SPRINGER OPTIMIZATION
AND ITS APPLICATIONS 45

Michela Milano
Pascal Van Hentenryck (Editors)

Hybrid Optimization

The Ten Years of CPAIOR

Springer

HYBRID OPTIMIZATION

Springer Optimization and Its Applications

VOLUME 45

Managing Editor
Panos M. Pardalos (University of Florida)

Editor—Combinatorial Optimization
Ding-Zhu Du (University of Texas at Dallas)

Advisory Board
J. Birge (University of Chicago)
C.A. Floudas (Princeton University)
F. Giannessi (University of Pisa)
H.D. Sherali (Virginia Polytechnic and State University)
T. Terlaky (McMaster University)
Y.Ye (Stanford University)

Aims and Scope

Optimization has been expanding in all directions at an astonishing rate during the last few decades. New algorithmic and theoretical techniques have been developed, the diffusion into other disciplines has proceeded at a rapid pace, and our knowledge of all aspects of the field has grown even more profound. At the same time, one of the most striking trends in optimization is the constantly increasing emphasis on the interdisciplinary nature of the field. Optimization has been a basic tool in all areas of applied mathematics, engineering, medicine, economics, and other sciences.

The *Springer Optimization and Its Applications* series publishes undergraduate and graduate textbooks, monographs, and state-of-the-art expository work that focus on algorithms for solving optimization problems and also study applications involving such problems. Some of the topics covered include nonlinear optimization (convex and nonconvex), network flow problems, stochastic optimization, optimal control, discrete optimization, multi-objective programming, description of software packages, approximation techniques and heuristic approaches.

For more titles published in this series, go to
http://www.springer.com/series/7393

HYBRID OPTIMIZATION

The Ten Years of CPAIOR

Edited By

MICHELA MILANO
Università di Bologna, Italy

PASCAL VAN HENTENRYCK
Brown University, USA

Springer

Editors
Michela Milano
Università di Bologna
Dipartimento di Elettronica
Informatica e Sistemistica
Viale Risorgimento 2
40136 Bologna
Italy
michela.milano@unibo.it

Pascal Van Hentenryck
Brown University
Department of Computer Science
02912 Providence Rhode Island
USA
pvh@cs.brown.edu

ISSN 1931-6828
ISBN 978-1-4419-1643-3 e-ISBN 978-1-4419-1644-0
DOI 10.1007/978-1-4419-1644-0
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2010937791

Mathematics Subject Classification (2010): 68Txx, 90-XX

© Springer Science+Business Media, LLC 2011
All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the publisher (Springer Science+Business Media, LLC, 233 Spring Street, New York, NY 10013, USA), except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed is forbidden.
The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

Cover illustration: Photo entitled 'Flaming Moss Reeds' taken by Charis Tyligadas

Printed on acid-free paper

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

Preface

This book celebrates the 10 years of CPAIOR, the International Conference on the integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming (CP). CPAIOR started in 1999 as a workshop series. It was first held in Ferrara, (Italy) and was co-organized by the Italian Association of Artificial Intelligence (AI*IA) and the Italian Association of Operations Research (AIRO). The workshop success went far beyond the expectations and received around 40 submissions and 70 participants from all over Europe and US. Such a large and lively community motivated researchers to organize the workshop for the following four editions, respectively, in Paderborn (Germany) in 2000, Ashford (United Kingdom) in 2001, Le Croisic (France) in 2002, and Montreal (Canada) in 2003. In 2004, CPAIOR became an International Conference with formal proceedings published as Springer LNCS. It was organized in Nice (France) with more than 100 participants. CPAIOR was then organized in Prague (Czech Republic) in 2005, Cork (Ireland) in 2006, Brussels (Belgium) in 2007, Paris (France) in 2008, Pittsburgh (USA) in 2009, while the upcoming meeting in 2010 will take place in Bologna (Italy). CPAIOR achieved in Paris the record number of 150 participants. CPAIOR main aim is to provide a forum for researchers in combinatorial optimization that exploits the hybridization of CP, AI, and OR techniques.

Constraint programming was always an integration technology as the original constraint programming languages already featured linear programming and satisfiability solvers. However, CP research in the early 1990s increasingly focused on the deep integration of OR techniques in finite domain solvers, which had become the essence of constraint programming. This cross-fertilization led to exciting results and have changed the way we look at combinatorial optimization. It led to new alleys for research, produced innovative optimization systems, provided more effective solutions to complex optimization problems, and enabled new application areas for optimization. This book glances through these exciting developments with the hope to draw readers to learn more and pursue this line of research.

The next 10 years of CPAIOR are likely to be interesting: They will certainly build on the foundations established in the last decade but will likely feature innovative, and far-reaching contributions that cannot be anticipated now.

The authors of the book chapters are eminent and well known researchers and have significantly contributed to the success story of the CPAIOR research field.

They come from Universities, Research Centers, and industry. We would like to thank all of them for their contribution.

Warm thanks go to reviewers of the collection for their invaluable contribution to the improvement of the chapter structure and content: Luca Benini, Agostino Dovier, Susanne Heipke, Holger Hoos, Narendra Jussien, Joao Marques-Silva, Meinolf Sellmann, Gilles Pesant, Jean-Charles Regin, Andrea Shaerf, Peter Stuckey, Mark Wallace, and Tallys Yunes.

Bologna, Italy — Michela Milano
Rhode Island, USA — Pascal Van Hentenryck
May 2010

Contents

Contributors

Tobias Achterberg IBM, CPLEX Optimization, Schoenaicher Str. 220, 71032 Boeblingen, Germany, achterberg@de.ibm.com

Allon Adir IBM Research – Haifa, Haifa University Campus, Haifa 31905, Israel, adir@il.ibm.com

Pedro Barahona Centro de Inteligência Artificial, Dep. de Informática, Universidade Nova de Lisboa, 2825 Monte de Caparica, Portugal, pb@di.fct.unl.pt

Christian Blum ALBCOM Research Group, Universitat Politècnica de Catalunya, Barcelona, Spain, cblum@lsi.upc.edu

Lucas Bordeaux Microsoft Research, Cambridge, UK, lucasb@microsoft.com

Pedro M. Castro Unidade de Modelação e Optimização de Sistemas Energéticos, Laboratório Nacional de Energia e Geologia, 1649-038 Lisboa, Portugal, pedro.castro@ineti.pt

Fabien Corblin TIMC-IMAG, Grenoble, France, Fabien.Corblin@imag.fr

Eric Fanchon TIMC-IMAG, Grenoble, France, Eric.Fanchon@imag.fr

Ignacio E. Grossmann Departement of Chemical Engineering, Carnegie Mellon University, Pittsburgh, PA 15213, USA, grossmann@cmu.edu

Youssef Hamadi Microsoft Research, 7 JJ Thomson Avenue, Cambridge, CB3 0FB, United Kingdom
and
LIX Ecole Polytechnique, F-91128 Palaiseau, France, youssefh@microsoft.com

Brahim Hnich Faculty of computer science, Izmir University of Economics, Izmir, Turkey, brahim.hnich@ieu.edu.tr

John N. Hooker Carnegie Mellon University, Pittsburgh, PA, USA, john@hooker.tepper.cmu.edu

Ludwig Krippahl Centro de Inteligência Artificial, Departamento de Informática, Universidade Nova de Lisboa, 2825 Monte de Caparica, Portugal, ludi@di.fct.unl.pt

Andrea Lodi DEIS, University of Bologna, Viale Risorgimento 2, 40136 Bologna, Italy, andrea.lodi@unibo.it

Michela Milano Università di Bologna, Dipartimento di Elettronica, Informatica e Sistemistica, Viale Risorgimento 2, 40136 Bologna, Italy, michela.milano@unibo.it

Eric Monfroy Universidad Santa María, Valparaíso, Chile, Eric.Monfroy@inf.utfsm.cl
and
LINA, Université de Nantes, France, Eric.Monfroy@univ-nantes.fr

Yehuda Naveh IBM Research – Haifa, Haifa University Campus, Haifa 31905, Israel, nahev@il.ibm.com

Olivier Perriquet Centro de Inteligência Artificial, Departamento de Informática, Universidade Nova de Lisboa, 2825 Monte de Caparica, Portugal, olivier@perriquet.net

Steven Prestwich Department of Computer Science, University College Cork, College Road, Cork, s.prestwich@cs.ucc.ie

Jakob Puchinger Mobility Department, Austrian Institute of Technology, Vienna, Austria, jakob.puchinger@ait.ac.at

Günther Raidl Institute of Computer Graphics and Algorithms, Vienna University of Technology, Vienna, Austria, raidl@ads.tuwien.ac.at

Philippe Refalo IBM, Les Taissounieres, 1681, route des Dolines, 06560 Sophia Antipolis, France, philippe.refalo@fr.ibm.com

Jean-Charles Régin Université de Nice-Sophia Antipolis, I3S/CNRS, 2000, Route des Lucioles - Les Algorithmes - bt. Euclide B BP 121 - 06903 Sophia Antipolis, Cedex, France, jcregin@gmail.com

Andrea Roli DEIS, Campus of Cesena, *Alma Mater Studiorum* Università di Bologna, Cesena, Italy, andrea.roli@unibo.it

Roberto Rossi Logistics, Decision and Information Sciences, Wageningen University, "De Leeuwenborch", Hollandseweg 1, 6706KN, Wageningen, roberto.rossi@wur.nl

Louis-Martin Rousseau Department de Mathématiques et Génie Industriel, École Polytechnique de Montréal, Montréal, Canada, louis-martin.rousseau@polymtl.ca

Frédéric Saubion LERIA, Université d'Angers, Angers, France, Frederic.Saubion@univ-angers.fr

Paul Shaw IBM, 1681 route des Dolines, 06560 Valbonne, France, paul.shaw@fr.ibm.com

S. Armagan Tarim Department of Management, Hacettepe University, Ankara, Turkey, armagan.tarim@hacettepe.edu.tr

Michael A. Trick Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA 15213, USA, trick@cmu.edu

Laurent Trilling TIMC-IMAG, Grenoble, France, Laurent.Trilling@imag.fr

Pascal Van Hentenryck Brown University, Department of Computer Science, 02912 Providence Rhode Island, USA, pvh@cs.brown.edu

Willem-Jan van Hoeve Tepper School of Business, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, USA, vanhoeve@andrew.cmu.edu

Tallys Yunes Department of Management Science, School of Business Administration, University of Miami, Coral Gables, FL 33124-8237, USA, tallys@miami.edu

The Ten Years of CPAIOR: A Success Story

Michela Milano and Pascal Van Hentenryck

Abstract The purpose of this chapter is to introduce the collection that celebrates the 10 years of CPAIOR. First, a short overview of research topics addressed by the ever growing CPAIOR community is presented. Then, a short chapter summary follows that describes the book structure and its content.

1 Introduction

This book celebrates the 10 years of CPAIOR, the international conference on the integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming (CP). CPAIOR started in 1999 as a workshop series (first held in Ferrara, Italy). In 2004, CPAIOR became an International Conference with formal proceedings [1, 2, 4–7] to provide a forum for research in combinatorial optimization that exploits the hybridization of CP, AI, and OR techniques. Constraint programming was always an integration technology as the original constraint programming languages already featured linear programming and satisfiability solvers.

Constraint programming from its inception featured an innovative "modeling" language. Instead of formulating an optimization problem as a set of linear inequalities, constraint programming describes an application by expressing its substructures (the so-called global constraints) and combining them with various combinators. These substructures capture important building blocks arising in many applications and are often (but not always) computationally tractable. In the early 1990s, part of constraint programming research focused on efficient algorithms and theoretical results for deciding the satisfiability of global constraints in conjunction with the domains of their variables and for filtering the variable domains. Edge-finding algorithms and the alldifferent constraints are beautiful illustrations of this line of research which significantly boosted the effectiveness of

M. Milano (✉)
Università di Bologna, Dipartimento di Elettronica, Informatica e Sistemistica,
Viale Risorgimento 2, 40136 Bologna, Italy
e-mail: michela.milano@unibo.it

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_1,
© Springer Science+Business Media, LLC 2011

constraint programming to tackle combinatorial applications in scheduling and rostering. Later in the 1990s, the integration of CP and OR moves from satisfiability to optimization through the concept of cost-based filtering which used reduced costs to filter domains in global optimization constraints. This direct link to linear programming, the emergence of modeling languages integrating constraint and mathematical programming, and the CPAIOR workshops provided a fertile ground for researchers to explore hybrid optimization techniques. It first led to a better understanding of the strengths and limitations of each paradigm, then to a recognition of their complementarities and, finally, to the design of truly hybrid optimization methods and tools.

Hybrid decomposition methods provide a compelling illustration of these synergies. These hybridizations recognize that CP is excellent to explore highly constrained combinatorial spaces quickly, while mathematical programming is particularly strong at deriving lower bounds. For instance, in logical Benders decomposition, the master problem, which ignores some combinatorial constraints, is solved with an MIP solver, while the subproblem tackles the combinatorial constraints using constraint programming. In constraint-based column generation, the master problem is a linear program and the columns are generated by using constraint programming to solve a combinatorial subproblem (e.g., a shortest path under side constraints). Other hybridizations, such as large neighborhood search, also exploit the strength of constraint programming to explore highly constrained combinatorial spaces to move from solutions to solutions in a local search. These beautiful results are not only effective from a computational standpoint; they also enable practitioners to use the rich language of constraint programming and its ability to handle side constraints.

Constraint programming models are often closer to applications than their OR counterparts. It is important, however, to recognize that they do not prescribe any optimization technology. CP models can be linearized automatically and solved by a MIP solver; they can be translated to conjunctive normal form and solved by a SAT solver; or they can be handled directly by constraint-based local search. Once again, these developments have opened new avenues for combinatorial optimization since the same models can now be solved jointly or separately by different technologies. In particular, global constraints can now provide a filtering algorithm for constraint programming, a linearization and cut generation algorithm for mathematical programming, and automatic differentiation for local search. Optimization systems are now being built along these lines, although practical implementations lag behind the conceptual frameworks.

The cross-fertilization between CP, AI, and OR also confronted some fundamental "philosophical" differences in how to build optimization systems. In mathematical programming, the solver is often seen as a black-box, while constraint programming solvers often leave considerable lattitude, including the ability to program the search. Significant research in constraint programming in recent years has focused on automating search in order to ease the use of the technology. At the same time, mathematical programming is moving toward a more expressive constraint language, showing how the two fields are impacting each other.

This book glances through exciting developments in the field of Hybrid Optimization with the hope to draw readers to learn more and pursue this line of research. It is unrealistic for an introduction to do justice to the wealth of research that emerged from CPAIOR in the last 10 years. Instead, it presents a small blueprint that should help readers approaching and understanding the existing results and the challenges in this area. This short summary highlights the flurry of activities and accomplishments at the intersection of CP, AI, and OR for combinatorial optimization. The field of hybrid optimization holds tremendous promises, and it is now attacking stochastic optimization and over-constraint problems, which are also ripe with hybridization opportunities. Hybrid optimization also faces many challenges. Paramount among these, at least in the short term, is ease of use. Optimization systems have to be developed in order to support the theoretical and conceptual progress, they should contain explanations and sensibility analysis of their results, and tools to understand performance. Theoretical progress is also needed to understand the strengths, limitations, and applicability of various contributions and help practitioners decide how to design effective hybrid solutions. In the long term, the challenges are even more daunting. Due to progress in telecommunications, globalization, and consolidation, optimization problems are increasingly large-scale and dynamic and integrate subproblems that used to be solved separately. The new interdisciplinary field of Computational Sustainability [3], that aims to apply techniques from computer and information science, operations research, applied mathematics, and statistics for balancing environmental, economic, and societal needs for sustainable development, poses new challenges to hybrid optimization. At the same time, computer devices are invading our homes, our cars, and our phones, generating a wealth of new optimization applications with new requirements. The next 10 years of CPAIOR are likely to be interesting: They will certainly built on the foundations established in the last decade but will likely feature innovative, and far-reaching contributions that cannot be anticipated now.

2 Chapter Overview

The contributions collected in the book cover some of the main topics of the CP-AI-OR literature. The book is conceptually divided into five parts:

- Modeling practice in hybrid systems (Chapter “Hybrid Modeling”)
- Hybrid problem solving techniques, including global constraints, hybrid solvers, decomposition method, and techniques for over-constrained problems (Chapters “Global Constraints: A Survey”, “Decomposition Techniques for Hybrid MILP/CP Models Applied to Scheduling and Routing Problems”, “Hybrid Solving Techniques” and “Over-Constrained Problems”)
- Search strategies that integrate tree search, local search, meta-heuristic and learning (Chapters “A Survey on CP-AI-OR Hybrids for Decision Making Under Uncertainty”, “Constraint Programming and Local Search Hybrids”, “Hybrid Metaheuristics”, “Learning in Search” and “What is Autonomous Search?”)

- Tools enabling integration and integrations of CP and MIP with SAT solvers (Chapters "Software Tools Enabling Integration" and "Connections and Integration with SAT Solvers: A Survey and a Case Study in Computational Biology")
- Applications that benefit from the integration (Chapters "Bioinformatics: A Challenge to Constraint Programming", "Sports Scheduling" and "Stimuli Generation for Functional Hardware Verification with Constraint Programming")

In the following, we give an outline of the book by providing a short description of each chapter.

Hybrid Modeling

The chapter provides a comprehensive list of practical guidelines and supporting theory for CP, Mixed Integer Linear Programming (MILP) modeling and for their integration.

CP modeling heavily relies on the use of global constraints that represent a structured problem component, compactly expressing a set of elementary constraints and in general encapsulates specialized, efficient, and effective filtering algorithms. On the other hand, MILP modeling relies on linear inequations. Some or all the variables are restricted to integer values. The solution process is based on linear relaxation that can be tightened by the addition of cutting planes. The key to building MILP formulation is to recognize the structure of feasible sets that are representable in MILP models.

The chapter discusses ways to integrate the two modeling styles and proposes guidelines for integrated modeling. A number of examples are provided to support the theory.

Global Constraints: A Survey

The chapter provides a comprehensive survey and an interesting classification of global constraints and their weighted version into the following categories: solution based, counting, balancing, combinations of basic constraints, sequencing, distance, geometric, summation-based, graph-based, order-based constraints. For each category, a list of global constraints along with details on their filtering algorithms are presented. Global constraints are a very powerful mean for the integration of OR techniques into CP. Many filtering algorithms embedded in global constraints are in fact graph algorithms and are based either on dynamic programming or on the linear programming relaxation or on lagrangean relaxation or on combinatorial relaxations. The chapter ends with a discussion on important aspects related to global constraint filtering algorithms.

Decomposition Techniques for Hybrid MILP/CP Models Applied to Scheduling and Routing Problems

Decomposition is a mean for integration: it applies when the original problem can be decomposed into two subproblems: one is MILP and another is CP subproblem.

Each model is solved separately and information obtained while solving one subproblem is used for the solution of the other subproblem.

The chapter overviews in detail two decomposition methods that are widely used in hybrid systems: Benders Decomposition and Column Generation.

The chapter shows two important applications tackled with decomposition techniques: scheduling and vehicle routing.

Hybrid Solving Techniques

The chapter describes hybrid algorithms that merge MIP techniques into CP and viceversa. Examples of the first kind of integration is the linearization of global constraints, the use of relaxations within global constraints, and cost based filtering. The most significant example of the second integration is the SCIP framework, whose description covers half of the chapter. The SCIP framework, born mainly in the OR community, is the first significant attempt to incorporate in MIP some of the expressiveness of CP into the final hybrid systems.

Over-Constrained Problem

In over-constrained problems, constraints do not allow any solution to the problem. After a brief historical review on the over-constrained problem literature, the chapter focusses on soft global constraints that can be violated. When a soft global constraint is violated, we measure the degree of violation and we wish to minimize the overall amount of violation.

Three soft global constraints are presented in detail, namely, the soft alldifferent, the soft global cardinality constraint, and the soft regular constraint, while many others are listed and references provided.

A Survey on CP-AI-OR Hybrids for Decision Making Under Uncertainty

This chapter considers problems where decision variables appear together with random variables that are not controllable by the decision makers but are observable.

An example is considered in the chapter, namely, single and multi-stage stochastic knapsack. First, a number of stochastic applications solved with hybrid approaches are listed. Then, algorithmic approaches are outlined and basically grouped into three categories: search and filtering based on stochastic reasoning, reformulation based and sample based approaches. The first two categories are in turn divided into general purpose and problem specific techniques, while sample based approaches are divided into the sample average approximation approach, forward sampling and sample aggregation. A discussion on related modeling frameworks concludes the chapter.

Constraint Programming and Local Search Hybrids

In the recent years, many attempts have been performed for integrating constraint programming and local search methods. The chapter, after describing the main local search methods used in conjunction with constraint programming, presents basically three ways of integrating the two. The first is a loose integration where constraint programming is simply used to check the consistency of a local search state. The second, tighter integration, uses constraint programming to explore the neighborhood of a given reference solution. The third way to integrate CP and LS concerns the use of local search for enhancing propagation and filtering.

Hybrid Metaheuristics

This chapter is devoted to the presentation of hybrid meta-heuristics: the authors provide examples and literature overviews concerning five important categories of hybrid metaheuristics. More specifically the chapter focuses on the hybridization of metaheuristics with (meta-)heuristics, constraint programming, tree search methods, problem relaxations, and dynamic programming. Each of the five categories mentioned above is treated in its own subsection. In each subsection, first, two representative examples are outlined, and then, a short literature overview is provided.

Learning in search

The chapter considers search in constraint programming, integer programming, and SAT and aims at describing techniques that help the search strategy to require as little as possible user intervention.

The first part of the chapter focuses on recent advances in strategy learning. Then restart is discussed as an effective method that plays a fundamental role together with no-good learning and clause learning in SAT solvers. Finally, the last part considers the combined effect of strategy learning, restart, and clause learning on the problem solving process.

What is Autonomous Search?

Autonomous search is a particular case of adaptive system that modifies and adapts itself to the problem considered in order to improve performance. The kind of adaptation can be either supervised or self-adaptation. The chapter focuses on the solver architecture containing the problem model/encoding, an evaluation function, the solving algorithms, the parameters to be configured, and the control component. An architecture for autonomous solvers is presented along with computation rules and their control. Finally, some case studies are discussed.

Software Tools Enabling Integration

A number of tools enabling integration have been developed both in the Constraint Programming and in the OR community.

This chapter first presents a number of features that facilitate integration and then describes a number of tools namely BARON, Comet, Ecl^{i}ps^e, G12, ILOG Optimizer and OPL development studio, SCIP, SIMPL, Xpress-Mosel, COIN-OR, Microsoft Solver Foundation, Prolog IV, SALSA, and TOoLS. For each tool, its main features are described with specific emphasis on the ones enabling of supporting integration.

Connections and Integration with SAT Solvers:A Survey and a Case Study in Computational Biology

An important breakthrough in constraint satisfaction over the past decade has been the advent of highly scalable solvers for Propositional Satisfiability (SAT). After an introduction on the integration of SAT solvers in CP, and their common features, the chapter focuses on gene regulatory network (GRN) deciphering. A GRN abstracts the interactions between several genes of a cell. The chapter describes the GRN deciphering modeling in CLP and in SAT and the corresponding experimental results.

Bioinformatics: A Challenge to Constraint Programming

The chapter is focussed on bioinformatics, a rapidly growing field at the intersection of biology and computer science. It reports the main problems where constraint programming, properly integrated with other techniques, plays an important role. The main applications covered by the chapter can be identified into: analysis of

sequence data, such as sequence comparison and pattern matching along with evolutionary methods and population genetics problems; RNA structures, for which there are several solutions based on local search and dynamic programming; protein structure prediction and determination problems along with protein interaction models; system biology modeling complex network of interactions of which life is made.

Sports Scheduling

The purpose of this chapter is to highlight the role integer programming, constraint programming, metaheuristics, and combinations thereof has played in advancing the theory and practice of sports scheduling. Two major sports scheduling problems are considered: The Constrained Break Optimization Problem and the Traveling Tournament Problem along with some variants. Both of these problems have been the subject of many papers in the last years, and both have proven to be very challenging problems. The chapter outlines some challenges and opportunities for the field.

Stimuli Generation for Functional Hardware Verification with Constraint Programming

Stimuli generation for functional hardware verification is the process of creating test cases with the intent of revealing unknown bugs in hardware designs, before the design is cast in silicon. After a description of the problem at hand, the chapter shows how CSP techniques are used for stimuli generation, and also some of the extension of these techniques that are needed for this specific domain. Then, the chapter concentrates on general aspects of modeling stimuli generation problems as CSPs, and describes specific areas of application within the domain and the general CSP model for each area. Challenges of current research interest are also highlighted at the end of the chapter.

References

1. Barták R, Milano M (eds) (2005) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems. In: Second international conference, CPAIOR 2005, Prague, Czech Republic, May 30–June 1, 2005, Proceedings. Lecture notes in computer science, vol 3524. Springer, Berlin
2. Beck JC, Smith BM (eds) (2006) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems. In: Third international conference, CPAIOR 2006, Cork, Ireland, May 31–June 2, 2006, Proceedings. Lecture notes in computer science, vol 3990. Springer, Berlin

3. Gomes CP (2009) Challenges for constraint reasoning and optimization in computational sustainability. In: Proceedings of the int.l conference on principles and practice of constraint programming – CP 2009, pp 2–4
4. Van Hentenryck P, Wolsey LA (eds) (2007) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems. In: 4th international conference, CPAIOR 2007, Brussels, Belgium, May 23–26, 2007, Proceedings. Lecture notes in computer science, vol 4510. Springer, Berlin
5. Perron L, Trick MA (eds) (2008) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems. In: 5th international conference, CPAIOR 2008, Paris, France, May 20–23, 2008, Proceedings. Lecture notes in computer science, vol 5015. Springer, Berlin
6. Régin J-C, Rueher M (eds) (2004) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems. In: First international conference, CPAIOR 2004, Nice, France, April 20–22, 2004, Proceedings. Lecture notes in computer science, vol 3011. Springer, Berlin
7. van Hoeve WJ, Hooker JN (eds) (2009) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems. In: 6th international conference, CPAIOR 2009, Pittsburgh, PA, USA, May 27–31, 2009, Proceedings. Lecture notes in computer science, vol 5547. Springer, Berlin

Hybrid Modeling

John N. Hooker

Abstract The modeling practices of constraint programming (CP), artificial intelligence, and operations research must be reconciled and integrated if the computational benefits of combining their solution methods are to be realized in practice. This chapter focuses on CP and mixed integer/linear programming (MILP), in which modeling systems are most highly developed. It presents practical guidelines and supporting theory for the two types of modeling. It then suggests how an integrated modeling framework can be designed that retains, and even enhances, the modeling power of CP while allowing the full computational resources of both fields to be applied and combined. A series of examples are used to compare modeling practices in CP, MILP, and an integrated framework.

