key contribution is the identification of properties of views that yield perfect derived propagators. The paper establishes a formal model that defines a view as a function and a derived propagator as functional composition of views (mapping values to values) with a propagator (mapping domains to domains). This model yields all the desired results, in that derived propagators are indeed propagators; faithfully implement the intended

constraints; preserve domain completeness of the original propagators; and preserve bounds completeness if the views satisfy additional properties.

2 Preliminaries

We assume a finite set of variables $Var = \{x_1, \dots, x_n\}$ and a finite set of values Val . An assignment $a \in Asn$ maps variables to values: $Asn = Var \rightarrow Val$. A constraint $c \in Con$ is a relation over the variables, represented as the set of all assignments that satisfy the constraint, $Con = 2^{Asn}$.

Constraints are implemented by propagators over domains, which are constructed as follows. A domain $d \in Dom$ maps each variable to a finite set of possible values, the *variable domain* $d(x) \subseteq Val$. We identify a domain d with a set of assignments $d \subseteq 2^{Asn}$, and therefore treat domains as constraints to simplify presentation. A domain d_1 is *stronger* than a domain d_2 (written $d_1 \subseteq d_2$), iff for all variables x , $d_1(x) \subseteq d_2(x)$.

Propagators, also called narrowing operators or filter functions, serve as implementations of constraints. A propagator is a function $p \in Dom \rightarrow Dom$ that is contracting ($p(d) \subseteq d$) and monotone ($d' \subseteq d \Rightarrow p(d') \subseteq p(d)$).

A propagator p implements its *associated constraint* $c_p = \{a \in Asn \mid p(\{a\}) = \{a\}\}$. Monotonicity implies that $c_p \cap d \subseteq p(d)$ for any domain d : no solution of c_p is ever removed by p . We say that p is *sound* for any $c \subseteq c_p$ and *weakly complete* for any $c' \supseteq c_p$ (it accepts all assignments in c and rejects all assignments not in c'). For any constraint c , we can find a propagator p such that $c = c_p$. Typically, there are several propagators, differing by *propagation strength* (see Sect. 3). Our definitions of soundness and different notions of completeness are based on and equivalent to Benhamou's [1] and Maher's [5].

3 Views and Derived Propagators

A *view* on a variable x is an injective function $\varphi_x \in Val \rightarrow Val'$, mapping values from Val to values from a possibly different set Val' . We lift a family of views φ_x point-wise to assignments as follows: $\varphi_{Asn}(a)(x) = \varphi_x(a(x))$. Finally, given a family of views lifted to assignments, we define a view $\varphi \in Con \rightarrow Con$ on constraints as $\varphi(c) = \{\varphi_{Asn}(a) \mid a \in c\}$. The inverse of that view is defined as $\varphi^-(c) = \{a \in Asn \mid \varphi_{Asn}(a) \in c\}$. Views can now be composed with a propagator: a *derived propagator* is defined as $\hat{\varphi}(p) = \varphi^- \circ p \circ \varphi$.

Example. Given a propagator p for the constraint $c \equiv (x = y)$, we want to derive a propagator for $c' \equiv (x = 2y)$ using a view φ such that $\varphi^-(c) = c'$. It is usually easier to think about the other direction: $\varphi(c') \subseteq c$. Intuitively, the function φ leaves x as it is and scales y by 2, while φ^- does the inverse transformation. We thus define $\varphi_x(v) = v$ and $\varphi_y(v) = 2v$. We have a subset relation because some tuples of c may be ruled out by φ .

This example makes clear why the set Val' is allowed to differ from Val . In this particular case, Val' has to contain all multiples of 2 of elements in Val .

The derived propagator is $\widehat{\varphi}(p) = \varphi^- \circ p \circ \varphi$. We say that $\widehat{\varphi}(p)$ “uses a scale view on” y , meaning that φ_y is the function defined as $\varphi_y(v) = 2v$. Similarly, using an identity view on x amounts to φ_x being the identity function on Val .

Given the assignment $a = (x \mapsto 2, y \mapsto 1)$, we first apply φ_{Asn} and get $\varphi_{Asn}(a) = (x \mapsto 2, y \mapsto 2)$. This is returned unchanged by p , so φ^- transforms it back to a . Another assignment, $a' = (x \mapsto 1, y \mapsto 2)$, is transformed to $\varphi_{Asn}(a') = (x \mapsto 1, y \mapsto 4)$, rejected ($p(\{\varphi_{Asn}(a')\}) = \emptyset$), and the empty domain is mapped to the empty domain by φ^- . Thus, $\widehat{\varphi}(p)$ implements $\varphi^-(c)$.

Common views capture linear transformations for integer variables, negation for Boolean variables, or complement for set variables. For example, in [8] the following views are introduced for a variable x and values v : a *minus view* on x is defined as $\varphi_x(v) = -v$, an *offset view* for $o \in \mathbb{Z}$ on x is defined as $\varphi_x(v) = v + o$, and a *scale view* for $a \in \mathbb{Z}$ on x is defined as $\varphi_x(v) = a \cdot v$.

The central properties of derived propagators are expressed in the following theorems (with proofs in the long version of this paper [9]):

Theorem 1. A derived propagator is a propagator: for all propagators p and views φ , $\widehat{\varphi}(p)$ is a monotone and contracting function in $Dom \rightarrow Dom$.

Theorem 2. If p implements c_p , then $\widehat{\varphi}(p)$ implements $\varphi^-(c_p)$.

Theorem 3. Views preserve contraction: for any domain d , if $p(\varphi(d)) \subseteq \varphi(d)$, then $\widehat{\varphi}(p)(d) \subseteq d$. This property makes sure that if the propagator makes an inference, then this inference will actually be reflected in a domain change.

A propagator is *domain complete* (or simply *complete*) for a constraint c if it establishes domain consistency. More formally, $\text{dom}(c)$ is the strongest domain including all valid assignments of a constraint, defined as $\min\{d \in Dom \mid c \subseteq d\}$. The minimum exists as domains are closed under intersection, and the definition is non-trivial because not every constraint can be captured by a domain. Now, for a constraint c and a domain d , $\text{dom}(c \cap d)$ refers to removing all values from d not supported by the constraint c . A propagator p is complete for a constraint c iff for all domains d , we have $p(d) \subseteq \text{dom}(c \cap d)$. A complete propagator thus removes all assignments from d that are inconsistent with c . One of the main results of this paper is that domain completeness is preserved by views.

Theorem 4. If p is complete for c , then $\widehat{\varphi}(p)$ is complete for $\varphi^-(c)$.

A propagator is *bounds complete* for a constraint c , if it only affects domain bounds, or only depends on domain bounds for its inferences. For our purposes, we only distinguish $\text{bounds}(\mathbb{Z})$ and $\text{bounds}(\mathbb{R})$ completeness (see [3] for an overview). Our definitions of bounds completeness are based on the *strongest convex domain* that contains a constraint, $\text{conv}(c) = \min\{d \in Dom \mid c \subseteq d \text{ and } d \text{ convex}\}$. A convex domain maps each variable to an interval, so that $\text{conv}(c)(x) = \{\min_{a \in c}(a(x)), \dots, \max_{a \in c}(a(x))\}$. Following Benhamou [1] and Maher [5], we define that p is $\text{bounds}(\mathbb{Z})$ complete for c iff $p(d) \subseteq \text{conv}(c \cap \text{conv}(d))$, and p is $\text{bounds}(\mathbb{R})$ complete for c iff $p(d) \subseteq \text{conv}(c_{\mathbb{R}} \cap \text{conv}_{\mathbb{R}}(d))$, where $\text{conv}_{\mathbb{R}}(d)$ is the convex hull of d in $\mathbb{R}$, and $c_{\mathbb{R}}$ is c relaxed to $\mathbb{R}$.

The result for domain completeness does not carry over directly to bounds completeness: we can only derive bounds complete propagators using views that satisfy certain additional properties. A constraint c is a φ *constraint* iff for all $a \in c$, there is a $b \in \text{Asn}$ such that $a = \varphi_{\text{Asn}}(b)$. A view φ is *interval injective* iff $\varphi^-(\text{conv}(c)) = \text{conv}(\varphi^-(c))$ for all φ constraints c . It is *interval bijective* iff it is interval injective and $\varphi(\text{conv}(d)) = \text{conv}(\varphi(d))$ for all domains d . The following table summarizes how completeness depends on view bijectivity:

<i>propagator</i>	<i>interval bijective view</i>	<i>interval injective view</i>	<i>arbitrary view</i>
domain	domain	domain	domain
bounds($\mathbb{Z}$)	bounds($\mathbb{Z}$)	bounds($\mathbb{R}$)	weakly
bounds($\mathbb{R}$)	bounds($\mathbb{R}$)	bounds($\mathbb{R}$)	weakly

Minus and offset views are interval bijective. A scale view for $a \in \mathbb{Z}$ on x is always interval injective and only interval bijective if $a = 1$ or $a = -1$. An important consequence is that a bounds($\mathbb{Z}$) complete propagator for the constraint $\sum_i x_i = c$, when instantiated with scale views for the x_i , results in a bounds($\mathbb{R}$) complete propagator for $\sum_i a_i x_i = c$.

Views are related to *indexicals* [2,10], propagators that prune a single variable and are defined over range expressions. However, views are not used to define propagators, but to derive new propagators from existing ones. Allowing the full expressivity of indexicals for views would imply giving up our completeness results. Another related concept are arithmetic expressions (as found in ILOG Solver [6]). In contrast to views, expressions are used for modeling, not for propagation, and, like indexicals, yield no completeness guarantees.

4 Extended Properties of Derived Propagators

A derived propagator permits further derivation: $\widehat{\varphi}(\widehat{\varphi}'(p))$ is perfectly acceptable, properties like correctness and completeness carry over. For instance, we can derive a propagator for $x - y = c$ from a propagator for $x + y = 0$ by combining an offset and a minus view on y .

A propagator is idempotent iff $p(p(d)) = p(d)$ for all domains d . Some systems require all propagators to be idempotent, others apply optimizations if the idempotence of a propagator is known [7].

Theorem 5. If a propagator is derived from an idempotent propagator, the result is idempotent again: If $p(p(d)) = p(d)$ for a propagator p and a domain d , then, for any view φ , $\widehat{\varphi}(p)(\widehat{\varphi}(p)(d)) = \widehat{\varphi}(p)(d)$.

A propagator is subsumed for a domain d iff for all stronger domains $d' \subseteq d$, $p(d') = d'$. Subsumed propagators do not contribute any propagation in the remaining subtree of the search, and can therefore be removed. Deciding subsumption is coNP-complete in general, but for most propagators an approximation can be decided easily. This can be used to optimize propagation.

Theorem 6. p is subsumed by $\varphi(d)$ iff $\widehat{\varphi}(p)$ is subsumed by d .

An alternative model of views is to regard a view φ as additional *view constraints*, implementing the *decomposition* of a constraint.

Example. Assume the equality constraint $c \equiv (x = y)$. In order to propagate $c' \equiv (x = y + 1)$, we could use a domain complete propagator p for c and a view φ with $\varphi_x(v) = v$, $\varphi_y(v) = v + 1$. The alternative model would contain additional variables x' and y' , a view constraint $c_{\varphi,x}$ for $x' = x$, a view constraint $c_{\varphi,y}$ for $y' = y + 1$, and $c[x/x', y/y']$, which enforces equality of x' and y' .

In general, every view constraint $c_{\varphi,i}$ shares exactly one variable with c and no variable with any other $c_{\varphi,i}$. Thus, the constraint graph is Berge-acyclic, and we can reach a fixpoint by first propagating all the $c_{\varphi,i}$, then propagating $c[x_1/x'_1, \dots, x_n/x'_n]$, and then again propagating the $c_{\varphi,i}$. This is exactly what $\varphi^- \circ p \circ \varphi$ does. In this sense, views can be seen as a way to specify a *perfect order of propagation*, which is usually not possible in constraint programming systems. Furthermore, if $\widehat{\varphi}(p)$ is domain complete for $\varphi^-(c)$, it achieves *path consistency* for $c[x_1/x'_1, \dots, x_n/x'_n]$ and all the $c_{\varphi,i}$ in the decomposition model.

Acknowledgements. We thank Mikael Lagerkvist and Gert Smolka for fruitful discussions about views and helpful comments on a draft of this paper.

References

1. Benhamou, F.: Heterogeneous Constraint Solving. In: Hanus, M., Rodríguez-Artalejo, M. (eds.) ALP 1996. LNCS, vol. 1139, pp. 62–76. Springer, Heidelberg (1996)
2. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Glaser, H., Hartel, P.H., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 191–206. Springer, Heidelberg (1997)
3. 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)
4. Gecode: Generic constraint development environment (2008), <http://www.gecode.org/>
5. Maher, M.J.: Propagation completeness of reactive constraints. In: Stuckey, P.J. (ed.) ICLP 2002. LNCS, vol. 2401, pp. 148–162. Springer, Heidelberg (2002)
6. Puget, J.-F., Leconte, M.: Beyond the glass box: Constraints as objects. In: Lloyd, J. (ed.) Proceedings of the International Symposium on Logic Programming, Portland, OR, USA, pp. 513–527. The MIT Press, Cambridge (1995)
7. Schulte, C., Stuckey, P.J.: Efficient constraint propagation engines. In: Transactions on Programming Languages and Systems (to appear, 2008)
8. Schulte, C., Tack, G.: Views and iterators for generic constraint implementations. In: Hnich, B., Carlsson, M., Fages, F., Rossi, F. (eds.) CSCLP 2005. LNCS (LNAI), vol. 3978, pp. 118–132. Springer, Heidelberg (2006)
9. Schulte, C., Tack, G.: Perfect derived propagators (June 2008), <http://arxiv.org/abs/0806.1806>
10. Van Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(FD). The Journal of Logic Programming 37(1–3), 139–164 (1998)

Refined Bounds for Instance-Based Search Complexity of Counting and Other #P Problems^{*}

Lars Otten and Rina Dechter

Bren School of Information and Computer Sciences
University of California, Irvine, CA 92697-3425, U.S.A.
{lotten, dechter}@ics.uci.edu

Abstract. We present measures for bounding the instance-based complexity of AND/OR search algorithms for solution counting and related #P problems. To this end we estimate the size of the search space, with special consideration given to the impact of determinism in a problem. The resulting schemes are evaluated empirically on a variety of problem instances and shown to be quite powerful.

1 Introduction

Inference algorithms like variable elimination have been known to be exponentially bounded by the tree width of a problem's underlying graph structure. More accurate bounds were derived by looking at the respective domain sizes of the variables in each cluster of a tree decomposition of the underlying graph [5], which was later also applied to search algorithms that explore the context-minimal AND/OR search graph [1].

We recently introduced a more informed upper-bounding scheme, that selectively takes determinism into account [6]. We demonstrated its effectiveness empirically over a set of Bayesian networks and showed that the bounds it provides can in some cases be better by orders of magnitude. These tighter bounds allow us, for instance, to better predict parameters of algorithms (like variable orderings) ahead of time.

In this paper we extend our earlier work in four ways: First, we refine the bounding scheme by “reusing” relations during the estimation process, projecting their scope down to the currently relevant variables. Secondly, we introduce a simple scheme for lower bounding, that uses a sampling-based SAT solution counting algorithm. Thirdly, we show that these schemes are applicable to constraint networks by presenting experiments on various sets of constraint problem instances. Finally, we investigate our bounds' ability to discriminate between different variable orderings and demonstrate that they are indeed informative in this respect.

2 Bounding Search Space Size

We will assume a *graphical model*, given as a set of variables $X = \{x_1, \dots, x_n\}$, their finite domains $D = \{D_1, \dots, D_n\}$, a set of functions or relations $R = \{r_1, \dots, r_m\}$, each of which is defined over a subset of X , and a combination operator (join, sum or

^{*} This work is supported in part by NSF grant IIS-0713118 and NIH grant R01-HG004175-02.

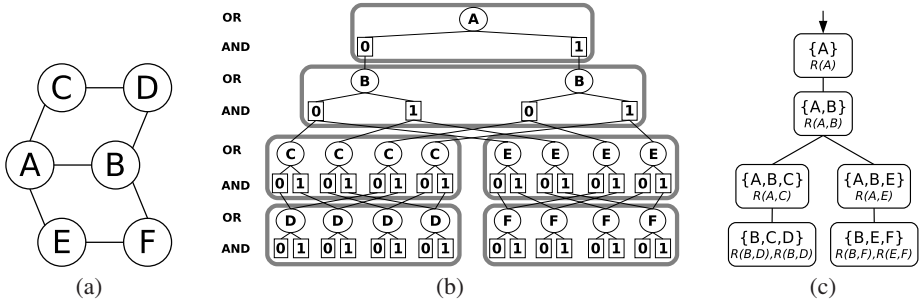


Fig. 1. (a) An example problem graph, (b) its AND/OR search graph along ordering $d = A, B, C, D, E, F$, and (c) the corresponding bucket tree decomposition

product) over all functions. The scope of a relation r_j , denoted $\text{scope}(r_j)$, is the subset of X on which r_j is defined, its tightness t_j is the number of valid tuples in the relation. A *constraint satisfaction problem* (CSP) is then a special kind of graphical model.

Given a variable ordering d , we furthermore assume the usual definition of a *tree decomposition*, decomposing the variables and relations into clusters of a certain tree width w (the max. number of variables in a cluster). If the variables in each cluster are covered by the scopes of the cluster's relations, we have a *hypertree decomposition*, with associated hypertree width hw (the max. number of functions in a cluster). It is known that, if $k = \max_i |D_i|$ and $t = \max_j t_j$, the time and space complexity of processing a tree decomposition is dominated by k^w , while a hypertree decomposition can yield a solution with time and space complexity dominated by t^{hw} [3].

AND/OR search, on the other hand, is a novel method to exploit problem decomposition during search. It introduces AND nodes into the search space that allow capturing the independence of subproblems. If we also apply context-based caching of identical subproblems, it is easy to see that the resulting AND/OR search space has a one-to-one correspondence to a (bucket) tree decomposition along the same ordering (cf. Fig. 1). Accordingly, similar asymptotic bounds can be proven [1].

For a more refined analysis of problem complexity, we can determine the size of the search space as follows: For each cluster of the bucket tree C_k , containing variables $X_k \subseteq X$ and relations $R_k \subseteq R$, we multiply the domain sizes of the variables in X_k – this represents all possible value combinations of the variables in X_k . Summing over all clusters we obtain an upper bound on the number of nodes in the AND/OR search space, denoted $twb := \sum_{k=1}^n \prod_{x_i \in X_k} |D_i|$ (this is essentially a fine-grained version of the asymptotic, tree width-exponential bound). The bound quality will depend on the degree of determinism present in the problem, which is not reflected in twb , but which can cause significant pruning of the search space in practice.

If we are working on a hypertree decomposition, we can take the product over the tightness of each relation in a cluster as an upper bound on the number of search nodes in that cluster and sum over clusters – a fine-grained version of the asymptotic hypertree width bound, thus accounting for determinism. However, since relation scopes typically overlap, this will be far from optimal. We therefore start from the twb bound, the product of variable domains, and iteratively pick relations whose tightness we can

Algorithm GreedyCovering

Input: Set of variables $X = \{x_1, \dots, x_r\}$ and set of relations $R = \{r_1, \dots, r_s\}$, with x_i having domain size $|D_i|$ and r_j having tightness t_j

Output: A subset of R (a partial covering of X)

Init: $Uncov := X$, $Covering := \emptyset$

- (1) Find j^* that minimizes $q_j = t_j / \prod_{x_k \in I_j} |D_k|$,
where $I_j = Uncov \cap scope(r_j)$.
- (2) If $q_{j^*} \geq 1$, terminate and return $Covering$.
- (3) Add r_{j^*} to $Covering$ and set
 $Uncov := Uncov \setminus scope(r_{j^*})$.
- (4) If $Uncov = \emptyset$, terminate and return $Covering$.
- (5) Goto (1).

Algorithm Compute-hwb

Input: A bucket tree decomposition with clusters $C_1, \dots, C_n$, where cluster C_k contains variables $X_k \subseteq X$ and relations $R_k \subseteq R$

Output: The bound hwb on the size of the search space

Init: $hwb := 0$

- (1) **for** $i = 1$ to n :
- (2) $R := R_k$.
- (3) For every relevant relation r from the ancestral buckets, project it onto $scope(r) \cap X_k$ and add it to R with updated tightness t'_r .
- (4) $G := GreedyCovering(X_k, R)$.
- (5) $hwb += \prod_{r_j \in G} t_j \cdot \prod_{x_i \in X_k \setminus G} |D_i|$.
- (6) **end for**.
- (7) Return hwb .

Fig. 2. Greedy covering algorithm and procedure to compute the overall bound hwb

use to improve the bound, greedily covering variables with relations, similar to a SET COVER problem [4]. This results in the algorithm *GreedyCovering* given in Fig. 2.

Propagating Cluster Size Downwards. When considering relations for the covering of variables X_k in cluster C_k , we can refine the above scheme even further: We collect all relations from the ancestral clusters in the rooted tree decomposition and project them down to X_k . This can be seen as propagation of information down the search tree. In practice, we found that for some problem instances it will decrease the bound by up to 30%.

Overall Bound and Complexity. The resulting overall algorithm *Compute-hwb* is given in Fig. 2. It computes the upper bound hwb on the number of nodes in the AND/OR search space. Its complexity can be shown to be time $\mathcal{O}(n \cdot m \cdot (t + w))$ and space $\mathcal{O}(m + t)$, where n and m are the number of variables and relations, respectively, w is the tree width of the problem along the given ordering, and t is the maximal tightness as before (proof in [7]).

Lower Bounds on Cluster Size. To obtain a lower bound on the size of the search space, we employ a different scheme: In each cluster we generate a SAT formula from all relevant relations (i.e., from within the current cluster and ancestral ones). We encode the invalid tuples of each relation as the nogoods of the SAT formula, thus the number of SAT solutions will correspond to the number of valid nodes in the cluster. We feed each cluster's formula to the sampling-based SAT solution counting algorithm *SampleSearch-LB* [2], which gives a (probabilistic) lower bound. To get a lower bound on the overall number of search nodes, we again sum the cluster bounds. We call this bound *satb*.

3 Experimental Results

We ran a variety of empirical tests on a large set of different problem instances from various domains. Here we present selected results, for the full set we refer to [7]. For every problem instance, we report the number of variables n , the number of relations

Table 1. *twb*, *hwb*, and *satb* bounds compared to true search space size $\#cm$

instance	<i>n</i>	<i>m</i>	<i>k</i>	<i>r</i>	<i>w</i>	<i>tr</i>	<i>twb</i>	<i>hwb</i>	<i>satb</i>	$\#cm$	Q_t	Q_h	Q_s
pret-60	60	40	2	3	4	0.50	1,534	1,102	839	998	1.51	1.10	0.89
pret-150	150	100	2	3	4	0.50	3,934	2,862	2,303	2,598	1.54	1.10	0.84
ssa-0432-003	435	738	2	5	31	0.75	4,244,330	2,059,616	1,116,669	1,868,283	2.27	1.10	0.60
ssa-2670-130	1359	2366	2	5	31	0.75	160,631,566	123,388,312	104,689,598	106,638,207	1.51	1.16	0.98
ssa-7552-038	1501	2444	2	6	63	0.75	308,861,278	115,499,146	6,815,140	36,718,327	8.41	3.15	0.19
ssa-7552-158	1363	1985	2	5	31	0.50	90,702	74,406	56,863	69,365	1.31	1.07	0.82
ssa-7552-159	1363	1983	2	5	31	0.50	92,238	73,586	48,929	68,694	1.34	1.07	0.71
BN_105	40	44	2	21	18	0.62	2,477,054	363	69	131	18909	2.77	0.53
BN_107	40	46	2	21	21	0.62	29,983,742	1,643	191	272	110234	6.04	0.70
BN_109	40	46	2	20	20	0.62	13,054,974	4,052	1,309	2,531	5158	1.60	0.52
BN_111	40	45	2	20	19	0.62	8,406,270	2,299	465	979	8587	2.35	0.47
BN_113	40	47	2	21	21	0.62	18,916,350	2,752	336	630	30026	4.37	0.53
aim-50-1-6-sat-1	50	77	2	3	18	0.88	2,517,118	2,053,046	931,492	1,813,906	1.39	1.13	0.51
aim-50-1-6-sat-2	50	76	2	3	16	0.88	767,678	626,955	73,180	551,659	1.39	1.14	0.13
aim-50-1-6-sat-3	50	78	2	3	20	0.88	4,742,590	3,859,278	2,023,465	3,848,835	1.23	1.00	0.53
aim-50-1-6-sat-4	50	77	2	3	19	0.88	3,615,166	2,616,824	2,079,752	2,532,968	1.43	1.03	0.82
aim-50-1-6-unsat-1	50	69	2	3	15	0.88	377,502	256,482	26,806	211,168	1.79	1.21	0.13
aim-50-1-6-unsat-2	50	77	2	3	19	0.88	3,484,734	2,551,090	16,995	1,908,441	1.83	1.34	0.01
aim-50-1-6-unsat-3	50	70	2	3	17	0.88	1,190,910	971,254	43,382	685,060	1.74	1.42	0.06
aim-50-1-6-unsat-4	50	76	2	3	20	0.88	7,236,702	5,195,870	2,386,893	3,873,236	1.87	1.34	0.62

m , the maximum variable domain size k , the maximum relation arity r , and the median tightness ratio tr over all its relations, defined as the ratio of valid tuples in a (full) relation table.

We build a bucket tree decomposition of the problem along a minfill ordering and report the tree width w . We then compute and report our bounds twb , hwb , and $satb$. Lastly we record the exact size of the AND/OR search graph, denoted $\#cm$. To make comparing values easier, we also report the ratios $Q_t = \frac{twb}{\#cm}$, $Q_h = \frac{hwb}{\#cm}$, and $Q_s = \frac{satb}{\#cm}$. The results are shown in Table 1. We note that computation of twb and hwb is performed within milliseconds, while $satb$ can take up to three seconds for ssa-7552-038 on our 2.66 GHz system (with 1000 samples generated in each cluster).

Bound Tightness. Going from twb to hwb , i.e., exploiting determinism, yields significantly tighter bounds across all instances, from, for instance, a 28% decrease on the *Pret* instances or 18% on aim-50-1-6-sat-2 to several orders of magnitude on the *BN* instances (which were generated with forced determinism and are thus amenable to our method). In many cases the hwb bound is indeed quite tight when compared to $\#cm$, getting to within 10% on the *Pret* and some *SSA* instances. Overall, the quality of the hwb bound seems to decrease with growing problem complexity. The current results for $satb$ are somewhat less impressive at this point, often being more than 50% smaller than $\#cm$ – yet they can sometimes give a rough idea and, on top of that, set the stage for future improvements.

Impact of Orderings. To investigate the power of our bounds in predicting good orderings, we processed the same problem instance 50 times along a randomized minfill ordering, each time computing twb and hwb as well as recording $\#cm$. The results for two instances are presented in Fig. 3. We find that hwb can provide valuable information: On aim-50-1-6-sat-4 the twb bound can not distinguish between orderings with tree width 19, yet hwb captures the different $\#cm$ values rather accurately. On aim-50-1-6-sat-1,

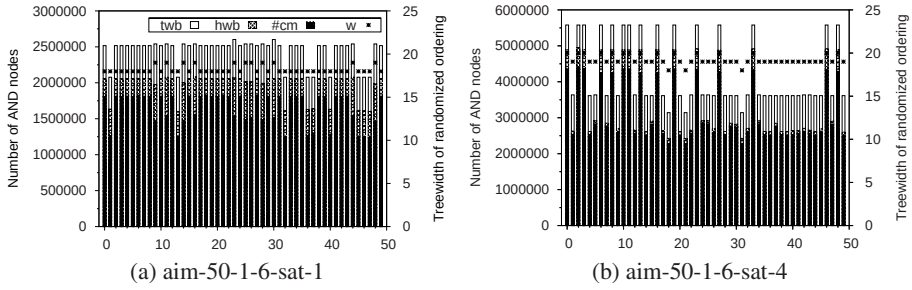


Fig. 3. Plots of the *twb* and *hwb* bounds versus the true search space size $\#cm$ on two problem instances, each over 50 randomized minfill variable orderings. Also shown is the tree width *w* for each ordering, which is plotted against a separate scale on the right.

some orderings yield a higher tree width of 19, yet have a smaller search space size $\#cm$, which is correctly indicated only by *hwb*.

4 Summary and Future Work

We have previously introduced a scheme that extends known methods for bounding the size of the search space by taking determinism in the relation specification into account [7]. In this work we expand upon this in four ways: We account for propagation of determinism down the search tree by reconsidering and projecting relations, we develop an approach for lower bounding search space size, we show empirically the applicability to constraint networks, and we demonstrate the bounds' ability to discriminate between variable orderings. Our experimental results show that the upper bounds we obtain can be quite tight and provide valuable information; the lower bounds, however, still leave room for improvement at this point.

We believe our bounding scheme can be extended to optimization tasks by using the cost function itself. By dynamically adapting the bounds throughout the search process, we plan to allow for run time parameter updates. Finally, recent advances in sampling-based counting should also allow us to improve the quality of the lower bounds.

References

1. Dechter, R., Mateescu, R.: AND/OR Search Spaces for Graphical Models. *Artificial Intelligence* 171, 73–106 (2007)
2. Gogate, V., Dechter, R.: Approximate Counting by Sampling the Backtrack-free Search Space. In: *Proceedings of AAAI 2007* (2007)
3. Gottlob, G., Leone, N., Scarcello, F.: A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence* 124, 243–282 (2000)
4. Johnson, D.S.: Approximation algorithms for combinatorial problems. In: *Proceedings of STOC 1973*, pp. 38–49 (1973)

5. Kjørulff, U.: Triangulation of Graphs – Algorithms Giving Small Total State Space. Research Report R-90-09, Dept. of Mathematics and Computer Science, Aalborg University (1990)
6. Otten, L., Dechter, R.: Bounding Search Space Size via (Hyper) tree Decompositions. In: Proceedings of UAI 2008 (2008)
7. Otten, L., Dechter, R.: Refined Bounds for Instance-Based Search Complexity of Counting and Other #P Problems. Technical Report, University of California, Irvine (2008)

Transforming Inconsistent Subformulas in MaxSAT Lower Bound Computation*

Chu Min Li^{1,2}, Felip Manyà^{3,4}, Nouredine Ould Mohamedou², and Jordi Planes⁴

¹ Hunan Normal University, Changsha, China

² MIS, Université de Picardie Jules Verne, 5 Rue du Moulin Neuf 80000 Amiens, France

³ Artificial Intelligence Research Institute (IIIA, CSIC), Campus UAB, 08193 Bellaterra, Spain

⁴ Computer Science Department, Universitat de Lleida, Jaume II 69, 25001 Lleida, Spain

Abstract. We define a new heuristic that guides the application of cycle resolution (CR) in MaxSAT, and show that it produces better lower bounds than those obtained by applying CR exhaustively as in Max-DPLL, and by applying CR in a limited way when unit propagation detects a contradiction as in MaxSatz.

1 Introduction

The lower bound (LB) computation method implemented in modern branch and bound MaxSAT solvers has two components: (i) the *underestimation component*, which detects disjoint inconsistent subformulas (typically using unit propagation (UP) [1] or unit propagation enhanced with failed literal detection [2]), and takes the number of detected inconsistent subformulas as an underestimation of the LB; and (ii) the *inference component*, which applies cost preserving inference rules that, in the best case, make explicit a contradiction by deriving an empty clause which allows to increment the LB.

We analyze more deeply than before the impact of the *cycle resolution* (CR) inference rule on the performance of MaxSAT solvers. It is well-known that Max-DPLL [3] applies CR exhaustively and does not combine its application with the underestimation component, while MaxSatz [4] applies CR when the underestimation component detects an inconsistent subformula via unit propagation which includes one particular unit clause and the premises of CR. In this paper, we provide evidence that the exhaustive application of CR is not the best option in general, and that combining its application with an underestimation component incorporating failed literal detection may produce better quality LBs. To better exploit the power of CR in MaxSAT solvers, we define a new heuristic that guides the application of CR during failed literal detection. Experiments on a new version of MaxSatz implementing this heuristic, called MaxSatz_c, show that MaxSatz_c substantially speeds up MaxSatz.

2 Inference Rules

MaxSatz [4] incorporates the following rules (also called Rule 1, Rule 2, Rule 3, and Rule 4 in this paper):

$$l_1, \bar{l}_1 \vee \bar{l}_2, l_2 \implies \square, l_1 \vee l_2 \quad (1)$$

* This research was funded by MEC research projects TIN2006-15662-C02-02, TIN2007-68005-C04-04, and CONSOLIDER CSD2007-0022, INGENIO 2010.

$$l_1, \bar{l}_1 \vee l_2, \bar{l}_2 \vee l_3, \dots, \bar{l}_k \vee l_{k+1}, \bar{l}_{k+1} \implies \Box, l_1 \vee \bar{l}_2, l_2 \vee \bar{l}_3, \dots, l_k \vee \bar{l}_{k+1} \quad (2)$$

$$l_1, \bar{l}_1 \vee l_2, \bar{l}_1 \vee l_3, \bar{l}_2 \vee \bar{l}_3 \implies \Box, l_1 \vee \bar{l}_2 \vee \bar{l}_3, \bar{l}_1 \vee l_2 \vee l_3 \quad (3)$$

$$l_1, \bar{l}_1 \vee l_2, \bar{l}_2 \vee l_3, \dots, \bar{l}_k \vee l_{k+1}, \bar{l}_{k+1} \vee l_{k+2}, \bar{l}_{k+1} \vee l_{k+3}, \bar{l}_{k+2} \vee \bar{l}_{k+3} \implies \Box, l_1 \vee \bar{l}_2, l_2 \vee \bar{l}_3, \dots, l_k \vee \bar{l}_{k+1}, l_{k+1} \vee \bar{l}_{k+2} \vee \bar{l}_{k+3}, \bar{l}_{k+1} \vee l_{k+2} \vee l_{k+3} \quad (4)$$

Max-DPLL [3] incorporates several rules for weighted MaxSAT, including chain resolution (which is equivalent to Rule 2 in the unweighted case) and CR restricted to 3 variables, which is as follows in the unweighted case:

$$\begin{array}{ccc} \bar{l}_1 \vee l_2 & & \bar{l}_1 \\ \bar{l}_1 \vee l_3 & \implies & l_1 \vee \bar{l}_2 \vee \bar{l}_3 \\ \bar{l}_2 \vee \bar{l}_3 & & \bar{l}_1 \vee l_2 \vee l_3 \end{array} \quad (5)$$

In the sequel, when we say CR we mean CR restricted to 3 variables. Rule 3 and Rule 4 in MaxSatz, and CR in Max-DPLL capture this special structure: $\bar{l}_1 \vee l_2, \bar{l}_1 \vee l_3, \bar{l}_2 \vee \bar{l}_3$, which we call *cycle structure*. Max-DPLL applies CR in an unlimited way: it replaces every subset of three binary clauses matching the cycle structure with one unit clause and two ternary clauses. MaxSatz applies CR in a limited way: it applies CR only when the underestimation component detects a contradiction containing the cycle structure by applying unit propagation.

3 CR and Failed Literal Detection

The next example shows that it is worth applying CR in scenarios not considered in the version of MaxSatz used in the 2007 MaxSAT Evaluation, called MaxSatz-07 in this paper. MaxSatz-07 computes underestimations using failed literal detection after no more contradictions can be derived using unit propagation, but does not check if a rule is applicable when a failed literal is detected. The applicability of rules is just checked when unit propagation (without failed literals) derives a contradiction. So, CR is applied only when unit propagation allows to apply Rule 3 or Rule 4. Assume that a MaxSAT instance ϕ contains

$$\begin{array}{l} x_1 \vee x_2, \bar{x}_2 \vee x_3, \bar{x}_2 \vee x_4, \bar{x}_3 \vee \bar{x}_4, \bar{x}_1 \vee x_5, \bar{x}_5 \vee x_6, \bar{x}_1 \vee x_7, \bar{x}_6 \vee \bar{x}_7 \\ x_8 \vee \bar{x}_2, x_8 \vee x_3, x_8 \vee x_4, \bar{x}_8 \vee x_9, \bar{x}_8 \vee x_{10}, \bar{x}_8 \vee x_{11}, \bar{x}_9 \vee \bar{x}_{10} \vee \bar{x}_{11} \end{array}$$

Since there is no unit clause, Rule 3 and Rule 4 are not applied. Failed literal detection just detects the inconsistent subformula in the first line (branching on the variable x_1). However, if CR is applied to $\bar{x}_2 \vee x_3, \bar{x}_2 \vee x_4, \bar{x}_3 \vee \bar{x}_4$, the underestimation component detects 2 inconsistent subformulas (one branching on x_1 and the other on x_8). We would like to highlight two features of this example: (i) the inconsistent subformula detected when CR is applied after branching on x_1 is smaller than when CR is not applied: $x_1 \vee x_2, \bar{x}_2, \bar{x}_1 \vee x_5, \bar{x}_5 \vee x_6, \bar{x}_1 \vee x_7, \bar{x}_6 \vee \bar{x}_7$ instead of $x_1 \vee x_2, \bar{x}_2 \vee x_3, \bar{x}_2 \vee x_4, \bar{x}_3 \vee \bar{x}_4, \bar{x}_1 \vee x_5, \bar{x}_5 \vee x_6, \bar{x}_1 \vee x_7, \bar{x}_6 \vee \bar{x}_7$; and (ii) the added ternary clauses

$(x_2 \vee \bar{x}_3 \vee \bar{x}_4$ and $\bar{x}_2 \vee x_3 \vee x_4)$ may contribute to detect further inconsistent subformulas: When branching on x_8 , it detects the inconsistent subformula $x_8 \vee \bar{x}_2$, $x_8 \vee x_3$, $x_8 \vee x_4$, $\bar{x}_8 \vee x_9$, $\bar{x}_8 \vee x_{10}$, $\bar{x}_8 \vee x_{11}$, $\bar{x}_9 \vee \bar{x}_{10} \vee \bar{x}_{11}$, $x_2 \vee \bar{x}_3 \vee \bar{x}_4$, which contains one of the added ternary clauses.

Proposition 1. *Let l be a failed literal in ϕ (i.e., $UP(\phi \wedge l)$ derives an empty clause), and let S_l be the set of clauses used to derive the contradiction in $UP(\phi \wedge l)$. If S_l contains the cycle structure $\bar{l}_1 \vee l_2$, $\bar{l}_1 \vee l_3$, $\bar{l}_2 \vee \bar{l}_3$, and S'_l is S_l after applying CR to the cycle structure, then $S'_l - \{l_1 \vee \bar{l}_2 \vee \bar{l}_3, \bar{l}_1 \vee l_2 \vee l_3\}$ is inconsistent.*

Proposition 1 states that if an inconsistent subformula containing the cycle structure is detected using unit propagation enhanced with failed literal detection and CR is applied, then the inconsistent subformula is smaller than the inconsistent subformula that can be derived without applying CR and, moreover, CR adds two ternary clauses that can be used to detect other inconsistent subformulas. This is the rationale behind the heuristic, defined in the next section, for guiding the application of CR during the process of detecting failed literals in the underestimation component.

4 A New Heuristic for Guiding the Application of CR

We have seen that CR can improve the lower bound if its application allows to reduce the size of an inconsistent subformula and liberate two ternary clauses. We define now a heuristic that guides the application of CR during the process of detecting failed literals with the aim of capturing situations in which CR could be beneficial. This heuristic is implemented in Algorithm 1, where $\text{occ2}(l)$ is the number of occurrences of literal l in binary clauses of ϕ , and S_l is an inconsistent subformula detected by applying unit propagation to $\phi \wedge l$.

Between the two literals of a variable x that are likely to be failed (since their satisfaction results in at least two new unit clauses), Algorithm 1 detects first the literal l with more occurrences in binary clauses. Note that l has less chances to be failed than $\bar{l}$ because its satisfaction produces fewer new unit clauses. If l is a failed literal and S_l contains a cycle structure, CR is applied in S_l before deciding whether $\bar{l}$ is failed. If $\bar{l}$ is also failed, the inconsistent subformula $S_l \cup S_{\bar{l}} - \{l, \bar{l}\}$ is transformed into a smaller inconsistent subformula by applying CR in S_l , and two ternary clauses are liberated. If $\bar{l}$ is not a failed literal in the current node, we do not have an inconsistent subformula that can be transformed using CR in S_l , but S_l is now smaller thanks to CR and will be easier to redetect it in the subtree. Note that $\bar{l}$ has higher chances to fail in the subtree and to produce an inconsistent subformula transformed using CR.

On the other hand, if l is not failed, Algorithm 1 does not detect an inconsistent subformula. It is not checked whether $\bar{l}$ is failed, and CR is not applied to $S_{\bar{l}}$ if $\bar{l}$ is failed, avoiding the application of CR that does not allow to transform an inconsistent subformula.

With the aim of evaluating the impact of Algorithm 1 in the performance of MaxSatz, we define the following solvers:

MaxSatz-07: Standard version of MaxSatz, implementing all MaxSatz inference rules, and failed literal detection, besides UP, in the underestimation component.

Algorithm 1. *flAndCycle*(ϕ, x), combining CR and failed literal detection**Input:** A MaxSAT instance ϕ and a variable x such that $\text{occ2}(x) \geq 2$ and $\text{occ2}(\bar{x}) \geq 2$ **Output:** ϕ in which CR is possibly applied, and an underestimation

```

1 begin
2   if  $\text{occ2}(x) > \text{occ2}(\bar{x})$  then  $l \leftarrow x$ ; else  $l \leftarrow \bar{x}$ ;
   underestimation  $\leftarrow 0$ ;
   if  $UP(\phi \wedge l)$  derives a contradiction then
3     apply CR in  $S_l$  if  $S_l$  contains a cycle structure;
     if  $UP(\phi \wedge \bar{l})$  derives a contradiction then
4       apply CR in  $S_{\bar{l}}$  if  $S_{\bar{l}}$  contains a cycle structure;
       underestimation  $\leftarrow 1$ ;
5   return new  $\phi$  and underestimation
6 end

```

MaxSatz: Optimized version of MaxSatz-07. The optimizations make MaxSatz substantially faster for Max-2SAT, but slightly slower for Max-3SAT when the clauses-to-variables ratio is small. All the following solvers are implemented on top of MaxSatz.

MaxSatz_c: The failed literal detection of MaxSatz is replaced by the following procedure: for every variable x such that $\text{occ2}(x) \geq 2$ and $\text{occ2}(\bar{x}) \geq 2$, call Algorithm 1. Compared with MaxSatz, after detecting failed literals l and $\bar{l}$, and incrementing the underestimation by 1, the inconsistent subformula $S_l \cup S_{\bar{l}} - \{l, \bar{l}\}$ is transformed by applying CR, so that MaxSatz_c has additional clauses for detecting other inconsistencies.

MaxSatz_c^p: MaxSatz_c but applying CR exhaustively at the root node as a preprocessing. For instances without binary clauses, MaxSatz_c^p is simply MaxSatz_c.

MaxSatz^p: MaxSatz but applying CR exhaustively at the root node as a preprocessing. For instances without binary clauses, MaxSatz^p is simply MaxSatz.

MaxSatz_c⁺: MaxSatz but applying CR exhaustively at each node after applying UP and the inference rules (Rule 1, Rule 2, Rule 3, and Rule 4), and before applying failed literal detection.

The exhaustive applications of CR in MaxSatz_c⁺ and in the preprocessing of MaxSatz_c^p and MaxSatz^p are not combined with failed literal detection. We do not know a priori whether they allow to transform an inconsistent subformula. Their comparison with MaxSatz_c will allow to see the effectiveness of CR combined with failed literal detection to transform inconsistent subformulas.

5 Experimental Results and Analysis

We compared the different versions of MaxSatz —on a Linux Cluster where the nodes have a 2GHz AMD Opteron processor with 1Gb of RAM— using sets of 100 random

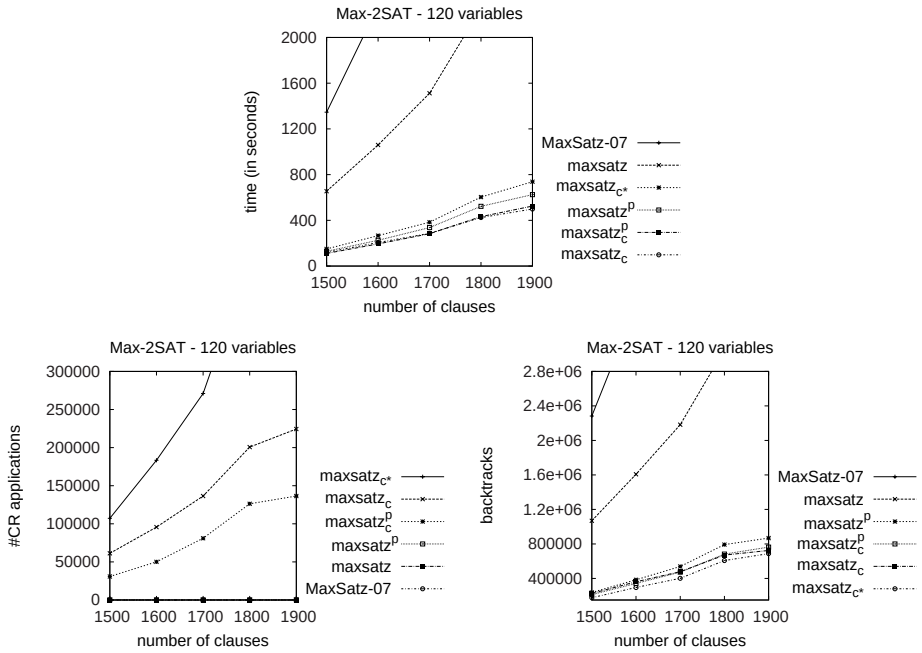


Fig. 1. Random Max-2SAT

Max-2SAT instances with 120 variables; the number of clauses ranged from 1500 to 1900. The results are shown in Figure 1: mean time needed to solve an instance of the set (top plot), mean number of CR applications without counting the applications of Rule 3 and Rule 4 (bottom left), and mean search tree size (bottom right).

It is quite easy to detect an inconsistent subformula in a Max-2SAT instance using unit propagation or failed literal detection, especially when the clauses-to-variables ratio is high. Since a cycle structure implies a failed literal, and its complementary literal easily fails during search, most cycle structures are likely contained in an inconsistent subformula detected using failed literal detection, explaining the behaviour of the exhaustive applications of CR in the preprocessing of MaxSatz_c^P and MaxSatz_c^P for random Max-2SAT, since most applications of CR probably allow to transform an inconsistent subformula. Nevertheless, MaxSatz_c, with guided CR applications aiming at transforming inconsistent subformulas, is always the best solver in terms of runtime: It is 5.4 times faster than MaxSatz for the hardest instances (1900 clauses). Notice that MaxSatz is substantially faster than MaxSatz-07 on Max-2SAT.

Additional experiments on Max-3SAT and Max-CUT instances, and on the instances from the 2007 MaxSAT Evaluation (not included here for lack of space) also provide empirical evidence that, in general, MaxSatz_c outperforms the rest of solvers, and applying CR exhaustively is the worst alternative, making MaxSatz_c* slower than MaxSatz for Max-3SAT.

References

1. Li, C.M., Manyà, F., Planes, J.: Exploiting unit propagation to compute lower bounds in branch and bound Max-SAT solvers. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 403–414. Springer, Heidelberg (2005)
2. Li, C.M., Manyà, F., Planes, J.: Detecting disjoint inconsistent subformulas for computing lower bounds for Max-SAT. In: Proceedings AAAI 2006, pp. 86–91 (2006)
3. Larrosa, J., Heras, F., de Givry, S.: A logical approach to efficient Max-SAT solving. *Artificial Intelligence* 172(2–3), 204–233 (2008)
4. Li, C.M., Manyà, F., Planes, J.: New inference rules for Max-SAT. *Journal of Artificial Intelligence Research* 30, 321–359 (2007)

Semi-automatic Generation of CHR Solvers for Global Constraints

Frank Raiser

Faculty of Engineering and Computer Sciences, University of Ulm, Germany
`Frank.Raiser@uni-ulm.de`

Abstract. Constraint programming often involves global constraints, for which various custom filtering algorithms have been published. This work presents a semi-automatic generation of CHR solvers for the subset of global constraints definable by specific automata. The generation is based on a constraint logic program modelling an automaton and an improved version of the Prim-Miner algorithm. The solvers only need to be generated once and achieve arc-consistency for over 40 global constraints.

1 Introduction

Global constraints are a combination of multiple constraints in order to improve filtering on the domains of involved variables. While it is common knowledge, that specialized filtering algorithms on global constraints achieve better domain pruning than generic approaches, the development and implementation of such algorithms requires a lot of effort. Thus, in [1] a generic method for deriving filtering algorithms from special checker automata is introduced.

Constraint Handling Rules (CHR) [2] is a multi-headed, guarded, and concurrent constraint programming language. CHR was designed for writing constraint solvers and is increasingly being used as a general-purpose programming language. However, there is no direct support for global constraints available in CHR so far.

The aim of this work is to adapt the results from [1] in order to generate CHR solvers for global constraints. To this end, we make use of the Prim-Miner algorithm proposed in [3] to generate rules from a constraint logic program (CLP). The necessary CLP is automatically created from the description of the automaton corresponding to a global constraint, as given by the global constraint catalog (GCC) [4,5]. This work presents a refined version of the Prim-Miner algorithm that adapts it to the problem at hand, which results in a significant improvement of runtime complexity.

2 Preliminaries

2.1 Automata for Global Constraints

In [1] automata for checking if a global constraint holds are introduced. The underlying idea is to compile a list of *signature arguments* from the arguments

of the global constraint and use this list to iterate through the automaton. These automata can use counters which are initialized to a value in the start state and can be modified at each transition. Additionally, the final state specifies a final value which has to hold for the counter in order for the global constraint to hold. We use $\mathcal{M} = \{1, \dots, |\mathcal{M}|\}$ for the set of transitions of an automaton.

Example 1. The **among** $(N, [X_1, \dots, X_k], \overline{V})$ constraint holds if exactly N variables from the set of variables $X_1, \dots, X_k$ take a value in the set $\overline{V}$. The corresponding automaton consists of two states. In the first state the value of the current variable X_i is checked for inclusion in the set $\overline{V}$ and a counter is incremented in that case. The second state is the final state where the final counter value is compared to N .