1 Modeling as a Key to Hybrid Problem Solving

The solution methods of constraint programming (CP), artificial intelligence, and operations research have complementary strengths. Recent research shows that these strengths can be profitably combined in hybrid algorithms, many of which are described in subsequent chapters of this book. Under the right conditions, one need not choose between CP, AI, and OR, but can have the best of all three worlds.

There is more to integration, however, than combining algorithmic techniques. There is also the issue of problem formulation. CP, AI, and OR have developed their own distinctive modeling styles, which poses the question of how to formulate problems that are to be solved by hybrid methods. Whereas the solution methods of the three fields can be seen as related and complementary, the modeling styles seem very different and possibly irreconcilable.

For example, one can contrast CP models with mixed integer/linear programming (MILP) models developed in the OR community. CP organizes its models

J.N. Hooker (✉)
Carnegie Mellon University, Pittsburgh, PA, USA
e-mail: john@hooker.tepper.cmu.edu

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_2,
© Springer Science+Business Media, LLC 2011

around high-level *global constraints*, each of which represents a structured collection of simpler constraints. The solver may have a large library of global constraints for which special-purpose algorithms have been designed. The modeler selects constraints that correspond to the major structural elements of the problem and combines them with some low-level constraints as needed to complete the formulation.

An advantage one might claim for this approach is that the selection of global constraints reveals the special structure of the problem and allows the solver to exploit it. It may also result in a fairly succinct model that is easier to read and debug because the global constraints reflect how the modeler thinks about the problem. Such a model can be seen as a theory or explanation that helps us understand the phenomenon described by the model. Scientific theories, after all, are essentially explanatory models of phenomena. On the other hand, a model that uses high-level global constraints may have to be reformulated for solvers that recognize different sets of constraints.

An MILP model takes the opposite approach. It uses a very small set of primitive terms, namely linear inequalities. The problem is broken down into elementary ideas that can be captured with inequality constraints, perhaps using auxiliary variables and other devices. Advanced modeling systems allow one to abbreviate the formulation with loops and if-then statements that generate a large number of constraints, but the formulation must nonetheless be conceived in terms of inequality constraints.

An advantage cited for this approach is the independence of model and method. Once the model is written, it can be submitted to any MILP solver. In fact, there are libraries of MILP instances that have been used, without alteration, as testbeds for several generations of solvers. In addition, the user need not be familiar with a library of meta-constraints. Finally, inequality constraints are suitable for the highly developed relaxation technology of MILP, including strong cutting planes, Lagrangean relaxation, and so forth. On the negative side, the model may be long, nonintuitive, and hard to debug. The solver may be unable to exploit substructure that global constraints would have revealed, aside from a few special types of structure that can be automatically recognized by more sophisticated solvers.

The modeling issue must be resolved if hybrid solvers are to harness the complementary strengths of CP and MILP. CP methods rely heavily on the application of specialized filtering methods to global constraints and must therefore know where global constraints appear in the problem. MILP methods rely heavily on relaxation methods and cutting planes that have been developed for inequality constraints and therefore require that inequality constraints appear in the model.

The modeling issue is important in practice. Developing a formulation and the associated data is an expensive undertaking. Models must typically be reformulated, updated, and debugged many times. While reasonably fast solution is desirable, it is equally essential that solution software support and simplify modeling activities. Practitioners frequently report that the greatest benefit of developing a model is not so much the ability to obtain a solution as the clearer understanding of the problem one obtains from the modeling exercise.

Practical application also requires interaction between modeling and the solution process. There are typically alternative ways to formulate a problem, some of which result in much faster solution than others. A practitioner can begin with a straightforward model and solve it on a small problem instance. The model can then be altered and refined as the instances are scaled up, so as to maintain tractability. A practical modeling system should support this kind of trial-and-error process.

This chapter focuses on integrating the modeling styles of CP and MILP in particular, because it is in these areas that modeling systems are most highly developed. After a brief review of current hybrid modeling systems and some basic terminology, the chapter is organized in three major sections. The first two sections develop guidelines for CP and MILP modeling, respectively. They introduce the necessary theory and present a series of examples to illustrate good modeling practice. The third section suggests how these modeling practices may be merged into a unified approach, based on ideas that have evolved in the CP-AI-OR community over the last decade or more. It illustrates the ideas by showing how each of the examples discussed earlier may be rendered in an integrated modeling framework. The chapter concludes by assessing the extent to which the particular strengths of CP and MILP modeling carry over into an integrated framework.

It is possible to develop modeling languages that specify the search procedure as well as the problem (e.g., [21, 40]). Although algorithmic modeling is beyond the scope of this chapter, it may become a key component of integrated modeling systems.

2 Modeling Systems

A number of modeling systems implement CP/MILP hybrid modeling to a greater or lesser extent. A pioneering effort is ECL^{i}PS^e [2, 4, 15, 50], a Prolog-based constraint logic programming system that provides an interface with linear and MILP solvers. ECL^{i}PS^e was recently revised to accept models written in MiniZinc, a CP-based modeling system [41]. OPL Studio [22] provides a modeling language that expresses both MILP and CP constraints. A script language allows one to write algorithms that call the CP and MILP solvers repeatedly. Mosel [17, 18] is a specialized programming language that interfaces with various solvers, including MILP and CP solvers.

SCIL [3] is an MILP solver with a modeling language that designates specially structured sets of inequalities. SIMPL [5, 62] is a hybrid solver with a high-level modeling language that integrates CP and MILP modeling. The solver processes each constraint with both CP-based and MILP-based techniques that are combined in a branch-infer-and relax algorithmic framework. SCIP [1] is a callable library that gives the user control of a solution process that can involve both CP and MILP solvers. G12 [53] is a CP-based and hybrid system that accepts models written in the Zinc modeling language and uses a mapping language (Cadmium) to associate the models with underlying solvers and/or search strategies.

There has been some investigation of integrated modeling beyond the CP/MILP interface. The modeling language in the Comet system [21], which evolved from an earlier system Localizer [40], allows CP and MILP constraints as well as high-level constraint-based specifications of local search. The global optimization package BARON [54, 55] combines nonlinear (as well as linear) integer programming with CP-style domain reduction, although it uses a modeling system (AIMMS) that does not support CP-style constraints. Some general discussions of integrated modeling include [26, 29, 62].

No attempt is made here to describe currently available modeling languages, because they evolve rapidly. The emphasis is on general principles that should inform the design of any integrated modeling system. In fact, none of the existing systems implement all of these principles or fully integrate CP and MILP modeling. Yet the necessary concepts and technology have reached a stage where a seamlessly integrated modeling system is within reach. Perhaps the discussion to follow will help encourage efforts in this direction.

3 Basic Terminology

For present purposes, a *problem* consists of a stock of *variables* $x_1, \ldots, x_n$ and a set of *constraints*. Each variable x_j is associated with a *domain* that can be viewed as a set of permissible values for x_j. A *solution* is any tuple $x = (x_1, \ldots, x_n)$ for which each x_j belongs to its domain. Each constraint is associated with a set of solutions that *satisfy* it. The goal is to find a *feasible* solution, which satisfies all the constraints.

When there is an *objective function* $f(x)$, the goal is to find an *optimal* solution, which can without loss of generality be defined as a feasible solution that minimizes $f(x)$ subject to the constraints. That is, an optimal solution $\bar{x}$ is one such that $f(\bar{x}) \leq f(x)$ for all feasible solutions x.

The terminology here is borrowed from both CP and OR, with constraints and domains defined roughly as in CP, and the various types of solutions as in OR. It is important to note that the domain of a variable need not consist of numbers, although this is the normal practice in OR. In CP, a domain may consist of arbitrary objects, or even sets of objects.

CP methods are typically designed to find feasible solutions and OR methods to find optimal solutions, but this is not a fundamental difference. CP methods can find optimal solutions by adding the constraint $f(x) \leq U$ to the problem and gradually reducing the bound U until no feasible solution can be found.

The *modeling* task is (a) to identify variables, domains, constraints, and perhaps an objective function that formulate the desired problem, and (b) to express the constraints and objective function in a form that allows solution by available software.

4 CP Modeling

The concepts of domain consistency, filtering, propagation, and global constraints are essential to understanding CP modeling practice. Once these are defined, guidelines for CP modeling can be stated and illustrated with a series of examples. These examples will recur in later sections to show how they can be formulated as MILP models and in an integrated modeling context. For further practice, one can consult the tutorials on CP modeling in [49, 52].

4.1 Consistency, Filtering, and Propagation

A central concept in CP solution technology is *domain consistency*, also known as *generalized arc consistency* or *hyperarc consistency*. A problem is domain consistent if every element of every domain is consistent with the constraint set. That is, for each element v in the domain of any variable x_j, there is at least one feasible solution in which $x_j = v$. Another way to put this is that each variable's domain is equal to the projection of the feasible set onto that variable.

CP solvers typically achieve or approximate domain consistency for individual constraints by means of *filtering* algorithms that remove inconsistent values from domains. The smaller domains obtained by filtering a constraint become the starting point for filtering another constraint, in a process known as *constraint propagation*. It is important to note that constraint propagation does not necessarily achieve domain consistency for the problem as a whole, even if the filtering algorithms achieve it for every individual constraint.

The advantage of filtering domains is that the search algorithm spends less time enumerating values that cannot be part of a feasible solution. If filtering reduces every domain to a singleton and achieves domain consistency as well, then a feasible solution is at hand.

A somewhat weaker form of consistency is *bounds consistency*, which applies when there is a natural ordering for the elements of a domain. A problem is bounds consistent if for each domain, its smallest value is consistent with the constraint set, and likewise for its largest value.

4.2 Global Constraints

CP modeling relies heavily on the use of *global constraints*. A global constraint represents a set of more elementary constraints that exhibit special structure when considered together. Each individual constraint typically involves only a few of the variables that appear in the global constraint and might be viewed as "local" in that sense.

A practice of using global constrains, rather than writing out the more elementary constraints, has several advantages: (a) it is more convenient; (b) it yields a more natural and readable model that is more easily debugged; (c) it alerts the solver that the model contains special structure that might have been overlooked if the elementary constraints had been written. In particular, filtering algorithms designed for a global constraint can generally remove more values than filtering algorithms designed for the more elementary constraints.

An example of a global constraint is the well-known *all-different* constraint, which can be written $\texttt{alldiff}(X)$ and requires that the variables in the set $X = \{x_1, \ldots, x_k\}$ take pairwise distinct values. It replaces a set of more elementary constraints in the form of inequations $x_i \neq x_j$ for $1 \leq i < j \leq k$.

Filtering is more effective when applied to an `alldiff` constraint than when applied to the individual inequations it represents. Suppose, for example, that variables x_1, x_2, x_3 all have domain $\{a, b\}$. The constraint $\texttt{alldiff}(\{x_1, x_2, x_3\})$ allows filtering to reduce each domain to the empty set if domain consistency is achieved. By contrast, achieving domain consistency for the individual inequations $x_1 \neq x_2$, $x_1 \neq x_3$, $x_2 \neq x_3$ removes no values from the domains. It is therefore better modeling practice to use the `alldiff`.

4.3 Example: Sudoku Puzzles

The popular sudoku puzzle (Fig. 1) illustrates how global constraints can be used in modeling, in this case `alldiff` constraints. A sudoku puzzle consists of a 9×9 grid whose cells must be filled with digits $1, \ldots, 9$ so that each row and each column of the grid contains nine distinct digits. In addition, each of the nine 3×3 subsquares of the grid must contain nine distinct digits. Some of the cells have preassigned digits.

The first task in formulating a model is normally to define the variables. In this case, a natural scheme is to let x_{ij} be the digit in row i and column j, so that each x_{ij} has domain $\{1, \ldots, 9\}$. The domain could of course be any set of nine distinct objects, not necessarily numbers, without affecting the problem. Let X_{i*} be the set of variables in the ith row, namely $\{x_{i1}, \ldots, x_{i9}\}$, and let X_{*j} be the set of variables

6		7	4	1	2			5
				8		4		
		4			3	8	2	6
	2					1	6	3
			5		6			
3	4	6					5	
4	7	3	2			5		
		9		5				
5			1	7	9	3		2

Fig. 1 A sudoku puzzle

in the jth column. Also let $X^{k\ell}$ contain the variables corresponding to the cells in the 3×3 square in position k, ℓ, for $k, \ell \in \{1, 2, 3\}$. Suppose the content of cell i, j is preassigned a_{ij} for all $(i, j) \in S$. Then, the problem can be formulated

$$\begin{aligned}
&\texttt{alldiff}(X_{i*}),\ \texttt{alldiff}(X_{*i}),\ i = 1, \ldots, 9\\
&\texttt{alldiff}(X^{k\ell}),\ k, \ell = 1, 2, 3\\
&x_{ij} = a_{ij},\ \text{all } (i, j) \in S\\
&x_{ij} \in \{1, \ldots, 9\},\ \text{all } i, j
\end{aligned} \tag{1}$$

Propagation is more effective if the `alldiff`s are filtered simultaneously. If the modeling system has a multiple `alldiff` constraint (not yet standard), the first two lines of (1) should be replaced with

$$\texttt{multiAlldiff}\begin{pmatrix} X_{i*}, X_{*i},\ i = 1, \ldots, 9, \\ X^{k\ell},\ k, \ell = 1, 2, 3 \end{pmatrix} \tag{2}$$

Moreover, the `alldiff`s in the first line of (1) have special structure, in that they define a Latin square. If at some point a specialized filter is developed for Latin squares, a global constraint `LatinSquare`(X) can be added to the model, where X is the matrix of variables x_{ij}. The new constraint is redundant of the `alldiff`s, but redundancy can result in better propagation.

4.4 CP Modeling Guidelines

At least four principles should guide the formulation of CP models. They will also carry over into an integrated modeling framework.

1. A specially-structured subset of constraints should be replaced by a single global constraint that captures the structure, when a suitable one exists. This produces a more succinct model and can allow more effective filtering and propagation.
2. A global constraint should be replaced by a more specific one when possible, to exploit more effectively the special structure of the constraints.
3. The addition of redundant constraints (i.e., constraints that are implied by the other constraints) can improve propagation.
4. When two alternate formulations of a problem are available, including both (or parts of both) in the model may improve propagation. This is especially helpful when some constraints are hard to write in one formulation but suitable for the other. The dual formulations normally contain different variables, which should be defined in terms of each other through the use of *channeling* constraints.

The sudoku formulation illustrates Principle 1, because each `alldiff` constraint represents many inequations. If the `multiAlldiff` constraint is used as in (2), this again accords with Principle 1, because the `multiAlldiff` replaces twenty-seven `alldiff` constraints with overlapping variable sets.

The sudoku model (1) also illustrates Principle 3, because one of the `alldiff` constraints is redundant. If the numbers in rows 1–8 are all different, and those in columns 1–9 are all different, then row 9 necessarily contains nine different numbers. Nonetheless, it is good modeling practice to include a redundant `alldiff` constraint for row 9.

A practice of including redundant constraints may appear contrary to the goal of writing perspicuous models, because it makes the models longer. Yet redundant constraints can sometimes result in a more intuitive statement of the problem, as in the sudoku example. When redundancy is introduced by dual formulations, the two formulations can be separated in the model statement for clarity. It is natural (as well as computationally advantageous) to make each one as complete as possible, rather than arbitrarily dropping some constraints for the sake of removing redundancy.

In other cases, however, the modeler may become aware of redundant constraints after completing the formulation. Adding them complicates the model, as in the case of the `LatinSquare` constraint in the sudoku model. Yet the redundant constraints can be written separately for clarity, and the model continues to provide an explanatory theory–perhaps even a better theory, because it includes some "theorems" (redundant constraints) along with the "axioms" (original constraints). It can be a good exercise to think through some of the consequences of a model by deriving redundant constraints.

4.5 Example: Car Sequencing

A car sequencing example, adapted from [52], introduces three important global constraints. An assembly line makes fifty cars a day. To simplify matters, suppose that only four types of cars are manufactured, although in practice there could be hundreds. Each car type is defined by the options installed, as indicated in Table 1. In this example, the only available options are air conditioning and a sun roof. The table also shows how many cars of each type are required on a given day.

The problem is to sequence the car types so as to meet production requirements while observing the capacity constraints of the assembly line. The constraints are that at most three cars in any sequence of five can be given air conditioning, and at most one in any sequence of three can be given a sun roof. Figure 2 shows a feasible solution for a smaller instance of the problem.

A natural decision variable for this problem is the type t_i of car to assign to each position i in the sequence. To make sure that the production requirements are

Table 1 Options and production level required for each car type

Type	Air cond.	Sun roof	Production
a	No	No	20
b	Yes	No	15
c	No	Yes	8
d	Yes	Yes	7

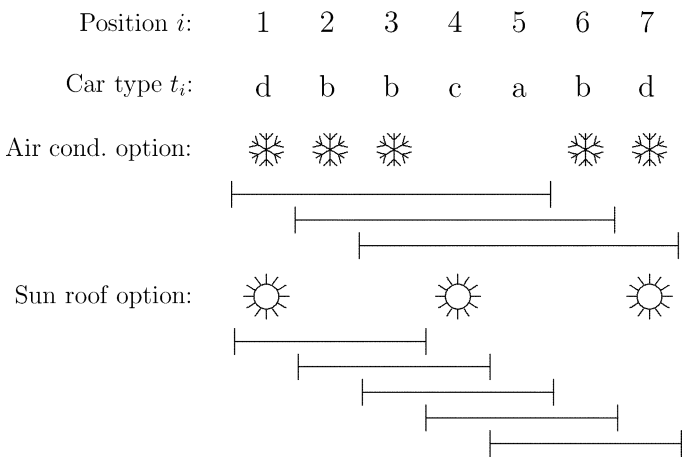


Fig. 2 A feasible solution for a small instance of the car sequencing problem in which the production requirements are $(D_a, D_b, D_c, D_d) = (1, 3, 1, 2)$. The brackets indicate subsequences in which assembly line capacity constraints are enforced

met, a constraint is needed that counts the number of times each type occurs in the sequence. The *cardinality* constraint, also known as the *generalized cardinality* or *gcc* constraint, serves the purpose [47, 48]. It is written `cardinality`$(X, \mathbf{v}, \mathbf{l}, \mathbf{u})$, where X is a set of variables, $\mathbf{v} = (v_1, \ldots, v_k)$ a tuple of values (not necessarily numerical), $\mathbf{l} = (\ell_1, \ldots, \ell_k)$ a tuple of lower bounds, and $\mathbf{u} = (u_1, \ldots, u_k)$ a tuple of upper bounds. The constraint says that, for each i, at least ℓ_i and at most u_i of the variables in X must take the value v_i. The production requirements for the car sequencing problem can be written with a single cardinality constraint,

$$\texttt{cardinality}\,(\{t_1, \ldots, t_{50}\}, (a, b, c, d), (20, 15, 8, 7), (20, 15, 8, 7)) \quad (3)$$

The `alldiff` constraint is a special case of a cardinality constraint in which each value is allowed to appear at most once. The `alldiff` constraints in the sudoku model (1) can therefore be replaced with cardinality constraints. It is best to follow Principle 2, however, by using the more specific `alldiff` constraint. Although filtering methods designed specifically for `alldiff` are likely to have the same result as `cardinality` filters (either typically achieves domain consistency), an `alldiff` filter generally runs faster.

The capacity constraints in the car sequencing problem limit the number of times each option occurs in subsequences of a specified length. However, the variables x_i indicate what type of car occurs in position i of the sequence, not which option. We know only that types b and d use the air conditioning option, and types c and d use the sun roof option. This requires a slightly different kind of counting than provided by the cardinality constraint. One must count the number of times type b or d occurs in a subsequence, and the number of times type c or d occurs. The *among* constraint was developed for such situations [10]. It can be written `among`(X, S, ℓ, u), where

X is a set of variables, S a set of values (not necessarily numerical), and ℓ and u are lower and upper bounds. The constraint requires that at least ℓ and at most u variables in X have a value that is among those in S.

Constraints of this kind can enforce the capacity constraints by applying them to every subsequence of five variables for air conditioners and every subsequence of three variables for sun roofs:

$$\begin{aligned}&\texttt{among}\left(\{t_j,\dots,t_{j+4}\},\{b,d\},0,3\right),\ \ j=1,\dots,46\\&\texttt{among}\left(\{t_j,t_{j+1},t_{j+2}\},\{c,d\},0,1\right),\ \ j=1,\dots,48\end{aligned}$$

Although this is a correct formulation, it fails to recognize that the among constraints are closely related. They apply to overlapping subsequences of variables of equal length. Faster and more effective filters can be designed if one exploits this structure (Principle 1). For this reason, scheduling formulations frequently use the constraint $\texttt{sequence}(\mathbf{x}, S, q, \ell, u)$, where $\mathbf{x} = (x_1,\dots,x_n)$ is a tuple of variables, S a set of values, and q an integer [10, 23, 38]. The constraint says that in any sequence of q consecutive variables $x_j,\dots,x_{j+q-1}$, at least ℓ and at most u variables must take a value in S. The capacity constraints can now be more compactly written as the second and third constraints in the complete car sequencing model that appears below.

$$\begin{aligned}&\texttt{cardinality}\left(\{t_1,\dots,t_{50}\},(a,b,c,d),(20,15,8,7),(20,15,8,7)\right)\\&\texttt{sequence}\left((t_1,\dots,t_{50}),\{b,d\},5,0,3\right)\\&\texttt{sequence}\left((t_1,\dots,t_{50}),\{c,d\},3,0,1\right)\\&t_i\in\{a,b,c,d\},\ \ i=1,\dots,50\end{aligned}\tag{4}$$

4.6 Example: Employee Scheduling

Employee scheduling is one of the most successful application areas for CP, and several well-studied global constraints have been developed for it. A small nurse scheduling problem, adapted from [16, 26, 49], illustrates dual formulations (Principle 4) and channeling constraints.

Four nurses are to be assigned to 8-h shifts. Shift 1 is the daytime shift, while shifts 2 and 3 occur at night. The schedule repeats itself every week. In addition,

1. Every shift is assigned exactly to one nurse.
2. Each nurse works at most one shift a day.
3. Each nurse works at least 5 days a week.
4. To ensure a certain amount of continuity, no shift can be staffed by more than two different nurses in a week.
5. To avoid excessive disruption of sleep patterns, a nurse cannot work different shifts on two consecutive days.
6. Also, a nurse who works shift 2 or 3 must do so at least 2 days in a row.

Table 2 Employee scheduling viewed as assigning workers to shifts. A feasible solution is shown

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Shift 1	A	B	A	A	A	A	A
Shift 2	C	C	C	B	B	B	B
Shift 3	D	D	D	D	C	C	D

Table 3 Employee scheduling viewed as assigning shifts to workers (shift 0 corresponds to a day off). A feasible solution is shown

	Sun	Mon	Tue	Wed	Thu	Fri	Sat
Worker A	1	0	1	1	1	1	1
Worker B	0	1	0	2	2	2	2
Worker C	2	2	2	0	3	3	0
Worker D	3	3	3	3	0	0	3

We can formulate the problem by assigning nurses to shifts or by assigning shifts to nurses. Rather than select one alternative, Principle 4 recommends using both. This not only sidesteps a difficult modeling decision but can result in faster solution than either model would permit if used in isolation. Table 2 displays a feasible assignment of nurses to shifts, and Table 3 displays the equivalent assignment of shifts to nurses.

We first write a model that assigns nurses to shifts. Let w_{sd} be the nurse assigned to shift s on day d, where the domain of w_{sd} is the set of nurses $\{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}\}$. Constraint 1 is satisfied automatically, by virtue of the notation. Constraint 2 says in effect that three different nurses work each day:

$$\texttt{alldiff}\,(w_{1d}, w_{2d}, w_{3d}), \quad d = 1, \ldots, 7 \tag{5}$$

Because there are 21 shifts in a week, constraint 3 implies that each nurse will work at least five and at most 6 days a week. This is readily expressed with a cardinality constraint:

$$\texttt{cardinality}\,(W, (\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}), (5, 5, 5, 5), (6, 6, 6, 6)) \tag{6}$$

where W is the set of variables w_{sd}.

Constraint 4 requires that for each shift s, at most two nurses are assigned to the variables $w_{s1}, \ldots, w_{s7}$ corresponding to the 7 days of the week. Thus, while constraint 2 counts the number of times a value occurs, constraint 3 counts the number of different values that occur. One therefore uses the $\texttt{nvalues}(X, \ell, u)$ global constraint [8, 11], which requires that the variables in X take at least ℓ and at most u different values.

$$\texttt{nvalues}\,(\{w_{s1}, \ldots, w_{s7}\}, 1, 2), \quad s = 1, 2, 3$$

Because `alldiff` is a special case of `nvalues` with $\ell = u = n$, one could have used `nvalues` rather than `alldiff` to express constraint 2. However, again following Principle 2, it is better to use the more specific constraint.

The remaining constraints are difficult to express in terms of variables w_{sd}, and there are no obvious global constraints that capture them. One can therefore move

to an alternative model in which variable t_{id} is the shift assigned to nurse i on day d, and where shift 0 denotes a day off. Constraint 1 is satisfied in this model by assigning different shifts to the nurses on each day:

$$\texttt{alldiff}\,(t_{\mathrm{A}d}, t_{\mathrm{B}d}, t_{\mathrm{C}d}, t_{\mathrm{D}d})\,, \quad d = 1, \ldots, 7$$

Constraint 2 is automatically satisfied by virtue of the notation. Constraint 3 says that each nurse gets at most 2 days off in a week (and at least one, because the other nurses get at most 2 days off):

$$\texttt{cardinality}\,(\{t_{i1}, \ldots, t_{i7}\}, 0, 1, 2)\,, \quad i = \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}$$

Thus, Constraints 1–3 are enforced in both models, a redundancy that can speed solution.

Constraint 4 is not easily expressed in terms of the variables t_{id}. However, these variables are suitable for Constraints 5 and 6, which refer to patterns of shifts that each nurse may work. This allows one to use the *stretch* constraint [13, 20, 45], written $\texttt{stretch}(x, v, \ell, u, P)$, where $x = (x_1, \ldots, x_n)$ and P is a set of *patterns*. The constraint requires, for each v_i, that any *stretch* of variables with value v_i in the sequence $x_1, \ldots, x_n$ has length at least ℓ_i and at most u_i. A stretch is a maximal subsequence of consecutive variables that take the same value. A pattern is a pair (v, v') of distinct values. The constraint requires that whenever a stretch of value v immediately precedes a stretch of value v', the pair (v, v') occurs in P. Constraints 5 and 6 can be written

$$\texttt{stretch-cycle}\,((t_{i1}, \ldots, t_{i7}), (2, 3), (2, 2), (6, 6), P)\,, \quad i = \mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}$$

where P consists of all patterns that include a day off

$$P = \{(s, 0), (0, s) \mid s = 1, 2, 3\}$$

One must use the cyclic version of `stretch` because the schedule is cyclic, and a stretch can extend across the weekend.

Constraints 5 and 6 can also be written as a `regular` constraint [46], which generalizes `stretch` and can often be processed at least as efficiently. A `regular` constraint models allowable sequences of values as expressions in a regular language, which can in turn be represented by a deterministic finite automaton. However, it is not straightforward to represent cyclic stretches with an automaton, due in part to the necessity of introducing additional variables. The more specific `stretch` constraint is therefore used in the present model.

Finally, the two models are linked with channeling constraints that relate the variables w_{sd} with the variables t_{id}:

$$\begin{aligned} w_{t_{id}d} &= i, \quad \text{all } i, d \\ t_{w_{sd}s} &= s, \quad \text{all } s, d \end{aligned} \tag{7}$$

The first constraint says that the nurse assigned to the shift to which nurse i is assigned on day d should be nurse i, and analogously for the second constraint.

For (7) to be valid, one must interpret w_{0d} as the nurse that is off duty on day d. When a problem instance requires more than one nurse to be off duty on a given day, it is necessary to define a dummy shift (representing a day off) for each nurse.

Note that variables occur as subscripts in the constraints (7). This powerful modeling device does not occur in MILP but is frequent in CP. A CP or integrated modeling system can parse an expression of the form x_t, where t is a variable with a finite domain, by replacing it with a new variable y and adding the constraint $\texttt{element}(t, (x_{v_1}, \ldots, x_{v_k}), y)$. The values $v_1, \ldots, v_k$ belong to the domain of t, and the `element` constraint requires that y take the same value as the variable in the list $x_{v_1}, \ldots, x_{v_k}$ whose subscript is the value of t. Thus, the modeling system interprets the constraints (7) as

$$\begin{aligned}
&y_{id} = i, \quad \text{all } i, d \\
&z_{sd} = s, \quad \text{all } s, d \\
&\texttt{element}\,(t_{id}, (w_{0d}, \ldots, w_{3d}), y_{id}), \quad \text{all } i, d \\
&\texttt{element}\,(w_{sd}, (t_{Ad}, \ldots, t_{Dd}), z_{sd}), \quad \text{all } s, d
\end{aligned}$$

Filtering methods have been developed for the `element` constraint as for any other global constraint, and they allow the channeling constraints (7) to improve propagation when the dual formulation is used. The dual formulation also speeds solution by allowing all the elements of the problem to be written as high-level global constraints (in one formulation or the other), which appears to be impossible if either formulation is used alone.

4.7 Assignment and Circuit Problems

Assignment and circuit problems illustrate several lessons in CP, MILP, and integrated modeling. The *assignment* problem can be viewed as one in which the objective is to find a minimum cost assignment of m tasks n to workers ($m \leq n$). Each task is assigned to a different worker, and no two workers are assigned the same task. If $m < n$, some of the workers will not be assigned tasks. If assigning worker i to task j incurs cost c_{ij}, the problem is simply stated:

$$\begin{aligned}
&\min \sum_i c_{ix_i} \\
&\texttt{alldiff}(x_1, \ldots, x_n), \quad x_i \in D_i, \quad i = 1, \ldots, n
\end{aligned} \tag{8}$$

where x_i is the worker assigned to task i. The domain D_i of x_i is the set of workers who can do task i.

The assignment problem can be viewed as a sequencing problem in which the cost depends on which item appears in each position of the sequence. The *circuit* problem, by contrast, is a sequencing problem in which the cost depends on which

item follows which in the sequence. The sequence is circular in the sense that the first item is viewed as following the last. The problem can be written

$$\begin{aligned} &\min \sum_i c_{x_i x_{i+1}} \\ &\texttt{alldiff}(x_1, \ldots, x_n), \quad x_i \in D_i, \quad i = 1, \ldots, n \end{aligned} \tag{9}$$

where $c_{n,n+1}$ is identified with c_{n1}. If each domain D_i is $\{1, \ldots, n\}$, the problem can be viewed as seeking a shortest hamiltonian circuit on a complete directed graph, where c_{ij} is the length of edge (i, j). This is the famous *traveling salesman problem*, in which the vertices are cities that the salesman must visit before returning to his home city. One can deal with edges (i, j) that are missing from the graph by setting $c_{ij} = \infty$. Unfortunately, filtering the `alldiff` has no effect because every domain is complete.

Although the two formulations (8) and (9) are almost identical, they represent vastly different problems. The assignment problem is very easy to solve, whereas the traveling salesman problem is notoriously hard.

An alternate formulation for the circuit problem uses the $\texttt{circuit}(y_1, \ldots, y_n)$ constraint [14, 32]. Here, y_i denotes the vertex after vertex i in the circuit, and the constraint requires that the y_is describe a hamiltonian circuit. The circuit problem can be written

$$\begin{aligned} &\min \sum_i c_{iy_i} \\ &\texttt{circuit}(y_1, \ldots, y_n), \quad y_i \in D'_i, \quad i = 1, \ldots, n \end{aligned} \tag{10}$$

This formulation has the advantage that missing edges can be explicitly represented in the domains. Domain D'_i of y_i contains j if and only if (i, j) is an edge of the graph. Filtering the `circuit` constraint can therefore have an effect. On the other hand, achieving domain consistency is much harder for `circuit` than for `alldiff` [26, 32].

Rather than choosing between formulations (9) and (10), Principle 4 recommends using both. The channeling constraints are $x_{i+1} = y_{x_i}$ for $i = 1, \ldots, n-1$, and $x_1 = y_{x_n}$. The "first" vertex in the circuit can be arbitrarily defined to be vertex 1 (i.e., $x_1 = 1$) without loss of generality. Propagation of incomplete domains D'_i through these constraints can remove elements from the domains D_i, so that `alldiff` filtering can now have an effect. Both objective functions can be used as bounds on cost, so that the dual formulation becomes

$$\begin{aligned} &\min \ z \\ &z \geq \sum_i c_{x_i x_{i+1}}, \quad z \geq \sum_i c_{iy_i} \\ &\texttt{alldiff}(x_1, \ldots, x_n), \quad x_i \in D_i, \quad i = 1, \ldots, n \\ &\texttt{circuit}(y_1, \ldots, y_n), \quad y_i \in D'_i, \quad i = 1, \ldots, n \\ &x_1 = y_{x_n} = 1, \quad x_{i+1} = y_{x_i}, \quad i = 1, \ldots, n-1 \\ &x_i \in \{1, \ldots, n\}, \quad y_i \in D_i, \quad i = 1, \ldots, n. \end{aligned} \tag{11}$$

5 MILP Modeling

An MILP model is an optimization problem in which the objective function is linear and the constraints are linear inequalities. Some or all of the variables are restricted to integer values. A major advantage of such a model is that it provides a ready-made continuous relaxation, obtained simply by dropping the integrality constraints. The relaxation is a linear programming problem whose solution provides a bound on the optimal value of the original problem. A "tighter" relaxation provides a bound that is closer to the optimal value. Such a bound can be very useful in a solution algorithm, for example, by pruning the search tree. In addition, the solution of the relaxation may be integral, in which case it solves the original MILP. Even when the solution is nonintegral, it may provide valuable clues on how to conduct a branching search.

Because integer-valued variables are present in an MILP model, its continuous relaxation can often be tightened by the addition of *cutting planes*. These are *valid* inequalities that are satisfied by all the feasible solutions of the MILP but "cut off" part of the feasible set of the continuous relaxation. MILP solution methods rely heavily on relaxation and cutting plane technology. Both general-purpose and special-purpose families of cutting planes have been developed, the latter for problems with special structure (see [39] for a survey).

A wide variety of problems can be given MILP models, although it is often not obvious how to do so, and the MILP formulation may require additional variables and constraints that obscure the underlying structure of the problem. Yet even when an MILP format is not appropriate for the original model, it may be advantageous for the solver to reformulate parts of the problem as an MILP to harness the power of the relaxation technology. The best strategy for using MILP formulations is therefore to write constraints in MILP format when it is natural to do from a modeling point of view and to allow the solver to reformulate other constraints as MILPs when this benefits the solution process. The present section focuses on how to write MILP formulations, while Sect. 6 explores the role of these formulations in integrated modeling.

The key to building MILP formulations is to recognize the structure of feasible sets that are representable by MILP models. It can be shown that MILP models are always equivalent to disjunctions of systems of knapsack inequalities. These lead to guidelines for writing models as disjunctions and for converting these to proper MILP models. Existing MILP modeling systems require the user to do the conversion by hand, but an integrated system would do so automatically.

This section therefore begins with knapsack modeling and then proceeds to show how knapsack modeling can be combined with disjunctive modeling to exploit the full resources of MILP problem formulation. Further examples and discussion of MILP modeling can be found in [19, 57–60].

The MILP modeling guidelines presented here omit some familiar modeling devices because they are subsumed by more general concepts of integrated modeling. These include special ordered sets, semi-continuous variables, and indicator constraints. These techniques are not actually part of MILP modeling but are extensions

that system developers have provided for convenience or computational efficiency. We will see that an integrated system provides the same capabilities, but in a more general and more principled way.

5.1 Knapsack Modeling

MILP formulations frequently involve counting ideas that can be expressed as *knapsack inequalities*. For present purposes, a knapsack inequality can be defined to be one of the form $ax \leq \beta$ (or $ax \geq \beta$), where some (or all) of the variables x_j may be restricted to integer values.

The term "knapsack inequality" derives from the fact that the *integer knapsack problem* can be formulated with such an inequality. The problem is to pack a knapsack with items that have the greatest possible value while not exceeding a maximum weight β. There are n types of items. Each item of type j has weight a_j and adds value c_j. If x_j is the number of items of type j put into the knapsack, the problem can be written

$$\begin{aligned} \max\ & cx \\ & ax \leq \beta \\ x_j \in \mathbb{Z},\ & \text{all } j \end{aligned} \tag{12}$$

A wide variety of modeling situations involve this same basic idea. A classic example is the capital budgeting problem, in which the objective is to allocate a limited amount of capital to projects so as to maximize revenue. Here, β is the amount of capital available. There are n types of projects, and each project j has initial cost a_j and earns revenue c_j. Variable x_j represents the number of projects of type j that are funded.

Typically, an MILP model contains a system of many knapsack inequalities. There may also be purely linear constraints in which all the variables are continuous. Some important special cases of knapsack systems include set packing, set covering, and set partitioning problems, as well as logical constraints.

Set packing. The set packing problem begins with a collection of finite sets S_j for $j = 1, \ldots, n$ that may partially overlap. It seeks a largest subcollection of sets that are pairwise disjoint.

Suppose, for example, that there are n surgeries to be performed, and the objective is to perform as many as possible this morning. Surgery j requires a specific set S_j of surgeons and other personnel. Because the surgeries must proceed in parallel, no two surgeries with overlapping personnel can be performed. This is a set packing problem.

The set packing problem can be formulated with 0–1 knapsack inequalities. Let $A_{ij}=1$ when item i belongs to set S_j, and $A_{ij} = 0$ otherwise. Let variable $x_j=1$ when set j is selected. The knapsack inequality $\sum_{j=1}^{n} A_{ij}x_{ij} \leq 1$ prevents the

selection of any two sets containing item i. Thus, the system $Ax \leq e$ of knapsack inequalities, where e is a vector of ones, prevents the selection of any two overlapping sets. The objective is to maximize $\sum_{j=1}^{n} x_j$ subject to $Ax \leq e$ and $x \in \{0,1\}^n$, which is an MILP problem.

Set covering. The set covering problem likewise begins with a collection of sets S_j but seeks the minimum subcollection that contains all the elements in the union of the sets. For example, one may wish to buy a minimum collection of songbooks that contains all the songs that appear in at least one book. Here, S_j is the set of songs in book j.

If A_{ij} and x_j are as before, the knapsack inequality $\sum_{j=1}^{n} A_{ij}x_j \geq 1$ ensures that item i is covered. The set covering problem is to minimize $\sum_{j=1}^{n} x_j$ subject to $Ax \geq e$ and $x \in \{0,1\}^n$. The objective function in this or the set packing problem can be generalized to cx by attaching a weight c_j to each set S_j, to represent the cost or benefit of selecting S_j.

Set partitioning. The set partitioning problem seeks a subcollection of sets such that each element is contained in exactly one of the sets selected. The constraints are therefore $Ax = e$, which are a combination of the knapsack constraints $Ax \leq e$ and $Ax \geq e$. The problem is to minimize or maximize cx subject to these constraints.

An important practical example of set partitioning is the airline crew rostering problem. Crews must be assigned to sequences of flight legs while observing complicated work rules. For example, there are restrictions on the number of flight legs a crew may staff in one assignment, the total duration of the assignment, the layover time between flight legs, and the locations of the origin and destination.

Let S_j be a set of flight legs that can be assigned to a single crew, where j indexes all possible such sets. A set S_j is selected ($x_j = 1$) when it is assigned to a crew, incurring cost c_j. This is a partitioning problem because each flight leg must be staffed by exactly one crew and must therefore appear in exactly one selected S_j. Although there may be millions of sets S_j and therefore millions of variables x_j, the model can be quite practical when its continous relaxation is solved by a column generation method; that is, by adding variables (and the associated columns of A) to the problem only when they can improve the solution. Typically, only a tiny fraction of the columns are generated.

Clique Inequalities. A collection of set packing constraints in a model can sometimes be replaced or supplemented by a *clique inequality*, which substantially tightens the continuous relaxation. For example, the 0–1 inequalities

$$\begin{aligned} x_1 + x_2 &\leq 1 \\ x_1 + x_3 &\leq 1 \\ x_2 + x_3 &\leq 1 \end{aligned} \tag{13}$$

are equivalent to the clique inequality $x_1 + x_2 + x_3 \leq 1$. One can see that the clique inequality provides a tighter relaxation from the fact that $x_1 = x_2 = x_3 = 1/2$ violates it but satisfies (13). It should therefore replace (13) in the model.

To generalize this idea, define a graph whose vertices j correspond to 0–1 variables x_j. The graph contains an edge (i, j) whenever $x_j + x_j \leq 1$ is implied by a constraint in the model. If the induced subgraph on some subset C of vertices is a clique, then $\sum_{j \in C} \leq 1$ is a valid inequality and can be added to the model.

Logical Conditions. Logical conditions on 0–1 variables can be formulated as knapsack inequalities that are similar to set covering constraints. Suppose, for example, that either plants 2 and 3 must be built, or else plant 1 must not be built. For the moment, regard x_j as a boolean variable that is true when plant j is built, and false otherwise. The condition can be written

$$\neg x_1 \vee x_2 \vee x_3 \tag{14}$$

where $\vee$ means "or" and $\neg$ means "not." Such a condition is a *logical clause*, meaning that it is a disjunction of *literals* (boolean variables or their negations). Because the clause states that at least one of the literals must be true, it can be written as an inequality $(1 - x_1) + x_2 + x_3 \geq 1$ by viewing x_j as true when $x_j = 1$ and false when $x_j = 0$. This is equivalent to the knapsack inequality $-x_1 + x_2 + x_3 \geq 0$.

Any logical condition built from "and," ($\wedge$) "or," "not," and "if" can be converted to a set of clauses and given an MILP model on that basis. "B if A" is written $A \Rightarrow B$ and is equivalent to $\neg A \vee B$. Consider, for example, the condition, "If plants 1 and 2 are built, then plants 3 and 4 must be built." It can be written

$$(x_1 \wedge x_2) \Rightarrow (x_3 \wedge x_4)$$

The implication can be eliminated to obtain $\neg(x_1 \wedge x_2) \vee (x_3 \wedge x_4)$. The negation is then brought inside using De Morgan's Law, resulting in the expression $\neg x_1 \vee \neg x_2 \vee (x_3 \wedge x_4)$. The conjunction is now distributed:

$$(\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_4)$$

This conjunction of two clauses can be written as two knapsack constraints:

$$\begin{aligned} -x_1 - x_2 + x_3 &\geq -1 \\ -x_1 - x_2 + x_4 &\geq -1 \end{aligned}$$

5.2 MILP Representability

MILP modeling achieves its full power when the model is allowed to contain auxiliary variables in addition to the variables of the original problem space, because this allows the model to represent disjunctions of discrete alternatives. The auxiliary variables can be either continuous or discrete and frequently appear in practical models. It is known precisely what kind of problems can be represented with MILP models in this way, due to theorems proved in [31] and generalized in [28].

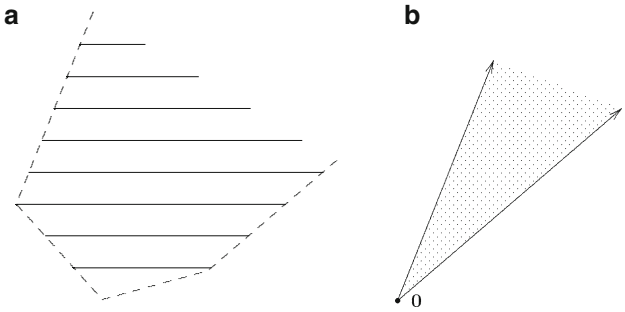


Fig. 3 (**a**) A mixed integer polyhedron Q (*horizontal lines*) where $Q = P \cap (\mathbb{R} \times \mathbb{Z})$ and P is bounded by the *dashed line*. (**b**) The recession cone of Q

Formally, a subset S of $\mathbb{R}^{n+p}$ is *MILP representable* if it is the projection onto x of the feasible set of a model of the form

$$Ax + Bu + D\delta \geq b$$
$$x \in \mathbb{R}^n \times \mathbb{Z}^p,\ u \in \mathbb{R}^m,\ \delta \in \{0,1\}^q$$

Some of the auxiliary variables are real-valued (u_j) and some are binary (δ_j).

To state the representability theorems, some definitions are necessary. Let a *mixed integer polyhedron* be the nonempty intersection of any polyhedron in $\mathbb{R}^{n+p}$ with $\mathbb{R}^n \times \mathbb{Z}^p$. Such a polyhedron is illustrated in Fig. 3. A vector $r \in \mathbb{R}^{n+p}$ is a *recession direction* of a polyhedron $P \in \mathbb{R}^{n+p}$ if one can go forever in the direction r without leaving P. That is, for any $x \in P$, $x + \alpha r \in P$ for all $\alpha \geq 0$. A rational vector r is a recession direction of a mixed integer polyhedron Q if it is a recession direction of a polyhedron whose intersection with $\mathbb{R}^n \times \mathbb{Z}^p$ is Q. The *recession cone* of a mixed integer polyhedron is the set of all its recession directions (Fig. 3).

Theorem 1. *A nonempty set $S \in \mathbb{R}^n \times \mathbb{Z}^p$ is MILP representable if and only if it is the union of finitely many mixed integer polyhedra in $\mathbb{R}^n \times \mathbb{Z}^p$ having the same recession cone.*

Each mixed integer polyhedron Q_k in the finite union can be described by a knapsack system. The theorem therefore states in effect that any MILP representable set can be modeled as a disjunction of knapsack systems (i.e., at least one of the systems must be satisfied). If $Q_k = \{x \in \mathbb{R}^n \times \mathbb{Z}^p \mid A^k x \geq b^k\}$, the disjunction is

$$\bigvee_{k \in K} \left(A^k x \geq b^k \right), \quad x \in \mathbb{R}^n \times \mathbb{Z}^p \tag{15}$$

This suggests a principle for creating MILP models, because practical problems frequently present a set of alternative actions or situations. If the alternatives can be modeled by knapsack systems, then the problem takes the form of a disjunction of such systems. An MILP model can now be written, provided the knapsack systems

describe mixed integer polyhedra with the same recession cone. The recession cone condition is normally satisfied in practice by adding a few innocuous constraints.

There remains the question, however, as how to convert the disjunction (15) of knapsack systems into an MILP model. There are two standard ways: a big-M formulation and a convex hull formulation. Both require 0–1 auxiliary variables, and the convex hull formulation requires continuous auxiliary variables as well.

First, the big-M formulation. Consider any nonempty MILP representable set $S \in \mathbb{R}^n \times \mathbb{Z}^p$. From Theorem 1, S is the union of mixed integer polyhedra Q_k defined earlier, where the Q_ks have the same recession cone. Then, S has the *sharp big-M formulation*

$$\begin{aligned} &A^k x \geq b^k - M^k(1-\delta_k), \quad k \in K \\ &x \in \mathbb{R}^n \times \mathbb{Z}^p, \quad \delta_k \in \{0,1\}, \quad k \in K \end{aligned} \tag{16}$$

where

$$M^k = b^k - \min_{\ell \neq k} \left\{ \min_x \left\{ A^k x \;\middle|\; A^\ell x \geq b^\ell,\ x \in \mathbb{R}^n \times \mathbb{Z}^p \right\} \right\} \tag{17}$$

Note that there are binary auxiliary variables δ_k. When $\delta_k = 1$, the kth knapsack system $A^k x \geq b^k$ is enforced. When $\delta_k = 0$, the kth system is deactivated by subtracting a vector M^k of large numbers from the right-hand side. The formulation is sharp in the sense that the components of M^k are chosen to be as small as possible, but it remains a valid formulation if larger values of M^k are used. If Q_k is the same as before,

Theorem 2. *A set $S \in \mathbb{R}^n \times \mathbb{Z}^p$ is MILP representable if and only if it has a sharp big-M representation (16), where $S = \bigcup_{k \in K} Q_k$.*

The convex hull formulation of (15) introduces continuous auxiliary variables by disaggregating x into a sum $\sum_{k \in K} x^k$, where each x^k corresponds to Q_k. The formulation is

$$\begin{aligned} &x = \sum_{k \in K} x^k \\ &A^k x^k \geq b^k \delta_k, \quad k \in K \\ &\sum_{k \in K} \delta_k = 1 \\ &x \in \mathbb{R}^n \times \mathbb{Z}^p, \quad \delta_k \in \{0,1\}, \quad k \in K \end{aligned} \tag{18}$$

Theorem 3. *A set $S \in \mathbb{R}^n \times \mathbb{Z}^p$ is MILP representable if and only if it can be formulated as (18), where $S = \bigcup_{k \in K} Q_k$. Furthermore, (18) is a convex hull formulation if $A^k \geq b^k$ is a convex hull formulation of Q_k for each $k \in K$.*

Model (18) is a *convex hull formulation* of S when its continuous relaxation describes the closure of the convex hull of S. The continuous relaxation is obtained by dropping the integrality constraints; that is, by replacing the last line of (18) with

$$x \in \mathbb{R}^{n+p}, \quad 0 \le \delta_k \le 1, \quad k \in K$$

The system $A^k x \ge b^k$ is a convex hull formulation of Q_k when it describes the closure of the convex hull of Q_k. A convex hull formulation has the tightest possible continuous relaxation.

It can be shown that (18) provides a convex hull relaxation of S when the polyhedra Q_k do not have the same recession cone, even though (18) is not a representation of S in this case. That is,

Theorem 4. *The continuous relaxation of (18), when projected onto x, describes the closure of the convex hull of* $S = \bigcup_{k \in K} Q_k$ *even when the* Q_k*s do not have the same recession cone.*

Similarly, the big-M formulation (16) provides a valid relaxation of S even when the polyhedra Q_k do not have the same recession cone.

In some modeling contexts, it is useful to associate user-defined 0–1 variables δ_k explicitly with each term k of a disjunction by writing (15) as

$$\bigvee_{k \in K} \left(A^k x \le b^k \right)_{\delta_k} \tag{19}$$

The variables δ_k become the 0–1 auxiliary variables in the MILP formulation of the disjunction and are available for use elsewhere in the problem as well. If the solver branches on the disjunction, it sets $\delta_k = 1$ when the kth disjunct is enforced. It enforces term k of the disjunction when δ_k is fixed to one and removes term k from the disjunction when $\delta_k = 0$.

5.3 Example: Fixed-Charge Problems

A simple fixed-charge problem illustrates MILP model construction. The cost z of manufacturing quantity x of some product is zero when $x = 0$ and is $f + cx$ when $x > 0$, where f is the fixed cost and c the unit variable cost. The problem is to minimize cost.

There are two alternatives, corresponding to a zero or positive production level, each giving rise to a different cost calculation. The feasible set S is illustrated in Fig. 4. It is described by a disjunction of linear systems that, in this case, contain only continuous variables:

$$\begin{pmatrix} x = 0 \\ z \ge 0 \end{pmatrix} \vee \begin{pmatrix} x \ge 0 \\ z \ge cx + f \end{pmatrix} \tag{20}$$

The disjuncts respectively describe the polyhedra P_1 and P_2 in Fig. 4a. The recession cone of P_1 is P_1 itself, and the recession cone of P_2 is $\{(z, x) \mid z \ge cx \ge 0\}$.