[1] further describes an arc-consistent filtering algorithm based on these automata and arc-consistent solvers for the ψ constraints as well as ϕ constraints, which are used to encode the transitions. In this work we generate CHR solvers for those constraints, which in combination with rules generating the necessary ψ and ϕ constraints allow arc-consistent filtering for global constraints. Note, however, that the arc-consistency result only holds for automata which do not involve counters. In all other cases the filtering algorithm and therefore the generated CHR solvers can still be used, but may not achieve arc-consistent filtering.

3 Semi-automatic Solver Generation

To generate a CHR solver for a global constraint, first the automaton definition is extracted from the global constraint catalog [5]. Then CHR rules for creating signature arguments, ψ and ϕ constraints are written, after which the solvers for ψ constraints and for ϕ constraints are generated. Optionally, the generated ruleset can be optimized in a post-processing step. The following sections present these steps in more detail.

3.1 Generation of ψ and ϕ Constraints

For the filtering algorithm proposed in [1] the generation of ψ and ϕ constraints for the automaton is required. Note that this step cannot be fully automated, as the signature arguments depend on the specific global constraint for which to generate a solver. In some cases, however, the generation of these constraints can be done in a canonical way as detailed in [6].

3.2 Generation of Solver for ψ Constraint

The generation of a solver for the ψ constraints assumes that all transition constraints C_i and their negations are available as arc-consistent built-in constraints. We also require that for all subsets of these constraints their union is available as a arc-consistent built-in constraint. This directly leads to the creation of rules of the following kind $\forall i \in \mathcal{M} : \psi(S, \Delta) \Rightarrow \Delta \notin C_i \mid S \in \mathcal{M} \setminus \{i\}$.

Intuitively, these rules make use of each transition i corresponding to a transition constraint C_i and state the fact that if this constraint does not hold the transition cannot be made. Thus, the corresponding identifier for the transition is removed from the possible transitions.

In order to achieve arc-consistency, however, further rules are required. Considering a domain restriction for $D(S) = \{i_1, \dots, i_k\}$ with $i_1, \dots, i_{|\mathcal{M}|}$ being a permutation of $\mathcal{M}$ and $0 < k \leq |\mathcal{M}|$, a constraint $C_{i_1} \cup \dots \cup C_{i_k}$ can be propagated: $\psi(S, \Delta) \wedge S \in \{i_1, \dots, i_k\} \Rightarrow \Delta \in (C_{i_1} \cup \dots \cup C_{i_k})$. Such rules are inserted for $D(S) = \{i_1, \dots, i_k\}$ being any of the possible subsets of $\mathcal{M}$ with the exception of $\emptyset$.

Assuming an automaton with $|\mathcal{M}|$ transition edges where each edge is labeled with a different constraint, $O(|\mathcal{M}|)$ rules of the first kind and $O(2^{|\mathcal{M}|})$ rules of the second kind are added. Using these $O(2^{|\mathcal{M}|})$ rules we have the following result:

Theorem 1. *The generated solver achieves arc-consistency for the ψ constraint.*

Example 2. The generated solver for the ψ constraint of the **among** global constraint consists of these rules:

-
1	$\psi(S, \Delta, \bar{V}) \Rightarrow \Delta \in \bar{V} \mid S \in \{0, 1\} \setminus \{0\}$
2	$\psi(S, \Delta, \bar{V}) \Rightarrow \Delta \notin \bar{V} \mid S \in \{0, 1\} \setminus \{1\}$
3	
4	$\psi(S, \Delta, \bar{V}) \wedge S \in \{0\} \Rightarrow \Delta \notin \bar{V}$
5	$\psi(S, \Delta, \bar{V}) \wedge S \in \{1\} \Rightarrow \Delta \in \bar{V}$
6	$\psi(S, \Delta, \bar{V}) \wedge S \in \{0, 1\} \Rightarrow \top$
-

3.3 Generation of Solver for ϕ Constraint

The solver for the ϕ constraint is based on the Prim-Miner algorithm. The basic idea is to encode the automaton in a constraint logic program P for the algorithm to work with and use all possible domains as candidate inputs.

Generation of CLP. Creating the CLP P for the ϕ constraints is a straightforward encoding of the automaton's transitions into CLP rules [6]. The generation of these CLPs can be fully automated given the definition of the automaton.

A check performed by the Prim-Miner algorithm against such a CLP consists of a backtracking search. If the given domain restrictions are consistent with one of the automaton's transitions the check succeeds and the check fails if the given domain restrictions do not allow for any of the transitions to fire.

Solver Generation. The generation of the solver for the ϕ constraint is performed by a modified version of the Prim-Miner algorithm, which we call the GC-Prim-Miner algorithm. It uses the previously generated CLP P against which to test goals. The resulting ruleset is a CHR solver for the ϕ constraint providing arc-consistency for global constraints whose automaton is free of counters:

Theorem 2. *For automata which do not involve counters the resulting rule set achieves arc-consistency for ϕ .*

As the runtime complexity of the direct application of the Prim-Miner algorithm is insufficient we can make use of the specifics of our application to improve it. By selecting the inputs in an advantageous way we can ensure, that the resulting ruleset still possesses the same propagation power, while at the same time drastically reducing the complexity of the algorithm. The details of this modification can be found in [6], along with the deduction of the runtime complexity now being $O(2^{3n+|\mathcal{M}|} + 2^{2n+2|\mathcal{M}|})$, whereas using the original Prim-Miner algorithm gives a complexity of $O(2^{2n} * 2^{2|\mathcal{M}|})$.

3.4 Post-processing of Rule Set

After generating a set of CHR rules with the GC-Prim-Miner algorithm the resulting rule set can be reduced. A large number of the generated rules is redundant, therefore, an additional post-processing of the rule set leads to a more concise solver.

In order to find redundant rules for removal, the results about operational equivalence of CHR programs in [7] can be applied. [7] presents a decidable, sufficient, and necessary syntactic condition to determine operational equivalence of CHR programs that are terminating and confluent [8]. We can apply this condition by removing each rule from the generated rule set successively, and check if the complete rule set and the rule set without that rule are operationally equivalent. If they are, the selected rule is redundant and can be removed.

Example 3. Using the GC-Prim-Miner algorithm on the CLP for the ϕ constraint for the **among** automaton and removing all redundant rules generates the following solver:

-
- 1 $\phi(Q, \overline{K}, S, Q', \overline{K'}) \Rightarrow Q \in \{s\}$
 - 2 $\phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge Q' \in \{t\} \Rightarrow Q \in \{s\} \wedge S \in \{\$ \}$
 - 3 $\phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge Q' \in \{s\} \Rightarrow Q \in \{s\} \wedge S \in \{0, 1\}$
 - 4 $\phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge S \in \{\$ \} \Rightarrow Q \in \{s\} \wedge Q' \in \{t\}$
 - 5 $\phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge S \in \{0\} \Rightarrow Q \in \{s\} \wedge Q' \in \{s\}$
 - 6 $\phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge S \in \{1\} \Rightarrow Q \in \{s\} \wedge Q' \in \{s\}$
 - 7 $\phi(Q, \overline{K}, S, Q', \overline{K'}) \wedge S \in \{0, 1\} \Rightarrow Q \in \{s\} \wedge Q' \in \{s\}$
-

4 Conclusion

In this paper we have shown a way to semi-automatically generate CHR solvers for the set of automata-describable global constraints. The process is not fully automated due to the generation of signature arguments and because signature constraints are not available in a suitable format in the global constraint catalog.

We have shown that by the use of the GC-Prim-Miner algorithm, and given the availability of arc-consistent built-in constraint solvers for transition constraints,

the generated CHR solvers achieve arc-consistency in those cases the automata-based filtering proposed in [1] allows for it. We have further shown, that the generality of the Prim-Miner algorithm can cause a runtime complexity problem, which can be alleviated by an order of magnitude if specialized for the problem at hand.

For future work the problems associated with the ψ constraint solver [6] need to be tackled. As there are few semantically different signature constraints used in the various automata it might be possible to develop arc-consistent solvers for these, including their negations and unions. Together with a way to automatically extract the signature constraints from the definitions given in the global constraint catalog this would allow for a fully automated generation of the CHR solvers.

References

1. 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)
2. Frühwirth, T.: Theory and practice of constraint handling rule. *Journal of Logic Programming, Special Issue on Constraint Logic Programming* 37(1-3), 95–138 (1998)
3. Abdennadher, S., Rigotti, C.: Automatic generation of CHR constraint solvers. *TPLP* 5(4-5), 403–418 (2005)
4. Beldiceanu, N., Demassey, S.: Global constraint catalog (2008), <http://www.emn.fr/x-info/sdemasse/gccat/>
5. Beldiceanu, N., Carlsson, M., Demassey, S., Petit, T.: Global constraint catalog: Past, present and future. *Constraints* 12(1), 21–62 (2007)
6. Raiser, F.: Semi-automatic generation of CHR solvers from global constraint automata. Technical Report UIB-2008-03, Ulmer Informatik Berichte, Universität Ulm (February 2008)
7. Abdennadher, S., Frühwirth, T.: Operational equivalence of CHR programs and constraints. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 43–57. Springer, Heidelberg (1999)
8. Abdennadher, S.: Operational semantics and confluence of constraint propagation rules. In: *Principles and Practice of Constraint Programming*, pp. 252–266 (1997)

Stochastic Local Search for the Optimal Winner Determination Problem in Combinatorial Auctions

Dalila Boughaci¹, Belaid Benhamou¹, and Habiba Drias²

¹ INCA/LSIS, CMI 39 rue Fredric Joliot-Curie Marseille 130013
boughaci@cmi.univ-mrs.fr, belaid.benhamou@cmi.univ-mrs.fr

² LRIA/USTHB- BP 32 El-Alia, Beb-Ezsoaur, 16111, Alger, Algérie
dboughaci@usthb.dz, hdrias@usthb.dz

Abstract. In this paper, a stochastic local search (SLS) is studied for the optimal winner determination problem (WDP) in combinatorial auctions. Experiments on various realistic instances of the considered problem are performed to show and compare the effectiveness of our approach. The computational experiments show that the SLS provides competitive results and finds solutions of a higher quality than Tabu search and Casanova methods.

1 Introduction

The combinatorial auction (CA) is the mechanism that allows agents (bidders) to bid on bundles of items (goods). It allows the bidders to express both complementarity¹ and substitutability² of their preferences within bids. The combinatorial auction avoids the risk to obtain incomplete bundles since the seller allows bids on bundles of items.

In this paper, we are interested in the optimal winner determination problem (WDP) in combinatorial auctions. The WDP is a complex problem and it is equivalent to a weighted set packing problem which is *NP-Complete* [5].

The optimal winner determination problem in combinatorial auctions can be stated as follows:

Let us consider a set of m items, $M = \{1, 2 \dots m\}$ to be auctioned and a set of n bids, $B = \{B_1, B_2 \dots B_n\}$. A bid B_j is a tuple $\langle S_j, P_j \rangle$ where S_j is a set of items, and P_j is the price of B_j ($P_j \succeq 0$). Further, consider a matrix $a_{m \times n}$ having m rows and n columns where $a_{ij} = 1$ iff the item i belongs to S_j , $a_{ij} = 0$, otherwise. Finally the decision variables are defined as follows: $x_j = 1$ iff the bid B_j is accepted (a winning bid), and $x_j = 0$ otherwise (a losing bid).

¹ Complementarity between items means that the value assigned to a collection of goods is greater than the sum of the values assigned to its individual's elements.

² Substitutability means that the value assigned to a collection of goods is lower than the sum of the value attached to its individual's elements.

The *WDP* is the problem of finding winning bids that maximize the auctioneer's revenue under the constraint that each item can be allocated to at most one bidder. The *WDP* can be modeled as the following integer program [2]:

$$\text{Maximize} \quad \sum_{j=1}^n P_j . x_j \quad (1)$$

$$\text{Under the constraints: } \sum_{j=1}^n a_{ij} x_j \leq 1 \quad i \in \{1 \dots m\} \quad (2)$$

$$x_j \in \{0, 1\} \quad (3)$$

The objective function (1) maximizes the auctioneer's revenue which is equal to the sum of the prices of the winning bids. The constraints (2) express the fact that each item can be allocated to at most one bidder. Due to the free disposal assumption, some items could be left uncovered.

Several exact algorithms to search optimal solutions for the WDP problem have been developed. Among them, we cite : the iterative deepening A^* , the Branch-on-Items (BoI), the Branch on Bids (BoB), and the Combinatorial Auctions BoB (CABoB)[6]. On other hand, different inexact methods are studied for the WDP. Among them Casanova [3] and the hybrid simulated annealing [2]. For more details about auctions and the WDP, the reader can refer to [6].

In this paper, we develop a SLS algorithm for the WDP. The proposed algorithm makes use of the random key encoding (RK) introduced by [1] and used mainly for ordering and scheduling problems. The RK encoding mechanism permits to generate and manipulate a feasible solution. Therefore, no additional penalties for invalid solutions are necessary. We note that bids are in conflict if they share an item. To maintain a feasible allocation along the search process, we have defined a conflict graph where bids are vertices and edges connect bids that cannot be accepted together. This graph is useful for removing any conflicting bids occurring in the current allocations when new bids are added.

2 Proposed Approach

In order to explore the most important part of the whole search space for attaining good solutions, we propose a search technique (SLS). A solution of the WDP problem can be defined as a combination of bids satisfying the target goals described in the objective function. We use an allocation A (a Vector with a variable length). Each of whose components A_i receives the winning bid number. The initial solution of the SLS is generated randomly according to the Random Key Encoding that operates as follows: we generate n real numbers sequenced by an r order where n is the number of bids and the r order is a permutation of keys values. To generate an allocation, first we select the bid having the highest order value to include in the allocation. Secondly, the bid having the second-highest order value is accepted if its acceptance with accepted bid currently in the allocation does not create any conflict, otherwise it is discarded. The process

is repeated until having examined the n bids. We obtain a subset of bids that can be a feasible solution to the WDP.

The proposed SLS starts with an initial allocation A generated randomly according to the Rk encoding. Then, it performs a certain number of local steps that consists in selecting a bid to be added in the current allocation A and in removing all conflicting bids that can be occurred in the current allocation. At each step, the bid to be accepted is selected according to one of the two following criteria:

1. The first criterion (*step1* of Algorithm 1) consists in choosing the bid in a random way with a fixed probability $wp > 0$.
2. The second criterion (*step2*) consists in choosing the best bid (the one maximizing the auctioneer's revenue when it is selected) to be accepted.

The process is repeated for a certain number of iterations called *maxiter* fixed empirically. The SLS algorithm is sketched in Algorithm 1.

Algorithm 1. The SLS method

Require: a WDP formula, an allocation A , *maxiter*, wp

Ensure: an improved allocation A

```

1: for  $I = 1$  to maxiter do
2:    $r \leftarrow$  random number between 0 and 1
3:   if  $r < wp$  then
4:      $bid =$  pick a random bid (*Step 1)
5:   else
6:      $bid =$  pick a best bid; (*Step 2)
7:   end if
8:    $A = A$  with  $bid$  included into it;
9:   remove from  $A$  any conflicting bids;
10: end for
return the best allocation found.
```

3 Computational Experiments

This section gives some preliminary results. The C programming language is used to implement our SLS algorithm for the WDP. The source codes are run on a Pentium- IV 2.8 GHz, 1GB of RAM.

To measure the performance of algorithms for the WDP, Lau and Goh provided benchmarks of various sizes consisting of up to 1500 items and 1500 bids [4]. These data sets allow for several factors such as a pricing factor, a bidder preference factor and a fairness factor in distributing items among bids.

In this paper, we use the realistic data pre-generated by [4] for which the CPLEX was unable to find the optimal solution in reasonable time [2]. The pre-generated data set includes 500 instances and it is available at the Zhuyi's home page³. These instances can be divided into 5 different groups of problems where each group contains 100 instances.

³ <http://logistics.ust.hk/~zhuyi/instance.zip>

3.1 Comparison with Casanova Local Search, Tabu Search and SAGII

To show the effectiveness of our approach, we compared the SLS with a tabu search algorithm (TS) which we have implemented. A comparative study with some algorithms of the state of the art concerning the WDP (Casanova and SAGII [2]) is given also in this section.

Tables 1 and 2 summarize the results for the 500 test instances with the 5 different problem sizes. The column μ corresponds to the arithmetic average solution of the 100 instances in each group, the column *time* corresponds to the average time in second. δ_1 is given by the expression $(\mu_{SLS} - \mu_{Casanova}) / \mu_{SLS}$. δ_2 is $(\mu_{SLS} - \mu_{TS}) / \mu_{SLS}$ and δ_3 equals to $(\mu_{SLS} - \mu_{SAGII}) / \mu_{SLS}$.

Table 1. Comparison of SLS with Casanova and TS

Groups of problems	SLS		Casanova		$\delta_1\%$	TS		$\delta_2\%$
	<i>time</i>	μ	<i>time</i>	μ		<i>time</i>	μ	
REL-500-1000	22.35	64216.14	119.46	37053.78	42,30	91,07	65286,94	-164
REL-1000-500	5.91	72206.07	57.74	51248.79	40,89	25,84	71985,34	0.30
REL-1000-1000	14.19	82120.31	111.42	51990.91	36,68	104,30	81633,63	0.60
REL-1000-1500	14.97	79065.08	168.24	56406.74	28,65	223,37	77931,41	1.43
REL-1500-1500	16.47	98877.07	165.92	65661.03	33,59	175,68	97824,64	1.06

Table 1 shows that SLS always gives a 28 to 43 percent improvement in results in comparison to Casanova. The SLS performs better than Casanova. It finds better solutions in shorter time. The difference between SLS and Casanova is even greater. Table 1 shows also good performances of the SLS in solving the WDP compared to TS.

The SLS and TS find good quality solutions for almost all the benchmarks efficiently. They are definitely better than the Casanova that fails to find good solutions for all the instances. It can be seen that the SLS is the fastest algorithm. However, for the REL 500-1000 group of problems, TS outperforms SLS in term of solutions quality.

Table 2. SLS vs. SAGII

Groups of problems	SLS		SAGII		$\delta_3\%$
	<i>time</i>	μ	<i>time</i>	μ	
REL-500-1000	22.35	64216.14	38.06	64922.02	-1.08
REL-1000-500	5.91	72206.07	24.46	73922.10	-2.32
REL-1000-1000	14.19	82120.31	45.37	83728.34	-1.92
REL-1000-1500	14.97	79065.08	68.82	82651.49	-4.33
REL-1500-1500	16.47	98877.07	91.78	101739.64	-2.81

Table 2 compares SLS and SAGII. The SLS produces quite similare results to SAGII in spite the sophisticate Branch-and-Bound and preprocessing tools used in SAGII.

It should be signaled that the SLS method and SAGII are not directly comparable, since the SLS does not use the pre-processor that was used in [2]. This pre-processor permits to improve on time and reduces mainly the required search effort. Our aim is to compute the power of the SLS comparing to other methods in particular local search family methods. However adding such pre-processor on SLS should improve it and permits to solve problems in shorter time.

4 Conclusion

A stochastic local search for the winner determination problem (WDP) is proposed in this paper with its different components. The proposed SLS method uses a random key encoding mechanism to generate a feasible combination of bids. The SLS method is evaluated on several realistic instances and compared with TS, SGAI and Casanova. The experimental results are very encouraging. To improve our algorithm on quality, new features will be integrated into the proposed algorithm such as the combination of the SLS and a Branch-and-Bound exact method. Our purpose is to find a good compromise when combining exact approaches with inexact ones. To improve on time, preprocessors will be added to exclude bids that can lead to suboptimal solutions.

References

1. Bean, J.C.: Genetics and random keys for sequencing and optimization. *ORSA Journal of Computing* 6(2), 154–160 (1994)
2. Guo, Y., Lim, A., Rodrigues, B., Zhu, Y.: Heuristics for a bidding problem. *Computers and Operations Research* 33(8), 2179–2188 (2006)
3. Hoos, H.H., Boutilier, C.: Solving combinatorial auctions using stochastic local search. In: *Proceedings of the 17th national conference on artificial intelligence*, pp. 22–29 (2000)
4. Lau, H.C., Goh, Y.G.: An intelligent brokering system to support multi-agent web-based 4th-party logistics. In: *Proceedings of the 14th international conference on tools with artificial intelligence*, pp. 54–61 (2002)
5. Rothkopf, M.H., Pekee, A., Ronald, M.: Computationally manageable combinatorial auctions. *Management Science* 44(8), 1131–1147 (1998)
6. Sandholm, T.: *Optimal Winner Determination Algorithms*. In: Cramton, P., et al. (eds.) *Combinatorial Auctions*, MIT Press, Cambridge (2006)

Revisiting the Upper Bounding Process in a Safe Branch and Bound Algorithm^{*}

Alexandre Goldsztejn¹, Yahia Lebbah^{2,3}, Claude Michel³, and Michel Rueher³

¹ CNRS / Université de Nantes 2, rue de la Houssinière, 44322 Nantes, France
`alexandre.goldsztejn@univ-nantes.fr`

² Université d'Oran Es-Senia B.P. 1524 EL-M'Naouar, 31000 Oran, Algeria
`ylebbah@gmail.com`

³ Université de Nice-Sophia Antipolis, I3S-CNRS, 06903 Sophia Antipolis, France
`{cpjm,rueher}@polytech.unice.fr`

Abstract. Finding feasible points for which the proof succeeds is a critical issue in safe Branch and Bound algorithms which handle continuous problems. In this paper, we introduce a new strategy to compute very accurate approximations of feasible points. This strategy takes advantage of the Newton method for under-constrained systems of equations and inequalities. More precisely, it exploits the optimal solution of a linear relaxation of the problem to compute efficiently a promising upper bound. First experiments on the Coconuts benchmarks demonstrate that this approach is very effective.

1 Introduction

Optimization problems are a challenge for CP in finite domains; they are also a big challenge for CP on continuous domains. The point is that CP solvers are much slower than classical (non-safe) mathematical methods on nonlinear constraint problems as soon as we consider optimization problems. The techniques introduced in this paper try to boost constraints techniques on these problems and thus, to reduce the gap between efficient but unsafe systems like BARON¹, and the slow but safe constraint based approaches. We consider here the global optimization problem $\mathcal{P}$ to minimize an objective function under nonlinear equalities and inequalities,

$$\begin{aligned} & \text{minimize } f(x) \\ & \text{subject to } g_i(x) = 0, \ i \in \{1, \dots, k\} \\ & \quad h_j(x) \leq 0, \ j \in \{1, \dots, m\} \end{aligned} \tag{1}$$

with $x \in \mathbf{x}$, $f : \mathbb{R}^n \rightarrow \mathbb{R}$, $g_i : \mathbb{R}^n \rightarrow \mathbb{R}$ and $h_j : \mathbb{R}^n \rightarrow \mathbb{R}$; Functions f , g_i and h_j are nonlinear and continuously differentiable on some vector $\mathbf{x}$ of intervals

^{*} An extended version of this paper is available at:

<http://www.i3s.unice.fr/%7Emh/RR/2008/RR-08.11-A.GOLDSZTEJN.pdf>

¹ See <http://www.andrew.cmu.edu/user/ns1b/baron/baron.html>

of $\mathbb{R}$. For convenience, in the sequel, $g(x)$ (resp. $h(x)$) will denote the vector of $g_i(x)$ (resp. $h_j(x)$) functions.

The difficulties in such global optimization problems come mainly from the fact that many local minimizers may exist but only few of them are global minimizers [3]. Moreover, the feasible region may be disconnected. Thus, finding feasible points is a critical issue in safe Branch and Bound algorithms for continuous global optimization. Standard strategies use local search techniques to provide a reasonable approximation of an upper bound and try to prove that a feasible solution actually exists within the box around the guessed global optimum. Practically, finding a guessed point for which the proof succeeds is often a very costly process.

In this paper, we introduce a new strategy to compute very accurate approximations of feasible points. This strategy takes advantage of the Newton method for under-constrained systems of equations and inequalities. More precisely, this procedure exploits the optimal solution of a linear relaxation of the problem to compute efficiently a promising upper bound. First experiments on the Coconuts benchmarks demonstrate that the combination of this procedure with a safe Branch and Bound algorithm drastically improves the performances.

2 The Branch and Bound Schema

The algorithm (see Algorithm 1) we describe here is derived from the well known Branch and Bound schema introduced by Horst and Tuy for finding a global minimizer. Interval analysis techniques are used to ensure rigorous and safe computations whereas constraint programming techniques are used to improve the reduction of the feasible space.

Algorithm 1 computes enclosers for minimizers and safe bounds of the global minimum value within an initial box $\mathbf{x}$. Algorithm 1 maintains two lists : a list $\mathcal{L}$ of boxes to be processed and a list $\mathcal{S}$ of proven feasible boxes. It provides a rigorous enclosure $[L, U]$ of the global optimum with respect to a tolerance ϵ .

Algorithm 1 starts with *UpperBounding*($\mathbf{x}, nbStarts$) which computes a set of feasible boxes by calling a local search with *nbStarts* starting points and a proof procedure.

The box around the local solution is added to $\mathcal{S}$ if it is proved to contain a feasible point. At this stage, if the box $\mathbf{x}'$ is empty then, either it does not contain any feasible point or its lower bound $\underline{\mathbf{f}}_{\mathbf{x}'}$ is greater than the current upper bound U . If $\mathbf{x}'$ is not empty, the box is split along one of the variables² of the problem.