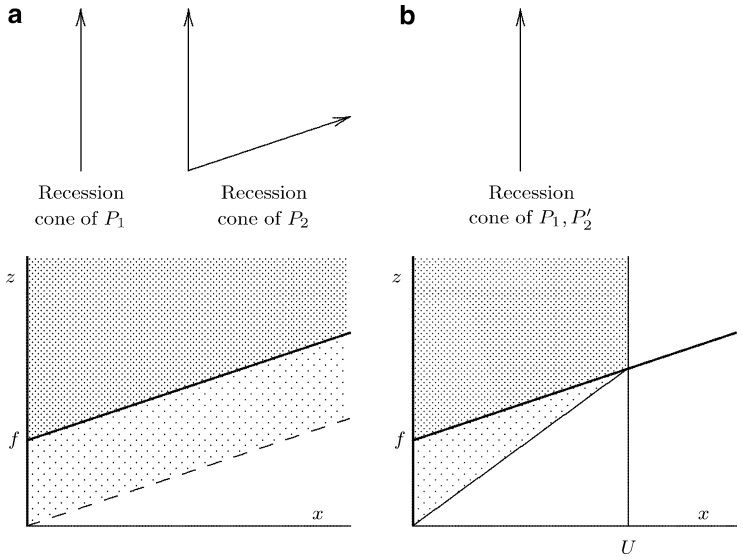


Fig. 4 (**a**) Feasible set of a fixed-charge problem, consisting of the union of polyhedra P_1 (*heavy vertical line*) and P_2 (*darker shaded area*). (**b**) Feasible set of the same problem with the bound $x \le U$, where P_2' is the *darker shaded area*. In both (**a**) and (**b**), the convex hull of the feasible set is the entire *shaded area*

Thus, by Theorem 1, S is not MILP representable. For example, one can write the big-M formulation (17) as

$$\begin{aligned} &0 \le x \le M_1^1 \delta && x \ge 0 \\ &z \ge -M_2^1 \delta && z \ge cx + f - M_2^2(1-\delta) \\ &\delta \in \{0,1\} \end{aligned}$$

where the 0–1 variables δ_1, δ_2 are replaced by $1-\delta$ and δ (because they sum to one). But this formulation is not well defined because from (17),

$$M_1^1 = -\min\{-x \mid x \ge 0,\ \ z \ge cx + f\} = \infty$$

Also the convex hull formulation (18) of (20) becomes

$$\begin{aligned} &x = x^1 + x^2 && z = z^1 + z^2 \\ &x^1 = 0 && x^2 \ge 0 \\ &z^1 \ge 0 && z^2 \ge cx^2 + f\delta \\ &\delta \in \{0,1\} \end{aligned} \tag{21}$$

One can eliminate the constraint $x^1 = 0$ (and the corresponding aggregation constraint $x = x^1 + x^2$) by replacing x^2 with x, and similarly for the constraint $z^1 \geq 0$. So (21) simplifies to

$$x \geq 0, \quad z \geq cx + f\delta, \quad \delta \in \{0, 1\}$$

which does not correctly represent the feasible set. However, its continuous relaxation correctly describes the closure of the convex hull of the feasible set, as predicted by Theorem 4. For if one replaces $\delta \in \{0, 1\}$ with $0 \leq \delta \leq 1$ and projects out δ, the result is $z \geq cx$, $x \geq 0$. This is illustrated in Fig. 4a.

The recession cones can be equalized by placing an upper bound U on x in the second disjunct of (20). The recession cone of each of the resulting polyhedra P_1, P_2' is the same, as illustrated in Fig. 4b, and the feasible set is therefore MILP representable. The big-M formulation becomes

$$\begin{aligned} &0 \leq x \leq M_1^1\delta && 0 \leq x \leq U + M_1^2(1-\delta) \\ &z \geq -M_2^1\delta && z \geq cx + f - M_2^2(1-\delta) \\ &\delta \in \{0, 1\} \end{aligned} \tag{22}$$

From (17), $(M_1^1, M_2^1, M_1^2, M_2^2) = (U, -f, 0, f)$. So (22) simplifies to

$$0 \leq x \leq U\delta, \quad z \geq cx + f\delta, \quad \delta \in \{0, 1\} \tag{23}$$

which is a correct formulation. The convex hull formulation becomes

$$\begin{aligned} &x = x^1 + x^2 && z = z^1 + z^2 \\ &0 = x^1 && 0 \leq x^2 \leq U\delta \\ &z^1 \geq 0 && z^2 \geq cx^2 + f\delta \\ &\delta \in \{0, 1\} \end{aligned}$$

which again simplifies to (23). In this case, the big-M formulation happens to be a convex hull formulation.

5.4 MILP Modeling Guidelines

Theorems 1–3 imply that an MILP model can be formulated by regarding the problem as a disjunction of knapsack systems corresponding to discrete alternatives. The disjunction is then converted to a big-M or convex hull formulation. In practice, it may be convenient to identify several disjunctions, each of which is converted to a set of MILP constraints. This suggests the following guidelines for MILP modeling:

1. Try to conceive the problem as posing one or more choices among discrete alternatives.
2. Formulate each alternative using a system of knapsack inequalities.

3. Write each choice of alternatives as a disjunction of knapsack systems. Some of the disjunctions may have only one disjunct, indicating that there is only one alternative.

At this point, the disjunctions are converted to MILP formulations as follows. The conversion would be automatic in an integrated modeling system.

4. Adjust the linear systems so that in each disjunction, all of the disjuncts describe mixed integer polyhedra with the same recession cone. This can be done manually as well; the options are discussed in Sect. 6.2.
5. Convert each disjunction to a big-M or a convex hull formulation. The convex hull formulation normally has a tighter continuous relaxation, unless the big-M model happens to be a convex hull formulation as well. The big-M formulation normally contains fewer variables, particularly when there are many disjuncts, unless the convex hull formulation can be simplified. The choice between the formulations rides on whether the tighter relaxation is worth the overhead of additional variables.

Not all useful MILP models evolve naturally from this disjunctive approach. An example is the standard 0–1 model for the circuit (traveling salesman) problem, discussed below.

5.5 Example: Facility Location

A capacitated facility location problem illustrates the above modeling guidelines. There are m possible locations for factories, and n customers who obtain products from the factories. A factory installed at location i incurs fixed cost f_i and has capacity C_i. Each customer j has demand D_j. Goods are shipped from factory i to customer j on trucks, each with capacity K_{ij}, and each incurring a fixed cost c_{ij}. The problem is to decide which facilities to install, and how to supply the customers, so as to minimize total cost (Fig. 5).

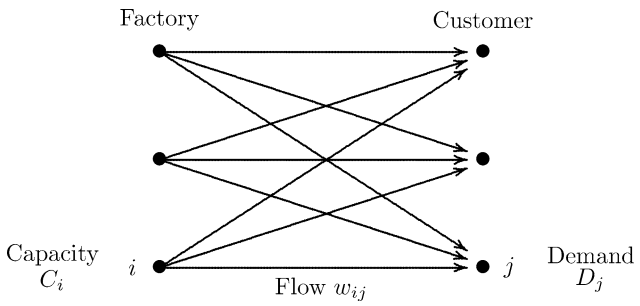


Fig. 5 A facility location problem

The basic decision, for each location i, is whether to install a factory at that location. This presents two discrete alternatives that can be represented as a disjunction. To describe each alternative with knapsack systems, let x_{ij} be the quantity of goods shipped from factory i to customer j, and let w_{ij} be the number of trucks on which they are transported. Then, if z_i is the total cost incurred at location i, the two alternatives for location i are represented by the disjunction

$$\begin{pmatrix} \sum_j x_{ij} \le C_i \\ 0 \le x_{ij} \le K_{ij} w_{ij}, \text{ all } j \\ z_i \ge f_i + \sum_j c_{ij} w_{ij} \\ w_{ij} \in \mathbb{Z}, \text{ all } j \end{pmatrix} \vee \begin{pmatrix} x_{ij} = 0, \text{ all } j \\ z_i \ge 0 \end{pmatrix} \tag{24}$$

The alternative on the left corresponds to installing a factory at location i. The first constraint enforces the factory's capacity limit, and the second does the same for the truck capacities. The third constraint computes the cost incurred at location i. Note that each w_{ij} is integer valued, which means that this disjunct describes a mixed integer polyhedron. The disjunct on the right corresponds to the case in which no factory is installed at location i.

Customer demand can be satisfied by imposing the constraint $\sum_i x_{ij} \ge D_j$ for each customer j. Each of these constraints can be viewed as a separate disjunction with only one alternative. The objective is to minimize total cost, given by $\sum_i z_i$.

This completes the modeler's task. At this point the modeling system takes over by converting (24) to an MILP formulation. It must first ensure that the two disjuncts in (24) have the same recession cone. As it happens, they do not. The cone for the first polyhedron is $\{(x_i, w_i, z_i) \mid x_i = 0,\ w_i \ge 0,\ z_i \ge \sum_j c_{ij} w_{ij}\}$ where $x_i = (x_{i1}, \ldots, x_{in})$ and $w_i = (w_{i1}, \ldots, w_{in})$, while the cone for the second is $\{(x_i, w_i, z_i) \mid x_i = 0,\ z_i \ge 0\}$. The cones can, in theory, be equalized if the innocuous constraints $w_{ij} \ge 0$ and $z_i \ge \sum_j c_{ij} w_{ij}$ are added to the second disjunct. This yields a disjunction that can be given an MILP model:

$$\begin{pmatrix} \sum_j x_{ij} \le C_i \\ 0 \le x_{ij} \le K_{ij} w_{ij}, \text{ all } j \\ z_i \ge f_i + \sum_j c_{ij} w_{ij} \\ w_{ij} \in \mathbb{Z}, \text{ all } j \end{pmatrix} \vee \begin{pmatrix} x_{ij} = 0, \text{ all } j \\ w_{ij} \ge 0, \text{ all } j \\ z_i \ge \sum_j c_{ij} w_{ij} \end{pmatrix} \tag{25}$$

Using (18), the convex hull formulation of (25) is

$$\begin{aligned}
&x_{ij} = x^1_{ij} + x^2_{ij}, \;\; w_{ij} = w^1_{ij} + w^2_{ij}, \;\; z_i = z^1_i + z^2_i, \;\; \text{all } j \\
&\sum_j x^1_{ij} \le C_i \delta_i && x^2_{ij} = 0, \; w^2_{ij} \ge 0 \;\; \text{all } j \\
&0 \le x^1_{ij} \le K_{ij} w^1_{ij}, \;\; \text{all } j && z^2_i \ge \sum_j c_{ij} w^2_{ij} \\
&z^1_i \ge f_i \delta_i + \textstyle\sum_j c_{ij} w^1_{ij} && \delta_i \in \{0,1\}, \;\; w_{ij} \in \mathbb{Z}, \;\; \text{all } j
\end{aligned}$$

Because the auxiliary 0–1 variables corresponding to the two disjuncts sum to one, they can be written as δ_i and $1-\delta_i$; the latter does not appear because the right-hand sides in the second disjunct are all zero. The constraints $x^2_{ij} = 0$ can be dropped (along with the aggregation constraints $x_{ij} = x^1_{ij} + x^2_{ij}$) if x^1_{ij} is replaced by x_{ij}, and similarly for the constraints $w^2_{ij} \ge 0$ and $z^2_i \ge \sum_j c_{ij} w^2_{ij}$. Because z_i can be replaced by $f_i \delta_i + \sum_j c_{ij} w_{ij}$ in the objective function $\sum_j z_j$, the complete MILP model becomes

$$\begin{aligned}
&\min \sum_i \left(f_i \delta_i + \sum_j c_{ij} w_{ij} \right) \\
&\sum_j x_{ij} \le C_i \delta_i, \;\; \text{all } i \\
&0 \le x_{ij} \le K_{ij} w_{ij}, \;\; \text{all } i, j \\
&\sum_i x_{ij} \ge D_j, \;\; \text{all } j \\
&\delta_i \in \{0,1\}, \;\; w_{ij} \in \mathbb{Z}, \;\; \text{all } i, j
\end{aligned} \tag{26}$$

Although each disjunction (25) is given a convex hull formulation in the MILP model (26), the model as a whole is not a convex hull formulation of the problem.

Using (16), the big-M model for the disjunction (25) is

$$\begin{aligned}
&\sum_j x_{ij} \le C_i + M^1_{1i}(1-\delta_i) && 0 \le x_{ij} \le M^2_{1ij} \delta_i, \;\; \text{all } i, j \\
&0 \le x_{ij} \le K_{ij} w_{ij} + M^1_{2ij}(1-\delta_i), \;\; \text{all } j && w_{ij} \ge -M^2_{2ij} \delta_i \;\; \text{all } i, j \\
&z_i \ge f_i \delta_i + \textstyle\sum_j c_{ij} w_{ij} - M^1_{3i}(1-\delta_i) && z^2_i \ge \sum_j c_{ij} w_{ij} - M^2_{3i} \delta_i, \;\; \text{all } i \\
&\delta_i \in \{0,1\}, \;\; w_{ij} \in \mathbb{Z}, \;\; \text{all } j
\end{aligned} \tag{27}$$

It can be verified from (17) that $M^1_{1i} = -C_i$, $M^2_{1ij} = C_i$, and all the other big-Ms are zero in the sharp formulation. The big-M formulation (27) therefore reduces to the same model as the convex hull formulation.

It is unclear how the disjunction (25) could be obtained automatically, and perhaps unrealistic to expect the user to equalize the recession cones in this way

manually. A more practical alternative is for the modeling system to equalize the recession cones by imposing reasonable lower and upper bounds on every variable, or to ask the user to provide bounds.

5.6 Examples: Piecewise Linear and Indicator Constraints

Piecewise linear and indicator constraints were not discussed in the section on CP but illustrate some important points for MILP modeling.

Piecewise linear functions are a very useful modeling tool because they provide a means to approximate separable nonlinear functions within a linear modeling framework. Typically, we wish to model a function of the form $g(x) = \sum_j g_j(x_j)$ by approximating each nonlinear term $g_j(x_j)$ with a piecewise linear function $f_j(x_j)$. The function $f_j(x_j)$ is set to a value equal to (or close to) $g_j(x_j)$ at a finite number of values of x_j and is defined between these points by a linear interpolation.

For convenience, we drop the subscripts and refer to $f_j(x_j)$ as $f(x)$. We suppose that $f(x)$ is piecewise linear in the general sense that it is linear on possibly disjoint intervals $[a_i, b_i]$ and undefined outside these intervals, as illustrated by Fig. 6. More precisely, $x \in \bigcup_{i \in I} [a_i, b_i]$ and

$$f(x) = \begin{cases} f(a_i) + \dfrac{x - a_i}{b_i - a_i}\,[f(b_i) - f(a_i)] & \text{if } x \in [a_i, b_i] \text{ and } a_i < b_i \\ f(a_i) & \text{if } x = a_i = b_i \end{cases} \tag{28}$$

In many applications, each $b_i = a_{i+1}$, which means $f(x)$ is continuous.

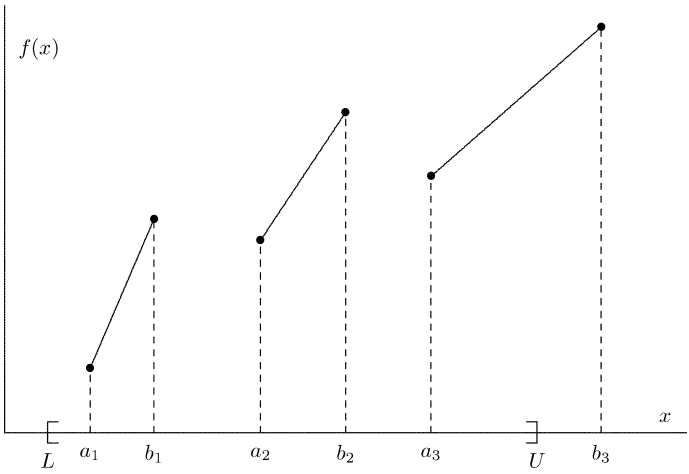


Fig. 6 A piecewise linear function. The domain of x is $[L, U]$

A disjunctive formulation is natural and convenient for a function of this form:

$$\bigvee_{i\in I}\begin{bmatrix} x = \lambda a_i + \mu b_i \\ z = \lambda f(a_i) + \mu f(b_i) \\ \lambda + \mu = 1,\ \ \lambda, \mu \geq 0 \end{bmatrix} \tag{29}$$

where disjunct i corresponds to $x \in [a_i, b_i]$, and z is a variable that represents $f(x)$.

Because the disjuncts of (29) define polyhedra with the same recession cone (all the polyhedra are bounded), the following convex hull formulation can be automatically generated:

$$\begin{aligned} & x = \sum_{i\in I} \lambda_i a_i + \mu_i b_i \\ & z = \sum_{i\in I} \lambda_i f(a_i) + \mu_i f(b_i) \\ & \lambda_i + \mu_i = \delta_i,\ i \in I \\ & \sum_{i\in I} \delta_i = 1 \\ & \lambda_i, \mu_i \geq 0,\ \delta_i \in \{0,1\},\ i \in I \end{aligned} \tag{30}$$

This is similar to a well-known textbook model that dispenses with the multipliers μ_i but applies only when $f(x)$ is continuous [34]:

$$\begin{aligned} & x = \sum_{i=1}^{k+1} \lambda_i a_i,\ \ z = \sum_{i=1}^{k+1} \lambda_i f(a_i),\ \ \sum_{i=1}^{k+1} \lambda_i = 1 \\ & \lambda_i \leq \delta_{i-1} + \delta_i,\ i = 2, \ldots, k \\ & \lambda_1 \leq \delta_1,\ \ \lambda_{k+1} \leq \delta_k,\ \sum_{i=1}^{k} \delta_i = 1 \\ & \lambda_i \geq 0,\ i = 1, \ldots, k+1;\ \ \delta_i \in \{0,1\},\ i = 1, \ldots, k \end{aligned} \tag{31}$$

where $a_{k+1} = b_k$. This model, however, is not as tight as (30). Moreover, (30) is "locally ideal," meaning that the 0–1 variables take integer values at all the vertices of the polyhedron described by the continuous relaxation [44]. Apparently, model (30) was unrecognized in the literature until described by Sherali [51] in 2001, but it is an immediate result of the disjunctive MILP formulation. Although Sherali proves that (30) is locally ideal, no proof is necessary, because any convex hull formulation of a disjunction is locally ideal. Model (30) can also be adapted to the case in which $f(x)$ is lower semi-continuous, as noted in [51].

For continuous functions $f(x)$, one can use the *incremental cost model*, which contains no more variables than (31), but is equivalent to the tight model (30) and locally ideal [51]:

$$\begin{aligned} & x = a_1 + \sum_{i=2}^{k+1} x_i,\ \ z = f(a_1) + \sum_{i=2}^{k+1} \frac{f_i'}{a_i'} x_i \\ & 0 \leq x_i \leq a_i',\ i = 2, \ldots, k+1 \end{aligned}$$

$$
\begin{aligned}
& a_i' \delta_i \le x_i \le a_i' \delta_{i-1}, \ i = 2, \ldots, k \\
& a_2' \delta_2 \le x_2 \le a_2', \ \ 0 \le x_{k+1} \le a_{k+1}' \delta_k \\
& \delta_i \in \{0, 1\}, \ i = 2, \ldots, k
\end{aligned} \tag{32}
$$

Here $a_i' = a_i - a_{i-1}$ and $f_i' = f(a_i) - f(a_{i-1})$.

Most MILP solvers allow one to model a continuous piecewise linear function by defining the multipliers λ_i in the textbook model (31) to be a *special ordered set of type 2* (SOS2). In this case, one need only to write the constraints on the first line of (31). The SOS2 condition requires that at most two of the variables λ_i be nonzero, where any two nonzero variables must be adjacent (i.e., λ_i and λ_{i+1} for some i). The condition is enforced directly by the branching mechanism. It simplifies the model by eliminating the 0–1 variables δ_i, but it sacrifices the tight continuous relaxation provided by the 0–1 model (30) or (32).

An integrated solver can implement SOS2 branching simply by branching on the terms of the disjunction (29) in the normal fashion. This has the effect of permitting only adjacent multipliers to be nonzero. In fact, this disjunctive approach is more general than SOS2 because it is not restricted to continuous functions. If desired, one can dispense with the 0–1 formulation (30) simply by instructing the solver not to generate a relaxation for the disjunction. Thus, there is no need for a separate SOS2 option in the modeling system.

There are recent proposals for modeling piecewise linear functions with a logarithmic number of 0–1 variables [36, 56]. However, we will see that in an integrated modeling context, piecewise linear functions can be efficiently modeled, and a relaxation provided, without the use of any auxiliary variables or special ordered sets.

Indicator constraints are constraints that are enforced only when a 0–1 variable is equal to one (or equal to zero). They, too, are naturally expressed in disjunctive form, and there is no need for a modeling system to offer this feature separately.

Suppose, for example, that we wish to enforce the system $Ax \ge b$ only when $\delta = 1$. The advantage of having an indicator constraint option in a modeling system is that it obviates the use of a big-M construction like

$$Ax \ge b - M(1 - \delta)$$

Yet one can achieve the same purpose by writing the disjunction

$$(Ax \ge b)_\delta$$

The second disjunct, corresponding to $\delta = 0$, is understood to be empty because it does not appear. The system will enforce $Ax \ge b$ when $\delta = 1$, as desired.

A *semi-continuous* variable x is a related idea in which x is forced to zero when $\delta = 0$ and to be within bounds $L \le x \le U$ when $\delta = 1$. One can define x to be semi-continuous by writing $(L \le x \le U)_\delta \vee (x = 0)$.

5.7 Example: Car Sequencing

It will be useful to compare an MILP model of the car sequencing problem with the CP model developed earlier. In this problem, there are four discrete alternatives at each position i in the manufacturing sequence–car types a, b, c, and d. Each alternative implies a choice of options. If we let $AC_i = 1$ when air conditioning is installed at position i, and $SR_i = 1$ when a sun roof is installed, the four alternatives can be written as follows for each position i:

$$\begin{pmatrix} AC_i = 0 \\ SR_i = 0 \end{pmatrix} \vee \begin{pmatrix} AC_i = 1 \\ SR_i = 0 \end{pmatrix} \vee \begin{pmatrix} AC_i = 0 \\ SR_i = 1 \end{pmatrix} \vee \begin{pmatrix} AC_i = 1 \\ SR_i = 1 \end{pmatrix}$$

The convex hull formulation of this disjunction is

$$\begin{aligned}
&AC_i = AC_i^a + AC_i^b + AC_i^c + AC_i^d \\
&SR_i = SR_i^a + SR_i^b + SR_i^c + SR_i^d \\
&\quad AC_i{}^a = 0, \quad AC_i{}^b = \delta_{ib}, \quad AC_i{}^c = 0, \quad AC_i{}^d = \delta_{id} \\
&\quad SR_i^a = 0, \quad SR_i^b = 0, \quad SR_i^c = \delta_{ic}, \quad SR_i^d = \delta_{id} \\
&\delta_{ia} + \delta_{ib} + \delta_{ic} + \delta_{id} = 1 \\
&\delta_{ij} \in \{0,1\}, \quad j = a,b,c,d
\end{aligned}$$

This simplifies to

$$\begin{aligned}
&AC_i = \delta_{ib} + \delta_{id}, \quad SR_i = \delta_{ic} + \delta_{id} \\
&\delta_{ib} + \delta_{ic} + \delta_{id} \le 1 \\
&\delta_{ij} \in \{0,1\}, \quad j = b,c,d
\end{aligned}$$

The complete MILP model can now be written by combining the above with constraints that meet demand and observe the assembly line capacity constraints:

$$\begin{aligned}
&AC_i = \delta_{ib} + \delta_{id}, \quad SR_i = \delta_{ic} + \delta_{id}, \quad i = 1,\dots,50 \\
&\delta_{ib} + \delta_{ic} + \delta_{id} \le 1, \quad i = 1,\dots,50 \\
&\delta_{ij} \in \{0,1\}, \quad j = b,c,d, \quad i = 1,\dots,50 \\
&\sum_{i=1}^{50} \delta_{ia} = 20, \quad \sum_{i=1}^{50} \delta_{ib} = 15, \quad \sum_{i=1}^{50} \delta_{ic} = 8, \quad \sum_{i=1}^{50} \delta_{id} = 7, \quad i = 1,\dots,50 \\
&\sum_{j=i}^{i+4} AC_j \le 3, \quad i = 1,\dots,46 \\
&\sum_{j=j}^{i+2} SR_j \le 1, \quad j = 1,\dots,48
\end{aligned} \tag{33}$$

5.8 Network Flow Models

The continuous relaxation of an MILP model sometimes describes an integral polyhedron, in the sense that the integer variables take integer values at every vertex. In such cases, one can easily solve the MILP model by solving its continuous relaxation with a linear programming algorithm that finds a vertex solution. There is a strong incentive to use an MILP formulation when it has this integrality property.

In practice, the most common MILP models with the integrality property are capacitated network flow models, of which assignment models are a special case. The matrix of constraint coefficients in these problems is totally unimodular, which ensures that the continuous relaxation of the model describes an integral polytope if all the right-hand sides are integral.

A network flow model is defined on a directed network in which the net supply S_i of flow at each node i is given. If (i, j) represents an arc that is directed from node i to node j, then C_{ij} is the arc capacity and variable y_{ij} represents the flow on (i, j). If arc (i, j) is missing from the network, one can nonetheless include y_{ij} in the model and set $C_{ij} = 0$. The flow model is

$$\begin{aligned} &\sum_j y_{ij} - \sum_j y_{ji} = S_i, \quad \text{all } i \\ &0 \le y_{ij} \le C_{ij}, \quad \text{all } i, j \end{aligned} \tag{34}$$

There is typically an objective function that measures cost, such as $\sum_{ij} c_{ij} y_{ij}$, where c_{ij} is the unit cost of sending flow on arc (i, j).

Due to total unimodularity, the model (34) describes an integral polytope if the supplies and capacities are all integral. This means that if the flows are restricted to be integral, the resulting MILP model can be solved by solving its continuous relaxation (34). This is a particularly easy problem to solve because there are specialized algorithms for computing minimum cost network flows.

5.9 Assignment Problems

The assignment problem discussed in Sect. 4.7 assigns m tasks to n workers ($m \le n$). It is a special case of a network flow problem, which means that the MILP model is totally unimodular. This provides a strong incentive to use an MILP formulation at some stage of the solution process.

The flow network corresponding to an assignment problem is bipartite, as illustrated in Fig. 7. If $m < n$, then dummy task nodes are created so that supply balances demand. The cost c_{ij} is set to zero for a dummy task i. A unit flow $y_{ij} = 1$ indicates that worker i is assigned task j. The flow model therefore reduces to

$$\begin{aligned} &\sum_{j=1}^{n} y_{ij} = \sum_{j=1}^{n} y_{ji} = 1, \quad i = 1, \ldots, n \\ &y_{ij} \ge 0, \quad \text{all } i, j \end{aligned} \tag{35}$$

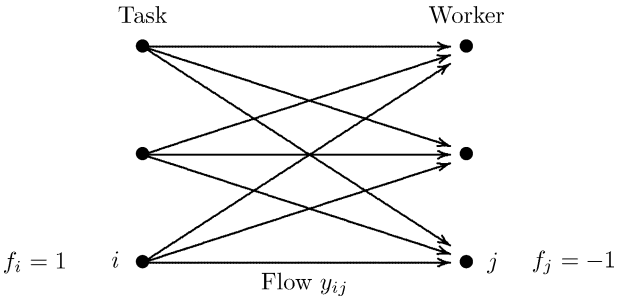


Fig. 7 A flow model for an assignment problem

This model can be solved very rapidly with specialized algorithms. Obviously, the solution is meaningful only if each $y_{ij} \in \{0, 1\}$, but this is assured by total unimodularity.

5.10 Circuit Problems

Circuit problems have been given several MILP formulations [42], but by far the most popular is the subtour elimination formulation. If binary variable $y_{ij} = 1$ when vertex j immediately follows i in the hamiltonian circuit, then the traveling salesman problem on n cities can be written

$$\begin{aligned}
&\min \sum_{ij} c_{ij} y_{ij} \\
&\sum_{j} y_{ij} = \sum_{j} y_{ji} = 1, \quad \text{all } i \\
&\sum_{(i,j)\in\delta(S)} y_{ij} \geq 1, \quad \text{all } S \subset \{1, \ldots, n\} \text{ with } 2 \leq |S| \leq n-1 \\
&y_{ij} \in \{0, 1\}, \quad \text{all } i, j
\end{aligned} \tag{36}$$

where S is a subset of vertices and $\delta(S)$ is the set of edges (i, j) for which $i \in S$ and $j \notin S$. The assignment constraints (line 2) ensure that exactly one vertex precedes, and exactly one vertex follows, each vertex in the tour. Line 3 contains the subtour elimination constraints, which rule out circuits on fewer than n vertices. This is accomplished by requiring, for each proper subset S of the vertices, that at least one edge in the circuit connect a vertex in S with a vertex outside S.

In practice, the formulation is not actually written out because it contains exponentially many constraints. Rather, a traveling salesman solver generates *separating* subtour elimination constraints as they are needed. A separating constraint is one that cuts off the solution of the current linear relaxation without cutting off any feasible solutions. Several families of strong cutting planes have been identified for the problem, along with separation heuristics. A survey of this work can be found in [6].

5.11 Example: Sudoku Puzzles

The sudoku problem can be formulated with assignment constraints, although a large number of 0–1 variables are necessary to do so. Let 0–1 variable $y_{ijt} = 1$ when digit t appears in cell i, j. Let $J_{k\ell}$ be the set of cells (i, j) in the 3×3 square in position k, ℓ. Then, the MILP model is

$$\sum_{j=1}^{9} y_{ijt} = \sum_{j=1}^{9} y_{jit} = 1, \quad i, t = 1, \ldots, 9 \qquad (a)$$

$$\sum_{(i,j)\in J_{k\ell}} y_{ijt} = 1, \quad k, \ell = 1, 2, 3, \quad t = 1, \ldots, 9 \qquad (b)$$

$$\sum_{t=1}^{9} y_{ijt} = 1, \quad i, j = 1, \ldots, 9 \qquad (c)$$

$$y_{ija_{ij}} = 1, \quad \text{all } (i, j) \in F \qquad (d)$$

$$y_{ijt} \in \{0, 1\}, \quad \text{all } i, j, t \qquad (37)$$

Constraints (a) enforce the `alldiff` condition for the rows and columns, and constraints (b) do the same for the 3×3 squares. Constraints (c) ensure that exactly one digit appears in each cell. Constraints (d) take care of the preassigned cells.

6 Integrated Modeling

It is time to address the question as to how the seemingly incompatible modeling styles of CP and MILP can be integrated. CP relies on the use of global constraints, so that it can exploit problem substructure with its filtering algorithms and propagation methods. MILP requires that the problem be reduced to linear inequality constraints, so that it can obtain linear relaxations and strengthen them with cutting planes.

A simple solution is to follow the CP practice of building a model around global constraints, but to create new global constraints that represent sets of inequalities. Different constraints can be designed for inequality sets with different kinds of special structure, so that the solver can take advantage of this structure when it generates cutting planes. A general-purpose constraint can be defined for an MILP inequality set that has no particular structure.

Because MILP models can always be constructed by conceiving the problem as a disjunction of linear systems, it may be more natural in many cases to write the problem in disjunctive form rather than translate the disjunctions to MILP models. The solver can make the translation automatically. It is therefore useful for a general-purpose MILP global constraint to accept disjunctions of linear systems as well as a single linear system.

The proposal, therefore, is that the modeler write parts of the problem with CP-style global constraints and other parts with global constraints that represent structured sets of linear inequalities, depending on which is more natural and best reveals the structure of the problem to the solver.

Such a model allows the solver to take full advantage of both CP and MILP solution technology. CP-style global constraints are explicitly present, which allows the solver to apply its repertory of filtering and propagation techniques. Inequality constraints are also explicitly present, identified by their structure, which allows the solver to generate MILP-based relaxations and cutting planes. This is already an advance over commercial MILP systems, which do not permit the user to identify most types of special structure in subsets of constraints.

Furthermore, MILP relaxation technology can be applied even when a constraint is written in CP style. If a CP constraint in the model has an alternative MILP formulation, the solver always has the option of generating the MILP formulation along with any useful cutting planes for the sake of obtaining a relaxation. Alternatively, the solver may generate a relaxation that is based on a polyhedral analysis of that particular constraint, rather than on an MILP model of it. The overall advantage of this scheme is that it allows the solver to exploit the wide variety of filtering and relaxation methods that appear in the CP, MILP, and CP-AI-OR literatures.

There are some technical issues that must be clarified if the generated inequality relaxations are to replicate the full advantage of MILP technology. One is a variable mapping issue that arises when MILP translations of global constraints create auxiliary variables. When the auxiliary variables map to the same set of variables in the original model, the solver must use the same auxiliary variables in all the translations. Also, care must be taken when simplifying the individual MILP translations, so that the combined translations provide a correct model.

Not only are the full computational resources of CP and MILP simultaneously available but they are mutually reinforcing in all the ways that have been described in the literature and the remainder of this book. CP-based filtering, for example, results in tighter MILP models and relaxations, which in turn provide bounds for more effective domain reduction. Thus, an integrated model supports integrated problem solving.

The remainder of this section begins with a summary of guidelines for integrated modeling. It then reviews the problems that have been so far introduced and indicates how they might be formulated in an integrated modeling system. It shows how relaxations can be generated and addresses the technical issues just mentioned. Because hybrid methods often use decomposition methods to combine solution techniques, the section concludes with two illustrations of how decomposition can be introduced into a model. This not only tells the solver how the problem may be decomposed but it may allow the model to be written with high-level global constraints that would not otherwise be applicable.

6.1 Integrated Modeling Guidelines

Because it is proposed that an integrated model be built around global constraints much as in CP modeling, CP modeling guidelines continue to apply. They must be augmented, however, with principles for incorporating MILP-based global constraints. In view of the above discussion, the following principles seem appropriate.

1. A specially-structured subset of constraints should be replaced by a single global constraint that captures the structure, when a suitable one exists.
2. A global constraint can be one familiar to the CP community or a collection of inequalities whose structure has been studied in the MILP literature.
3. Constraints that are more naturally formulated as disjunctions of linear systems should normally be left in this form, rather than converting them to MILP models.
4. A global constraint should be replaced by a more specific one when possible.
5. Redundant constraints can improve propagation. However, the solver should be relied upon to generate the MILP equivalent of a CP constraint in the model.
6. When two formulations of a problem are natural and intuitive, both (or parts of both) may be included in the model to improve propagation. This is especially helpful when some constraints are hard to write in one formulation but suitable for the other. Channeling constraints should be used to define variables in the two formulations in terms of each other.
7. Decomposition can be introduced into a model when it would alert the solver to a useful decomposition strategy or when it would permit the use of high-level global constraints that would not otherwise be applicable. This can be accomplished with a `subproblem` global constraint, described below.

6.2 Example: Facility Location

The facility location problem is naturally expressed as an MILP model. The primary elements of the problem are a disjunction of alternatives (install the factory or not) and additional linear inequalities (customer demand). Following Principle 3, it should be written in disjunctive form rather than converting it to an MILP.

It is therefore convenient to invent a general-purpose MILP global constraint $\texttt{linear}(\vee_k S_k)$, which enforces the disjunction of the linear systems S_k. A special case is $\texttt{linear}(S)$, which enforces a single linear system S. Any integrality restrictions on the variables are given when the variable domains are specified. The facility location model can be written

$$\texttt{linear}\left(\begin{pmatrix}\sum_j x_{ij} \le C_i \\ 0 \le x_{ij} \le K_{ij} w_{ij}, \text{ all } j \\ z_i \ge f_i + \sum_j c_{ij} w_{ij} \\ w_{ij} \in \mathbb{Z}, \text{ all } j\end{pmatrix} \vee \begin{pmatrix} x_{ij} = 0, \text{ all } j \\ z_i \ge 0 \end{pmatrix}\right), \quad \text{all } i$$

$$\texttt{linear}\begin{pmatrix}\min \sum_i z_i \\ \sum_i x_{ij} \ge D_j, \quad \text{all } j\end{pmatrix}$$

$$x_{ij}, z_i \in \mathbb{R}, \quad w_{ij} \in \mathbb{Z}, \quad \text{all } i, j \tag{38}$$

The objective function is placed in a `linear` constraint because it can be viewed as a linear inequality, namely $\sum_i z_i \le U$, where U is any upper bound on the minimum cost.

As noted in Sect. 5.5, the mixed integer polyhedra described by the two terms of the disjunction in (38) have different recession cones. It is therefore impossible for the solver to generate an MILP model for the disjunction. However, the purpose of creating an MILP model is to obtain its continuous relaxation. The continuous relaxations of the convex hull and big-M formulations are valid relaxations for the problem, even though the formulations do not correctly model the problem.

This suggests that the `linear` constraint can be accompanied by a parameter that has three possible values.

Relaxation only. The solver simply generates a valid relaxation of the disjunction based on the big-M or convex hull formulation. It does not strengthen the relaxation with cutting planes, because these formulations may not be valid MILP models.

User-defined recession cones. The solver assumes that the modeler has equalized the recession cones by hand, perhaps simply by placing reasonable bounds on the variables. The solver creates a valid MILP model and perhaps strengthens it with cutting planes.

System-defined recession cones. The solver automatically equalizes the recession cones, again perhaps by placing bounds on the variables. It is a research issue how these bounds can be adjusted automatically so that the resulting relaxation is reasonably tight.

6.3 Examples: Piecewise Linear and Indicator Constraints

A piecewise linear function (28) can be modeled with a specialized global constraint as well as with a disjunctive constraint similar to (29). Such a global constraint might take the form

$$\texttt{piecewiselinear}(x, z, \mathbf{a}, \mathbf{b}, \mathbf{f(a)}, \mathbf{f(b)})$$

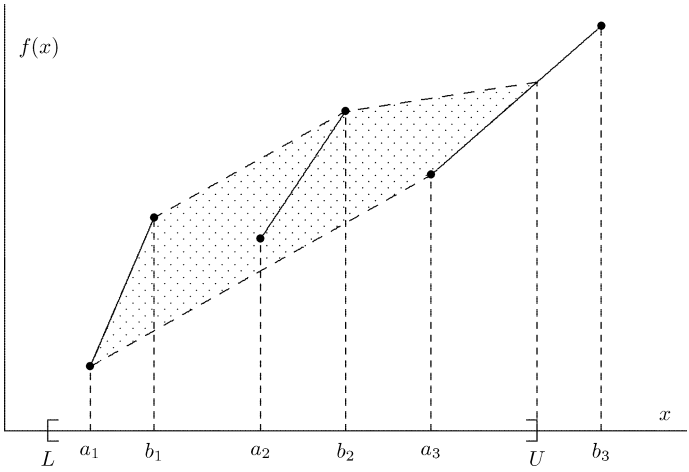


Fig. 8 Convex hull relaxation of a piecewise linear function (*shaded area*)

where $\mathbf{a} = (a_1, \ldots, a_m)$, $\mathbf{f(a)} = (f(a_1), \ldots, f(a_m))$, and similarly for $\mathbf{b}$ and $\mathbf{f(b)}$. The variable z represents $f(x)$. The `piecewiselinear` constraint is not only convenient but can have computational advantages. It can provide a convex hull relaxation without introducing any continuous or 0–1 auxiliary variables, and it allows for intelligent branching.

The relaxation is obtained by computing the convex envelope of the graph of f, as illustrated in Fig. 8. This can be quickly accomplished with computational geometry techniques. The relaxation consists of the few inequalities that define the convex hull. The solver can branch on the constraint by splitting the domain of x. For example, if the value of x in the solution of the current relaxation is between b_1 and a_2 in Fig. 8, the domain is split into intervals $[a_1, b_1]$ and $[a_2, U]$. The convex hull relaxation is recomputed for each branch and becomes tighter. There is evidence that this approach can reduce computation substantially relative to standard MILP techniques [43, 62].

An indicator constraint can also be expressed as a global constraint, namely a conditional constraint. In general, a conditional constraint has the form $A \Rightarrow B$, where A is a set of constraints on discrete variables, and B is an arbitrary set of constraints that are enforced when A becomes satisfied. An indicator constraint that enforces $Ax \geq b$ when $\delta = 1$ can be written $(\delta = 1) \Rightarrow (Ax \geq b)$.

However, when B consists of linear inequality constraints, it may be best to write a conditional constraint as a disjunction of linear systems, because this invokes the generation of convex-hull relaxations. Thus, $(\delta = 1) \Rightarrow (Ax \geq b)$ can be written in the disjunctive form $(Ax \geq b)_\delta$ as suggested earlier. To equalize recession cones, the modeling system can automatically add user-supplied bounds $L \leq x \leq U$ to the disjunction, which becomes

$$\begin{pmatrix} Ax \geq b \\ L \leq x \leq U \end{pmatrix}_\delta \vee (L \leq x \leq U) \tag{39}$$

The system now generates a convex-hull MILP model for (39).

Similarly, a set of conditional constraints of the form $(\delta_i = 1) \Rightarrow (A^i x \geq b^i)$ for $i \in I$ should be written

$$\bigvee_{i \in I} \left(A^i x \geq b^i\right)_{\delta_i}$$

The disjunct corresponding to $\delta_i = 0$ for all $i \in I$ does not appear because it is empty.

6.4 Network Flow Problems

An MILP model is the preferred choice for a network flow problem, not only because it is a natural and intuitive formulation, but also because it has the substantial advantage of total unimodularity. However, there is no need to write out the individual network flow constraints, and even if one did, the solver might not recognize the network flow structure. Principles 1 and 2 call for a global constraint to represent the inequality set. Such a constraint might be written

$$\texttt{networkFlow}(\mathbf{y}, \mathbf{f}, \mathbf{C}) \tag{40}$$

where $\mathbf{y}$ is a matrix of flow variables y_{ij}, $\mathbf{f}$ is a vector of net supplies f_i, and $\mathbf{C}$ is a matrix of arc capacities C_{ij}. On encountering a constraint of the form (40), the modeling system automatically generates the MILP model (34).

A minimum cost network flow problem can be stated by minimizing the objective function $\sum_{ij} y_{ij}$ subject to (40).

6.5 Assignment Problems

The assignment problem is naturally written in its CP form (8), using the `alldiff` constraint. The MILP model (35) is also useful due to its total unimodularity, but the solver should be relied upon to generate it (Principle 5). Generating this model raises the important technical issue of variable mapping.

The solver generates the MILP assignment constraints (35) when it encounters the `alldiff` constraint in the CP model. But this alone is not adequate. Fast algorithms for the MILP assignment model use an objective function $\sum_{ij} c_{ij} y_{ij}$ that is expressed in terms of the 0–1 variables y_{ij}, and not the objective function $\sum_i c_{ix_i}$ that appears in the CP model (8).

To make this more precise, recall that the variable subscript in $\sum_i c_{ix_i}$ is parsed by generating an `element` constraint. The CP model actually sent to the solver is therefore

$$\begin{aligned}
&\min \sum_i z_i \\
&\texttt{element}(x_i, (c_{i1}, \ldots, c_{in}), z_i), \quad i = 1, \ldots, n \\
&\texttt{alldiff}(x_1, \ldots, x_n), \quad x_i \in D_i, \quad i = 1, \ldots, n
\end{aligned} \tag{41}$$

The solver can now create MILP translations for the `element` constraints as well as the `alldiff`. The ith element constraint can be given an MILP model by writing a convex hull formulation of the disjunction $\bigvee_j (z_i = c_{ij})$:

$$z_i = \sum_{j=1}^{n} c_{ij} y'_{ij}, \quad \sum_{j=1}^{n} y'_{ij} = 1, \quad y'_{ij} \in \{0,1\}, \quad j = 1,\ldots,n \tag{42}$$

The MILP translation of (41) that results in

$$\begin{aligned} &\min \sum_i z_i \\ &z_i = \sum_{j=1}^{n} c_{ij} y'_{ij}, \quad \sum_{j=1}^{n} y'_{ij} = 1, \quad i = 1,\ldots,n \\ &\sum_{j=1}^{n} y_{ij} = \sum_{j=1}^{n} y_{ji} = 1, \quad i = 1,\ldots,n \\ &y'_{ij} \in \{0,1\}, \quad i = 1,\ldots,n \end{aligned} \tag{43}$$

The variables y_{ij}, y'_{ij} are related to the original variables x_i by way of variable mapping constraints

$$x_i = \sum_{j=1}^{n} j y_{ij}, \quad x_i = \sum_{j=1}^{n} j y'_{ij}, \quad i = 1,\ldots,n \tag{44}$$

The difficulty is that (43) is not an assignment problem with the integrality property, unless variables y_{ij} are identified with variables y'_{ij}. The solver can accomplished this by mapping the variables x_i to the *same* set of 0–1 variables y_{ij} whenever it creates an MILP model containing variables defined as in (44). This means that the y_{ij}s become *global* variables rather than local to a specific MILP translation. If the original model contains only an `alldiff` constraint and an objective function, the solver can now exploit the total unimodularity of the MILP translation once it verifies that the translated objective function has the right form.

The practice of mapping variables in the original model to global variables in the MILP translations can be called *global variable mapping*. Such a practice ensures that one can use the succinct CP model (8) for an assignment problem without sacrificing any of the advantages of an MILP model. However, MILP variables that are not mapped to an original variable should remain local. This is illustrated in the car sequencing example below.

6.6 Circuit Problems

From a modeling point of view, a CP formulation of the circuit (traveling salesman) problem is superior to an MILP formulation. A CP formulation is more natural and

contains only one constraint, as opposed to exponentially many constraints in the most popular MILP model. The question remains, however, as to which of the two CP formulations is better in an integrated modeling context–the `alldiff` formulation (9) or the `circuit` formulation (10).

The `circuit` formulation seems more intuitive, because it is conceived in terms of a circuit, as opposed to an assignment. The variables y_i in the `circuit` formulation refer to the next vertex in the hamiltonian circuit, which allows one to indicate missing edges from the graph by removing elements from the variable domains.

The `circuit` formulation is superior from a technical point of view as well, because it allows for more effective propagation and relaxation. Consider the situation with the `alldiff` formulation. No filtering can take place because the variable domains are complete. In addition, the totally unimodular MILP model of `alldiff` constraint is of little use in the context of a circuit problem, because it has no variables in common with the MILP translation of the objective function. To see this, recall that the $\texttt{alldiff}(x_1, \ldots, x_n)$ constraint is translated

$$\sum_{j=1}^{n} y_{ij} = \sum_{j=1}^{m} y_{ji} = 1, \quad y_{ij} \in \{0,1\}, i = 1, \ldots, n \tag{45}$$

where the x_is are mapped to the auxiliary variables y_{ij} by

$$x_i = \sum_j j y_{ij} \tag{46}$$

However, the objective function $\sum_i c_{x_i x_{i+1}}$ of (9) is parsed as

$$\sum_i z_i, \quad \texttt{element}\left((x_i, x_{i+1}), \mathbf{C}, z_i\right), \quad i = 1, \ldots n \tag{47}$$

where $\mathbf{C}$ is the matrix of coefficients c_{ij}. The ith element constraint sets z_i equal to the element of $\mathbf{C}$ in position (x_i, x_{i+1}). This can be translated to an MILP model by writing a convex hull formulation of the disjunction $\bigvee_{jk}(z_i = c_{jk})$:

$$z_i = \sum_{jk} c_{jk}\delta_{ijk}, \quad \sum_{jk} \delta_{ijk} = 1, \quad \delta_{ijk} \in \{0,1\}, \quad \text{all } j,k$$

where the x_is are mapped to the auxiliary variables δ_{ijk} by

$$\delta_{ijk} = 1 \Leftrightarrow (x_i, x_{i+1}) = (j,k) \tag{48}$$

Because (46) and (48) are different mappings, the solver does not (and should not) identify the variables δ_{ijk} with the variables y_{ij}. So the MILP translation (45) of the `alldiff` has no variables in common with the MILP translation (47) of the objective function, and the resulting MILP model of the problem is useless as a relaxation.

The `circuit` formulation, on the other hand, allows for filtering, even though achieving domain consistency is much harder than for `alldiff`. Also the `circuit` constraint allows one to harness the advanced relaxation methods that have been developed for the MILP formulation (36) of the constraint. The solver would not actually generate the entire MILP formulation, because of its exponential size, but would generate separating subtour elimination constraints and strong separating cuts as needed, much as a specialized traveling salesman solver would do. The difficulty that arose with the `alldiff` constraint does not occur here, because the variables y_{ij} that occur in the MILP translation (36) of `circuit` also occur in the MILP translation of the objective function $\sum_i c_{iy_i}$. The latter is simply $\sum_{ij} c_{ij} y_{ij}$, and both MILP translations use the same variable mapping $y_i = \sum_j j y_{ij}$.

In addition, it is possible to write a relaxation of the `circuit` constraint solely in terms of the original variables y_i, provided they take numerical values. It is argued in [33] that a proper choice of these values can exploit structure in the objective function.

The dual model (11), which uses both `alldiff` and `circuit` constraints, may be advantageous when some other constraints in the problem are best expressed in terms of the x_i variables (Principle 6). In such cases, the MILP-based relaxation of `alldiff` may be useful even though it does not connect with the objective function.

6.7 Example: Sudoku Puzzles

The sudoku puzzle is most naturally modeled with `alldiff` constraints, as in (1). It may also be advantageous for the solver to generate the MILP model (37), which is not totally unimodular but may provide a useful relaxation. The solver can easily generate the assignment constraints for each `alldiff`. If the solver uses global variable mapping, the combined assignment constraints provide the desired relaxation.

MILP-based relaxations for `alldiff` contain 0–1 variables y_{ij}, and not the original variables x_i. Polyhedral relaxations in the original variables have been studied for the `alldiff` constraint [24, 61] and the `multiAlldiff` constraint [35, 37], provided those variables take numerical values such as $1, \ldots, n$. The solver may choose to generate these in addition to the MILP model, particularly, if the y_{ij}s do not occur in the relaxations of the objective function or other constraints.

6.8 Example: Car Sequencing

The CP-based formulation (4) of the car sequencing model is very appropriate for an integrated modeling context, due to its simplicity and the fact that it harnesses

the filtering power of `cardinality` and `sequence` constraints. It may be useful for the modeling system to generate the MILP model (33) automatically, because it provides a relaxation. However, the generation of such a model raises another important technical point.

The `cardinality` constraint in the CP model (4) translates immediately to the desired MILP constraints in (33), namely

$$\sum_{i=1}^{50} \delta_{ia} = 20, \quad \sum_{i=1}^{50} \delta_{ib} = 15, \quad \sum_{i=1}^{50} \delta_{ic} = 8, \quad \sum_{i=1}^{50} \delta_{id} = 7, \quad i = 1, \ldots, 50 \tag{49}$$

using the variable mapping

$$(\delta_{ij} = 1) \Rightarrow (t_i = j), \quad j = \mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{d} \tag{50}$$

There may appear to be a difficulty in translating the two `sequence` constraints, however. For the first `sequence` constraint in (4), disjunctions of the form

$$(AC_i = 0) \vee (AC_i = 1) \vee (AC_i = 0) \vee (AC_i = 1) \tag{51}$$

are converted to the MILP model

$$\begin{aligned} & AC_i = \delta_{ib} + \delta_{id}, \quad i = 1, \ldots, 50 \\ & \delta_{ia} + \delta_{ib} + \delta_{ic} + \delta_{id} = 1 \\ & \sum_{j=i}^{i+4} AC_j \le 3, \quad i = 1, \ldots, 46 \end{aligned} \tag{52}$$

which simplifies to

$$\begin{aligned} & AC_i = \delta_{ib} + \delta_{id}, \quad i = 1, \ldots, 50 \\ & \delta_{ib} + \delta_{id} \le 1 \\ & \sum_{j=i}^{i+4} AC_j \le 3, \quad i = 1, \ldots, 46 \end{aligned} \tag{53}$$

If global variable mapping is used, the variables δ_{ij} in (53) are the same as those in (49), because they are mapped to the same original variables t_i using (50). Similarly, the second `sequence` constraint yields the MILP model

$$\begin{aligned} & SR_i = \delta_{ic} + \delta_{id}, \quad i = 1, \ldots, 50 \\ & \delta_{ia} + \delta_{ib} + \delta_{ic} + \delta_{id} = 1 \\ & \sum_{j=i}^{i+2} SR_j \le 1, \quad i = 1, \ldots, 48 \end{aligned} \tag{54}$$

which simplifies to

$$\begin{aligned} &SR_i = \delta_{ic} + \delta_{id}, \quad i = 1, \ldots, 50 \\ &\delta_{ic} + \delta_{id} \leq 1 \\ &\sum_{j=i}^{i+2} SR_j \leq 1, \quad i = 1, \ldots, 48 \end{aligned} \tag{55}$$

The variables AC_i and SR_i are local to MILP translation because they are not mapped to any of the original variables.