In the main loop, algorithm 1 selects the box with the lowest lower bound of the objective function. The *Prune* function applies filtering techniques to reduce the size of the box $\mathbf{x}'$. In the framework we have implemented, *Prune* just uses a 2B-filtering algorithm [2]. Then, *LowerBound*($\mathbf{x}'$) computes a rigorous lower bound $\underline{\mathbf{f}}_{\mathbf{x}'}$ using a linear programming relaxation of the initial problem. Actually, function *LowerBound* is based on the linearization techniques of the Quad-framework [1]. *LowerBound* computes a safe minimizer $\underline{\mathbf{f}}_{\mathbf{x}'}$ thanks to the techniques introduced by Neumaier et al.

² Various heuristics are used to select the variable the domain of which has to be split.

Algorithm 1. Branch and Bound algorithm**Function** BB(IN $\mathbf{x}$, ϵ ; OUT $\mathcal{S}$, $[L, U]$)

```

%  $\mathcal{S}$ : set of proven feasible points
%  $\mathbf{f}_{\mathbf{x}}$  denotes the set of possible values for  $f$  in  $\mathbf{x}$ 
%  $nbStarts$ : number of starting points in the first upper-bounding
 $\mathcal{L} \leftarrow \{\mathbf{x}\}; \quad (L, U) \leftarrow (-\infty, +\infty); \quad \mathcal{S} \leftarrow UpperBounding(\mathbf{x}', nbStarts);$ 
while  $w([L, U]) > \epsilon$  do
   $\mathbf{x}' \leftarrow \mathbf{x}''$  such that  $\underline{\mathbf{f}}_{\mathbf{x}''} = \min\{\underline{\mathbf{f}}_{\mathbf{x}''} : \mathbf{x}'' \in \mathcal{L}\}; \quad \mathcal{L} \leftarrow \mathcal{L} \setminus \mathbf{x}'; \quad \bar{\mathbf{f}}_{\mathbf{x}'} \leftarrow \min(\bar{\mathbf{f}}_{\mathbf{x}'}, U);$ 
   $\mathbf{x}' \leftarrow Prune(\mathbf{x}'); \quad \underline{\mathbf{f}}_{\mathbf{x}'} \leftarrow LowerBound(\mathbf{x}'); \quad \mathcal{S} \leftarrow \mathcal{S} \cup UpperBounding(\mathbf{x}', 1);$ 
  if  $\mathbf{x}' \neq \emptyset$  then  $(\mathbf{x}'_1, \mathbf{x}'_2) \leftarrow Split(\mathbf{x}'); \quad \mathcal{L} \leftarrow \mathcal{L} \cup \{\mathbf{x}'_1, \mathbf{x}'_2\};$ 
  if  $\mathcal{L} = \emptyset$  then  $(L, U) \leftarrow (+\infty, -\infty)$ 
  else  $(L, U) \leftarrow (\min\{\underline{\mathbf{f}}_{\mathbf{x}''} : \mathbf{x}'' \in \mathcal{L}\}, \min\{\bar{\mathbf{f}}_{\mathbf{x}''} : \mathbf{x}'' \in \mathcal{S}\})$ 
endwhile

```

Algorithm 1 maintains the lowest lower bound L of the remaining boxes $\mathcal{L}$ and the lowest upper bound U of proven feasible boxes. The algorithm terminates when the space between U and L becomes smaller than the given tolerance ϵ .

The Upper-bounding step (see Algorithm 2) performs a multistart strategy where a set of $nbStarts$ starting points are provided to a local optimization solver. The solutions computed by the local solver are then given to a function *InflateAndProve* which uses an existence proof procedure based on the Borsuk test. However, the proof procedure may fail to prove the existence of a feasible point within box $\mathbf{x}_p$. The most common source of failure is that the “guess” provided by the local search lies too far from the feasible region.

3 A New Upper Bounding Strategy

The upper bounding procedure described in the previous section relies on a local search to provide a “guessed” feasible point lying in the neighborhood of a local optima. However, the effects of floating point computation on the provided local optima are hard to predict. As a result, the local optima might lie outside the feasible region and the proof procedure might fail to build a proven box around this point.

We propose here a new upper bounding strategy which attempts to take advantage of the solution of a linear outer approximation of the problem. The lower bound process uses such an approximation to compute a safe lower bound of $\mathcal{P}$. When the LP is solved, a solution x_{LP} is always computed and, thus, available for free. This solution being an optimal solution of an outer approximation of $\mathcal{P}$, it lies outside the feasible region. Thus, x_{LP} is not a feasible point. Nevertheless, x_{LP} may be a good starting point to consider for the following reasons:

- At each iteration, the branch and bound process splits the domain of the variables. The smaller the box is, the nearest x_{LP} is from the actual optima of $\mathcal{P}$.
- The proof process inflates a box around the initial guess. This process may compensate the effect of the distance of x_{LP} from the feasible region.

Algorithm 2. Upper bounding build from the LP optimal solution x_{LP}^*

Function UpperBounding(IN $\mathbf{x}$, x_{LP}^* , $nbStarts$; OUT $\mathcal{S}'$)

```
%  $\mathcal{S}'$ : list of proven feasible boxes;   $nbStarts$ : number of starting points
%  $x_{LP}^*$ : the optimal solution of the LP relaxation of  $\mathcal{P}(\mathbf{x})$ 
 $\mathcal{S}' \leftarrow \emptyset$ ;   $x_{corr}^* \leftarrow \text{FeasibilityCorrection}(x_{LP}^*)$ ;   $\mathbf{x}_p \leftarrow \text{InflateAndProve}(x_{corr}^*, \mathbf{x})$ ;
if  $\mathbf{x}_p \neq \emptyset$  then  $\mathcal{S}' \leftarrow \mathcal{S}' \cup \mathbf{x}_p$ 
return  $\mathcal{S}'$ 
```

However, while x_{LP} converges to a feasible point, the process might be quite slow. To speed up the upper bounding process, we have introduced a light weight, though efficient, procedure which compute a feasible point from a point lying in the neighborhood of the feasible region. This procedure which is called *FeasibilityCorrection* will be detailed in the next subsection.

Algorithm 2 describes how an upper bound may be build from the solution of the linear problem used in the lower bounding procedure.

4 Computing Pseudo-feasible Points

This section introduces an adaptation of the Newton method to under-constrained systems of equations and inequalities which provides very accurate approximations of feasible points at a low computational cost. When the system of equations $g(x) = 0$ is under-constrained there is a manifold of solutions. $l(x)$, the linear approximation is still valid in this situation, but the linear system of equations $l(x) = 0$ is now under-constrained, and has therefore an affine space of solutions. So we have to choose a solution $x^{(1)}$ of the linearized equation $l(x) = 0$ among the affine space of solutions. As $x^{(0)}$ is supposed to be an approximate solution of $g(x) = 0$, the best choice is certainly the solution of $l(x) = 0$ which is the closest to $x^{(0)}$. This solution can easily be computed with the Moore-Penrose inverse: $x^{(1)} = x^{(0)} - A_g^+(x^{(0)})g(x^{(0)})$, where $A_g^+ \in \mathbb{R}^{n \times m}$ is the Moore-Penrose inverse of $A_g \in \mathbb{R}^{m \times n}$, the solution of the linearized equation which minimizes $\|x^{(1)} - x^{(0)}\|$. Applying previous relation recursively leads to a sequence of vectors which converges to a solution close to the initial approximation, provided that this latter is accurate enough.

The Moore-Penrose inverse can be computed in several ways: a singular value decomposition can be used, or in the case where A_g has full row rank (which is the case for $A_g(x^{(0)})$ if $x^{(0)}$ is non-singular) the Moore-Penrose inverse can be computed using $A_g^+ = A_g^T(A_g A_g^T)^{-1}$.

Inequality constraints are changed to equalities by introducing slack variables: $h_j(x) \leq 0 \iff h_j(x) = -s_i^2$. So the Newton method for under-constrained systems of equations can be applied.

5 Experiments

In this Section, we comment the results of the experiments with our new upper bounding strategy on a significant set of benchmarks. Detailed results can be found in the research report ISRN I3S/RR-2008-11-FR³). All the benchmarks come from the collection of benchmarks of the Coconuts project⁴. We have selected 35 benchmarks where Icos succeeds to find the global minimum while relying on an unsafe local search. We did compare our new upper bounding strategy with the following upper bounding strategies:

- S1: This strategy directly uses the guess from the local search, i.e. this strategy uses a simplified version of algorithm 1 where the proof procedure has been dropped. As a consequence, it does not suffer from the difficulty to prove the existence of a feasible point. However, this strategy is unsafe and may produce wrong results.
- S2: This strategy attempts to prove the existence of a feasible point within a box build around the local search guess. Here, all provided solutions are safe and the solving process is rigorous.
- S3: Our upper bounding strategy where the upper bounding relies on the optimal solution of the problem linear relaxation to build a box proved to hold a feasible point. A call to the correction procedure attempts to compensate the effect of the outer approximation.
- S4: This strategy applies the correction procedure to the output of the local search (to improve the approximate solution given by a local search).
- S5: This strategy mainly differs from S3 by the fact that it does not call the correction procedure

S3, our new upper bounding strategy is the best strategy: 31 benchmarks are now solved within the 30s time out; moreover, almost all benchmarks are solved in much less time and with a greater amount of proven solutions. This new strategy improves drastically the performance of the upper bounding procedure and competes well with a local search.

Current work aims at improving and generalizing this framework and its implementation.

References

1. Lebbah, Y., Michel, C., Rueher, M., Daney, D., Merlet, J.-P.: Efficient and safe global constraints for handling numerical constraint systems. *SIAM Journal on Numerical Analysis* 42(5), 2076–2097 (2004)
2. Lhomme, O.: Consistency techniques for numeric CSPs. In: *Proceedings of IJCAI 1993*, Chambéry, France, pp. 232–238 (1993)
3. Neumaier, A.: Complete search in continuous global optimization and constraint satisfaction. *Acta Numerica* (2004)

³ <http://www.i3s.unice.fr/%7Emh/RR/2008/RR-08.11-A.GOLDSZTEJN.pdf>

⁴ See <http://www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Benchmark.html>.

Computing All Optimal Solutions in Satisfiability Problems with Preferences

Emanuele Di Rosa, Enrico Giunchiglia, and Marco Maratea

DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy
{emanuele,enrico,marco}@dist.unige.it

Abstract. The problem of finding an optimal solution in a constraint satisfaction problem with preferences has attracted a lot of researchers in Artificial Intelligence in general, and in the constraint programming community in particular. As a consequence, several approaches for expressing and reasoning about satisfiability problems with preferences have been proposed, and viable solutions exist for finding one optimal solution. However, in many cases, it is not desirable to find just one solution. Indeed, it might be desirable to be able to compute more, and possibly all, optimal solutions, e.g., for comparatively evaluate them on the basis of other criteria not captured by the preferences.

In this paper we present a procedure for computing all optimal solutions of a satisfiability problem with preferences. The procedure is guaranteed to compute all and only the optimal solutions, i.e., models which are not optimal are not even computed.

1 Introduction

The problem of finding an optimal solution in a constraint satisfaction problem with qualitative preferences has attracted a lot of researchers in Artificial Intelligence in general, and in the constraint programming community in particular.

As a consequence, several approaches for expressing and reasoning about satisfiability (SAT) problems with preferences have been proposed, and viable solutions exist for finding one optimal solution, see, e.g., [1,2]. However, in many cases, it is not desirable to find just one solution. Indeed, it might be desirable to be able to compute more, and possibly all, optimal solutions, e.g., for comparatively evaluate them on the basis of other criteria not captured by the preferences. As an example of the practical importance of the issue, from the ILOG web page¹: "ILOG CPLEX 11 introduces the solution pool feature, which allows users to consider multiple solutions to a MIP model. In practice, a single, even optimal, solution is not always sufficient, because every aspect of a problem cannot always be perfectly captured in a MIP model. The solution pool feature offers a mechanism for exploring the effects of subjective preferences on the solution space without enforcing them as constraints in the model".

A simple approach for finding all optimal solutions consists in first enumerating all (non necessarily optimal) solutions, and then eliminating a solution μ if there exists another solution μ' which is "preferred" to μ . The first obvious drawback of this approach is that it requires the computation of all solutions, even the non optimal ones.

¹ <http://www.ilog.com/products/cplex/news/whatsnew.cfm>

The second drawback is that each solution has to be stored and compared with the others. In [3], in the context of CP-nets, the authors noticed that by imposing an ordering on the splitting heuristic used for searching solutions, it is possible to mitigate the second drawback by comparing a solution only with the previously generated ones, which are already guaranteed to be optimal: In this way, only the so far generated optimal solutions need to be stored. Still, the number of optimal solutions can be exponential and all the solutions (even the non optimal ones) are computed.

In this paper we present a procedure for computing all optimal solutions of a SAT problem with qualitative preferences which is guaranteed to compute all and only the optimal solutions, i.e., models which are not optimal are not even computed. In our setting, a qualitative preference is a partially ordered set of literals $S, \prec$: S is the set of literals that we would like to have satisfied, and $\prec$ is partial order on S expressing the relative importance of fulfilling the literals in S . For this result, it is essential that the splitting heuristic follows the partial order on the expressed preferences: Imposing such ordering can lead to significant degradations in the performances of the solver [4], though this has been shown to happen only when the number of preferences is very high (in the order of the number of variables in the problem [2]), and this is not the case for many applications, see, e.g., [5].

2 Satisfiability and Qualitative Preferences

Consider a finite set P of variables. A *literal* is a variable x or its negation $\neg x$. A *formula* is either a variable or a finite combination of formulas using the n -ary connectives $\wedge, \vee$ for conjunction and disjunction ($n \geq 0$), and the unary connective $\neg$ for negation. We use the symbols $\perp$ and $\top$ to denote the empty disjunction and conjunction respectively. If l is a literal, we write $\bar{l}$ for $\neg l$ and we assume $\overline{\overline{x}} = x$. This notation is extended to sets S of literals, i.e., $\overline{S} = \{\bar{l} : l \in S\}$. Formulas are used to express hard constraints that have to be satisfied. For example, given the 4 variables *Fish*, *Meat*, *RedWine*, *WhiteWine*, the formula

$$(\overline{Fish} \vee \overline{Meat}) \wedge (\overline{RedWine} \vee \overline{WhiteWine}) \quad (1)$$

models the fact that we cannot have both fish (*Fish*) and meat (*Meat*), both red (*RedWine*) and white (*WhiteWine*) wine.

An *assignment* is a consistent set of literals. If $l \in \mu$, we say that both l and $\bar{l}$ are *assigned* by μ . An assignment μ is *total* if each literal l is assigned by μ . A total assignment μ *satisfies*

- a literal l if and only if $l \in \mu$,
- $(\varphi_1 \vee \dots \vee \varphi_n)$ ($n \geq 0$) if and only if μ satisfies at least one φ_i with $1 \leq i \leq n$,
- $(\varphi_1 \wedge \dots \wedge \varphi_n)$ ($n \geq 0$) if and only if μ satisfies all φ_i with $1 \leq i \leq n$,
- the negation of a formula $\neg\psi$ if and only if μ does not satisfy ψ .

A *model* of a formula φ is a total assignment satisfying φ . A formula φ *entails* a formula ψ if the models of φ are a subset of the models of ψ . For instance, (1) has 9 models. In the following, we represent a total assignment as the set of variables assigned to

true. For instance, $\{Fish, WhiteWine\}$ represents the total assignment in which the only variables assigned to true are *Fish* and *WhiteWine*, i.e., the situation in which we have fish and white wine.

A (qualitative) *preference (on literals)* is a partially ordered set of literals, i.e., a pair $S, \prec$ where (i) S is a set of literals, called the *set of preferences*, which represents the set of literals that we would like to have satisfied; and (ii) $\prec$ is a partial order on S : $l \prec l'$ models the fact that we prefer l to l' . For example,

$$\{Fish, Meat, \overline{RedWine}\}, \{Fish \prec Meat\} \quad (2)$$

models the case in which we prefer to have both fish and meat, and avoid red wine; in the case in which it is not possible to have both fish and meat, we prefer to have the fish more than the meat.

A qualitative preference $S, \prec$ on literals can be extended to the set of total assignments as follows: Given two total assignments μ and μ' , we say that μ is *preferred to* μ' ($\mu \prec \mu'$) if and only if (i) there exists a literal $l \in S$ with $l \in \mu$ and $\bar{l} \in \mu'$; and (ii) for each literal $l' \in S \cap (\mu' \setminus \mu)$, there exists a literal $l \in S \cap (\mu \setminus \mu')$ such that $l \prec l'$. A model μ of a formula φ is *optimal* if it is a minimal element of the partially ordered set of models of φ . For instance, considering the qualitative preference (2), the formula (1) has only two optimal models, i.e., $\{Fish\}$ and $\{Fish, WhiteWine\}$.

Consider a formula φ , a qualitative preference $S, \prec$ and a set Γ of optimal models of φ . Γ is said to be *complete* (wrt $S, \prec$) if contains all the optimal models of φ . With this notion, there can be only one complete set of optimal models, which, in the case of (1) and the preference (2), is the set $\{\{Fish, WhiteWine\}, \{Fish\}\}$. Other notions of completeness are possible.

3 Computing All Optimal Solutions

Given a formula φ and a preference, we now show how it is possible to compute a complete set of models of φ by extending the Davis-Logemann-Loveland procedure (DLL) [6] and the procedure in [2] for computing one optimal solution. DLL is the most used decision procedure for checking satisfiability of formulas. However, DLL does not directly handle arbitrary formulas, but finite sets of clauses, where a *clause* is a finite set of literals to be interpreted disjunctively. This is not a limitation because of well known clause form transformation procedures (see, e.g., [7,8]).

In the following, we will continuously switch between formulas and sets of clauses, intuitively meaning the same thing.

Consider a formula φ and a preference $S, \prec$. An assignment μ dominates an assignment μ' (wrt $S, \prec$) if $\mu \prec \mu'$. The problem of computing a complete set of optimal models of φ wrt $S, \prec$ can be solved by considering the following crucial condition which enables us to say which are the assignments that are dominated by μ (wrt $S, \prec$). We therefore define a formula whose models are dominated by μ . Consider a total assignment μ .

1. $n(\mu)$ is the set of preferences not satisfied by μ , i.e., $n(\mu) = S \cap \bar{\mu}$
2. for each $l \in S$, $d(l, \mu)$ is the set of literals in μ which are preferred to l according to $\prec$, i.e., $d(l, \mu) = \{l' : l' \in \mu, l' \prec l \text{ or } \bar{l'} \prec l\}$.

Then the μ -dominates formula (wrt $S, \prec$) is

$$\neg((\bigvee_{l \in n(\mu)} (\bigwedge_{l' \in d(l, \mu)} l' \wedge l)) \vee (\bigwedge_{l \in \mu, l \in (S \cup \overline{S})} l \wedge (\bigvee_{l' \in \mu, l' \notin (S \cup \overline{S})} \overline{l'}))) \quad (3)$$

The total assignment μ dominates a total assignment μ' wrt $S, \prec$ iff μ' satisfies the corresponding μ -dominates formula, as stated by the following theorem.

Theorem 1. *Let $S, \prec$ be a qualitative preference on literals. A total assignment dominates a total assignment μ' wrt $S, \prec$ if and only if μ' satisfies the μ -dominates formula wrt $S, \prec$.*

For example,

1. if $\mu_1 = \{Fish\}$ and $S, \prec$ is as in (2), then
 - (a) $n(\mu_1)$ is $\{Meat\}$, and $d(Meat) = \{Fish\}$,
 - (b) the μ_1 -dominates formula is $\neg((Fish \wedge Meat) \vee (Fish \wedge \overline{Meat} \wedge \overline{RedWine} \wedge \overline{WhiteWine}))$: Any total assignment which does not satisfy $(Fish \wedge Meat)$ or $(Fish \wedge \overline{RedWine} \wedge \overline{WhiteWine})$ is dominated by $\{Fish\}$. Notice that the total assignment $\{Fish, WhiteWine\}$ is not dominated by $\{Fish\}$, as expected.
2. if $\mu_2 = \{Meat\}$ and $S, \prec$ is as in (2), then
 - (a) $n(\mu_2)$ is $\{Fish\}$, and $d(Fish) = \emptyset$,
 - (b) the μ_2 -dominates formula is $\neg(Fish \vee (\overline{Fish} \wedge Meat \wedge \overline{RedWine} \wedge \overline{WhiteWine}))$: μ_2 does not dominate the total assignments satisfying $Fish$ or $(\overline{Fish} \wedge Meat \wedge \overline{RedWine} \wedge \overline{WhiteWine})$.

Notice that since $\mu_1 \prec \mu_2$, the μ_1 -dominates formula is entailed by the μ_2 -dominates formula: μ_1 dominates a superset of the total assignments dominated by μ_2 .

It is thus possible to generalize the DLL-based procedure presented in [2] for computing an optimal model, in order to return complete sets of optimal models. The resulting procedure is represented in Figure 1. In the figure,

- It is assumed that the input formula φ is a set of clauses; μ is an assignment; ψ is an initially empty set of clauses;
- $(\varphi \cup \psi)_\mu$ is the set of clauses obtained from $\varphi \cup \psi$ by (i) deleting the clauses $C \in \varphi \cup \psi$ with $\mu \cap C \neq \emptyset$, and (ii) substituting each clause $C \in \varphi \cup \psi$ with $C \setminus \overline{\mu}$;
- $Reason(\mu)$ returns a set of clauses equivalent to the negation of the μ -dominates formula.
- $ChooseLiteral_1(\varphi \cup \psi, \mu)$ returns an unassigned literal l such that
 - if there exists a literal in S which is not assigned by μ , then each literal l' with $l' \prec l$ has to be assigned by μ , and
 - l is an arbitrary literal occurring in $\varphi \cup \psi$, otherwise.

$nOPT\text{-}DLL$ has to be invoked with φ and μ set to the input formula and the empty set respectively. $nOPT\text{-}DLL$ prints a complete set of optimal models, as stated by the following theorem.

Theorem 2. *Let $S, \prec$ be a qualitative preference on literals. Let φ be a set of clauses. $nOPT\text{-}DLL(\varphi, \emptyset)$ prints a complete set of optimal models for φ .*

$S, \prec :=$ a qualitative preference on literals;
 $\psi := \emptyset$;

```

function nOPT-DLL( $\varphi \cup \psi, \mu$ )
1  if ( $\perp \in (\varphi \cup \psi)_\mu$ ) return FALSE;
2  if ( $\mu$  is total)
3    Print( $\mu$ );
4     $\psi := \psi \cup \text{Reason}(\mu)$ ;
5  return FALSE;
6  if ( $\{l\} \in (\varphi \cup \psi)_\mu$ ) return nOPT-DLL( $\varphi \cup \psi, \mu \cup \{l\}$ );
7   $l := \text{ChooseLiteral}_1(\varphi \cup \psi, \mu)$ ;
8  return nOPT-DLL( $\varphi \cup \psi, \mu \cup \{l\}$ ) or
      nOPT-DLL( $\varphi \cup \psi, \mu \cup \{\bar{l}\}$ ).

```

Fig. 1. The algorithm of nOPT-DLL

4 Conclusions

In this paper we have presented an algorithm for computing all solutions in SAT problems with preferences. The algorithm computes only optimal models, by following the given partial order on preferences, but is not ensured to work in polynomial space. Future work comprises the design of algorithms which are guaranteed to work in polynomial space.

References

1. Boutilier, C., Brafman, R.I., Domshlak, C., Hoos, H.H., Poole, D.: CP-nets: A tool for representing and reasoning with conditional ceteris paribus preference statements. *J. Artif. Intell. Res (JAIR)* 21, 135–191 (2004)
2. Giunchiglia, E., Maratea, M.: Solving optimization problems with DLL. In: *Proc. of 17th European Conference on Artificial Intelligence (ECAI)*, pp. 377–381 (2006)
3. Boutilier, C., Brafman, R.I., Domshlak, C., Hoos, H.H., Poole, D.: Preference-based constrained optimization with CP-nets. *Computational Intelligence* 20(2), 137–157 (2004)
4. Järvisalo, M., Junttila, T., Niemelä, I.: Unrestricted vs restricted cut in a tableau method for Boolean circuits. *Annals of Mathematics and Artificial Intelligence* 44(4), 373–399 (2005)
5. Giunchiglia, E., Maratea, M.: Planning as satisfiability with preferences. In: *Proc. of 22nd AAAI Conference on Artificial Intelligence*, pp. 987–992. AAAI Press, Menlo Park (2007)
6. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem proving. *Communication of ACM* 5(7), 394–397 (1962)
7. Tseitin, G.: On the complexity of proofs in propositional logics. *Seminars in Mathematics* 8 (1970); Reprinted in [9]
8. Plaisted, D.A., Greenbaum, S.: A Structure-preserving Clause Form Translation. *Journal of Symbolic Computation* 2, 293–304 (1986)
9. Siekmann, J., Wrightson, G. (eds.): *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, vol. 1-2. Springer, Heidelberg (1983)

On the Efficiency of Impact Based Heuristics

Marco Correia and Pedro Barahona

Centro de Inteligência Artificial, Departamento de Informática,
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
`{mvc,pb}@di.fct.unl.pt`

Abstract. Recently proposed impact based heuristics have been shown to outperform other instances of the first-fail policy such as the common dom and dom/deg heuristics. This paper compares the behaviour of a constraint and a variable centered impact based heuristic and relates it to the amount of constraint propagation inherent to the model of the problem. Additionally, it presents results which suggest that a look-ahead impact heuristic we recently proposed might be the best choice for problems with low locality and where constraint propagation plays an important role.