The difficulty is that when the MILP formulations (53) and (55) are merged, the constraints involving δ_{ij}s are not equivalent to the corresponding constraints in the desired MILP model (33):

$$\begin{aligned} &AC_i = \delta_{ib} + \delta_{id}, \quad SR_i = \delta_{ic} + \delta_{id}, \quad i = 1, \ldots, 50 \\ &\delta_{ib} + \delta_{ic} + \delta_{id} \leq 1, \quad i = 1, \ldots, 50 \end{aligned} \tag{56}$$

The two inequalities $\delta_{ib} + \delta_{id} \leq 1$ and $\delta_{ic} + \delta_{id} \leq 1$ are not equivalent to the inequality $\delta_{ib} + \delta_{ic} + \delta_{id} \leq 1$ in (56).

The problem is that (52) and (53) are equivalent only in the sense that they specify the same disjunction (51). Because the auxiliary variables δ_{ij}s are global variables, the two formulations must be equivalent in the space that includes the δ_{ij}s as well as the AC_is. In general, when an MILP translation based on disjunctions is simplified, it must be simplified to a formulation that is equivalent in the auxiliary variables as well as the variables in the disjuncts, if the auxiliary variables are global. It is therefore essential to use the formulation (52) rather than (53), and similarly to use (54) rather than (55). If this is done, the result is the desired MILP model (33).

6.9 Example: Employee Scheduling

The CP idiom is especially well suited for employee scheduling problems, because several global constraints are expressly designed for this purpose–such as the `stretch` constraint in the CP model of Sect. 4.6. Writing an MILP model for `stretch` is not straightforward and should not be attempted in the original model. How the solver might generate an MILP translation presents an interesting research issue, as does the task of finding linear relaxations that are not based on MILP formulations.

6.10 Decomposition: Machine Assignment and Scheduling

A machine assignment and scheduling problem illustrates the usefulness of decomposition in modeling (Principle 7). A set of n jobs are to be assigned to m machines. The jobs assigned to each machine must be scheduled so that they do not overlap and are processed within their time windows. The time window for each job j

consists of a release time r_j and a deadline d_j. Each job j has a processing time of p_{ij} on machine i. For simplicity, suppose the cost of assigning job j to machine i is a constant c_{ij}.

The assignment portion of the problem is modeled simply by letting x_j be the machine assigned to job j. For the scheduling component, a number of well-studied global constraints are available. The simplest is a disjunctive scheduling constraint, which might be written $\texttt{disjunctive}(\{t_1, \ldots, t_n\})$, where t_j is the start time of job j. The constraint requires that start times be set so that each job runs inside its time window and starts after the previous job finishes. The time window of each job j is enforced by setting the domain of each x_j to the interval $[r_j, d_j - p_j]$, where p_j is the processing time of job j. Filtering algorithms for the constraint use edge finding and other methods to reduce the domains of the t_js [7].

One would like to write a model for the assignment and scheduling problem that uses `disjunctive` constraints:

$$\begin{aligned} &\texttt{linear}\left(\begin{array}{l}\min \displaystyle\sum_j c_{x_j j} \\ r_j \le t_j \le d_j - p_{x_j j}, \quad \text{all } j\end{array}\right) \\ &\texttt{disjunctive}\left(\{t_j \mid x_j = i\}\right), \quad \text{all } i \\ &t_j \in \mathbb{R}, \quad x_j \in \{1, \ldots, m\}, \quad \text{all } j \end{aligned} \tag{57}$$

Each `disjunctive` constraint schedules the jobs on one of the machines. The difficulty is that the `disjunctive` constraints are not well defined until the values of the x_js are known, because the variable list in the constraints depends on the x_js. In principle, an enhanced `disjunctive` constraint could be designed to filter the t_j and x_j domains simultaneously, but there is apparently no such enhanced constraint in current systems.

By introducing decomposition into the model, however, one can retain the `disjunctive` constraints. One approach is to define a global constraint that specifies a subproblem in which the values of certain variables are assumed to be known. The constraint could be written

$$\texttt{subproblem}(X, C_1, \ldots, C_k)$$

to enforce constraints $C_1, \ldots, C_k$ after the values of the variables in set X are fixed. The model (57) can be written

$$\begin{aligned} &\texttt{linear}\left(\min \sum_j c_{x_j j}\right) \\ &\texttt{subproblem}\left(\begin{array}{l}\{x_1, \ldots, x_n\}, \\ \texttt{disjunctive}\left(\{t_j \mid x_j = i\}\right), \quad \text{all } i, \\ \texttt{linear}\left(r_j \le t_j \le d_j - p_{x_j j}, \quad \text{all } j\right)\end{array}\right) \\ &t_j \in \mathbb{R}, \quad x_j \in \{1, \ldots, m\}, \quad \text{all } j \end{aligned} \tag{58}$$

The variables x_j function as constants inside the subproblem, which means that the `disjunctive` constraints are well defined. Also the time window constraints set the variable domains to ranges appropriate for the assigned machine.

The solver may be able to exploit the decomposition structure that is identified in the model. In this case, it might use a Benders method, because the outer problem and the subproblem use disjoint sets of variables (the x_js and the t_js, respectively). The `disjunctive` constraint would be associated with an algorithm that generates a logic-based Benders cut. The cut is added to the main problem, which is then re-solved. For example, if the scheduling problem on machine i is infeasible, the `disjunctive` filter may discover that a small subset J of the jobs assigned to machine i are responsible for the infeasibility. Then, a Benders cut $\bigvee_{j \in J}(x_j \neq i)$ can be added to the constraint set outside the subproblem. This constraint set is then re-solved to obtain new trial values of the x_is, and so on until an optimal solution is obtained. This process has been used in a number of contexts (e.g., [12, 24, 25, 27, 30]) and is discussed further in Chapter "Decomposition Techniques for Hybrid MILP/CP Models Applied to Scheduling and Routing Problems".

An important generalization of disjunctive scheduling is *cumulative* scheduling, which allows tasks to be run simultaneously subject to one or more resource constraints. Each task consumes each resource at a certain rate, and there is a limit on the total rate of consumption–a limit that may vary with time. Several versions of the `cumulative` global constraint exist for this situation, and logic-based Benders cuts have been developed for some of them as well as for the `disjunctive` constraint. Although filtering technology for `disjunctive` and `cumulative` is highly developed [7], it may be useful to generate MILP formulations or linear relaxations that are not based on MILP models [26].

6.11 Decomposition: Routing and Frequency Assignment

A final example, adapted from [52], illustrates decomposition in a more complex setting. The arcs of a directed network represent optical fibers with limited capacity (Fig. 9). There are requests for communication channels to be established between certain pairs of nodes. The channels must be routed over the network, and they must be assigned frequencies so that all the channels passing along any given arc have different frequencies. There are a limited number of frequencies available, and it may not be possible to establish all the channels requested. The objective is to maximize the number of channels established.

The problem can be decomposed into its routing and frequency-assignment elements. The routing problem is amenable to an MILP formulation, and the frequency assignment problem is conveniently written with `alldiff` constraints–provided that a `subproblem` constraint is used to fix the flows before the frequency assignment problem is stated.

The routing problem is similar to the well-known multi-commodity network flow problem. This problem generalizes the capacitated network flow problem discussed above by distinguishing several commodities that must be transported over the network. There is a net supply of each commodity at each node, and the total

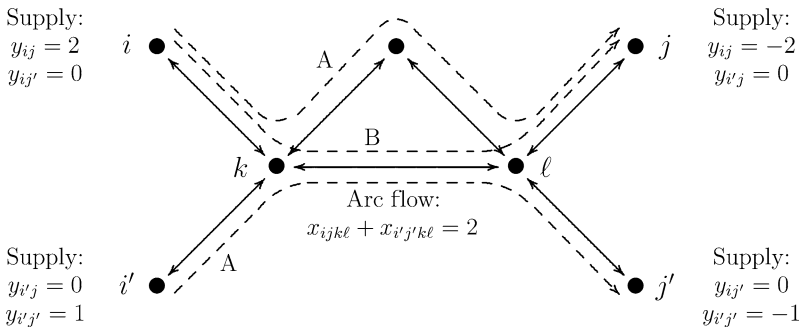


Fig. 9 A message routing and frequency assignment problem. Two message channels are requested from i to j and one from i' to j'. The arcs have capacity 2, and frequencies A, B are available. The dashed lines show an optimal solution

flow on each arc must be within the arc capacity. In the message routing problem, each origin-destination pair represents a different commodity.

The message routing problem is not identical to the multicommodity flow problem because the net supplies are not fixed due to the fact that some requests may not be satisfied. As a result, one would not be able to use a global constraint designed for multicommodity flow problems, even if one existed. Nonetheless, it is fairly easy to write the MILP constraints directly.

For each pair of nodes (i, j), let D_{ij} be the number of i-to-j channels requested (possibly zero). A key decision is which requests to honor, and one can therefore let integer variable y_{ij} be the number of channels from i to j that are actually established. (It is assumed here that different channels from i to j can be routed differently.) The net supply of commodity (i, j) is y_{ij} at node i, $-y_{ij}$ at node j, and zero at other nodes. Let $x_{ijk\ell}$ be the flow of commodity (i, j) on arc (k, ℓ), and $C_{k\ell}$ the capacity of the arc. To simplify notation, arcs missing from the network can be viewed as arcs with a capacity of zero. The flow model is

$$
\texttt{linear}\left(\begin{array}{l}
\max \displaystyle\sum_{ij} y_{ij} \\
\displaystyle\sum_{\ell \neq i} x_{iji\ell} - \sum_{k \neq i} x_{ijki} = y_{ij}, \quad \text{all } i, j \\
\displaystyle\sum_{\ell \neq j} x_{ijj\ell} - \sum_{k \neq i} x_{ijkj} = -y_{ij}, \quad \text{all } i, j \\
\displaystyle\sum_{\ell \neq i,j,k} x_{ijk\ell} - \sum_{\ell \neq i,j,k} x_{ij\ell k} = 0, \quad \text{all } i, j, k \text{ with } k \neq i, j \\
\displaystyle\sum_{ij} x_{ijk\ell} \leq C_{k\ell}, \quad \text{all } k, \ell \\
x_{ijk\ell} \geq 0, \quad \text{all } i, j, k, \ell \\
0 \leq y_{ij} \leq D_{ij}, \quad \text{all } i, j
\end{array}\right)
$$

$$
x_{ijk\ell}, y_{ij} \in \mathbb{Z}, \quad \text{all } i, j, k, \ell
$$

Once the communications channels are routed, a frequency f_{ij} can be assigned to each pair i, j so that the frequencies assigned to channels passing through any given arc are all different. The model is therefore completed by writing

$$\texttt{subproblem}\left(\begin{array}{l}\left\{x_{ijk\ell},\ \text{ all } i,j,k,\ell\right\},\\ \texttt{alldiff}\left(\left\{f_{ij} \mid x_{ijk\ell} > 0\right\}\right),\ \text{ all } k,\ell\end{array}\right)$$
$$f_{ij} \in F,\ \text{ all } i,j \text{ with } i \neq j$$

where F is the set of available frequencies.

7 Conclusions

A scheme for integrating CP and MILP modeling styles has been proposed, in which structured sets of MILP inequalities appear as global constraints alongside the global constraints of CP. It remains to assess, however, whether a scheme of this sort can deliver the advantages of both CP and MILP modeling in a single framework.

It has already been argued that the full computational resources of both CP and MILP are available in an integrated setting, where they can also be combined for greater effect. CP-style global constraints continue to appear in the model whenever they provide the best modeling approach, and they can be subjected to any filtering or propagation methods available to a CP solver. MILP relaxation technology can also be brought to bear, even in cases where it would not be applied in a commercial MILP solver, because structured sets of inequalities are identified. Even when constraints are not written in inequality form, the integrated solver can generate MILP translations when they are useful. This results in an MILP-based relaxation that is as effective as any obtained from a conventional MILP model, provided certain technical issues are handled correctly. Global constraints can be given linear relaxations that are not based on an MILP model and therefore cannot be used in MILP solvers. Beyond this, the potential of integrated problem solving can be tapped, because CP filtering and MILP relaxations are mutually reinforcing.

The key advantages of CP-based modeling are the power of the modeling language, the relative conciseness and naturalness of its formulations, and their ability to reveal problem structure to the solver. These advantages are clearly retained, and again enhanced, because the lexicon of global constraints is increased to encompass structural ideas from MILP.

The key advantages of MILP modeling–apart from its ability to harness MILP relaxation technology, which is retained–are its reliance on a small set of primitives (linear inequalities) and the relative independence of model and solution method. Integrated modeling obviously sacrifices the first advantage, because it relies on a sizable collection of global constraints. Yet it must be asked whether this is actually a sacrifice.

The difficulty in using global constraints is presumably that one must be familiar with a large number of them to be able to write a model. Yet one frequently writes an MILP model, or at least important parts of it, by identifying such patterns as flow balances, fixed charges, packing or covering constraints, and so forth, and then reducing them to inequality form. One must therefore be familiar with a number of modeling ideas in any case. Integrated modeling only spares one the labor (and errors) of writing micro-constraints that can be generated automatically. Moreover, a well-organized list of constraints can alert the modeler to patterns that might otherwise have been overlooked in the problem. It can provide a vocabulary that helps one to learn and distinguish modeling ideas, much as a technical vocabulary assists learning in any field.

Global constraints seem to be proliferating day by day, but new constraints tend to be variations on old ones. A well-known global constraints catalog [9] lists 313 constraints, but on close examination, one can identify about thirty basic modeling ideas among these, of which the other constraints are variations and extensions. A good modeling system can organize constraints along various dimensions, so that one can generally find what one needs, much as one finds the relevant function in a spreadsheet.

Independence of model and method presents a more serious challenge to integrated modeling, and the matter deserves careful examination. It should be acknowledged at the outset, however, that the issue is not independence of model and *method*, because no modeling language achieves it, but independence of model and *solver*. Good MILP modelers know that one must think about the solution method when writing a model. The constraints must be chosen to result in a tight relaxation, the variables chosen to allow effective branching, redundant constraints added, symmetry considered, special ordered sets used when applicable, and so forth. Nonetheless, MILP does achieve independence of model and solver (with the possible exception of special ordered sets), because a given MILP model will run on almost any solver.

Reliance on global constraints undeniably links the model with the solver, because different solvers offer different libraries of constraints. The library of available constraints grows as solvers advance, and the best way to write a model evolves accordingly. Static collections of models used for benchmarking software, such as MIPLIB, are an impossibility, because the formulations must change with the solution technology to take full advantage of the software.

Independence of model and solver is frequently discussed as though it was an unmitigated advantage, when in fact it has both positive and negative aspects. A fixed modeling language provides the convenience of being able to run a model on any solver, but that very characteristic blocks progress in solution technology. Integrated methods, for example, can sometimes yield orders of magnitude in computational speedup, but only if one is willing to move beyond traditional MILP modeling.

As for benchmarking, a standard set of MILP models allows one to compare a wide variety of solvers, but at the cost of restricting one's attention to certain kinds of solvers. Benchmarking sets can equally well consist of problem *statements* (as opposed to models), so that one can reformulate the problems as necessary for

new solvers. This allows one to monitor progress in modeling practices as well as algorithms. Some popular benchmarking libraries, such as MIPLIB, contain models for which the underlying problems are actually unknown, which means that they cannot be reformulated. This practice does not seem optimal for progress in either modeling or solution technology.

In summary, integrated modeling forgoes the convenience of solver independence, but it compensates with more convenient modeling and a wider repertory of solution methods. Even the inconvenience of incompatible solvers may fade over time, because software vendors will have an incentive to converge toward a universal set of global constraints. They may want to satisfy as many customers as possible by implementing all the global constraints they prefer to use.

The discussion here has focused on CP and MILP, but integrated modeling can in principle be broadened to encompass nonlinear constraints, local search heuristics and other AI-based search procedures, stochastic models and methods, and even simulation. Ideally, a single modeling system would allow one to write problem formulations to which the solver can apply any combination of methods that might be effective.

References

1. Achterberg T, Berthold T, Koch T, Wolter K (2008) A new approach to integrate CP and MIP. In: Perron L, Trick MA (eds) Proceedings of the international workshop on integration of artificial intelligence and operations research techniques in constraint programming for combinatorial optimization problems (CPAIOR 2008). Lecture notes in computer science, vol 5015. Springer, Berlin, pp 6–20
2. Ajili F, Wallace M (2004) Hybrid problem solving in ECLiPSe. In: Milano M (ed) Constraint and integer programming: toward a unified methodology. Kluwer, Dordrecht, pp 169–206
3. Althaus E, Bockmayr A, Elf M, Kasper T, Jünger M, Mehlhorn K (2002) SCIL–Symbolic constraints in integer linear programming. In: 10th European symposium on Algorithms (ESA 2002). Lecture notes in computer science, vol 2461. Springer, New York, pp 75–87
4. Apt K, Wallace M (2006) Constraint logic programming using ECLiPSe. Cambridge University Press, Cambridge
5. Aron I, Hooker JN, Yunes TH (2004) SIMPL: a system for integrating optimization techniques. In: Régin JC, Rueher M (eds) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR 2004). Lecture notes in computer science, vol 3011. Springer, Berlin, pp 21–36
6. Balas E, Fischetti M (2002) Polyhedral theory for the asymmetric traveling salesman problem. In: Gutin G, Punnen AP (eds) The traveling salesman problem and its variations. Kluwer, Dordrecht, pp 117–168
7. Baptiste P, Pape CL, Nuijten W (2001) Constraint-based scheduling: applying constraint programming to scheduling problems. Kluwer, Dordrecht
8. Beldiceanu N (2001) Pruning for the *minimum* constraint family and for the *number of distinct values* constraint family. In: Walsh T (ed) Principles and practice of constraint programming (CP 2001). Lecture notes in computer science, vol 2239. Springer, London, pp 211–224
9. Beldiceanu N, Carlsson M, Rampon JX (2009) Global constraint catalog. http://www.emn.fr/x-info/sdemasse/gccat/
10. Beldiceanu N, Contejean E (1994) Introducing global constraints in CHIP. Math Comput Model 12:97–123

11. Bessière C, Hebrard E, Hnich B, Kiziltan Z, Walsh T (2005) Filtering algorithms for the nvalue constraint. In: Barták R, Milano M (eds) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR 2005). Lecture notes in computer science, vol 3524. Springer, Berlin, pp 79–93
12. Bockmayr A, Pisaruk N (2003) Detecting infeasibility and generating cuts for mixed integer programming using constraint progrmaming. In: Gendreau M, Pesant G, Rousseau LM (eds) Proceedings of the international workshop on integration of artificial intelligence and operations research techniques in constraint programming for combintaorial optimization problems (CPAIOR 2003). Montréal, Canada
13. Bourdais S, Galinier P, Pesant G (2003) Hibiscus: a constraint programming application to staff scheduling in health care. In: Rossi F (ed) Principles and practice of constraint programming (CP 2003). Lecture notes in computer science, vol 2833. Springer, Berlin, pp 153–167
14. Caseau Y, Laburthe F (1997) Solving small TSPs with constraints. In: Naish L (ed) Proceedings, fourteenth international conference on logic programming (ICLP 1997), vol 2833. MIT, Cambridge, pp 316–330
15. Cheadle AM, Harvey W, Sadler AJ, Schimpf J, Shen K, Wallace MG (2003) ECLiPSe: a tutorial introduction. Technical Report IC-Parc-03-1, IC-Park, Imperial College London
16. Cheng BMW, Lee JHM, Wu JCK (1996) Speeding up constraint propagation by redundant modeling. In: Freuder EC (ed) Principles and practice of constraint programming (CP 1996). Lecture notes in computer science, vol 1118. Springer, Berlin, pp 91–103
17. Colombani Y, Heipcke S (2002) Mosel: an extensible environment for modeling and programming solutions. In: Jussien N, Laburthe F (eds) Proceedings of the international workshop on integration of artificial intelligence and operations research techniques in constraint programming for combinatorial optimization problems (CPAIOR 2002). Le Croisic, France, pp 277–290
18. Colombani Y, Heipcke S (2004) Mosel: an overview. white paper, DASH Optimization
19. Guéret C, Heipcke S, Prins C, Sevaux M (2000) Applications of optimization with Xpress-MP. White paper, Dash Optimization
20. Hellsten L, Pesant G, van Beek P (2004) A domain consistency algorithm for the stretch constraint. In: Wallace M (ed) Principles and practice of constraint programming (CP 2004). Lecture notes in computer science, vol 3258. Springer, Berlin, pp 290–304
21. Hentenryck PV, Michel L (2005) Constraint based local search. MIT, Cambridge
22. Hentenryck PV, Michel L, Perron L, Régin JC (1999) Constraint programming in OPL. In: International conference on principles and practice of declarative programming (PPDP 1999). Paris
23. van Hoeve WJ, Pesant G, Rousseau LM, Sabharwal A (2006) Revisiting the sequence constraint. In: Benhamou F (ed) Principles and practice of constraint programming (CP 2006). Lecture notes in computer science, vol 4204. Springer, Berlin, pp 620–634
24. Hooker JN (2000) Logic-based methods for optimization: combining optimization and constraint satisfaction. Wiley, New York
25. Hooker JN (2004) A hybrid method for planning and scheduling. In: Wallace M (ed) Principles and practice of constraint programming (CP 2004). Lecture notes in computer science, vol 3258. Springer, Berlin, pp 305–316
26. Hooker JN (2007) Integrated methods for optimization. Springer, Heidelberg
27. Hooker JN (2007) Planning and scheduling by logic-based Benders decomposition. Oper Res 55:588–602
28. Hooker JN (2009) A principled approach to mixed integer/linear problem formulation. In: Chinneck JW, Kristjansson B, Saltzman M (eds) Operations research and cyber-infrastructure (ICS 2009 proceedings). Springer, Berlin, pp 79–100
29. Hooker JN, Kim HJ, Ottosson G (2001) A declarative modeling framework that integrates solution methods. Ann Oper Res 104:141–161
30. Jain V, Grossmann IE (2001) Algorithms for hybrid MILP/CP models for a class of optimization problems. INFORMS J Comput 13:258–276
31. Jeroslow RG (1987) Representability in mixed integer programming, I: characterization results. Discrete Appl Math 17:223–243

32. Kaya LG, Hooker JN (2006) A filter for the circuit constraint. In: Benhamou F (ed) Principles and practice of constraint programming (CP 2006). Lecture notes in computer science, vol 4204. Springer, Berlin, pp 706–710
33. Kaya LG, Hooker JN (2008) The circuit polytope. Manuscript, Carnegie Mellon University
34. Keha AB, de Farias IR, Nemhauser GL (2004) Models for representing piecewise linear cost functions. Oper Res Lett 32:44–48
35. Kruk S (2009) Some facets of multiple alldifferent predicate. In: Belotti P (ed) Workshop on bound reduction techniques for constraint programming and mixed-integer nonlinear programming, at CPAIOR
36. Li HL, Lu HC, Huang CH, Hu NZ (2009) A superior representation method for piecewise linear functions. INFORMS J Comput 21:314–321
37. Magos D, Mourtos I, Appa G (2008) A polyhedral approach to the *alldifferent* system. Technical report, Technological Educational Institute of Athens
38. Maher MJ, Narodytska N, Quimper CG, Walsh T (2008) Flow-based propagators for the SEQUENCE and related global constraints. In: Stuckey PJ (ed) Principles and practice of constraint programming (CP 2008). Lecture notes in computer science, vol 5202. Springer, Heidelberg, pp 159–174
39. Marchand H, Martin A, Weismantel R, Wolsey L (2002) Cutting planes in integer and mixed integer programming. Discrete Appl Math 123:397–446
40. Michel L, Hentenryck PV (1999) Localizer: a modeling language for local search. INFORMS J Comput 11:1–14
41. Nethercote N, Stuckey PJ, Becket R, Brand S, Duck GJ, Tack G (2007) Minizinc: towards a standard CP modelling language. In: Bessière C (ed) Principles and practice of constraint programming (CP 2007). Lecture notes in computer science, vol 4741. Springer, Heidelberg, pp 529–543
42. Orman AJ, Williams HP (2006) A survey of different integer programming formulations of the travelling salesman problem. In: Kontoghiorghes EJ, Gatu C (eds) Optimisation, economics and financial analysis, advances in computational management science. Springer, Berlin, pp 933–106
43. Ottosson G, Thorsteinsson E, Hooker JN (2002) Mixed global constraints and inference in hybrid CLP-IP solvers. Ann Math Artif Intell 34:271–290
44. Padberg M (2000) Approximating separable nonlinear functions via mixed zero-one programs. Oper Res Lett 27:1–5
45. Pesant G (2001) A filtering algorithm for the stretch constraint. In: Walsh T (ed) Principles and practice of constraint programming (CP 2001). Lecture notes in computer science, vol 2239. Springer, Berlin, pp 183–195
46. Pesant G (2004) A regular language membership constraint for finite sequences of variables. In: Wallace M (ed) Principles and practice of constraint programming (CP 2004). Lecture notes in computer science, vol 3258. Springer, Berlin, pp 482–495
47. Quimper CG, López-Ortiz A, van Beek P, Golynski A (2004) Improved algorithms for the global cardinality constraint. In: Wallace M (ed) Principles and practice of constraint programming (CP 2004). Lecture notes in computer science, vol 3258. Springer, Berlin, pp 542–556
48. Régin JC (1996) Generalized arc consistency for *global cardinality* constraint. In: National conference on artificial intelligence (AAAI 1996). AAAI, Portland, pp 209–215
49. Régin JC (2004) Modeling problems in constraint programming. In: Tutorial presented at conference on Principles and Practice of constraint programming (CP 2004). Toronto
50. Rodošek R, Wallace M, Hajian M (1999) A new approach to integrating mixed integer programming and constraint logic programming. Ann Oper Res 86:63–87
51. Sherali HD (2001) On mixed-integer zero-one representations for separable lower-semi-continuous piecewise-linear functions. Oper Res Lett 28:155–160
52. Simonis H (2009) Modelling in CP: tutorial presented at CPAIOR 2009. http://4c.ucc.ie/hsimonis/slidescpaior.pdf
53. Stuckey PJ, de la Banda MG, Maher M, Marriott K, Slaney J, Somogyi Z, Wallace M, Walsh T (2005) The G12 project: mapping solver independent models to efficient solutions. In: van Beek P (ed) Principles and practice of constraint programming (CP 2005). Lecture notes in computer science, vol 3668. Springer, Berlin, pp 314–327

54. Tawarmalani M, Sahinidis NV (2002) Convexification and global optimization in continuous and mixed-integer nonlinear programming: theory, algorithms, software, and applications. Springer, Berlin
55. Tawarmalani M, Sahinidis NV (2004) Global optimization of mixed-integer nonlinear programs: a theoretical and computational study. Math Program 99:563–591
56. Vielma JP, Nemhauser GL (2008) Modeling disjunctive constraints with a logarithmic number of binary variables and constraints. In: Integer programming and combintorial optimization proceedings (IPCO 2008). Lecture notes in computer science, vol 5035, pp 199–213
57. Williams HP (1999) Model building in mathematical programming, 4th edn. Wiley, New York
58. Williams HP (2006) The formulation and solution of discrete optimization models. In: Appa G, Pitsoulis L, Williams HP (eds) Handbook on modelling for discrete optimization. Springer, Berlin, pp 3–38
59. Williams HP (2009) Logic and integer programming. Springer, Berlin
60. Williams HP, Brailsford SC (1997) The splitting of variables and constraints in the formulation of integer programming models. Eur J Oper Res 100:623–628
61. Williams HP, Yan H (2001) Representations of the all_different predicate of constraint satisfaction in integer programming. INFORMS J Comput 13:96–103
62. Yunes TH, Aron I, Hooker JN (2010) An integrated solver for optimization problems. Oper Res 58:342–356

Global Constraints: A Survey

Jean-Charles Régin

Abstract Constraint programming (CP) is mainly based on filtering algorithms; their association with global constraints is one of the main strengths of CP because they exploit the specific structure of each constraint. This chapter is an overview of these two techniques. A collection of the most frequently used global constraints is given and some filtering algorithms are detailed. In addition, we try to identify how filtering algorithms can be designed. At last, we identify some problems that deserve to be addressed in the future.