1 Introduction

Despite the success of Constraint Programming in addressing combinatorial NP problems non trivial instances still require appropriate search strategies to find solutions efficiently. The most general such strategy is possibly the first-fail principle: solve first the most difficult sub-problems. This principle is usually implemented in the variable selection heuristics, by starting enumeration on the variables with domains of least cardinality. This heuristics does not take into account the “structure” of the problem, which is more likely to be accounted for by selecting the variable that participates in more constraints (hence, harder to enumerate). The combination of both ideas results in the popular “dom/deg” heuristic, which performs reasonably well in a number of problems.

Still, other heuristics have been recently proposed that implement the first fail principle more efficiently, by measuring during search the impact of the decisions made, either focusing on the constraints violated, as the wdeg heuristics [2] or the impact on the search space as the variable centered heuristics presented in [12]. However we have found that the singleton arc consistency heuristic we proposed [4] clearly outperforms the dom/wdeg heuristics in a class of CSP problems with some identified features.

In this paper, we firstly describe in more detail the above heuristics. Then we show their results in a number of benchmark problems, for which we study their features, namely the propagation that is achieved, and the correlation that exists between these features and the efficiency shown by the heuristics under study. We conclude with some general comments on this preliminary study and some suggestions for further work.

2 Preliminaries

A constraint network consists of a set of variables $\mathcal{X}$, a set of domains $\mathcal{D}$, and a set of constraints $\mathcal{C}$. Every variable $x \in \mathcal{X}$ has an associated domain $D(x)$ denoting its possible values. Every k -ary constraint $c \in \mathcal{C}$ is defined over a set of k variables $(x_1, \dots, x_k)$ by the subset of the Cartesian product $D(x_1) \times \dots \times D(x_k)$ which are consistent values. The constraint satisfaction problem (CSP) consists in finding an assignment of values to variables such that all constraints are satisfied.

A CSP is arc-consistent iff it has non-empty domains and every consistent instantiation of a variable can be extended to a consistent instantiation involving an additional variable. A problem is generalized arc-consistent (GAC) iff for every value in each variable of a constraint there exist compatible values for all the other variables in the constraint.

A CSP P is singleton θ -consistent (SC), iff it has non-empty domains and for any value $a \in \text{dom}(x)$ of every variable $x \in \mathcal{X}$, the resulting subproblem $P|_{x=a}$ can be made θ -consistent. SC time complexity is in $O(n^2 d^2 \theta)$, θ being the time complexity of the algorithm that achieves θ -consistency on the constraint network. Restricted singleton consistency (RSC) [11] considers each variable only once and has runtime complexity $O(nd\theta)$.

3 Impact Based Heuristics

Impact based heuristics use information collected dynamically during search to score each variable according to its first-failness. We now revise three impact based strategies which use distinct sources of information.

The method presented in [2] associates a counter with each constraint expressing the number of times it was violated since the beginning of search. This information is projected to each variable by summing the counters for all constraints where the variable participates (referred to as the *weighted degree* of the variable). Heuristics then select the variable with largest degree (wdeg), or with smallest ratio between the size of the domain and the degree (dom/wdeg).

The impact based heuristic introduced in [12] measures the size of the search space, given by a function $\sigma(P)$, before and after the enumeration of a variable. The impact of a variable x , given by $(\sigma(P) - \sigma(P|_{x=a})) / \sigma(P)$, is averaged over all its previous enumerations and the highest impact so far indicates the next variable to instantiate.

In [4] we extended this idea by explicitly computing the actual impact of every variable before each enumeration, which is available if simultaneously maintaining (restricted) singleton consistency. In this case we give preference for the variable with a smaller sum of search space size for every possible instantiation, formally

$$\text{var}_{\text{LA}}(P) = \arg \min_{x \in \mathcal{X}(P)} \left(\sum_{a \in D(x)} \sigma(P|_{x=a}) \right)$$

which also corresponds to the minimize promise variable heuristic defined in [7]. As said, this heuristic requires lookahead computations which may be inefficient,

however it is expected that the reduction in dead end failures compensates such inefficiency.

4 Experiments and Discussion

In this section we compare the heuristics described previously on a set of three well known benchmarks. The dom/dweg heuristic was used while maintaining arc-consistency (dom/wdeg). We also tested the dom/wdeg heuristic while maintaining restricted singleton arc consistency, and the dom and wdeg heuristics independently but their performance were consistently worse (results are omitted for space reasons). The var_{LA} heuristic was used with restricted singleton arc consistency (1a) further restricted to the subset of variables with domain size $d = 2$ as described in [4], to avoid expensive lookahead computation with variables unlikely to be selected.

Firstly, 200 random CSP instances were generated using model C [9] (generalized to 5-ary CSPs) with 25 variables and domains of size 4. Instances were created by varying the looseness of the constraints in $[0.1 \dots 0.8]$ and setting the density such that the constrainedness of the instances is $\kappa = 0.95$ [8] (phase transition occurs typically at $\kappa \approx 1$). For more details see [4]. Solutions were stored as positive table constraints and GAC-Schema [1] was used for filtering.

Graph coloring assigns n colors to m nodes of a given graph such that no pair of connected nodes have the same color. We used a k -colorable graph generator [5] to obtain 200 instances of 10-colorable graphs with 65 nodes at the phase transition by varying the average node degree d uniformly around 0.6 [3]. Difference binary constraints were posted for every pair of connected nodes.

The Latin squares problem places N colors in a $N \times N$ grid, such that each color occurs exactly once on the same row or column. A partial Latin squares problem has several preassigned cells, and the goal is to complete the puzzle. We generated 200 instances of satisfiable partial Latin squares of size 35, with 396 cells preassigned, using `lsencode-v1.1` [10]. The problem was modelled using alldifferent (GAC) constraints¹.

The results in fig. 1 (obtained using CaSPER² on a Pentium4, 1.7GHz with 512Mb RAM, timeout set to 1800 seconds, and time given in seconds) show that there is no clear winner across all problems.

As possibly expected, for problems where propagation achieves less pruning, impact based heuristics are less effective. This is the case of both the random and the graph coloring problems, as illustrated in fig. 2, that shows, in log scale, the reduction of the size of the search space, subsequent to each enumeration. In random problems the propagation is poor given the lack of structure of the problem.

The networks for the graph coloring problem exhibit some locality, and propagation mostly affect variables in the same cluster of the variable being assigned,

¹ The dual encoding model, as proposed in [6], was also considered but never improved over the direct model on the presented instances.

² Code available from <http://proteina.di.fct.unl.pt/casper>, revision 333.

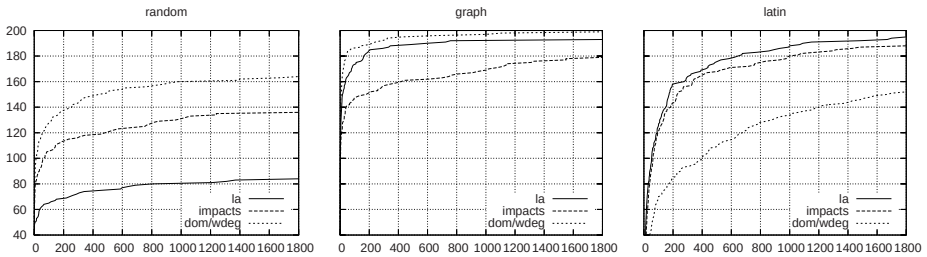


Fig. 1. Number of problems solved (vertical axis) versus time (horizontal axis)

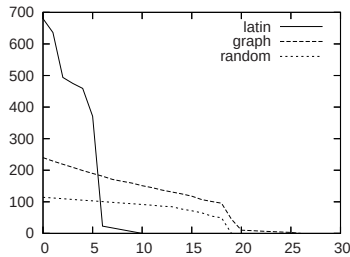


Fig. 2. Search space size during solving of a typical instance in each problem

with limited propagation to variables far away in the network. Apparently, the dom/wdeg heuristics, by reasoning at constraint level, is able to "infer" such locality and take advantage of it for variable selection.

In contrast, in the latin square instances modeled by means of all different global constraints locality is not so marked (two variables often share the same constraint, if in the same row or column, or are separated by two constraints, one row and one column, and seldom by more than that, when both row and column pivot elements are already ground). Moreover, generalized arc consistency propagation virtually affects all variables after variable enumeration. The greater impact achieved in these problems, together with the lack of locality to be exploited by a constraint centered impact heuristic such as dom/wdeg, makes variable centered impact heuristics more adequate in this problem. We tried both heuristics in a set of smaller latin square instances modelled with pairwise distinct constraints while maintaining (singleton) node consistency and observed a different ranking, which confirms our thesis (see table 1).

Table 1. Results for finding the first solution to latin-15 with a selected strategy

strategy	#timeouts	avgfails	stddevfails	avgtime	stddevtime
dom/wdeg	2	63549	422439	13.7	94.6
la	42	8022	30312	208.3	366.4

5 Conclusion

This paper focused on a class of heuristics exploiting some form of impact that decisions may have had in the past (dom/wdeg and impact) or will have in the near future (la). We have shown that the performance of these heuristics is correlated with the model used for the problem, more specifically with the amount and locality of constraint propagation.

The experiments reported in the paper show that work on impact based heuristics is far from over. The fact that there is no clear winner heuristic suggests combinations of these and other impact measures to obtain an efficient compromise between past and future impact information, and/or variable and constraint centered impacts.

Notwithstanding this future work, the new lookahead heuristic that we present in this paper already outperform, to our knowledge, all other heuristics on finite domain encodings of the latin square problems, and we believe that will also outperform other heuristics on problems with the same characteristics (low locality, large reduction of search space per choice).

References

1. Bessière, C., Régin, J.-C.: Arc consistency for general constraint networks: preliminary results. In: *Procs. of IJCAI 1997*, Nagoya, Japan, pp. 398–404 (1997)
2. Boussemart, F., Hemery, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: *ECAI, Spain*, pp. 146–150. IOS Press, Amsterdam (2004)
3. Cheeseman, P., Kanefsky, B., Taylor, W.M.: Where the really hard problems are. In: *Proceeding of IJCAI 1991*, pp. 331–340 (1991)
4. Correia, M., Barahona, P.: On the integration of singleton consistency and lookahead heuristics. In: *Procs. of the annual ERCIM workshop on constraint solving and constraint logic programming*, Rocquencourt, France (June 2007)
5. Culberson, J.: Graph coloring resources, <http://web.cs.ualberta.ca/~joe/Coloring/Generators/generate.html>
6. Dotu, del Val, Cebrian: Redundant modeling for the quasigroup completion problem. In: *ICCP: Int. Conf. on Constraint Programming (CP)*. LNCS (2003)
7. Geelen, P.A.: Dual viewpoint heuristics for binary constraint satisfaction problems. In: *Procs. of ECAI 1992*, New York, NY, USA, pp. 31–35. John Wiley & Sons, Inc., Chichester (1992)
8. Gent, I.P., MacIntyre, E., Prosser, P., Walsh, T.: The constrainedness of search. In: *Proceedings of AAAI 1996*, vol. 1, pp. 246–252 (1996)
9. Gent, I.P., MacIntyre, E., Prosser, P., Smith, B.M., Walsh, T.: Random constraint satisfaction: Flaws and structure. *Constraints* 6(4), 345–372 (2001)
10. Kautz, Ruan, Achlioptas, Gomes, Selman, Stickel: Balance and filtering in structured satisfiable problems. In: *Procs. of IJCAI 2001* (2001)
11. Prosser, P., Stergiou, K., Walsh, T.: Singleton consistencies. In: *Dechter, R. (ed.) CP 2000*. LNCS, vol. 1894, pp. 353–368. Springer, Heidelberg (2000)
12. Refalo, P.: Impact-based search strategies for constraint programming. In: *Wallace, M. (ed.) CP 2004*. LNCS, vol. 3258, pp. 557–571. Springer, Heidelberg (2004)

Probabilistically Estimating Backbones and Variable Bias: Experimental Overview^{*}

Eric I. Hsu, Christian J. Muise, J. Christopher Beck, and Sheila A. McIlraith

Department of Computer Science, University of Toronto
{eihsu, cjmuise, sheila}@cs.toronto.edu, jcb@mie.utoronto.ca

Abstract. Backbone variables have the same assignment in all solutions to a given constraint satisfaction problem; more generally, *bias* represents the proportion of solutions that assign a variable a particular value. Intuitively such constructs would seem important to efficient search, but their study to date has been from a mostly conceptual perspective, in terms of indicating problem hardness or motivating and interpreting heuristics. Here we summarize a two-phase project where we first measure the ability of both existing and novel probabilistic message-passing techniques to directly estimate bias and identify backbones for the Boolean Satisfiability (SAT) Problem. We confirm that methods like Belief Propagation and Survey Propagation—plus Expectation Maximization-based variants—do produce good estimates with distinctive properties. The second phase demonstrates the use of bias estimation within a modern SAT solver, exhibiting a correlation between accurate, stable, estimates and successful backtracking search. The same process also yields a family of search heuristics that can dramatically improve search efficiency for the hard random problems considered.

1 Introduction

Probabilistic message-passing algorithms like Survey Propagation (SP) and Belief Propagation (BP), plus variants based on Expectation Maximization (EM), have proved very successful for random SAT and CSP problems [2,3,4]. This success would appear to result from the ability to implicitly sample from the space of solutions and thus estimate variable bias: the percentages of solutions that have a given variable set true or false. However, such bias estimation ability has never been measured directly, and its actual usefulness to heuristic search has also escaped systematic study. Similarly, backbones, or variables that must be set a certain way in any solution to a given problem, have also drawn a good deal of recent interest [5,6,7,8,9], but they have not been directly targeted for discovery within arbitrary problems. Since backbones must have 100% positive or negative bias, bias determination generalizes the task of backbone identification.

Isolating the performance of probabilistic techniques when applied to bias estimation improves our understanding of both the estimators and of bias itself, ultimately directing the design of a complete problem-solving system. Thus the first stage of our study compares the basic accuracy of six message-passing techniques and two control methods when applied to hard, random, satisfiable SAT problems as stand-alone bias estimators. The second stage assesses how such comparisons translate when we move the

^{*} This article summarizes a more detailed description that is available as a technical report [1].

algorithms into the realm of full-featured search, by embedding them as variable/value ordering heuristics within the MiniSat solver [10]. While it is intuitive that bias should relate to how we set variables during search, it is not obvious that bias should be a key to efficiency in the presence of modern features like restarts and clause learning.

2 Definitions

Definition 1 (SAT instance). A (CNF) SAT instance is a set C of m clauses, constraining a set V of n Boolean variables. Each clause $c \in C$ is a disjunction of literals built from the variables in V . An assignment $X \in \{0, 1\}^n$ to the variables satisfies the instance if it makes at least one literal true in each clause.

Definition 2 (Bias, Survey). For a satisfiable SAT instance $\mathcal{F}$, the **estimated bias distribution** θ_v of a variable v attempts to represent the fraction of solutions to $\mathcal{F}$ wherein v appears positively or negatively. Thus it consists of a **positive bias** θ_v^+ and a **negative bias** θ_v^- , where $\theta_v^+, \theta_v^- \in [0, 1]$ and $\theta_v^+ + \theta_v^- = 1$. A vector of bias distributions, one for each variable in a theory, will be called a **survey**, denoted $\Theta(\mathcal{F})$.

Equivalently, it can be useful to think of a variable’s bias as the probability of finding the variable set positively or negatively when randomly sampling from the space of satisfying assignments. Less formally, the “**strength**” of a bias distribution indicates the margin by which it favors one value over the other.

3 Probabilistic Methods for Estimating Bias

We compare six distinct message-passing techniques for measuring variable bias: Belief Propagation (BP), EM Belief Propagation-Local/Global (EMBP-L and EMBP-G), Survey Propagation (SP), and EM Survey Propagation-Local/Global (EMSP-L and EMSP-G). On receiving a SAT instance $\mathcal{F}$, each of the propagation methods begins by formulating an initial survey at random. Each algorithm proceeds to successively refine its estimates over multiple iterations. An iteration consists of a single pass through all variables, where the bias for each variable is updated with respect to the other variables’ biases, according to the characteristic rule for a method. If no variable’s bias has changed between two successive iterations, the process ends with convergence; otherwise a method terminates by timeout or some other parameter. EM-type methods are “convergent”, or guaranteed to converge naturally, while regular BP and SP are not [4].

BP estimates bias via Pearl’s Belief Propagation, also known as the Sum-Product algorithm [11, 12]. **SP** (Survey Propagation) extends BP to measure not only the probabilities of a variable being positively or negatively *constrained* in a solution, but also the probability that it could have been set either way [2]. Such extra sensitivity changes the dynamics between iterations, but in the final survey any mass for the third “joker state” is evenly divided between the positive and the negative for the purposes of estimating bias. Both methods can be re-formulated within the Expectation Maximization

framework [13], producing the four EM-based methods. **EMBP-L** and **EMBP-G** use the two-state model of BP and differ in exploiting variational approximations [14] based on local (generalized arc-) consistency and global consistency, respectively. Similarly, **EMSP-L** and **EMSP-G** apply local or global consistency to the three-state model represented by SP. Global methods embody a tighter bound on the variational approximation, but in general take longer to compute than local methods. For experimental comparison, we created two non-probabilistic control methods. **LC** (“Literal Count”) greatly simplifies the backbone-inspired heuristic at the core of a highly successful system for refuting *unsatisfiable* SAT instances [8]. **CC** (“Clause Count”) is an even simpler baseline method that just counts the number of clauses containing a given variable as a positive literal, and the number wherein it appears negatively. The relative weights of these two counts determine the variable’s estimated bias distribution.

4 Summary of Experiments

The first phase of experiments compared the eight methods as stand-alone bias estimators for randomly generated problems whose true biases were found via exhaustive model counting. Figure 1 depicts basic root-mean-squared error, with the global EM-based methods performing the best, followed by the controls and the local EM-based methods. Probabilistic methods that do not use EM have the worst basic accuracy, and on most measures the SP variants do better than those based on BP’s simpler model.

However, a second pattern arises across experiments that emphasize strong biases and estimates: “backbone identification rate” and “rank of first wrong bias” (graphs available in longer presentation [1]). The former measures the proportion of actual backbone variables that an estimator biases toward the correct polarity. The latter examines the variables to which a method assigns the strongest biases, and finds the highest-ranking estimate that was actually toward the wrong polarity. For both of these

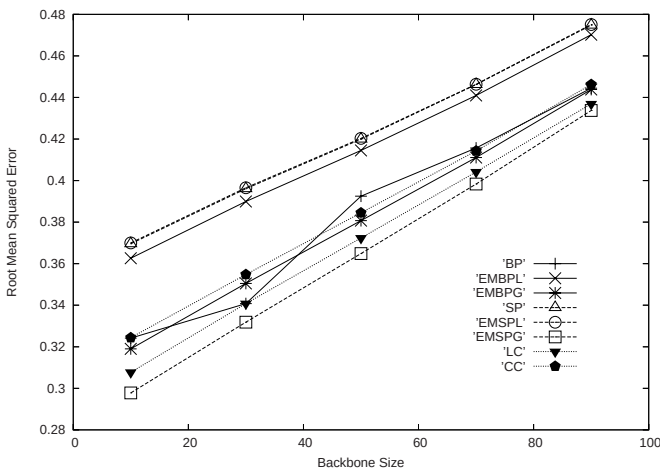


Fig. 1. RMS error over 500 instances of increasing backbone size, $n = 100$

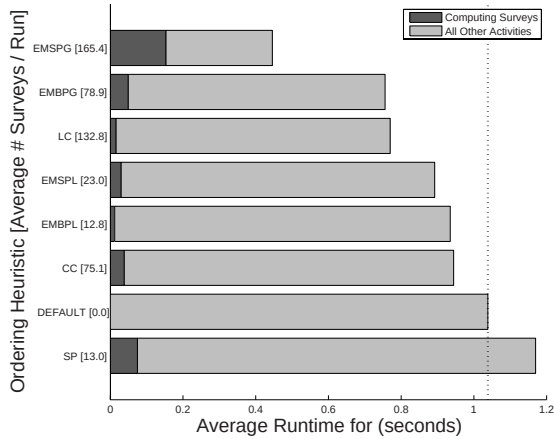


Fig. 2. Total/Survey runtimes averaged over 100 random problems, $n = 250$ and $\alpha = 4.11$

measures, BP actually performs the best, followed by EMSP-G, LC, SP, EMBP-G, the local EM methods, and CC.

These two sets of measures (basic accuracy and accuracy on strong biases) turned out to be the most important predictors of success in the second phase of study. Here, the methods were embedded as variable-ordering heuristics within the MiniSat solver [10]. Part of this process meant determining a good way to use surveys: based on our experimental experiences, the results reflect the intuitive “succeed-first” approach of setting the most strongly-biased variable in a survey in the direction of its stronger bias.

The overall runtime results in Figure 2 exhibit a correlation between good bias estimates and significant improvement in search efficiency. The graph presents average runtimes broken down to show the proportion that was devoted to computing surveys.

The “DEFAULT” method represents regular MiniSat without bias estimation, and BP performed so badly as to not appear on the graph. When scaling to larger experiments, the relative efficiency of the best method (EMSP-G) exhibits exponential growth. For instance, when $n = 450$, it requires an average of 5 minutes per problem, while default MiniSat typically requires half an hour, or times out after three hours.

Of the many measures examined during the first phase of experiments, basic accuracy seems to be the most important predictor of success as a heuristic: global EM methods and controls outperform local EM methods and SP/BP, and within each of these bands the SP version is a more effective heuristic than the BP version. Exceptions to this correlation indicate the importance of the secondary strength-oriented accuracy measures mentioned above. For instance, CC compares relatively well in basic accuracy, but ranks lower as a heuristic—it scored the worst with the accuracy measures that focus on strong biases. Still, the influence of such secondary measures remains dominated by overall accuracy—BP is the best estimator of strong biases, but has poor overall accuracy and constitutes the worst heuristic by far.

5 Conclusions

The main findings of these experiments indicate that probabilistic message-passing techniques can be comparatively successful at estimating variable bias and identifying backbone variables, and that successful bias estimation has a positive effect on heuristic search efficiency within a modern solver. Secondary contributions include a novel family of EM-based bias estimators, and a series of design insights culminating in a fast solver for hard random problems.

However, many important issues remain. For instance, structured and unsatisfiable instances have not yet been considered by this bias estimation framework. This may require a finer-grained analysis of accuracy that considers variance across multiple runs with various random seeds. Further, the best way to use surveys for variable ordering cannot be settled conclusively by the limited span of branching strategies that have been studied to date. For instance, we waste some of the SP framework's power when we split the probability mass for the joker state between positive and negative bias; future branching strategies might favor variables with low joker probabilities.

References

1. Hsu, E., Muise, C., Beck, J.C., McIlraith, S.: Applying probabilistic inference to heuristic search by estimating variable bias. Technical Report CSRG-577, University of Toronto (2008)
2. Braunstein, A., Mezard, M., Zecchina, R.: Survey propagation: An algorithm for satisfiability. *Random Structures and Algorithms* 27, 201–226 (2005)
3. Dechter, R., Kask, K., Mateescu, R.: Iterative join-graph propagation. In: Proc. of 18th Int'l Conference on Uncertainty in A.I (UAI 2002), Edmonton, Canada, pp. 128–136 (2002)
4. Hsu, E., Kitching, M., Bacchus, F., McIlraith, S.: Using EM to find likely assignments for solving CSP's. In: Proc. of 22nd National Conference on Artificial Intelligence (AAAI 2007), Vancouver, Canada (2007)
5. Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Troyansky, L.: Determining computational complexity from characteristic phase transitions. *Nature* 400(7), 133–137 (1999)
6. Kilby, P., Slaney, J., Thiébaux, S., Walsh, T.: Backbones and backdoors in satisfiability. In: Proc. of 20th National Conference on A.I (AAAI 2005), Pittsburgh, PA (2005)
7. Singer, J., Gent, I., Smaill, A.: Backbone fragility and the local search cost peak. *Journal of Artificial Intelligence Research* 12, 235–270 (2000)
8. Dubois, O., Dequen, G.: A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In: Proc. of 17th International Joint Conference on Artificial Intelligence (IJCAI 2001), Seattle, WA (2001)
9. Zhang, W.: Configuration landscape analysis and backbone guided local search. Part I: Satisfiability and maximum satisfiability. *Artificial Intelligence* 158(1), 1–26 (2004)
10. 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)
11. Pearl, J.: Probabilistic Reasoning in Intelligent Systems. Morgan Kaufmann, San Francisco (1988)
12. Kschischang, F.R., Frey, B.J., Loeliger, H.A.: Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory* 47(2) (2001)
13. Dempster, A., Laird, N., Rubin, D.: Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society* 39(1), 1–39 (1977)
14. Jordan, M., Ghahramani, Z., Jaakkola, T., Saul, L.: An introduction to variational methods for graphical models. In: Jordan, M. (ed.) *Learning in Graphical Models*. MIT Press, Cambridge (1998)

A New Empirical Study of Weak Backdoors

Peter Gregory, Maria Fox, and Derek Long

University of Strathclyde

Glasgow, UK

{pg,maria,derek}@cis.strath.ac.uk

Abstract. Work by Kilby, Slaney, Thiebaux and Walsh [1] showed that the backdoors and backbones of unstructured Random 3SAT instances are largely disjoint. In this work we extend this study to the consideration of backdoors in SAT encodings of *structured* problems. We show that the results of Kilby *et al.* also apply to structured problems. Further, we analyse the frequency with which individual variables appear in backdoors for specific problem instances. In all problem classes there are variables with particularly high frequencies of backdoor membership. Backbone variables that do appear in backdoors typically appear in very few.

1 Introduction and Background

There has been considerable research into the hidden structure of constraint satisfaction problems. One example is the *backdoor* structure, which can be informally characterised as sets of choices that make it trivial to solve a problem. Previous empirical analysis by Kilby *et al.* [1] studied backdoors in unstructured Random 3SAT instances. They showed that the backdoors and backbones of these instances tend to be largely disjoint. In this work we extend this study to the consideration of backdoors in SAT encodings of structured problems.

A SAT instance, P , is defined as a set of boolean variables, X , and a set of disjunctive clauses, C over X . A solution to P is a total mapping $X \mapsto \{true, false\}$ which satisfies all $c \in C$. The SAT decision problem class is defined to be $SAT = (D, Y)$ with D the domain of all instances of SAT , and $Y \subseteq D$ the set of all “yes” instances. A subproblem of SAT is any $SAT' = (D', Y')$, such that $D' \subseteq D$ and $Y' = Y \cap D'$. A subproblem of SAT is obtained whenever additional restrictions are placed on the general SAT problem class. A *subsolver*, A , is a polynomial time algorithm that solves a subproblem, SAT' , of SAT rejecting all other instances of SAT .

The definition of backdoors is relative to a specific subsolver. A previous definition was introduced by Williams *et al.* [2] and our definition is compatible with theirs. Given a subsolver, A , and an instance, P , of SAT , a *weak backdoor*, B , for P is a set of variables in X such that there is at least one assignment, b , to B for which A determines P satisfiable, given b . Since a backdoor is defined with respect to a given subsolver, A , we must choose one for the purposes of our studies. In the rest of this paper the term backdoor is used to refer to weak backdoors of SAT problems, where A is *unit propagation* [3].

The *backbone* structure is important in our characterisation of the backdoor. Given a SAT instance P with variable set $\mathcal{X}$, the backbone $\mathcal{I} \subseteq \mathcal{X}$ is defined as $x \in \mathcal{X}$ such that exactly one of either $C \wedge x$ or $C \wedge \neg x$ is satisfiable.

2 Problem Set and Finding Backdoors

The SAT problems we analyse are derived by translation from five different problem classes: Driverlog, Blocksworld (planning problems), Graph 3-Colouring, partially completed Latin Squares and Random 3SAT. Each problem set contains 100 separate instances, all of which are satisfiable. The planning domains were taken from previous International Planning Competitions [4,5] and were translated to SAT using the Blackbox [6] translator. The Random 3SAT and Graph 3-Colouring instances are from the satlib [7] benchmarks. The Latin Square instances were generated, and translated to SAT instances, by Carla Gomes' Is-encode program¹.

In this work we are interested in studying minimal backdoors and we find them using Algorithm 1. This algorithm depends on a sub-procedure we call `bounded_dppll`, which performs DPLL search over a subset of the variables. If `bounded_dppll` returns a satisfying assignment, then the variables it searched across *must* form a backdoor.

If an instance is satisfiable then the entire set of variables must be a backdoor. Algorithm 1 finds a minimal backdoor by iteratively removing variables from the *candidate* set, and testing if the remaining structure is still a backdoor. If the removal of a variable leads to `bounded_dppll` being unable to find a solution, then that variable forms part of the minimal backdoor and is placed in the *member* set. Once the *candidate* set is exhausted, the *member* set can be returned as the found minimal backdoor.

Algorithm 1. MINIMAL BACKDOOR

```

1: candidate  $\leftarrow X$ 
2: member  $\leftarrow \emptyset$ 
3: while candidate  $\neq \emptyset$  do
4:    $c \leftarrow x_i, x_i \in \text{candidate}$ 
5:   candidate  $\leftarrow \text{candidate} \setminus c$ 
6:   if bounded_dppll(candidate  $\cup$  member) = reject then
7:     member  $\leftarrow \text{member} \cup \{c\}$ 
8:   end if
9: end while
10: return member

```

3 A New Empirical Characterisation of Backdoors

The results of Kilby *et al.* show that for Random 3SAT instances, there is little overlap between backdoors and backbones. They do not demonstrate that what holds for unstructured Random 3SAT problems also holds for structured SAT problems. In this section we consider structured problems to determine whether backbone variables are more or less likely to be backdoor variables. To test this claim, four instances have been taken from each of the problem sets defined in Section 2. Table 1 shows the comparative statistics for 20 different instances, four from each problem domain. Exactly 100 unique backdoors were found for each instance, using Algorithm 1.

¹ Is-encode is currently available at <http://www.cs.cornell.edu/gomes/SOFT/Isencode-v1.1.tar.Z>

Table 1. Characteristics of 20 different SAT instances, four from each problem class. #V refers to the number of variables, #C refers to number of clauses, $\mu|\mathcal{B}|$ refers to mean backdoor size, μO refers to mean backbone overlap.

	#V	#C	$ \mathcal{I} $	$\mu \mathcal{B} $	$ \mathcal{B}^* $	μO		#V	#C	$ \mathcal{I} $	$\mu \mathcal{B} $	$ \mathcal{B}^* $	μO
BW 1	250	3141	218	3.63	107	1.81	GC 1	150	545	0	7.00	140	0.00
BW 2	91	323	91	2.88	52	2.88	GC 2	150	545	0	7.75	141	0.00
BW 3	148	1211	124	3.34	63	1.15	GC 3	150	545	0	8.24	144	0.00
BW 4	148	1213	116	3.42	76	1.13	GC 4	150	545	0	8.74	129	0.00
DL 1	148	358	60	8.65	67	0.00	3SAT 1	75	325	28	4.86	41	0.24
DL 2	179	625	130	10.23	73	0.84	3SAT 2	75	325	29	8.39	74	1.61
DL 3	307	1466	247	12.14	81	0.67	3SAT 3	75	325	72	8.45	69	5.45
DL 4	180	609	114	10.86	94	1.22	3SAT 4	75	325	50	9.96	70	2.14
LS 1	283	1708	0	8.30	266	0.00	LS 3	278	1643	6	7.90	259	0.05
LS 2	275	1758	3	7.14	249	0.00	LS 4	283	1757	0	8.29	267	0.00

Backbone Size. It can be seen that for some problem classes, the backbone is large as a proportion of the variables: 86% of the Blocksworld variables, 68% of the Driverlog variables and 60% of the Random 3SAT instances are backbone variables. In the other two domains, Latin Squares and Graph Colouring, the backbones are much smaller: $< 0.01\%$ for the Latin Square problems, and there are no backbone variables in the Graph Colouring instances. The Graph Colouring result is obvious since colours can be permuted to create new solutions.

Backdoor Size. Table 2 shows the average size of the backdoors and their variances for each problem class. The results are averages over all of the 100 instances in each problem set. Each value is the result of 100 backdoors, and hence the overall average is the average over 10,000 different backdoors. The averages are also displayed as proportions of the total number of variables in each instance. For example, the $\mu|\mathcal{B}|$ result for Blocksworld means that, on average, the fraction 0.02 of the variables formed each minimal backdoor. Of the five problem classes, the Random 3SAT instances clearly have an unusually large average size backdoor. They also have a larger variance than the other problem classes. This provides evidence that the random problems are characteristically different from the structured problems.

Backbone / Backdoor Variable Overlap. The overlap values in Table 1 show that the backdoors and backbones overlapped most in the Random 3SAT instances and the Blocksworld instances. In the Driverlog instances, there is little overlap, even though the backbone is large in proportion to the total number of variables. In Table 2, the

Table 2. Qualities of the entire problem sets. $\mu|\mathcal{B}|/|\mathcal{X}|$ refers to the average backdoor size as a proportion of the variables. $\sigma^2|\mathcal{B}|/|\mathcal{X}|$ refers to the variance of the backdoor sizes as a proportion of the total variables, μO refers to the average overlap as a proportion of the backdoor size, $\sigma^2 O$ refers to the variance of the overlaps as a proportion of the backdoor size.

	$\mu \mathcal{B} / \mathcal{X} $	$\sigma^2 \mathcal{B} / \mathcal{X} $	μO	$\sigma^2 O$
BW	0.02	1.97×10^{-5}	0.45	7.01×10^{-2}
DL	0.03	4.67×10^{-5}	0.03	1.56×10^{-3}
LS	0.03	5.74×10^{-6}	0.00	1.52×10^{-6}
GC	0.05	5.52×10^{-5}	0.00	0.00
3SAT	0.10	8.85×10^{-5}	0.31	2.93×10^{-2}

Table 3. Maximum absolute backdoor frequencies of variables in different instances

BW	1	2	3	4	DL	1	2	3	4	LS	1	2	3	4	GC	1	2	3	4	3SAT	1	2	3	4
	11	16	17	17		55	52	12	100		8	8	9	7		28	44	39	40		42	48	100	42

averages over the entire problem sets are shown. Notice that the mean overlap now represents the mean number of overlapping variables as a proportion of the backdoor size. The highest proportion of overlapping variables occurs in the Blocksworld instances, followed by the Random 3SAT, followed by the remainder of the problems, all sharing approximately zero values.

Frequency of Variables in Backdoors. The only other property of the instances in Table 1 not discussed so far is the $|B^*|$ property. This represents the total number of variables that occur in *any* of the 100 minimal backdoors found by Algorithm 1. These figures are typically quite high. None of the instances in the table have full coverage, but some come close. So this result tells us that many of the variables appear in at least one backdoor. What it does not tell us is whether the variables appear in many backdoors. To answer this question, further analysis is required. Figure 1 plots the frequency with which each variable appears in the 100 backdoors. The x-axes represent the variables in each of the instances, the y-axes represent the number of backdoors that each variable occurred in. The y-axes on each graph are scaled to the most frequent variable in that specific problem instance. The maximum frequency in each of the graphs is given in Table 3. The variables are sorted so that they appear in descending frequency of backdoor membership. These graphs show that there are some variables that are members of large numbers of backdoors, while other variables are either very infrequently members of backdoors, or are actually never backdoor members.

Frequency of Backbone Variables in Backdoors. In Figure 1 it was seen that in particular instances of the studied problem set, there are particular, small set of variables that are members of many of the backdoors that were found for those instances. Figure 2 refines this picture in order to show which variables in the frequency plots are backbone variables. If a variable is a backbone variable, then there is a black bar plotted underneath its frequency. In all three of the graphs, it is clear that backbone variables also tend to low backdoor frequency variables. The Blocksworld instance deviates slightly from this rule, although there are still fewer backbone members in the high-frequency area. The reader will recall that the backbone size of this instance is 218 variables from

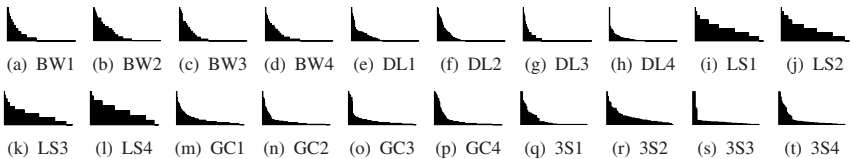


Fig. 1. Backdoor frequencies for 20 SAT instances

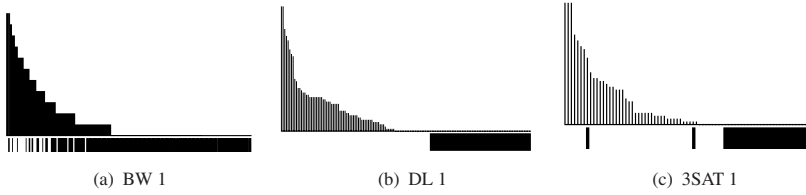


Fig. 2. Backdoor Frequencies for 3 Problem Instances with backbone variables highlighted. Black bar indicates backbone membership.

Table 4. Changes in the various measures with clause learning enabled. $\Delta \mu |\mathcal{B}|$ refers to change in the average backdoor size, $\Delta |\mathcal{B}^*|$ refers to the change in backdoor coverage, $\Delta \mu O$ refers to the change in average overlap between the backdoors and the backbone.

Problem Class	$\Delta \mu \mathcal{B} $	$\Delta \mathcal{B}^* $	$\Delta \mu O$
Blocksworld	-1.76	-52	-1.58
Driverlog	-0.81	-17.5	-0.68
Latin Squares	-0.02	-0.5	-0.01
Graph Colouring	-1.68	-31	0.00
Random 3SAT	-2.75	-37.75	-2.36

a possible 250 variables (Table 1), and as such little over 10% of the variables are non-backbone variables.

Backdoors and Clause Learning. In order to study the effect of clause learning on backdoors, 100 new backdoors of the same instances studied in this section are found. However, when they are now found, clause learning is enabled in the implementation. Table 4 summarises the differences between the measures in both of the experiments. Notice that in the vast majority of cases, the backdoors found when clause learning is enabled are, on average, smaller than they were before.

4 Conclusions

The work presented in this paper first shows that the results of Kilby *et al* hold more generally in structured problems. In addition, our work extends this previous analysis to provide a deeper analysis of backdoor variables. The total variable coverage across many backdoors, reported in the earlier work, does not distinguish between variables that occur in backdoors frequently and those that do not. We have refined the analysis by studying the frequency with which individual variables occur in backdoors.

References

1. Kilby, P., Slaney, J., Thiebaux, S., Walsh, T.: Backbones and backdoors in satisfiability. In: Proceedings of AAAI 2005 (2005)
2. Williams, R., Gomes, C., Selman, B.: Backdoors to typical case complexity. In: IJCAI 2003 (2003)

3. Davis, M., Putnam, H.: A computing procedure for quantification theory. *Journal of the ACM* 7(3), 201–215 (1960)
4. Long, D., Fox, M.: The 3rd IPC: Results and analysis. *JAIR* 20, 1–59 (2003)
5. Bacchus, F.: The AIPS 2000 Planning Competition. *AI Magazine* 22(3), 47–56 (2001)
6. Kautz, H.A., Selman, B.: Unifying SAT-based and graph-based planning. In: *IJCAI*, pp. 318–332 (1999)
7. Hoos, H.H., Stützle, T.: SATLIB: An Online Resource for Research on SAT. In: Gent, I., van Maaren, H., Walsh, T. (eds.) *SAT2000: Highlights of Satisfiability Research in the year 2000*. *Frontiers in Artificial Intelligence and Applications*, pp. 283–292. Kluwer Academic, Dordrecht (2000)

Adding Search to Zinc

Reza Rafeh¹, Kim Marriott¹, Maria Garcia de la Banda¹,
Nicholas Nethercote², and Mark Wallace¹

¹ Clayton School of IT, Monash University, Australia

² NICTA Victoria Research Laboratory, University of Melbourne, Australia

Abstract. We describe a small, non-intrusive extension to the declarative modelling language Zinc that allows users to define model-specific search. This is achieved by providing a number of generic search patterns that take Zinc user-defined functions as parameters. We show the generality of the approach by using it to implement three very different kinds of search: backtracking search, branch-and-bound search, and local search. Our approach is competitive with hand-coded search strategies.

1 Introduction

Recent approaches to solving combinatorial problems divide the task into two steps: developing a *conceptual model* of the problem that gives a declarative specification without consideration as to how to actually solve it, and *solving* the problem by mapping the conceptual model into an executable program called the *design model*.

The declarative modelling language Zinc [7,5] is a first-order functional language designed to support experimentation with different solving techniques. In its current implementation, conceptual models in Zinc can be automatically mapped into design models that use one of the following three solving approaches: standard constraint programming (CP); a Mixed Integer Programming (MIP) solver; and incomplete search using local search methods.

While the default search used by the automatically mapped design models usually performs well for MIP, this is not the case for CP and local search whose efficiency often depends on the modeller providing an effective, model-specific search strategy. However, allowing users to define their search routines requires the integration of a conceptual model and a search strategy, something that is difficult to achieve cleanly since while the former is best expressed declaratively, the latter is inherently procedural.

Here we describe an extension to Zinc to support model-specific search. The extension consists of three high-level search patterns for backtracking search, branch and bound, and local search, respectively, that take complex expressions, functions and predicates as parameters. This combined with user-defined functions give Zinc modellers a degree of flexibility to tailor the search only found previously in procedural search languages. While the actual mechanism of search must still be understood procedurally, the Zinc specification is declarative and requires no additional language features.

2 Using Search in Zinc

We illustrate the use of our search patterns with a simple example, the N-queens problem, which tries to place n queens on an $n \times n$ chess board in such a way that no two queens can take each other. A Zinc model for this problem is:

```
int: n;
type Domain = 1..n;
array[Domain] of var Domain :q;
predicate noattack(Domain: i,j, var Domain: qi,qj) =
    qi != qj /\ qi + i != qj + j /\ qi - i != qj - j;
constraint forall(i,j in Domain where i<j)
    noattack(i,j,q[i],q[j]);
solve satisfy;
```

The model defines the integer variable `n` to be a parameter, `Domain` to be a new type for the range `1..n`, and `q` to be an array of `n` finite domain decision variables (indicated by the keyword `var`) over that range. For our purposes, the most interesting feature of Zinc is that it allows the user to define new predicates and functions. In the example, the modeller has defined the `noattack` predicate, which succeeds if queens `qi` and `qj` of rows `i` and `j` respectively, cannot attack each other (`/\` denotes conjunction). The constraint uses the `forall` expression to make sure the `noattack` predicate holds for each pair of queens. The last line declares the model to be a satisfaction problem. Since the `solve` item has no annotation for search, Zinc uses the default search to solve the model.

Backtracking search

Modellers can use Zinc's depth-first search pattern `backtrack(init,expand)` for solving satisfaction problems with backtracking search using a propagation solver. The first argument, *init*, is the state of the root node in the search tree. This is often the list of variables to label, but can be anything the modeller needs to create choice points, and can include extra information such as a counter to implement iterative deepening. Its second argument, *expand*, is a (possibly user-defined) function that takes the state for the current node and returns its children as a list of pairs of the form (ns, c) , where *ns* is the child's state, and *c* the constraint that should be posted right before this child becomes the current node. Note that *expand* has implicit access to the solver state and, thus, can call standard propagation solver reflection functions such as `domain(V)`, which returns the current domain of variable *V*.

As an example, we can use a standard labeling search for the N-queens model by annotating the `solve` item as follows:

```
solve satisfy::backtrack(q,std_label);
```

where the initial local state is the list of variables `q` and function `std_label` is the *expand* function. It is defined by:

```

function list of tuple(list of $T, var bool): std_label(list of $T:Vs) =
  if Vs = [] then []
  else [ (tail(Vs), head(Vs) == d) | d in domain(head(Vs))]
endif;

```

which takes a list of variables *Vs* (with polymorphic type `list of $T`) and (by using a list comprehension) for each variable *V* in *Vs* returns a list of tuples in which the first element is the remaining variables (and will be the state of the children nodes) and the second element is an equality constraint between the variable and a value from its domain. The `head` and `tail` functions are provided in the Zinc library and return the head and tail of the input list, respectively. Since the output of `domain` is a set and Zinc's sets are ordered, the domain values are considered from smallest to largest. To instantiate each variable, the backtracking search tries the constraints returned by `std_label` in order.

Branch-and-bound

For optimization problems, Zinc provides a variant of the backtracking pattern extended with branch and bound: `backtrack(init, expand, bound, flag)`. This is used as an annotation to the `solve` item which is either in the form `solve maximize expr` or `solve minimize expr` for maximization and minimization problems, respectively.

The first two arguments of the pattern are as before. The two extra arguments are a function, *bound*, for computing the new bound from the previous and current bounds, and a flag to indicate the kind of branch-and-bound search performed. The flags are similar to those provided in ECLiPSe [1], and include *restart* (to restart the search from the root of the search tree), *continue* (to continue the search from the current node in the search tree), and *dichotomic* (to do dichotomic search).

Local search

A common class of techniques for solving combinatorial optimization problems are so-called *local search* methods (such as hill-climbing or simulated annealing) which iteratively improve a single valuation by moving to a neighbour. Zinc provides the pattern `local_search(init_valn, init_state, move, finish)`, which takes as arguments the initial valuation (list of variable/value pairs), the initial state information, a function *move* that takes the current state and returns the new valuation to move to (this needs only to give the values for variables that have been changed in the move) along with the new state, and a function *finish* that takes the state and indicates whether the search should finish.

These functions can use the following Zinc's local search solver reflection functions similar to those provided in Comet [8]: `val(V)` gives the value of variable *V* in the valuation, `var_penalty(V)` the degree of violation associated with variable *V*, `penalty(C)` the violation of constraint *C*, `current_penalty` the total penalty for the current valuation, and `new_penalty(Val)` the total penalty that will result if the changes in valuation *Val* are applied to the current valuation.

The modeller can then specify, for example, a simple hill-climbing search routine by annotating the `solve` item as:

```
solve satisfy::local_search([(q[i],i)|i in Domain],1000,move,finish);
```

where initially the i^{th} queen is placed on row i and the initial state is simply the maximum permitted number of moves. The *move* function is:

```
function valuation: swap($T: v1, $T: v2) = [(v1,val(v2)),(v2,val(v1))];
function tuple(int, valuation): move(int: nmovesleft) =
  let {int: i=maximizes(q,var_penalty),
      int: j=minimizes([swap(q[i],q[k])|k in Domain], new_penalty)
  } in
    (nmovesleft-1,swap(q[i],q[j]));
function has_ended: finish(int: nmovesleft) =
  if current_penalty == 0 then sol(get_valuation)
  elseif nmovesleft <= 0 then end(get_valuation)
  else continue
endif;
```

where function `swap` takes two variables and returns a valuation in which the values of variables have been swapped. The built-in type `valuation` is defined in Zinc as a list of variable/value pairs. The built-in functions `minimizes` and `maximizes` take a list and a function and return the position of the element in the list that minimizes and maximizes the function, respectively. The `move` function chooses the most violated queen `q1` and determines the queen `q2` with which it can be swapped to reduce the overall violation. The number of moves left for the next iteration is decremented. After each move, the function `finish` is invoked which decides upon the state whether the search should finish. The enumerated type `has_ended` is defined in Zinc's library as:

```
enum has_ended = {sol(valuation),end(valuation),continue};
```

to indicate if the search has found a solution, it has not but it must end, or should continue.

It is worth pointing out that Zinc allows the modeller to override the default violation of constraints and variables by using annotations that can take complex expressions and functions. Also, the Zinc modeller does not have to explicitly set up invariants (or functional constraints). These are inferred automatically from the choice of driver variables based on the initial valuation and the model constraints. The compiler generates an error if some non-driver variable cannot be computed from the driver variables.

Evaluation

To evaluate the expressiveness of our approach, we chose a set of 8 well known benchmarks and searched the literature for the best tree and local search strategy for each problem. The three search patterns in Zinc were expressive enough to implement the best search algorithms for all models (models can be found at [6]).

Our implementation maps Zinc models into ECLiPSe programs (ECLiPSe was chosen because it supports all target solving techniques). Our results show that the models with user-defined search are often orders of magnitude faster than the equivalent models using the default search, and that the mapped models are competitive with hand-written models in ECLiPSe that use the same search algorithm (on average, the overhead is less than 10%).

3 Discussion

Our extension to Zinc allows users to run the same conceptual model with different solving methods and, when the default search is too slow, to tailor it with user-defined search. This can be achieved by simply writing functions in Zinc to pass as the required parameters to one of the search templates: backtracking, branch and bound, and local search. The success of this pragmatic solution to an inherently difficult problem is only possible because: (1) the modelling language is powerful enough to allow the user to provide user-defined functions to tailor the search; and (2) a limited number of different generic search schemas covers most of the useful search routines.

Modelling languages for combinatorial problems have traditionally been declarative. Early languages such as AMPL [3] had search built into the solvers and provided only a few simple parameters for controlling it. This approach is too inflexible. The main alternative approach used, for example, in Mosel [2] and OPL [4], is to allow users to specify the search with the model. However, this requires the modelling language to be extended with non-declarative procedural constructs, something that is avoided in our approach.

The closest predecessor to Zinc's search appears to be the ECLiPSe search predicate [1] (which in turn was preceded by that of CHIP). While most search parameters in ECLiPSe are multiple-choice ones, some can be user-defined predicates. The key difference is that ECLiPSe is not fully declarative: modelling and search are both performed by procedural statements.

Acknowledgements. We thank members of the G12 team at National ICT Australia for helpful discussions, in particular Ralph Becket and Peter Stuckey.