1 Introduction

Constraint Programming (CP) is mainly based on the exploitation of the structure of the constraints and CP accepts to have constraints whose structure are different. This idea seems to be exploited only in CP: we do not want to loose the structure of the constraints. Other techniques such as SAT or MIP impose to model the problem while respecting some rules: having only boolean variables and 3 clauses for SAT, or having only linear constraints for MIP.

This specificity of CP allows the use of any kind of algorithm for solving a problem. We could even say that we want to exploit as much as possible the capability to use different algorithms. Currently, when a problem is modelled in CP, it is possible that a large variety of algorithms are used at the same time and communicate with each other. It is really conceivable to have at the same time flow algorithms, dynamic programming, automaton transformations, etc.

J.-C. Régin (✉)
Université de Nice-Sophia Antipolis, I3S/CNRS, 2000, route des
Lucioles - Les Algorithmes - bt. Euclide B BP 121 - 06903 Sophia Antipolis, Cedex, France
e-mail: jcregin@gmail.com

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_3,
© Springer Science+Business Media, LLC 2011

1.1 CP Principles

In CP, a problem is defined from variables and constraints. Each variable is associated with a domain containing its possible values. A constraint expresses properties that have to be satisfied by a set of variables.

In CP, a problem can also be viewed as a conjunction of sub-problems for which we have efficient resolution methods. These sub-problems can be very easy like $x < y$ or complex like the search for a feasible flow. These sub-problems correspond to constraints. Then, CP uses for each sub-problem the available resolution method associated with it in order to remove from the domains the values that cannot belong to any solution of the sub-problem. This mechanism is called *filtering*. By repeating this process for each sub-problem, so for each constraint, the domains of the variables are going to be reduced.

After each modification of the variable domains, it is useful to reconsider all the constraints involving this variable, because that modification can lead to new deductions. In other words, the domain reduction of one variable may lead to deduce that some other values of some other variables cannot belong to a solution and so on. This mechanism is called *propagation*.

Then, and in order to reach a solution, the search space will be traversed by assigning successively a value to each variable. The filtering and propagation mechanisms are, of course, triggered when a modification occurs. Sometimes, an assignment may lead to the removal of all the values of a domain : we say that a failure occurs, and the latest choice is reconsidered: there is a backtrack and a new assignment is tried. This mechanism is called the *search*.

So, CP is based on three principles : filtering, propagation and search. We could represent it by reformulating the famous Kowalski's definition of Algorithm (Algorithm = Logic + Control) [78] as:

$$\text{CP} = \text{filtering} + \text{propagation} + \text{search} \tag{1}$$

where filtering and propagation correspond to Logic and Search to Control.

1.2 Global Constraints

One of the most interesting properties of a filtering algorithm is arc consistency. We say that a filtering algorithm associated with a constraint establishes arc consistency if it removes all the values of the variables involved in the constraint that are not consistent with the constraint. For instance, consider the constraint $x + 3 = y$ with the domain of x equals to $D(x) = \{1, 3, 4, 5\}$ and the domain of y equal to $D(y) = \{4, 5, 8\}$. Then, the establishing of arc consistency will lead to $D(x) = \{1, 5\}$ and $D(y) = \{4, 8\}$.

Since constraint programming is based on filtering algorithms, it is quite important to design efficient and powerful algorithms. Therefore, this topic caught the attention of many researchers, who discovered a large number of algorithms.

As we mentioned it, a filtering algorithm directly depends on the constraint it is associated with. The advantage of using the structure of a constraint can be emphasized on the constraint $x \leq y$. Let $\min(D)$ and $\max(D)$ be, respectively, the minimum and the maximum value of a domain. It is straightforward to establish that all the values of x and y in the range $[\min(D(x)), \max(D(y))]$ are consistent with the constraint. This means that arc consistency can be efficiently and easily established by removing the values that are not in the above ranges. Moreover, the use of the structure is often the only way to avoid memory consumption problems when dealing with non-binary constraints. In fact, this approach prevents you from explicitly representing all the combinations of values allowed by the constraint.

Thus, researchers interested in the resolution of real life applications with constraint programming, and notably those developing languages that encapsulate CP (like PROLOG), designed specific filtering algorithms for the most common simple constraints (like $=, \neq, <, \leq, \ldots$). They also developed general frameworks to exploit efficiently some knowledge about binary constraints (like AC-5 [150]). However, they have been confronted with two new problems: the lack of expressiveness of these simple constraints and the weakness of domain reduction of the filtering algorithms associated with these simple constraints. It is, indeed, quite convenient when modelling a problem in CP to have at one's disposal some constraints corresponding to a set of constraints. Moreover, these new constraints can be associated with more powerful filtering algorithms because they can take into account the simultaneous presence of simple constraints to further reduce the domains of the variables. These constraints encapsulating a set of other constraints are called *global constraints*.

Initially, global constraints were defined as a set of constraints having the same type for which an efficient algorithm were known. Then, this latter point has been relaxed.

One of the most famous examples is the ALLDIFF constraint, especially because the filtering algorithm associated with this constraint is able to establish arc consistency in a very efficient way.

An ALLDIFF constraint defined on X, a set of variables, states that the values taken by variables must be all different. This constraint can be represented by a set of binary constraints. In this case, a binary constraint of difference is built for each pair of variables belonging to the same constraint of difference. But the pruning effect of arc consistency for these constraints is limited. In fact, for a binary ALLDIFF constraint between two variables i and j, arc-consistency removes a value from domain of i only when the domain of j is reduced to a single value. Let us suppose we have a CSP with 3 variables x_1, x_2, x_3 and an ALLDIFF constraint involving these variables with $D(x_1) = \{a, b\}$, $D(x_2) = \{a, b\}$ and $D(x_3) = \{a, b, c\}$. Establishing arc consistency for this ALLDIFF constraint removes the values a and b from the domain of x_3, while arc-consistency for the ALLDIFF represented by binary constraints of difference does not delete any value. We will see later that the filtering algorithm associated with a global constraint is stronger than the conjunction of the independent filtering algorithms of the local constraints corresponding to the global constraint.

Fig. 1 An assignment timetable

	Mo	Tu	We	Th	...
peter	D	N	O	M	
paul	D	B	M	N	
mary	N	O	D	D	
...					

A = {M,D,N,B,O}, P = {peter, paul, mary, ...}
W = {Mo, Tu, We, Th, ...}
M: morning, D: day, N: night B: backup, O: day-off

We can further emphasize the advantage of global constraints on a more realistic example that involves global cardinality constraints (GCC).

A GCC is specified in terms of a set of variables $X = \{x_1, \ldots, x_p\}$ which take their values in a subset of $V = \{v_1, \ldots, v_d\}$. It constrains the number of times a value $v_i \in V$ is assigned to a variable in X to be in an interval $[l_i, u_i]$. GCCs arise in many real life problems. For instance, consider the example derived from a real problem and given in [37] (cf. Fig. 1). The task is to schedule managers for a directory-assistance center, with five activities (set A), seven persons (set P) over seven days (set W). Each day, a person can perform an activity from the set A. The goal is to produce an assignment matrix that satisfies the following global and local constraints:

- *General constraints* restrict the assignments. First, for each day we have a minimum and maximum number for each activity. Second, for each week, a person has a minimum and maximum number for each activity. Thus, for each row and each column of the assignment matrix, there is a global cardinality constraint.
- *Local constraints* mainly indicate incompatibilities between two consecutive days. For instance, a morning schedule cannot be assigned after a night schedule.

Each general constraint can be represented by as many min/max constraints as the number of involved activities. Now, these min/max constraints can be easily handled with, for instance, the **atmost/atleast** operators proposed in [149]. Such operators are implemented using local propagation. But as noted in [37]: "The problem is that efficient resolution of a timetable problem requires a global computation on the set of min/max constraints, and not the efficient implementation of each of them separately." Hence, this way is not satisfactory. Therefore, global cardinality constraints associated with efficient filtering algorithms (like ones establishing arc consistency) are needed.

In order to show the difference in global and local filtering, consider a GCC associated with a day (cf Fig. 2). The constraint can be represented by a bipartite graph called a value graph (left graph in Fig. 2). The left set corresponds to the person set, the right set to the activity set. There exists an edge between a person and an activity when the person can perform the activity. For each activity, the numbers between parentheses express the minimum and the maximum number of times the activity has to be assigned. For instance, John can work in the morning or in the day but

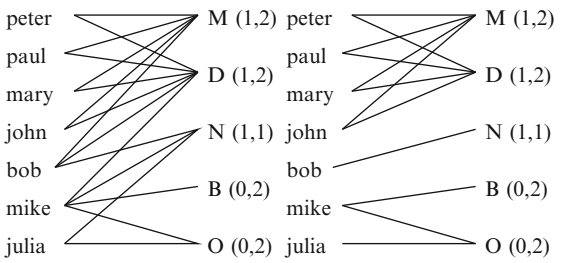


Fig. 2 An example of a global cardinality constraint (GCC)

not in the night; one manager is required to work in the morning, and at most two managers work in the morning. We recall that each person has to be associated with exactly one activity.

Encoding the problem with a set of atmost/atleast constraints leads to no deletion. Now, we can carefully study this constraint. Peter, Paul, Mary, and John can work only in the morning and during the day. Moreover, morning and day can be assigned together to at most four persons. Thus, no other persons (i.e. Bob, Mike, nor Julia) can perform activities M and D. So we can delete the edges between Bob, Mike, Julia and D, M. Now, only one possibility remains for Bob: N, which can be assigned at most once. Therefore, we can delete {mike,N} and {julia,N} edges. This reasoning leads to the right graph in Fig. 2. It corresponds to the establishing of arc consistency for the global constraint.

Filtering is a local mechanism because it is associated to each constraint independently. If a constraint is decomposed into some other constraints, then the set of filtering are less efficient because they have less information. We can formally emphasize this idea by the following property:

Property 1. *The establishing of arc consistency on* $\mathcal{C}=\wedge\{C_1, C_2, \ldots, C_n\}$*, the conjunction of the constraints* $C1, C2, \ldots C_n$ *is stronger (that is, cannot remove fewer values) than the establishing of arc consistency of the network* $(\cup_{C\in\mathcal{C}} X(C), \mathcal{D}_{X(\mathcal{C})}, \{C_1, C_2, \ldots, C_n\})$.

Proof. The set of tuples of $\mathcal{C} = \wedge\{C_1, C_2, \ldots, C_n\}$ corresponds to the set of solution of $(\cup_{C\in\mathcal{C}} X(C), \mathcal{D}_{X(\mathcal{C})}, \{C_1, C_2, \ldots, C_n\})$. Therefore, the establishing of arc consistency of $\wedge\{C_1, C_2, \ldots, C_n\}$ removes all the values that do not belong to a solution of $(\cup_{C\in\mathcal{C}} X(C), \mathcal{D}_{X(\mathcal{C})}, \{C_1, C_2, \ldots, C_n\})$ which is stronger than the arc consistency of the previous network. □

Therefore, arc consistency on global constraints is a strong property. The following proposition is an example of the gap between arc consistency for a global constraint and arc consistency for the network corresponding to this global constraint.

Property 2. *Arc Consistency for* $C=\text{ALLDIFF}(X)$ *corresponds to the arc consistency of a Constraint Network with an exponential number of constraints defined by:*

$\forall A \subseteq X: |D(A)| = |A| \Rightarrow D(X-A)$ *is reduced to* $D(X)-D(A)$*, where* $D(A)$ *is the union of domain variable of A.*

Proof. Hall's theorem states that an assignment problem in the bipartite graph $G = (X, Y, E)$ has a solution covering X if and only if $\forall A \subseteq X \ |A| \leq |\Gamma(A)|$. In addition, we can easily prove that if there is $A \subseteq X$ such that $|A| = |\Gamma(A)|$, then no element of $X - A$ can be assigned to an element of $\Gamma(A)$. Thus, by defining for each subset A of X the constraint $|D(A)| = |A| \Rightarrow D(X - A)$ is reduced to $D(X) - D(A)$, we establish arc consistency of the constraint. □

However, in practice, it is possible to observe results that are not so marked. We can emphasize this idea on the following graph colouring problem: choose colours for the nodes of a graph so that adjacent nodes are not the same color. The kind of graph that we will colour is one with $n * (n+1)/2$ nodes, where n is odd and where every node belongs to exactly two maximal cliques of size n.

For example, for $n = 5$, there is a graph consisting of the following maximal cliques:

$$c0 = \{0, 1, 2, 3, 4\},\ c1 = \{0, 5, 6, 7, 8\},\ c2 = \{1, 5, 9, 10, 11\},$$
$$c3 = \{2, 6, 9, 12, 13\},\ c4 = \{3, 7, 10, 12, 14\},\ c5 = \{4, 8, 11, 13, 14\}$$

The minimum number of colours needed for this graph is n since there is a clique of size n. Consequently, our problem is to find out whether there is a way to color such a graph in n colours.

We compare the results obtained with the ALLDIFF constraint and without it (that is only binary constraints of difference are used). Times are expressed in seconds:

	Clique size							
	27		31		51		61	
	#fails	time	#fails	time	#fails	time	#fails	time
binary $\neq$	1	0.17	65	0.37	24512	66.5	?	$>6\,h$
ALLDIFF	0	1.2	4	2.2	501	25.9	5	58.2

These results show that using global constraints establishing arc consistency is not systematically worthwhile when the size of the problem is small, even if the number of backtracks is reduced. However, when the size of problem is increased, efficient filtering algorithms are needed.

Thus, we can recapitulate some strong advantages of global constraints:

- Expressiveness: it is more convenient to define one constraint corresponding to a set of constraints than to define independently each constraint of this set.
- Powerful filtering algorithms can be designed because the set of constraints can be taken into account as a whole. Specific filtering algorithms make it possible to use Operations Research techniques or graph theory.

A lot of global constraints have been developed. Simonis proposed the first state of the art [141], Régin wrote a book chapter about them [117], Beldiceanu defined a catalogue [9] which tries to be exhaustive and van Hoeve and Katriel gave a recent presentation of some of them [63]. In this chapter, we try to present the most important ones by considering the number of applications, the number of references

or the number of papers that are dedicated to them. We do not claim that we are totally objective, because we also speak about the constraints we know the best.

This chapter is organized as follows. First, we recall some preliminaries about Constraint Programming. Then, we present a general arc consistency algorithm, and we recall the complexity of table constraints. We also explain how flow can be computed and detailed some flow properties that are useful to build filtering algorithms. Then, we propose a collection of global constraints based on the constraint type. For most of them, we explain the ideas on which the filtering algorithms are based. Next, we consider the designs of filtering algorithm and we try to identify some general principles. Before concluding, we look at some problems that deserve to be addressed in the future.

2 Preliminaries and Notations

A finite **constraint network** $\mathcal{N}$ is defined as a set of n **variables** $X = \{x_1, \ldots, x_n\}$, a set of current **domains** $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ where $D(x_i)$ is the finite set of possible **values** for variable x_i, and a set $\mathcal{C}$ of **constraints** between variables. We denote by $\mathcal{D}_0 = \{D_0(x_1), \ldots, D_0(x_n)\}$ the set of initial domains of $\mathcal{N}$. Indeed, we consider that any constraint network $\mathcal{N}$ can be associated with an initial domain $\mathcal{D}_0$ (containing $\mathcal{D}$), on which constraint definitions were stated.

A **constraint** C on the ordered set of variables $X(C) = (x_1, \ldots, x_r)$ is a subset $T(C)$ of the Cartesian product $D_0(x_1) \times \cdots \times D_0(x_r)$ that specifies the **allowed** combinations of values for the variables $x_1, \ldots, x_r$. An element of $D_0(x_1) \times \cdots \times D_0(x_r)$ is called a **tuple on** $X(C)$. $|X(C)|$ is the **arity** of C.

We will use the following notations:

- (x, a) denotes the value a of the variable x
- $\text{var}(C, i)$ represents the ith variable of $X(C)$
- $\text{index}(C, x)$ is the position of variable x in $X(C)$
- $\tau[k]$ denotes the kth value of the tuple τ
- $\tau[x]$ represents $\tau[\text{index}(C, x)]$ when no confusion is possible
- $D(X)$ denotes the union of domains of variables of X (i.e. $D(X) = \cup_{x \in X} D(x)$)
- $\#(a, \tau)$ is the number of occurrences of the value a in the tuple τ

Let C be a constraint. Here are some definitions:

- A tuple τ on $X(C)$ is **valid** if $\forall (x, a) \in \tau, a \in D(x)$.
- C is **consistent** if there exists a tuple τ of $T(C)$ which is valid.
- A tuple τ of $T(C)$ involving (x, a) (that is with $a = \tau[\mathit{index}(C, x)]$) is called a **support** for (x, a) on C.
- A value $a \in D(x)$ is **consistent with** C if $x \notin X(C)$ or there exists a valid support for (x, a) on C (i.e. a valid tuple τ with $(x, a) \in \tau$).
- A constraint is **arc consistent** if $\forall x \in X(C), D(x) \neq \emptyset$ and $\forall a \in D(x), a$ is consistent with C.

A **filtering algorithm** associated with a constraint C is an algorithm which removes some values that are inconsistent with C; and that does not remove any consistent values. If the filtering algorithm removes all the values inconsistent with C, we say that it establishes the arc consistency of C, or that C is domain consistent.

The **propagation** is the mechanism that consists of calling the filtering algorithm associated with the constraints involving a variable x each time the domain of this variable is modified. Note that if the domains of the variables are finite, then this process terminates because a domain can be modified only a finite number of times.

We introduce a theorem that will be useful in this chapter. This theorem is based on hypergraph and is due to [67]. Unfortunately, we were not able to find the original paper. Thus, we propose to reformulate it in a simpler form which is easier to understand.

First, we recall the definition of Bipartite Constraint Graph introduced by Jegou [68].

Definition 1 (Jegou). Let $\mathcal{C}$ be a set of constraints. The **bipartite constraint graph** of $\mathcal{C}$ is the bipartite graph $BCG(\mathcal{C}) = (X_B, Y_B, E_B)$ where X_B, Y_B are node sets and E_B an edge set defined as follows:

- Each constraint $C \in \mathcal{C}$ is associated with a node y_i
- $X_B = \cup_{C \in \mathcal{C}} X(C)$
- $Y_B = \{y_i \text{ s.t. } C_i \text{ is associated with } y_i\}$
- $E_B = \{\{x_i, y_j\} \text{ s.t. } x_i \in X_B, y_j \in Y_B \text{ and } x_i \in X(C_j)\}$

Then, we have the expected theorem

Theorem 1 (Janssen and Villarem). *Let $\mathcal{C}$ be a set of constraints. If the bipartite constraint graph of $\mathcal{C}$ has no cycle, then establishing arc consistency for the constraint network $N = (\cup_{C \in \mathcal{C}} X(C), \mathcal{D}_{X(\mathcal{C})}, \mathcal{C})$ is equivalent to establish arc consistency for the constraint defined by the conjunction of the constraints of $\mathcal{C}$ that is $\wedge\{C_1, C_2, \ldots, C_n\}$.*

Proof. Clearly, if a value (x, a) is consistent with $\wedge\{C_1, C_2, \ldots, C_n\}$, then it is also consistent with N. On the other hand, the bipartite constraint graph of N has no cycle, and therefore, two constraints have at most one variable in common and so $\mathcal{C}$ can be represented by disjoint paths, each one of the form $C_1, x_1, C_2, x_2, \ldots, x_{m-1}, C_m$ where x_i are variables and C_i constraints of $\mathcal{C}$ and for each constraint C_i, $i = 1 \ldots (m-1)$: $X(C_i) \cap X(C_{i+1}) = \{x_i\}$ and for each pair of constraints C_i and C_j: $|X(C_i) \cap X(C_j)| \leq 1$ and C_1 may only share a variable with C_2 and C_m may only share a variable with C_{m-1}. If a value (x_i, a) of this path is consistent with the two constraints in which it can be involved, then it is straightforward to extend its support to a complete assignment of all the variables and so (x_i, a) is consistent with the conjunction of constraints $\wedge\{C_1, C_2, \ldots, C_n\}$ □

We have an immediate useful corollary

Corollary 1. *Let C be a constraint and $(X(C), \mathcal{D}_{X(C)}, \{C_1, \ldots, C_m\})$ be a constraint network equivalent to C. If the bipartite constraint graph of the constraint set of this network has no cycle, then establishing arc consistency of C is equivalent to establish arc consistency of the constraint network.*

3 Global Constraints Collection

We can identify at least five different categories of global constraints:

- *Classical Constraints*. This category contains all usual constraints, such as ALLDIFF, GCC, REGULAR, SEQUENCE, PATH, etc.
- *Weighted Constraints*. This category contains constraints which are associated with some costs, such as the cardinality with cost (COST-GCC), the SHORTER PATH, the KNAPSACK, BIN-PACKING, etc. Usually, a summation is implied and there is a limit on it. A lot of NP-Hard constraint are in this category. The name of these constraints often contains "weighted", "cost based", "with cost", etc.
- *Soft Constraints*. This category contains the relaxation of classical or weighted constraints when they cannot be satisfied. In general, the soft version of a constraint involves an additional cost variable which measures the distance to the satisfaction. Formally, these constraints have been introduced by Petit et al. [101], and the well known local search based language Comet is mainly based on these constraints [151].
- *Constraints on Global Variables*. This category contains the constraints that are not defined on classical variable, but rather on set variables or on graph variables. Set variables have been proposed independently by Gervet [55] and Puget [103]. Régin implemented global constraints on set variables in ILOG Solver [120]. The HDR thesis of Gervet [57] contains a lot of interesting ideas and is certainly the best reference on this topic. Some information can be found in [56, 58, 128, 129, 153].
- *Open Constraints*. This category is new and has been proposed by van Hoeve and Régin [63]. The idea is to define constraint on set of variables that are not close at the definition. More precisely, we do not know exactly the variables that will be involved in the constraint: we only know that some variables are involved and that some others could be involved. van Hoeve and Régin presented an efficient AC filtering algorithm for some open global cardinality constraints and extended this result to conjunctions of them, in case they are defined on disjoint sets of variables.[1] Maher studied some variations of the model proposed by van Hoeve and Régin [89].

In this chapter, we consider only constraints belonging to the two first categories: the classical and weighted constraints.

[1] van Hoeve and Régin gave an example of a scheduling alternative: consider a set of activities and suppose that each activity can be processed either on the factory line 1 formed by the set of unary resources R1, or on the factory line 2 formed by the set of unary resources R2. Thus, at the beginning, the set of resources that will be used by an activity is not known. Also the set of activities that will be processed by a resource is not known. However, it is useful to express that the activities that will be processed on each line must be pairwise different. This can be done by defining two ALLDIFF constraints, involving the start variables of each activity, and by stating that a start variable will be involved in exactly one ALLDIFF constraint. van Hoeve and Régin showed how arc consistency can be efficiently establish for the conjunction of these 2 alldiff constraints.