References

1. Apt, K.R., Wallace, M.G.: Constraint Logic programming using ECLiPSe. Cambridge University Press, Cambridge (2006)
2. Colombani, Y., Heipcke, S.: Mosel: An overview (2007), <http://www.dashoptimization.com/home/downloads/pdf/mosel.pdf>
3. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A Modeling Language for Mathematical Programming. Duxbury Press (2002)
4. Van Hentenryck, P., Perron, L., Puget, J.F.: Search and strategies in OPL. ACM Transactions on Computational Logic 1(2), 285–320 (2000)

5. 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)
6. Rafeh, R.: The Zinc modelling language home page,
<http://www.csse.monash.edu.au/~rezar/Zinc>
7. Rafeh, R., Garcia de la Banda, M., Marriott, K., Wallace, M.: From Zinc to design model. In: Hanus, M. (ed.) *PADL 2007. LNCS*, vol. 4354, pp. 215–229. Springer, Heidelberg (2006)
8. Van Hentenryck, P., Michel, L.: *Constraint-Based Local Search*. MIT Press, Cambridge (2005)

Experimenting with Small Changes in Conflict-Driven Clause Learning Algorithms

Gilles Audemard¹ and Laurent Simon²

¹ Univ Lille-Nord de France CRIL / CNRS UMR8188,
Lens, F-62307
audemard@cril.fr

² Univ Paris-Sud, LRI / CNRS UMR8623 / INRIA Saclay
Orsay, F-91405
simon@lri.fr

Abstract. Experimentation of new algorithms is the usual companion section of papers dealing with SAT. However, the behavior of those algorithms is so unpredictable that even strong experiments (hundreds of benchmarks, dozen of solvers) can be still misleading. We present here a set of experiments of very small changes of a canonical Conflict Driven Clause Learning (CDCL) solver and show that even very close versions can lead to very different behaviors. In some cases, the best of them could perfectly have been used to convince the reader of the efficiency of a new method for SAT. This observation can be explained by the lack of real experimental studies of CDCL solvers.

1 Introduction

Conflict-Driven Clause Learning algorithms (CDCL) have been one of the major breakthroughs in the practical solving of industrial SAT problems. Since the introduction of ZChaff in 2001 [8], a lot of progresses have been made [3], and solvers can now tackle problems of millions of clauses. All techniques and methods embedded in “modern” solvers are well known: dynamic heuristics [4,8], learning [9], restarts [1,6] and lazy data structures [8]. Efficient solvers can nowadays be written from scratch in less than a thousand lines of code.

However, we believe that the underlying mechanisms are still not understood. They result from extensive tests rather than strong experimental studies, where paradigms would be proposed and tested against observations. We believe that new breakthroughs in the next years may only come if we begin to really understand the reasons of solvers performances. A new technique may be good, but can still be thrown away and not published because of a dramatic side effect of a previously unknown behavior of CDCL solvers. It is thus crucial to begin an in-depth study of modern solvers, without trying to improve their performances at first. In this short paper, we try to consider a typical CDCL solver, MINISAT [3], as a physical system that we try to test against well admitted ideas. Our final aim here is more to cast new questions to the community, given some observations of MINISAT performances, rather than proposing a full and tested paradigm of CDCL solvers. As a side effect of our studies, we illustrate how far one

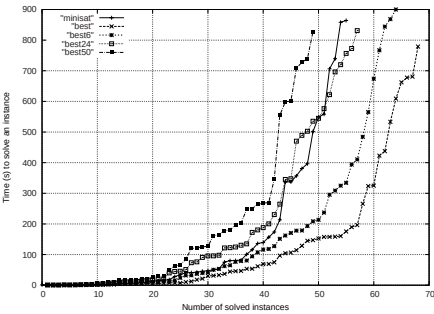
may improve MINISAT performances with only a couple lines hack. This last observation may for instance be a standard to know whether or not new solvers bring really new ideas or may result from a side effect of small changes of a canonical CDCL solver, MINISAT.

2 Shuffling Effects: The Lisa Syndrome, Revisited

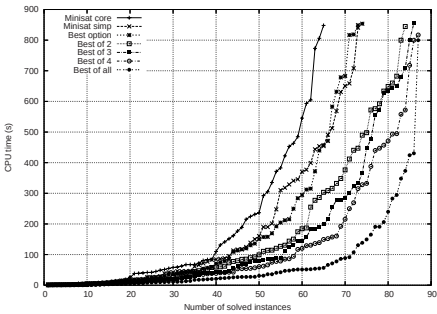
It is well admitted that shuffling instances have a negative effect on industrial benchmarks (see the so-called “Lisa” Syndrom in [7], related to [2]). This observation has motivated, in SAT Contests and SAT Races, to consider only proper benchmarks. Thus, it is admitted that modern solvers explicitly use a heuristic that suppose a non-shuffled instance. Over all explanations, one intuition is that first variables have more chances to be input variables than additional variables introduced in order to avoid combinatorial explosion. During the first phase, MINISAT chooses decision variables in lexicographic order (some solvers choose variables according to their occurrences in the input formula). However, it is not clear how much one may lose by shuffling an instance. If the above explanation holds, and if order of clauses and variables are related to some important structural property between clauses and literals, then one should loose a lot by shuffling an instance before calling MINISAT.

All experiments are done with instances of the SAT Race 2006. Shuffling is done on variables order, clauses order and literals order in each clause (like in [2]).

Figure 1.a shows the traditional performance plot for solvers comparison. It gives the CPU time (in seconds) needed to solve a given number of instances. We can read that MINISAT (without SATELITE preprocessing) is able to solve 55 problems. Curves “best” (respectively “best6”, “best24” and “best50”) plot the result of virtual solvers which would have the best CPU time obtained on all 50 shuffled runs (respectively the 5th, 24th and 50th (median) percentile). Thus, a very simple shuffling (50 times) of instances allows to solve 69 instances in less than 900 seconds (in comparison to the 55 instances solved with only one run on the original problem). What is more striking, is



a. Shuffle Neighborhood



b. One-Parameter Neighborhood

Fig. 1. Studying Shuffling (left) and Parameters (right) (median time on 40 launches)

that there is 24% of chances to obtain better results by shuffling instance (see “best24” curve). This result is clearly higher than one would have predicted if it was only justified by the topology of the original problem (input/output variables encoding).

3 Parameters Effects

When tuning the solver, a number of parameters have to be set (like the randomness of the heuristics, the number of conflicts before restarts, ...). We study how performances can be enhanced by changing only one of these values in MINISAT. We took 10 different magic values of MINISAT parameters and studied all (1-parameter) neighbors as they were different solvers. For each value, we tried both MINISAT with SATELITE (called *simp*) and without it (called *core*), on all original benchmarks. Between 5 and 8 different values were tested for each of 10 parameters¹, which give us 126 solvers (half with SATELITE).

Figure 1.b gives the results for some *virtual* solvers based on parameter neighborhood. Each *best of N* curve corresponds to the subset of N solvers that give the best results, if the N solvers were ran in parallel on N computers. First observation: using two versions of MINISAT (the best couple of solvers were core with RESTARTINC=1 and simp with MINISAT default values) can pay a lot. It seems that keeping a very fast restart policy, but without preprocessing, may pay. This shed a new light on recent works on restart policies. We report the best of 3 solvers, also based on variants of restart policies: MINISAT simp with default values, MINISAT core with RESTARTINC=1 and MINISAT core with RESTARTINC=1.1.

The second observation is based on the proximity of all best-N curves (except for the best of all, that even though, joins all best-N curves at the end), which means that MINISAT really reaches its limits there. One may cast doubts on the real improvement of CDCL solvers if any brand new solver does not really improve this “hard” limit.

Figure 2.a reports another experiment: we took MINISAT and, each time one of the 10 constants was requested, we added 10% random noise to it. We can see that the “noisy” MINISAT now behaves like another solver. When new methods exhibit similar performance plot w.r.t MINISAT, nothing can be really drawn from it. This can only be due to some hidden noise. Last observation we made: When considering the whole neighborhood, using SATELITE as a preprocessor is not so important. We measured that differences between best of all simp versions and best of all core versions are only by one more bench solved for the first version.

¹ VARDECAY (inverse of the variable activity decay) $\in \{0.5, 0.75, 0.85, 0.90, 0.95, 0.99, 0.999\}$; VARINC (init. amount to bump vars) $\in \{1, 2, 5, 10, 50\}$; RESTARTINC (factor by which the restart limit is multiplied after restarts) $\in \{1, 1.1, 1.25, 1.5, 1.75, 2, 4, 8\}$; RESTARTFIRST (init. restart limit) $\in \{10, 50, 100, 200, 500, 1000, 5000, 10000, 50000\}$; RANDOMVARFREQ (frequency with which MINISAT choose a random variable rather than the heuristics based one) $\in \{0, 0.001, 0.002, 0.003, 0.01, 0.05, 0.1, 0.5\}$; LEARNTSIZEINC (factor that increases the limit of learnt clauses) $\in \{0.5, 0.8, 1, 1.1, 1.2, 1.5, 2, 4\}$; LEARNTSIZEFACTOR (limit for learnt clauses as a factor of the total number of clauses) $\in \{1, 1.5, 2, 3, 4, 5, 8\}$; CLAINC (init.amount to bump clauses with) $\in \{1, 2, 5, 10, 50\}$; CLAUSEDECAY (inverse of the clause activity decay factor) $\in \{0.5, 0.75, 0.85, 0.90, 0.95, 0.99\}$; POLARITYMODE (branching) $\in \{false, true\}$.

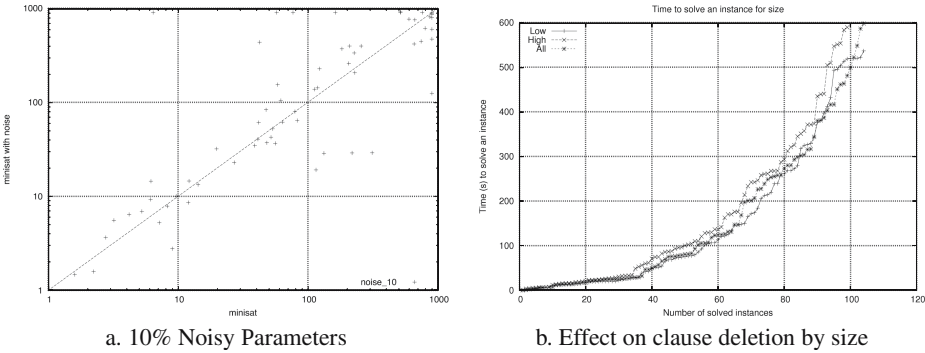


Fig. 2. Noisy parameters (left, median on 40 runs) ; Clauses deletion (right, median on 20 runs)

4 Learning Large or Short Clauses?

In order to avoid memory explosion, modern solvers clean out learnt clauses database. Clauses with less activity (the number of times that these clauses were directly, and recently, considered when analyzing the reasons for the conflict) are deleted. However, it is not necessary for CDCL solver completeness to keep learnt clauses until the end. They just have to provide a reason for current asserting literals. This reason, represented as a clause, can be forgotten when it becomes unnecessary. We analyze here the behavior of MINISAT when one forces it to forget some clauses. Our first goal is to know whether some classes of clauses may be removed without degrading MINISAT performances. The second is more important. We believe that improvements of future CDCL solvers are related to highlighting "important" learnt clauses (yet another time, in a multi-core context, it would be worth sharing a clause between processes only if it is *important*, see [5] for example).

We conducted this experiment as follows. First, we run MINISAT on shuffled instances (20 times), and store, for each benchmark, the median size of learnt clauses. Then, we run 3 versions of MINISAT. The first one forgets 25 % of learnt clauses of any size. In the second (resp. third), it forgets 50% of clauses of size less than (resp. greater than) the computed median size (for a given benchmark). For each parameter, and each benchmark, we consider the median CPU time over 20 shuffled instances. This experiment should show what is highly believed: the size of learnt clauses matters.

Results are summarized in figure 2.b and, contrary to what is usually believed, it seems that short clauses are not *significantly* more important than large ones. Indeed, removing short, large or any clauses produces approximatively the same results at the end. This was already pointed out in previous, theoretical, works, that shown that some proofs need large clauses, but it is surprising to measure in practice that deleting 50% of short clauses is not so different than deleting 50% of large clauses. We also tried to characterize important clauses with other parameters (number of resolutions step during conflict analysis, minimal resolution depth of clauses), but results are identical, and *important* clauses are very hard to characterize, with a global measure.

5 Conclusion

In [2], it was already proposed to use shuffling techniques to characterize the behavior of solvers, and to begin a real experimental study of them. However, this is not a sufficient framework to really test solvers against hypothesis, as they were physical systems. This work is a first step in this direction. We took a canonical, well known, solver, MINISAT, and built experimental studies in order to validate or invalidate some well admitted ideas. So, what can be drawn from our very simple experiments? First, shuffling instances is not as bad as one may have expected. In 25% of the case, it may pay, which is probably too high to confirm that the locality of variables and the order of clauses in real world problems really matters. It is often argued that shuffling instances is useless and has no meaning at all from a practical point of view. However, if one wants to add good learnt clauses somewhere in a formula, by any preprocessing technique, then it is essential to understand where to add it, and if the order really matters and how. At last, we showed that it is not possible to consider short clauses as globally more important than large clauses, which is highly counter-intuitive and was believed to be false. We also show that, by moving parameters, one may obtain really different solvers.

In the next years, CDCL framework will probably be extended to multi-core architectures, which will increase their complexity and their “unpredictability”. If one wants to understand their behavior, a lot of effort has to be made now. Would it be satisfactory to use the multi-core ability of next processors generation only by using different shuffled instances of the same benchmarks? We shown in this paper that a lot of progress has to be done in order to really, deeply, understand why CDCL are so efficient, and what mechanisms are essential. In the quest for efficiency, it is urgent to begin to study them, from a real, deep, experimental perspective.

References

1. Biere, A.: Adaptive restart strategies for conflict driven SAT solvers. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 28–33. Springer, Heidelberg (2008)
2. Brglez, F., Li, X.Y., Stallmann, M.F.: On SAT instance classes and a method for reliable performance experiments with SAT solvers. *Annals of Mathematics and Artificial Intelligence* 43, 1–34 (2005)
3. 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)
4. Goldberg, E., Novikov, Y.: BerkMin: A fast and robust SAT-solver. *Discrete Applied Mathematics* 155(12), 1549–1561 (2007)
5. Hamadi, Y., Jabbour, S., Sais, L.: ManySAT: Solver description. In: System description for the SAT-RACE (2008)
6. Huang, J.: The effect of restarts on the efficiency of clause learning. In: proceedings of IJCAI 2007, pp. 2318–2323 (2007)
7. Le Berre, D., Simon, L.: Essentials of the SAT 2003 competition. In: SAT 2003 (2003)
8. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of DAC 2001, pp. 530–535 (2001)
9. Silva, J.P.M., Sakallah, K.A.: Grasp - a new search algorithm for satisfiability. In: ICCAD, pp. 220–227 (1996)

Search Space Reduction for Constraint Optimization Problems

Kenil C.K. Cheng and Roland H.C. Yap

School of Computing, National University of Singapore
{chengchi, ryap}@comp.nus.edu.sg

Abstract. In a constraint optimization problem (COP), many feasible assignments have the same objective value. This usually means huge search space and poor propagation among the objective variables (which appear in the objective function) and the problem variables (which do not). In this paper, we investigate a search strategy that focuses on the objective function, namely, the objective variables are assigned before the problem variables.¹ Despite the larger search space, we may indeed solve a COP faster, provided that the constraint propagation is strong — the search can reach the optimal objective value faster in the objective space, and by strong propagation it knows if the constraints are unsatisfiable with little search in the problem space. To obtain strong propagation, we study the use of dual encoding [1] for COPs. Our COP formulation and search strategy are general and can handle any dual COPs.

1 Background

A *constraint optimization problem* (COP) $\mathcal{Q} = (X, \mathcal{C})$ consists of a finite set X of variables and a finite set $\mathcal{C}$ of constraints. Every *variable* $x_i \in X$ can only take values from its *domain* $\text{dom}(x_i)$ which is a finite set of integers. An *assignment* (x_i, a) denotes $x_i = a$. A r -ary *constraint* $C(x_1, \dots, x_r) \in \mathcal{C}$ is a subset of the Cartesian product $\prod_{i=1}^r \text{dom}(x_i)$ that restricts the values the variables in C can take simultaneously. The *arity* of C is r and the *scope* is $\text{var}(C)$. A set of assignments $\theta = \{(x_1, a_1), \dots, (x_r, a_r)\}$ *satisfies* C , and is a *solution* of C , iff $\theta \in C$. Solving a COP requires finding a value for each variable from its domain so that all constraints are satisfied and the *objective* $x_0 \in X$ is maximized.

For simplicity of presentation, we consider COPs of the following form: Let $\mathcal{Q} = (X \cup Y \cup \{z\}, \mathcal{C})$ be a COP. Each variable is either a *problem variable* ($\mathcal{P}$ -var) $x_i \in X$, an *objective variable* ($\mathcal{O}$ -var) $y_j \in Y$, or the *objective* z . The objective is defined by the *objective function* $z = \sum_{y \in Y} y$ (a constraint in $\mathcal{C}$). For every $y_j \in Y$, there is in $\mathcal{C}$ an *objective constraint* ($\mathcal{O}$ -constraint) of the form $y_j = F_j(x_1, \dots, x_r)$, where $\{x_1, \dots, x_r\} \subseteq X$ and F_j is a function that maps every $\{(x_1, a_1), \dots, (x_r, a_r)\}$ to some $b \in \text{dom}(y_j)$. Any remaining constraint $C \in \mathcal{C}$ is a *problem constraint* ($\mathcal{P}$ -constraint), where $\text{var}(C) \subseteq X$. Finally, the Cartesian

¹ The usual branch-and-bound search labels only the problem variables; the objective variables are assigned indirectly by constraint propagation.

product of the domains of the $\mathcal{P}$ -vars is the *problem space* ($\mathcal{P}$ -space), and that of the $\mathcal{O}$ -vars is the *objective space* ($\mathcal{O}$ -space). As a remark, our restriction on the $\mathcal{O}$ -constraints and the objective function is to simplify presentation; our ideas are applicable to COPs whose $\mathcal{O}$ -constraints have more than one $\mathcal{O}$ -var, or when the objective function is arbitrary (e.g., non-linear).

In [2], we proposed the idea of searching in the $\mathcal{O}$ -space in conjunction with Russian Doll Search and used the Still-Life problem as a benchmark. This paper focuses on how to do dual encoding for general COPs. In contrast to [2], we show that $\mathcal{O}$ -space search also benefits pure optimization problems.

2 Dual Encoding of a COP

Since enforcing arc consistency (AC) on the dual model is strictly stronger than enforcing generalized arc consistency (GAC) on the original model [3], a dual model will be more suitable when the search explores the $\mathcal{O}$ -space and the $\mathcal{P}$ -space simultaneously. Here we explain how to transform a COP into an equivalent COP using dual encoding [1]. Our transformation is only on the $\mathcal{P}$ -vars and the $\mathcal{P}$ -constraints; the $\mathcal{O}$ -vars and the objective function are unchanged. This is because each $\mathcal{O}$ -constraint and the objective function need only share exactly one $\mathcal{O}$ -var, and so dual encoding on them is not useful.

To describe dual encoding we need some extra notation. Let θ be a set of assignments, X a set of $\mathcal{P}$ -vars and y an $\mathcal{O}$ -var. We denote $\theta|_y = b$ if $(y, b) \in \theta$, and $\theta|_X = \{(x, a) \in \theta : x \in X\}$. Also, two sets of assignments θ and θ' are *compatible* iff for every $(x_i, a) \in \theta$ and $(x_j, a') \in \theta'$, $i = j$ implies $a = a'$.

Finally, the *dual encoding* of a COP $\mathcal{Q} = (X \cup Y \cup \{z\}, \mathcal{C})$ is another COP $\mathcal{Q}' = (X' \cup Y \cup \{z\}, \mathcal{C}')$ constructed as follows:

- For every $C \in \mathcal{C}$, if $\text{var}(C)$ intersects X , there is a *dual $\mathcal{P}$ -var* $x' \in X'$ whose domain is $\{\theta|_X : \theta \in C\}$, a set of projected solutions of C , which can be simply mapped to distinct integers.
- For every $C(x_1, \dots, x_r, y) \in \mathcal{C}$, where $x_1, \dots, x_r \in X$ and $y \in Y$, there is a *dual $\mathcal{O}$ -constraint* $C'(x', y) \in \mathcal{C}'$ such that $\{(x', \theta|_X), (y, \theta|_y)\}$ is a solution of C' iff θ is a solution of C .
- For every $C_i, C_j \in \mathcal{C}$, if $\text{var}(C_i) \cap \text{var}(C_j) \cap X$ is non-empty, there is a *dual $\mathcal{P}$ -constraint* $C'_{i,j}(x'_i, x'_j) \in \mathcal{C}'$ such that $\{(x'_i, \theta_i|_X), (x'_j, \theta_j|_X)\}$ is a solution of $C'_{i,j}$ iff $\theta_i \in C_i$, $\theta_j \in C_j$, and $\theta_i|_X$ and $\theta_j|_X$ are compatible.

$\mathcal{Q}'$ is the *dual model* (of $\mathcal{Q}$) and $\mathcal{Q}$ is the *primal model* (of $\mathcal{Q}'$). Both $\mathcal{Q}$ and $\mathcal{Q}'$ have the same $\mathcal{O}$ -vars Y and the same objective function. By construction, there is a one-to-one mapping between the solutions of the two COPs.

3 Experimental Results

To evaluate the search behaviors under various sizes of the $\mathcal{P}$ -space and the $\mathcal{O}$ -space, we experiment with random pure COPs (no $\mathcal{P}$ -constraint). In the instance $\langle n, d, m, e, r, p \rangle$, there are n $\mathcal{P}$ -vars with domain $\{0, \dots, d-1\}$, m $\mathcal{O}$ -vars with

domain $\{0, \dots, e-1\}$, and m $\mathcal{O}$ -constraints with arity $r+1$ (r $\mathcal{P}$ -vars and one $\mathcal{O}$ -var). For every solution $\{(x_1, a_1), \dots, (x_r, a_r), (y, b)\}$ of an $\mathcal{O}$ -constraint, with probability p we randomly choose $b \in \{1, \dots, e-1\}$, or else $(1-p)$ we set $b = 0$.² Thus for small p most assignments to the $\mathcal{P}$ -vars map to the same objective value (zero), making the problem difficult. Each benchmark has 10 instances.

We used Gecode 2.1.1 on a 2 GHz Core 2 Duo MacBook with 1 GB RAM. The built-in table constraint extensional [4] is used to enforce GAC on the primal $\mathcal{O}$ -constraints. For the dual constraints, we implemented the arc consistency (AC) algorithm by Samaras and Stergiou [5], which exploits the piecewise functional nature of the dual constraints. During the depth-first branch-and-bound search, the variables are assigned in a static order, and the largest value in the domain is always tried first. To be more specific, when the primal model is used, the $\mathcal{P}$ -var that appears in most constraints is selected first, i.e., only the $\mathcal{P}$ -space is searched. We call this strategy “primal/x.” When the dual model is used, the $\mathcal{O}$ -space is explored before the $\mathcal{P}$ -space — the $\mathcal{O}$ -var whose corresponding dual $\mathcal{P}$ -var appears in most constraints is assigned first; after all $\mathcal{O}$ -vars are assigned, the dual $\mathcal{P}$ -vars are assigned in the same order. We call this “dual/yx.”

Fig. 1 plots the number of backtracks and the solving time by dual/yx against primal/x, on $\langle 25, 5, \{20, 21\}, \{2, 50\}, 4, p \rangle$, where p ranges from 0.01 to 0.04. Note that the size of the $\mathcal{P}$ -space is always $5^{25} \approx 3e17$ while the size of the $\mathcal{O}$ -space is between $2^{20} \approx 1e6$ and $50^{21} \approx 4.77e35$. From the graph we can see that using dual/yx often results in 10 to 200 times less backtracks (and never more) than primal/x, although the search space for dual/yx is bigger.³ This is because dual encoding is suitable for sparse problems [5] and exploring the $\mathcal{O}$ -space is more preferred for small p , when the $\mathcal{O}$ -vars take the value zero for most assignments of the $\mathcal{P}$ -vars. Fig. 1b shows that the improvement on solving time by dual/yx is significant for instances which are hard for primal/x (region to the right of $y = x$; when search on the $\mathcal{P}$ -vars takes about > 10 seconds). Actually, when dual/yx is faster, it has 56.3 (median) times less backtracks than primal/x; but when it is slower, the ratio is 12.5, and hence the cost of the dual constraints outweighs the benefit. Overall, our results show that the search moves to the optimal value quickly, and during the traversal in the $\mathcal{O}$ -space, the constraint propagation is strong enough to prune many unsatisfiable regions in the $\mathcal{P}$ -space.

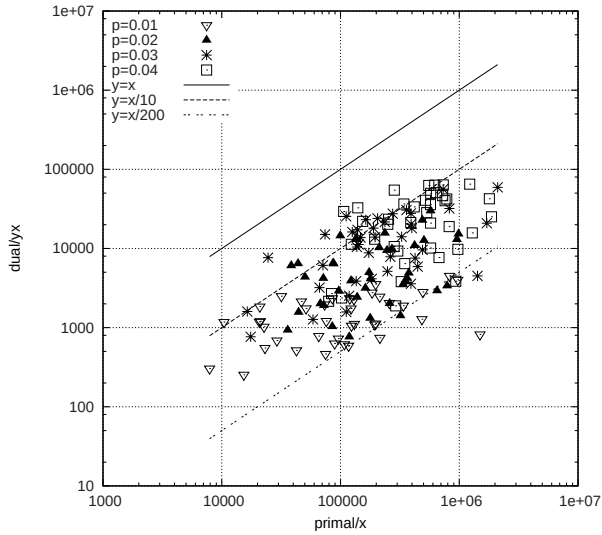
Finally, no instance were solved within 5 minutes when the $\mathcal{P}$ -space was explored with the dual model. This is because the domains of the dual $\mathcal{P}$ -vars are large and the values are often mapped to the same value (zero, in the worst case) of the $\mathcal{O}$ -var.

4 Related Work and Concluding Remarks

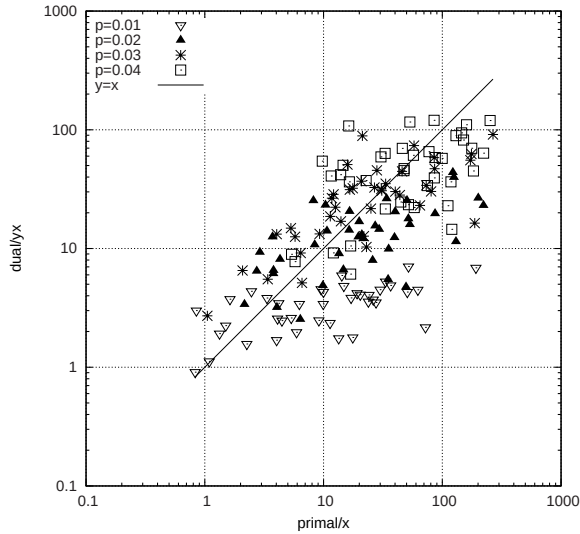
Decomposing the search space is not a new idea. Sachenbacher and Williams made a similar observation on the relationship between the $\mathcal{O}$ -space and the

² Tuples with objective value $b = 0$ are akin to no-goods as the objective is maximized.

³ An $\mathcal{O}$ -constraint is functional from $\mathcal{P}$ -vars to $\mathcal{O}$ -var, so that the value of $\mathcal{O}$ -var is determined when all $\mathcal{P}$ -vars are assigned, but usually not vice versa.



(a) Number of backtracks



(b) Solve time in seconds

Fig. 1. Empirical results on dual/xy against primal/x

$\mathcal{P}$ -space. Their *conflict-directed A* search* [6] solves a COP in two interleaving steps. Upon all the $\mathcal{O}$ -vars are assigned using a traditional A* search, the current sub-problem is solved with a satisfiability (SAT) solver. If it is unsatisfiable, the A* search resumes. There is no propagation between the $\mathcal{O}$ -vars and the $\mathcal{P}$ -vars.

Decomposition is also used in hybrid algorithms (e.g., [7]) that combine integer programming and constraint programming; however, communication between the solvers is limited (e.g., one-way).

The above mentioned algorithms are in fact instances of the *logic-based Benders decomposition* [8], which decomposes a combinatorial problem into a master problem and one or more sub-problems. Once the master problem is solved, we check whether all the sub-problems are satisfiable. If not, the unsatisfied sub-problems create and insert *Benders cuts* to the master problem. A Benders cut is an implied constraint on the variables in the master problem, which prevents the master problem from reaching the same unsatisfiable state in the future. The master problem is then solved again for a different solution. This continues until a solution of the original problem is found or unsatisfiability is proved.

In contrast to a rigid separation of the $\mathcal{P}$ -space and the $\mathcal{O}$ -space, we simply give more freedom to the variable ordering heuristic, which can now choose a $\mathcal{P}$ -var or an $\mathcal{O}$ -var. This is achieved via the strong propagation among the objective function, the $\mathcal{O}$ -constraints and the $\mathcal{P}$ -constraints. We believe dynamic variable ordering heuristics will improve the performance of our search strategy.

The usefulness of dual encoding for COPs was demonstrated with the Still-Life problem [2,9]. Our preliminary experiments show that dual encoding can also be useful for general COPs. Interestingly, due to the functional nature of a dual $\mathcal{O}$ -constraint, assigning an $\mathcal{O}$ -var divides the domain of the corresponding $\mathcal{P}$ -var, and can be seen as a value ordering heuristic. It would be interesting to see whether the optimization approach can apply to the soft CSP framework.

References

1. Dechter, R., Pearl, J.: Tree clustering for constraint networks. *Artificial Intelligence* 38, 353–366 (1989)
2. Cheng, K.C.K., Yap, R.H.C.: Search space reduction and Russian doll search. In: *National Conference on Artificial Intelligence*, pp. 179–184 (2007)
3. Stergiou, K., Walsh, T.: Encodings of non-binary constraint satisfaction problems. In: *National Conference on Artificial Intelligence*, pp. 163–168 (1999)
4. Bessière, C., Régin, J.C., Yap, R.H.C., Zhang, Y.: An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence* 165(2), 165–185 (2005)
5. Samaras, N., Stergiou, K.: Binary encodings of non-binary constraint satisfaction problems: Algorithms and experimental results. *Journal of Artificial Intelligence Research* 24, 641–684 (2005)
6. Sachenbacher, M., Williams, B.C.: Conflict-directed A* search for soft constraints. In: *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pp. 182–196 (2006)
7. Beck, J.C., Refalo, P.: A hybrid approach to scheduling with earliness and tardiness costs. *Annals of Operations Research* 118, 49–71 (2003)
8. Hooker, J.N., Ottoson, G.: Logic-based benders decomposition. *Mathematical Programming* 96(1), 33–60 (2003)
9. Smith, B.M.: A dual graph translation of a problem in ‘life’. In: *International Conference on Principles and Practice of Constraint Programming*, pp. 402–414 (2002)

Engineering Stochastic Local Search for the Low Autocorrelation Binary Sequence Problem

Steven Halim, Roland H.C. Yap, and Felix Halim

School of Computing, National University of Singapore
{`stevenha, ryap, halim`}@comp.nus.edu.sg

Abstract. This paper engineers a new state-of-the-art Stochastic Local Search (SLS) for the Low Autocorrelation Binary Sequence (LABS) problem. The new SLS solver is obtained with white-box visualization to get insights on how an SLS can be effective for LABS; implementation improvements; and black-box parameter tuning.

1 Introduction

Low Autocorrelation Binary Sequence (LABS) problem is a hard problem with simple formulation: find a binary sequence $s = \{s_0, s_1, \dots, s_{n-1}\}$, $s_i \in \{-1, 1\}$ of length n that minimizes the objective function $E(s)$ – the *quadratic* sum of the autocorrelation function C_k , or equivalently, maximizes the merit factor $F(s)$:

$$C_k(s) = \sum_{i=0}^{n-k-1} s_i s_{i+k} \quad E(s) = \sum_{k=1}^{n-1} (C_k(s))^2 \quad F(s) = \frac{n^2}{2E(s)}$$

The LABS problem dates from 1960s and was first posed in the Physics community. It has applications in many communication and electrical engineering problems. More recently, LABS has been investigated by the optimization community using both exact and incomplete solvers [1,2,3,4,5].

In 2006, Dotú and van Hentenryck [4] proposed a *simple* SLS: Tabu Search (TS) *with frequent restarts*. This could find optimal LABS solutions for $n \leq 48$ much quicker than the exact Branch & Bound [2]. It was roughly on par with another good SLS solver for LABS problem: Kernighan-Lin [3].

In 2007, Gallardo *et al.* [5] proposed an SLS: MA_{TS} , combining a Memetic Algorithm with a similar TS. MA_{TS} was shown to be “one order of magnitude” faster than the *pure* TS [4] and was the fastest LABS solver in 2007.

In this paper, we show how an integrated white+black box approach [6] using an SLS engineering tool Viz [7,8] can be used to successfully engineer a new state-of-the-art LABS SLS starting from [4]. For more details, please visit <http://sls.visualization.googlepages.com>.

2 In-Depth Analysis of LABS Fitness Landscape

Previous researchers, e.g. [1,2,5] have shown several features of LABS fitness landscape. As LABS is unconstrained, each LABS instance n has 2^n valid solutions with several Global Optima (GO) (≥ 4) (an approximation of $|GO|$ for

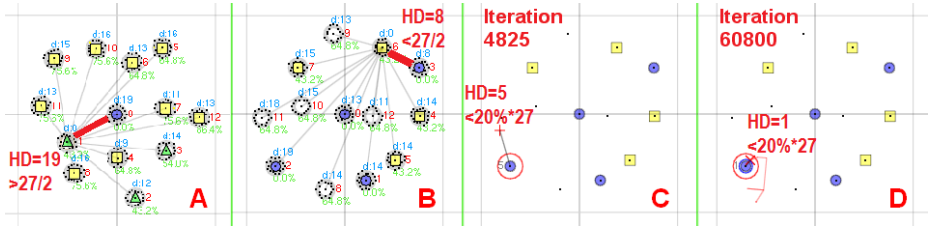


Fig. 1. FLST visualization for LABS $n = 27$ with 4 GO (dark blue circles). LO are deep and isolated, shown by black dots (poor solutions) around each LO in part A&B.

$3 \leq n \leq 64$ is in [3]). These GO are spread like ‘golf holes’ (deep and isolated) in irregular LABS fitness landscape. The fitness landscape of LABS causes difficulties for standard SLS algorithms to work well especially with large n .

In order to get more insights about LABS fitness landscape, we use the Fitness Landscape Search Trajectory (FLST) visualization in Viz [7,8]. To obtain this FLST visualization, we run *our* initial implementation (called **TSv1**) of the TS algorithm from [4] to sample diverse and high quality Local Optima (LO) from the fitness landscape of medium-sized LABS instances ($n \leq 40$). Our sampling strategy exploits the symmetries in LABS: when **TSv1** reaches a solution with Objective Value (OV) equals with the known optimal value (a GO) for that particular medium-sized LABS instance, we can immediately generate all the symmetries of this GO solution. This sampling strategy is used to get a clearer picture of the LABS fitness landscape (compare Fig. 1.A with 1.B).

In Fig. 1.A, we see that without symmetry in GO/LO sampling, we are not immediately aware of the existence of other GO and the *Hamming Distance* (HD) from the current LO to the nearest GO found seems to be large, $HD > n/2$.

By exploiting symmetry, all 4 GO are also ‘found’ when **TSv1** hits a GO. In Fig. 1.B, we can now see that the positions of GO (dark blue circles) are spread out. This suggests that wherever the current solution is, it should be nearer to one GO (the *nearest GO*) than to other GOs. Further observations reveal that LO (light colored non-blue circles) are usually not too close to the nearest GO. By using exact enumeration for LABS $3 \leq n \leq 24$, we have checked that around 85% of the 2nd best solution (which is an LO) have HD around $[n/4 \dots 2n/5]$ bits away from the nearest GO. We exploit this insight.

3 Improving the Tabu Search Algorithm of [4]

In [4], a rather simple yet successful TS algorithm for LABS is presented. We are grateful to the TS code (we call this **TSv0**) from the authors. We benchmarked **TSv0** on our test machine, a 2 GHz Centrino Duo laptop (see the scattered black circles around magenta line in Fig. 2). Benchmarking shows that our test machine has similar performance to the 3 GHz P4 PC used in [4].

Before obtaining the **TSv0** code, we implemented the TS algorithm using *our own* understanding of the pseudo-code in the paper. We call our original

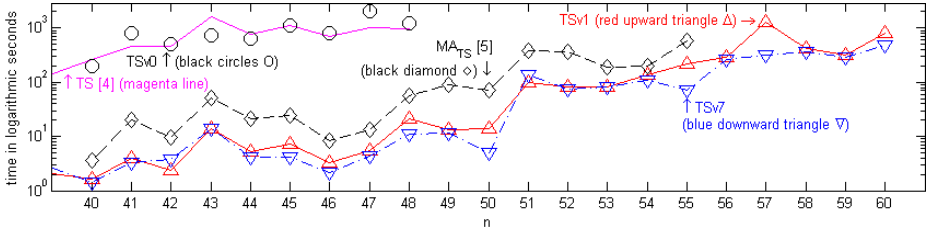


Fig. 2. Comparison of average runtimes (20 runs) between $\{\mathbf{TS} [4], \mathbf{TSv0} \text{ O}\}$, $\{\mathbf{MA}_{TS} [5] \diamond\}$, and $\{\mathbf{TSv1} \Delta, \mathbf{TSv7} \nabla\}$ for LABS with known optimal OV's ($40 \leq n \leq 60$)

implementation, **TSv1**. **TSv1** is already much faster by about one order of magnitude than **TSv0** (see the **red** line with upward triangle in Fig. 2).

Visualization in Viz shows clearly the speed difference as the search trajectory in **TSv1** animates much faster than **TSv0**. Analysis of the source codes reveals the following two major differences. First, while both codes use a form of “incremental computation” to speed up the naïve $O(n^2)$ $E(s)$ computation, the actual sub-algorithms for this part turns out to be different. Since this part is not described in [4], we implemented **TSv1** with the incremental $O(n)$ *ValueFlip* technique used by $\mathbf{MA}_{TS}$ [5]. It turns out that although there is some incremental calculation in **TSv0**, the computation of $E(s)$ is still $O(n^2)$.

Second, though both codes use an $O(1)$ tabu table mechanism, they have different `TABU_TENURE` settings. We know that `TABU_TENURE` cannot be $\approx n$ as it will quickly forbid (almost) all 1-bit flip moves. Black-box tuning on several constant values $[0.1, 0.2, 0.3]n$ on training instances $n = \{27, 30, 41, 42\}$ helps us to set small `TABU_TENURE` = $0.2n$ for **TSv1**. But, **TSv0** use `TABU_TENURE` = n . Thus **TSv0** does more frequent random restart (every $n+1$ iterations) than the pre-determined `MAX_STABLE` parameter as no more valid moves are available.

We can see that **TSv1** runtimes are already comparable to the recent state-of-the-art $\mathbf{MA}_{TS}$ (**TSv1** on 3 GHz P4 PC is about 1.7 to 5.6 times faster than $\mathbf{MA}_{TS}$ [5] for LABS $40 \leq n \leq 55$ and the 3 GHz P4 PC is (probably) at most 1.25 times faster than the 2.4 GHz P4 PC in [5]). We remark that this shows that the random restart strategy in **TSv0/TSv1** is good and better than the benchmarking in [5] would indicate.

4 A State-of-the-Art Tabu Search for LABS

We wanted to get better results. We analyzed **TSv1** search trajectory using the same FLST visualization. During visualization, a circle is drawn on the nearest sampled GO/LO if the current solution is “near” (near is $\text{HD} \leq 20\% * n$).

Using this feature, we observe the following behavior, shown in Fig. 1.C and Fig. 1.D: **TSv1** happens to be near a GO in the *earlier phase* of the search (1.C), but **TSv1** does not immediately navigate there. **TSv1** then wanders to another region near to another GO, perhaps due to random restart strategy. *Thousands*

of iterations later, **TSv1** gets near to the first GO again (1.D) and this time **TSv1** manages to find the GO.

Such observations and insights about the LABS fitness landscape in Sec. 2 lead us to engineer a better SLS strategy. The resulting TS variant is called **TSv7**. (We experimented with other variants also using the white+black box approach described here but **TSv7** had the best performance). Upon experiencing stagnation, **TSv7** restarts to a region around HD $n/4$ bits away from the current LO. Basically, **TSv7** searches for the *nearest* GO. Only if the current LO region is saturated, **TSv7** does a diversification.

The implementation improvements using faster incremental calculation, improved TABU_TENURE settings and the new strategy engineered from white-box analysis using visualization gives a new SLS, **TSv7**. However, we are not done. To obtain a state-of-the-art result, we configured the parameter values for **TSv7** using black-box tuning. We ran a full factorial design of logical parameter values on a set of training instance and picked the best one. Due to space constraints, we are unable to show the algorithm and its parameters. More details and the source code for **TSv7** can be found on the webpage.

Fig. 2 shows the performance of TS (timings from [4]), MA_{TS} (timings from [5]), and **TSv0/TSv1/TSv7** (2 GHz Centrino Duo laptop). We see that **TSv7** strategy is better than the original random restart strategy used in **TSv0/TSv1**. The performance gap is easily noticeable on larger $n = \{50, 55, 57, 60\}$.

To analyze the results, we used the Wilcoxon signed-ranks test. It detected a significant difference between the average runtimes of **TSv1** and **TSv7** on LABS $40 \leq n \leq 60$ (21 pairs, $T = 27.5, p < .01$). Since both TS variants use the same

Table 1. Best found LABS solutions using **TSv7**: $61 \leq n \leq 77$. These runs are performed on a 2.33 GHz Core2 Duo PC.

n	E(s)	F(s)	Runtime	Limit	Best Found LABS in Run Length Notation [2]
61	226	8.23	3 m	1.1 h	33211112111235183121221111311311
62	235	8.18	8 m	1.5 h	112212212711111511121143111422321
63	207	9.59	4 m	2.0 h	2212221151211451117111112323231
64	208	9.85	47 m	2.7 h	223224111341121115111117212212212
65	240	8.80	2.2 h	3.7 h	132323211111711154112151122212211
66	265	8.22	3.1 h	4.9 h	2432112312311211212412318111111311
67	241	9.31	4.1 h	6.6 h	12112111211222B222111111112224542
68	250	9.25	6.6 h	8.8 h	1111111141147232123251412112221212
69	274	8.69	8.2 h	11.8 h	11111111141147232123251412112221212
70	295	8.31	12.4 h	15.8 h	232441211722214161125212311111111
71	275	9.17	7.8 h	10.0 h	241244124172222111113112311211231121
72	300	8.64	2.4 h	10.0 h	111111411144417115112214212224222
73	308	8.65	1.2 h	10.0 h	1111112311231122113111212114171322374
74	349	7.85	0.2 h	10.0 h	11321321612333125111412121122511131111
75	341	8.25	8.0 h	10.0 h	12122132121211211111131111618433213232
76	338	8.54	4.6 h	10.0 h	111211112234322111134114212211221311B11
77	366	8.10	3.9 h	10.0 h	11111191342222431123312213411212112112

incremental OV computation and run on the same hardware, this difference in average runtimes can be attributed to the new stochastic strategy.

The least square fit on the logarithm of the average runtimes gives an estimated running time of $O(5.03e-6 * 1.37^n)$ and $O(1.03e-5 * 1.34^n)$ seconds for **TSv1** and **TSv7**, respectively. We believe **TSv7** to be the *current* state-of-the-art SLS algorithm for LABS. Due to lack of space, we do not show the error bars but the experiments show that **TSv7** is more robust than MA_{TS} and **TSv1**.

Table 1 explores the frontier of LABS instances, $61 \leq n \leq 77$, where optimal values have yet to be proven. For $61 \leq n \leq 70$, we use a runtime limit roughly based on the estimated runtime from Fig. 2. For $n > 70$, we use a runtime limit of 10 hours. We see that **TSv7** manages to obtain relatively good LABS solutions (by the merit factor) in reasonable running time.

5 Conclusion

The contributions of this paper are twofold. First, we show that one has to analyse the search trajectory and not just the timings for a SLS. Our implementation (**TSv1**) of the pseudo-code in [4] shows that it is actually a good strategy. The conclusion in [5] that MA_{TS} is faster than the original TS by ‘one order of magnitude’ is in part due to the less incremental implementation.

Second, we have shown how to engineer a new state-of-the-art LABS solver. Though the changes from **TSv1** to the final **TSv7** may seem small, it is often that small changes to an SLS causes big differences. Our changes are derived from reasoning on the LABS fitness landscape structure and TS trajectory behavior and thus serve as a rationale supported by empirical experiments. The resulting **TSv7** is also simpler than the hybrid MA_{TS} code.

Acknowledgements

We thank Iván Dotú and Pascal van Hentenryck for sharing their source code.

References

1. CSPLIB: A Problem Library for Constraints, <http://www.csplib.org>
2. Mertens, S.: Exhaustive search for low-autocorrelation binary sequences. *Journal of Physics A: Mathematical and General* 29, 473–481 (1996)
3. Brglez, F., Xiao, Y.L., Stallmann, M.F., Miltzer, B.: Reliable Cost Predictions for Finding Optimal Solutions to LABS Problem: Evolutionary and Alternative Algorithms. In: *Intl. Workshop on Frontiers in Evolutionary Algorithms* (2003)
4. Dotú, I., van Hentenryck, P.: A Note on Low Autocorrelation Binary Sequences. In: *Constraint Programming*, pp. 685–689 (2006)
5. Gallardo, J.E., Cotta, C., Fernández, A.J.: A Memetic Algorithm for the Low Autocorrelation Binary Sequence Problem. In: *GECCO*, pp. 1226–1233 (2007)

6. Halim, S., Yap, R.H.C., Lau, H.C.: An Integrated White+Black Box Approach for Designing and Tuning SLS. In: Constraint Programming, pp. 332–347 (2007)
7. Halim, S., Yap, R.H.C., Lau, H.C.: Viz: A Visual Analysis Suite for Explaining Local Search Behavior. In: ACM User Interface Software & Technology, pp. 57–66 (2006)
8. Halim, S., Yap, R.H.C.: Designing and Tuning SLS through Animation and Graphics: an Extended Walk-through. In: Engineering SLS Algorithms, pp. 16–30 (2007)

Author Index

- Anbulagan 550
Ansótegui, Carlos 298, 560
Araya, Ignacio 342
Audemard, Gilles 630
- Bacchus, Fahiem 478
Barahona, Pedro 608
Beck, J. Christopher 613
Béjar, Ramón 298, 560
Beldiceanu, Nicolas 220
Benedetti, Marco 463
Benhamou, Belaid 593
Benhamou, Frédéric 205
Benini, Luca 21
Bessiere, Christian 175
Botea, Adi 550
Boughaci, Dalila 593
Brito, Ismel 387
- Cambazard, Hadrien 418
Campêlo, Manoel B. 555
Carlsson, Mats 220
Cebrián, Manuel 82
Chase, Michael 97
Cheng, Kenil C.K. 509, 635
Christie, Marc 205
Cire, Andre A. 36
Clote, Peter 82
Collavizza, Hélène 327
Colmerauer, Alain 1
Correia, Marco 608
- de Souza, Cid C. 36
Dechter, Rina 534, 576
Di Rosa, Emanuele 603
Dotu, Ivan 82
Drias, Habiba 593
- Fernàndez, César 298, 560
Fox, Maria 618
- Garcia de la Banda, Maria 624
Gelain, Mirco 402
Giunchiglia, Enrico 603
Gogate, Vibhav 534
- Goldsztejn, Alexandre 190, 205, 598
Granvilliers, Laurent 190
Green, Martin J. 372
Gregory, Peter 618
- Hadzic, Tarik 448
Halim, Felix 640
Halim, Steven 640
Hebrard, Emmanuel 358
Heller, Daniel 539
Hnich, Brahim 235
Hooker, John N. 448
Hsu, Eric I. 613
Huang, Jinbo 144
- Jaffar, Joxan 493
Jeavons, Peter G. 112, 524
Jefferson, Christopher 372, 529
- Kadioglu, Serdar 251
Kitching, Matthew 478
Kumar, T.K. Satish 282
- Lallouet, Arnaud 463
Lebbah, Yahia 598
Lecoutre, Christophe 128
Lesaint, David 67
Li, Chendong 545
Li, Chu Min 313, 582
Lombardi, Michele 21
Long, Derek 618
Lopes, Tony M.T. 36
- Maher, Michael 159
Maier, Paul 566
Malik, Abid M. 97
Malitsky, Yuri 266
Manyà, Felip 582
Maratea, Marco 603
Marisetti, Satyanarayana 545
Marriott, Kim 624
Martin, Julien 220
Mateu, Carles 298, 560
McIlraith, Sheila A. 613
Mehta, Deepak 67

- Meseguer, Pedro 387
 Michel, Claude 598
 Michelin, Philippe 555
 Milano, Michela 21
 Mohamedou, Nouredine Ould 582
 Moura, Arnaldo V. 36
 Muise, Christian J. 613

 Narodytska, Nina 159
 Nethercote, Nicholas 624
 Neveu, Bertrand 342
 Normand, Jean-Marie 205

 O'Sullivan, Barry 52, 67, 358, 418, 433, 448
 Otten, Lars 576

 Panda, Aurojit 539
 Papadopoulos, Alexandre 433
 Petrie, Karen E. 529
 Pini, Maria Silvia 402
 Planes, Jordi 582
 Prestwich, Steven 235

 Quesada, Luis 67
 Quimper, Claude-Guy 159

 Rafeh, Reza 624
 Raiser, Frank 588
 Razgon, Igor 358
 Rodrigues, Carlos Diego 555
 Rossi, Francesca 402
 Rossi, Roberto 235
 Rueher, Michel 327, 598

 Ruggiero, Martino 21
 Russell, Tyrel 97

 Sachenbacher, Martin 566
 Salamon, András Z. 524
 Santosa, Andrew E. 493
 Schulte, Christian 571
 Sellmann, Meinolf 251, 266, 539
 Simon, Laurent 630
 Simonis, Helmut 52

 Tack, Guido 571
 Tarim, S. Armagan 235
 Tiedemann, Peter 448
 Trombettoni, Gilles 342

 van Beek, Peter 97
 Van Hentenryck, Pascal 82, 327
 van Hove, Willem-Jan 266
 Vautard, Jérémie 463
 Venable, K. Brent 402
 Verger, Guillaume 175
 Voicu, Răzvan 493

 Wallace, Mark 624
 Walsh, Toby 159, 402
 Wei, Wanxia 313
 Wilson, Nic 67

 Yap, Roland H.C. 509, 545, 635, 640
 Yip, Justin 539

 Zhang, Harry 313
 Zhang, Yuanlin 545
 Živný, Stanislav 112