We can identify two main groups among these constraints: the constraints that are mainly defined by their functions and the constraints that are defined by the underlined technique they use. This latter group corresponds to *Formal Language based Constraints*, which mainly contains REGULAR and GRAMMAR constraints, whereas the former group contains several types of constraints:

- *Solution based Constraints*. It mainly contains constraints that are defined from any problem P and TABLE constraints.
- *Counting Constraints*. It mainly contains: ALLDIFF, PERMUTATION, global cardinality (GCC), global cardinality with cost (COST-GCC) and cardinality matrix constraints (CARD-MATRIX).
- *Balancing Constraints*. It mainly contains: BALANCE, DEVIATION and SPREAD constraints.
- *Constraint Combination based Constraints*. It mainly contains: MAX-SAT, OR and AND constraints.
- *Sequencing Constraints*. It mainly contains: AMONG, SEQUENCE, generalized sequence (GEN-SEQUENCE), and global sequencing constraints (GSC).
- *Distance Constraints*. It mainly contains: INTER-DISTANCE and sum of inequalities constraints (SUM-INEQ).
- *Geometric Constraints*. It mainly contains: DIFF-N constraints.
- *Summation based Constraints*. It mainly contains: SUBSET-SUM and KNAPSACK constraints.
- *Packing Constraints*. It mainly contains: symmetric alldiff (SYM-ALLDIFF), STRETCH, K-DIFF, number of distinct value (NVALUE), and BIN-PACKING constraints.
- *Graph based Constraints*. It mainly contains: CYCLE, PATH, TREE, and weighted spanning tree (WST) constraints.
- *Order based Constraints*. It mainly contains: lexicographic (LEXICO$\leq$) and SORT constraints.

First, we will consider individually each type of these constraints and then we will study the formal language based constraints.

3.1 Solution Based Constraints

Often when we are solving a real problem, the various simple models that we come up with cannot be solved within a reasonable period of time. In such a case, we may consider a sub-problem of the original problem, say P. Then, we build a *global* constraint that is the conjunction of the constraints involved in that sub-problem.

The main issue is to define an efficient filtering algorithm associated with P. This task can be difficult. There are several ways to try to solve it, and we will discuss this question in Sect. 4 of this chapter. However, we will focus our attention here on two possibilities:

- A generic algorithm is used.
- Or all the solutions of P are enumerated and a Table constraint is used.

3.1.1 Generic Constraint (GENERIC)

Suppose that you are provided with a function, denoted by EXISTSOLUTION(P), which is able to know whether a particular problem $P = (X, \mathcal{C}, \mathcal{D})$ has a solution or not. In this section, we present two general filtering algorithms establishing arc consistency for the constraint corresponding to P, that is the global constraint $C(P) = \wedge\mathcal{C}$.

These filtering algorithms correspond to particular instantiations of a more general algorithm: GAC-Schema [29].

For convenience, we denote by $P_{x=a}$ the problem P in which the domain of X is restricted to a, in other words $P_{x=a} = (X, \mathcal{C} \cup \{x = a\}, \mathcal{D})$.

Establishing arc consistency on $C(P)$ is done by looking for supports for the values of the variables in X. A support for a value (x, a) on $C(P)$ can be searched by using any search procedure since a support for (x, a) is a solution of problem $P_{x=a}$.

A First Algorithm

A simple algorithm consists of calling the function EXISTSOLUTION with $P_{x=a}$ as a parameter for every value a of every variable x involved in P, and then to remove the value a of x when EXISTSOLUTION$(P_{x=a})$ has no solution. Algorithm 1 is a possible implementation.

This algorithm is quite simple but it is not efficient because each time a value will be removed, the existence of a solution for all the possible assignments needs to be recomputed.

If $O(P)$ is the complexity of function EXISTSOLUTION(P) then we can recapitulate the complexity of this algorithm as follows:

	Consistency checking		Establishing arc consistency	
	Best	Worst	Best	Worst
From scratch	$\Omega(P)$	$O(P)$	$nd \times \Omega(P)$	$nd \times O(P)$
After k Modifications	$k \times \Omega(P)$	$k \times O(P)$	$knd \times \Omega(P)$	$knd \times O(P)$

Algorithm 1: Simple general filtering algorithm establishing arc consistency

```
SIMPLEGENERALFILTERINGALGORITHM(C(P): constraint; deletionSet: list): Bool
for each x ∈ X do
    for each a ∈ D(x) do
        if ¬ EXISTSOLUTION(P_{x=a}) then
            remove a from D(x)
            if D(x) = ∅ then return False
            add (x, a) to deletionSet
return True
```

A Better General Algorithm

This section shows how a better general algorithm establishing arc consistency can be designed provided that function EXISTSOLUTION(P) returns a solution when there is one instead of being Boolean.

First, consider that a value (x, a) has been removed from $D(x)$. We must study the consequences of this deletion. So, for all the values (y, b) that were supported by a tuple containing (x, a) another support must be found. The list of the tuples containing (x, a) and supporting a value is the list $S_C(x, a)$; and the values supported by a tuple τ is given by $S(\tau)$.

Therefore, Line 1 of Algorithm 2 enumerates all the tuples in the S_C list and Line 2 enumerates all the values supported by a tuple. Then, the algorithm tries to find a new support for these values either by "inferring" new ones (Line 3) or by explicitly calling function EXISTSOLUTION (Line 4).

Here is an example of this algorithm: Consider $X = \{x_1, x_2, x_3\}$ and $\forall x \in X$, $D(x) = \{a, b\}$; and $T(C(P)) = \{(a, a, a), (a, b, b), (b, b, a), (b, b, b)\}$ (i.e. these are the possible solutions of P.)

Algorithm 2: function GENERALFILTERINGALGORITHM

```
   GeneralFilteringAlgorithm(C(P): constraint; x: variable; a: value, deletionSet:
   list): Bool
   // S_C(x,a): list of tuples supported by (x,a)
   // S(τ) : list of values supported by the tuple τ
   // this function studies the consequence of the deletion of the value a of D(x)
1  for each τ ∈ S_C(x,a) do
       for each (z,c) ∈ τ do remove τ from S_C(z,c)
2      for each (y,b) ∈ S(τ) do
           // (x,a) was the valid support of (y,b)
           remove (y,b) from S(τ)
           if b ∈ D(y) then
               // we search for another valid support for (y,b)
               // first by inference
3              σ ← SEEKINFERABLESUPPORT(y,b)
               if σ ≠ nil then add (y,b) to S(σ)
               else
                   // second we explicitly check if P has a solution when y = b
4                  σ ← EXISTSOLUTION(P_{y=b})
                   if σ ≠ nil then
                       // a valid support is found
                       add (y,b) to S(σ)
                       for k = 1 to |X(C)| do add σ to S_C(var(C(P),k), σ[k])
                   else
                       // there is no valid support : b is deleted from D(y)
                       remove b from D(y)
                       if D(y) = ∅ then return False
                       add (y,b) to deletionSet
   return True
```

First, a support for (x_1, a) is sought: (a, a, a) is computed and (a, a, a) is added to $S_C(x_2, a)$ and $S_C(x_3, a)$, (x_1, a) in (a, a, a) is added to $S((a, a, a))$. Second, a support for (x_2, a) is sought: (a, a, a) is in $S_C(x_2, a)$ and it is valid, therefore it is a support. There is no need to compute another solution.

Then, a support is sought all the other values. Now, suppose that value a is removed from x_2, then all the tuples in $S_C(x_2, a)$ are no longer valid: (a, a, a) for instance. The validity of the values supported by this tuple must be reconsidered, that is the ones belonging to $S((a, a, a))$, so a new support for (x_1, a) must be sought and so on.

The programme which aims to establish arc consistency for $C(P)$ must create and initialize the data structures (S_C, S), and call function GENERALFILTERINGALGORITHM$(C(P), x, a, \mathit{deletionSet})$ (see Algorithm 2) each time a value a is removed from a variable x involved in $C(P)$, in order to propagate the consequences of this deletion. *deletionSet* is updated to contain the deleted values not yet propagated. S_C and S must be initialized in a way such that:

- $S_C(x, a)$ contains all the allowed tuples τ that are the current support for some value, and such that $\tau[\text{index}(C(P), x)] = a$.
- $S(\tau)$ contains all values for which τ is the current support.

Function SEEKINFERABLESUPPORT of GENERALFILTERINGALGORITHM "infers" an already checked allowed tuple as support for (y, b) if possible, in order to ensure that it never looks for a support for a value when a tuple supporting this value has already been checked. The idea is to exploit the property: "If (y, b) belongs to a tuple supporting another value, then this tuple also supports (y, b)". Therefore, elements in $S_C(y, b)$ are good candidates to be a new support for (y, b). Algorithm 3 is a possible implementation of this function.

The complexity of the GENERALFILTERINGALGORITHM is given in the following table:

	Consistency checking		Establishing Arc consistency	
	best	worst	best	worst
From scratch	$\Omega(P)$	$O(P)$	$nd \times \Omega(P)$	$nd \times O(P)$
After k modifications	$\Omega(1)$	$k \times O(P)$	$nd \times \Omega(P)$	$knd \times O(P)$

Moreover, the space complexity of this algorithm is $O(n^2 d)$, where d is the size of the largest domain and n is the number of variables involved in the constraint.

Algorithm 3: function SEEKINFERABLESUPPORT

```
SEEKINFERABLESUPPORT(y: variable, b: value): tuple
// we search whether (y, b) belong to a valid tuple supporting another value
while S_C(y, b) ≠ ∅ do
    σ ← first(S_C(y, b))
    if σ is valid then return σ    /* σ is a support */
    else remove σ from S_C(y, b)
return nil
```

This space complexity depends on the number of tuples needed to support all the values. Since there are nd values and only one tuple is required per value, we obtain the above complexity.

3.1.2 Table Constraint (TABLE)

A TABLE constraint is a constraint defined explicitly by the list of allowed tuples or by the list of forbidden tuples. This constraint is one of the most useful constraints. GAC-Schema or one of its recent variations [54, 76, 85, 87] can be used for establishing arc consistency.

We would like to emphasize the complexity of such an algorithm because we will reuse it several times in this chapter. Consider for instance a TABLE constraint involving r variables and a tuple set containing T elements. For every value a of every variable x, we can define $T(x,a)$ the subset of T containing the tuples involving (x,a). For a given variable x, all these lists are disjoint and the sum of their size is bounded by $|T|$. Searching for a valid support for (x,a) is equivalent to find a valid tuple in $T(x,a)$. We can use a method which do not repeat several time the same validity check of an element of $T(x,a)$. Therefore, all the searches for a valid support for (x,a) can be done with at most $|T(x,a)|$ validity checks. So, for one variable, the overall cost for all the values in the domain is $r|T|$, where r is the cost of one validity check because the $T(x,a)$ sets are disjoint. We can amortize the cost of validity checks: if we discover that a tuple is no longer valid, then we can remove it from each of the set $T(x,a)$ for each value (x,a) it contains. This does not cost more than checking the validity of the tuple once. However, this means that globally a tuple can be checked unvalid only once, so globally the cost of all checks is in $O(r|T|)$. This number is also the maximum number of time the support of a value of x can be no longer valid. Therefore, the time complexity for establishing arc consistency and for maintaining it for one branch of the tree search is in $O(r|T|)$:

Proposition 1. *Let C be a* TABLE *constraint involving r variables and defined by the set T of allowed tuples. The time complexity for establishing arc consistency for C and for maintaining it for one branch of the tree search is in $O(r|T|)$.*

3.2 Counting Constraints

Counting constraints ensure rules defined on the number of times values are taken in any solution. These constraints express conditions that are strongly related to assignment problems that can be solved by the flow theory. Therefore, we suggest to consider first the Flow theory and then to present the counting constraints and the way their filtering algorithms are based on flow theory. This presentation also clearly shows how OR algorithms can be integrated into CP.

3.2.1 Flow Theory

Preliminaries

The definitions about graph theory come from [146]. The definitions, theorems and algorithms about flow are based on [2, 26, 82, 146].

A **directed graph** or **digraph** $G = (X, U)$ consists of a **node set** X and an **arc set** U, where every arc (u, v) is an ordered pair of distinct nodes. We denote by $X(G)$ the node set of G and by $U(G)$ the arc set of G.

A **path** from node v_1 to node v_k in G is a list of nodes $[v_1, \ldots, v_k]$ such that (v_i, v_{i+1}) is an arc for $i \in [1 \ldots k-1]$. The path **contains** node v_i for $i \in [1 \ldots k]$ and arc (v_i, v_{i+1}) for $i \in [1 \ldots k-1]$. The path is **simple** if all its nodes are distinct. The path is a **cycle** if $k > 1$ and $v_1 = v_k$.

If $\{u, v\}$ is an edge of a graph, then we say that u and v are the **ends** or the **extremities** of the edge. A **matching** M on a graph is a set of edges and no two of which have a common node. The **size** $|M|$ of M is the number of edges it contains. The **maximum matching problem** is that of finding a matching of maximum size. M **covers** X when every node of X is an endpoint of some edge in M.

Let G be a graph for which each arc (i, j) is associated with two integers l_{ij} and u_{ij}, respectively, called the **lower bound capacity** and the **upper bound capacity** of the arc.

A **flow** in G is a function f satisfying the following two conditions:

- For any arc (i, j), f_{ij} represents the amount of some commodity that can "flow" through the arc. Such a flow is permitted only in the indicated direction of the arc, i.e., from i to j. For convenience, we assume $f_{ij} = 0$ if $(i, j) \notin U(G)$.
- A **conservation law** is observed at each node: $\forall j \in X(G) : \sum_i f_{ij} = \sum_k f_{jk}$.

We will consider two problems of flow theory:

- **The feasible flow problem**: Does there exist a flow in G that satisfies the **capacity constraint**? That is, find f such that $\forall (i, j) \in U(G)\ l_{ij} \leq f_{ij} \leq u_{ij}$.
- **The problem of the maximum flow for an arc** (i, j): Find a feasible flow in G for which the value of f_{ij} is maximum.

Without loss of generality (see p. 45 and p. 297 in [2]), and to overcome notation difficulties, we will consider that:

- If (i, j) is an arc of G, then (j, i) is not an arc of G.
- All boundaries of capacities are non-negative integers.
  In fact, if all the upper bounds and all the lower bounds are integers and if there exists a feasible flow, then for any arc (i, j) there exists a maximum flow from j to i which is integral on every arc in G (See [82] p. 113.)

Flow Computation

Consider, for instance, that all the lower bounds are equal to zero and suppose that you want to increase the flow value for an arc (i, j). In this case, the flow of zero

on all arcs, called the **zero flow**, is a feasible flow. Let P be a path from j to i different from $[j, i]$, and $val = min(\{u_{ij}\} \cup \{u_{pq} \text{ s.t. } (p, q) \in P\})$. Then, we can define the function f on the arcs of G such that $f_{pq} = val$ if P contains (p, q) or $(p, q) = (i, j)$ and $f_{pq} = 0$ otherwise. This function is a flow in G. (The conservation law is obviously satisfied because (i, j) and P form a cycle.) We have $f_{ij} > 0$, hence it is easy to improve the flow of an arc when all the lower bounds are zero and when we start from the zero flow. It is, indeed, sufficient to find a path satisfying the capacity constraint.

The main idea of the basic algorithms of flow theory is to proceed by successive modifications of flows that are computed in a graph in which all the lower bounds are zero and the current flow is the zero flow. This particular graph can be obtained from any flow and is called the residual graph:

Definition 2. The **residual graph** for a given flow f, denoted by $R(f)$, is the digraph with the same node set as in G. The arc set of $R(f)$ is defined as follows: $\forall (i, j) \in U(G)$:

- $f_{ij} < u_{ij} \Leftrightarrow (i, j) \in U(R(f))$ and upper bound capacity $r_{ij} = u_{ij} - f_{ij}$.
- $f_{ij} > l_{ij} \Leftrightarrow (j, i) \in U(R(f))$ and upper bound capacity $r_{ji} = f_{ij} - l_{ij}$.

All the lower bound capacities are equal to 0.

Figures 3 and 4 are examples of flow and residual graph of the example given in Fig. 2.

Instead of working with the original graph G, we can work with the residual graph $R(f^o)$ for some f^o. From f' a flow in $R(f^o)$, we can obtain f another flow in G defined by: $\forall (i, j) \in U(G) : f_{ij} = f^o_{ij} + f'_{ij} - f'_{ji}$. And from a path in $R(f^o)$ we can define a flow f' in $R(f^o)$ and so a flow in G:

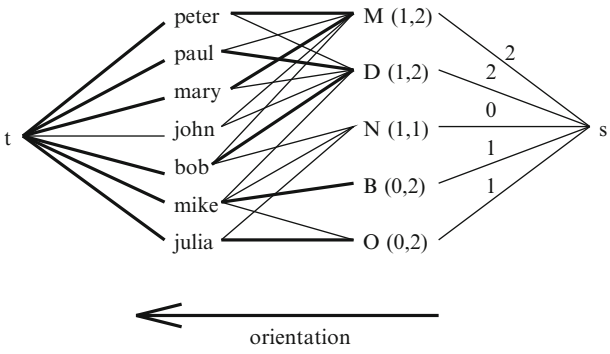


Fig. 3 A flow from s to t. For convenience, the arc (t, s) is omitted. All arcs have a minimum capacity of 0 and a maximum capacity of 1, excepted the outgoing arcs from s where the capacities are given in parenthesis. For instance, the arc (s, D) has a minimum capacity equals to 1 and a maximum equals to 2

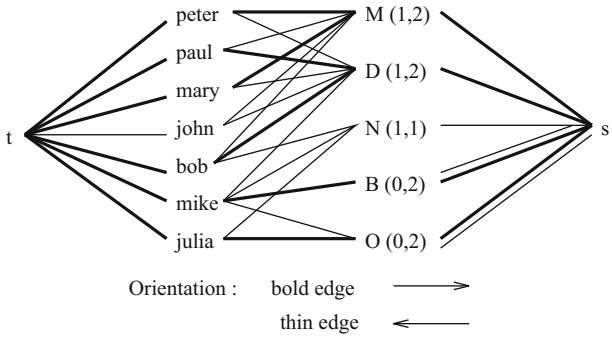


Fig. 4 The residual graph for the flow given in Fig. 3. For convenience, the arc (t, s) and (s, t) are omitted. All arcs have a minimum capacity of 0 and a maximum capacity of 1

Definition 3. We will say that f **is obtained from** f^o **by sending** k **units of flow along a path** P **from** j **to** i if:

- P is a path in $R(f^o) - \{(j, i)\}$
- $k \leq \min(\{r_{ij}\} \cup \{r_{uv} s.t. (u, v) \in P\})$
- f corresponds in $R(f^o)$ to the flow f' defined by:
  - $f'_{pq} = k$ for each arc $(p, q) \in P \cup \{(i, j)\}$
  - $f'_{pq} = 0$ for all other arcs.

If k is not mentioned, it will be assumed that $k = min(\{r_{ij}\} \cup \{r_{uv} s.t. (u, v) \in P\})$.

In the previous definition, the path must be different from $[j, i]$, otherwise f' will be the zero flow.

The following proposition shows that the existence of a path in the residual graph is a necessary and sufficient condition:

Theorem 1. *Let f^o be any feasible flow in G, and (i, j) be an arc of G.*

- *There is a feasible flow f in G with $f_{ij} > f^o_{ij}$ if and only if there exists a path from j to i in $R(f^o) - \{(j, i)\}$.*
- *There is a feasible flow f in G with $f_{ij} < f^o_{ij}$ if and only if there exists a path from i to j in $R(f^o) - \{(i, j)\}$.*

Proof. see [82] p. 112. □

Maximum Flow Algorithm

Theorem 1 gives a way to construct a maximum flow in an arc (i, j) by iterative improvement, due to Ford and Fulkerson:

Begin with any feasible flow f^0 and look for a path from j to i in $R(f^0) - \{(j, i)\}$. If there is none, f^0 is maximum. If, on the other hand, we find such a path P, then define f^1 obtained from f^0 by sending flow along P. Now, look for a

path from j to i in $R(f^1) - \{(j,i)\}$ and repeat this process. When there is no such path for f^k, then f^k is a maximum flow.

A path can be found in $O(m)$, thus we have[2]:

Property 3. *A maximum flow of value v in an arc (i, j) can be found from a feasible flow in $O(mv)$.*

Feasible flow algorithm

For establishing a feasible flow, several methods exist. For instance, it is possible to transform this problem into one in which all the lower bounds capacities are equal to zero and searching for a particular maximum flow value for one arc. (See [2] p. 169.) However, there is a simple method which repeatedly searchs for maximum flows in some arcs:

Start with the zero flow f^o. This flow satisfies the upper bounds. Set $f = f^o$, and apply the following process while the flow is not feasible:

(1) pick an arc (i, j) such that f_{ij} violates the lower bound capacity in G (i.e. $f_{ij} < l_{ij}$).
(2) Find P a path from j to i in $R(f) - \{(j,i)\}$.
(3) Obtain f' from f by sending flow along P; set $f = f'$ and goto (1)

If, at some point, there is no path for the current flow, then a feasible flow does not exist. Otherwise, the obtained flow is feasible.

Property 4. *Let k_{ij} be the infeasibility number w.r.t. the zero flow of each arc (i, j) in G. We can find a feasible flow in G or prove there is none in $O(m \sum_{(i,j)\in U(G)} k_{ij})$.*

Flow Properties

The most interesting property for us is a Corollary of Theorem 1.

Corollary 2. *Let f^o be any feasible flow in G, and (i, j) be an arc of G. The flow value f_{ij} is constant for any feasible flow f if and only if:*

- *There is no path from j to i in $R(f^o) - \{(j,i)\}$.*
- *There is no path from i to j in $R(f^o) - \{(i,j)\}$.*

We will also consider a specific case which is useful for our purpose:

Corollary 3. *Let f^o be any feasible flow in G, and (i, j) be an arc of G with $f^o_{ij} = 0$. The flow value f_{ij} is equal to 0 for any feasible flow f if and only if i and j belong to two different strongly connected components of $R(f^o)$.*

[2] This complexity comes from the integer capacities. In this case, the flow is augmented by at least one for each iteration.

The search for strongly connected components can be done in $O(m+n+d)$ [146]. The advantage of this proposition is that all the arcs (i, j) with a constant 0 flow value can be identified by only one identification of the strongly connected components in $R(f^o)$.

This corollary is used in the following way: suppose that i represents a variable and j a value of i. Now, if the constraint is equivalent to the search of a feasible flow in a graph where (i, j) is an arc if j belongs to $D(i)$, then the corollary gives us a necessary and sufficient condition to determine when (i, j) is consistent with the constraint.

This is exactly the reasoning used for the global cardinality constraint as we will see later in this chapter.

3.2.2 Alldiff and Permutation Constraints (ALLDIFF, PERMUTATION)

The ALLDIFF constraint constrains the values taken by a set of variables to be pairwise different. The PERMUTATION constraint is an ALLDIFF constraint in which $|D(X(C))| = |X(C)|$.

Definition 4. An **alldiff** constraint is a constraint C defined by

$$\text{ALLDIFF}(X) = \{\tau \text{ s.t. } \tau \text{ is a tuple on } X(C) \text{ and } \forall a_i \in D(X(C)) : \#(a_i, \tau) \leq 1\}$$

This constraint is used in a lot of real world problems such as rostering or resource allocation. It is quite useful to express that two things cannot be at the same place at the same moment.

A filtering algorithm establishing arc consistency for the ALLDIFF is given in [110]. Its complexity is in $O(m)$ with $m = \sum_{x \in X} |D(x)|$, after the computation of the consistency of the constraint which requires $O(\sqrt{n}m)$. When the domains of the variables are intervals, [93] proposed a filtering algorithm establishing bound consistency with a complexity which is asymptotically the same as for sorting the internal endpoints. If the interval endpoints are from an integer range of size $O(n^k)$ for some constant k, the algorithm runs in linear time. Therefore, Melhorn's algorithm is linear for a permutation constraint. Lopes et al. [88] have designed an original and simple algorithm having the same complexity. A comparison between several algorithms is available in [144].

On the other hand, [84] has proposed an algorithm which considers that the domains are intervals, but which can create "holes" in the domain, that is the resulting domain will be union of intervals. His filtering algorithm is in $O(n^2 d)$.

In the original paper, Régin's filtering algorithm is based on matching theory, but we can also use the flow theory in order to obtain almost the same algorithm. We will not describe this algorithm here, because we prefer to detail a more general constraint : the global cardinality constraint. From this constraint, we can immediately obtain a filtering algorithm for the ALLDIFF constraint.

3.2.3 Global Cardinality Constraint (GCC)

A global cardinality constraint (GCC) constrains the number of times every value can be taken by a set of variables. This is certainly one of the most useful constraints in practice. Note that the ALLDIFF constraint corresponds to a GCC in which every value can be taken at most once.

Definition 5. A **global cardinality constraint** is a constraint C in which each value $a_i \in D(X(C))$ is associated with two positive integers l_i and u_i with $l_i \leq u_i$ defined by

$$\text{GCC}(X, l, u) = \{\tau \text{ s.t. } \tau \text{ is a tuple on } X(C) \text{ and } \forall a_i \in D(X(C)) : l_i \leq \#(a_i, \tau) \leq u_i\}$$

This constraint is present in almost all rostering or car-sequencing problems.

A filtering algorithm establishing arc consistency for this constraint has been proposed by Régin [112]. The consistency of the constraint can be checked in $O(nm)$ and the arc consistency can be computed in $O(m)$ providing that a maximum flow has been defined. Two other algorithms establishing bound consistency for this constraint have been developed by Quimper et al. [105] and Katriel et al. [74]. The first one is original whereas the second is an adaptation of Régin's algorithm [112].

We propose to describe Régin's algorithm here.

This algorithm is mainly based on the following observation: a GCC C is consistent if there is a flow in a directed graph $N(C)$ called the value network of C [112]:

Definition 6. Given $C = gcc(X, l, u)$ a GCC; the **value network** of C is the directed graph $N(C)$ with lower bound capacity and upper bound capacity on each arc. $N(C)$ is obtained from the value graph $GV(C)$, by:

- Orienting each edge of $GV(C)$ from values to variables. For such an arc (u, v): $l_{uv} = 0$ and $u_{uv} = 1$.
- Adding a node s and an arc from s to each value. For such an arc (s, a_i): $l_{sa_i} = l_i$, $u_{sa_i} = u_i$.
- Adding a node t and an arc from each variable to t. For such an arc (x, t): $l_{xt} = 1$, $u_{xt} = 1$.
- Adding an arc (t, s) with $l_{ts} = u_{ts} = |X(C)|$.

Figures 3 and 4 are examples of flow and residual graph of the example of the GCC given in Fig. 2.

Proposition 2. *Let C be a* GCC *and $N(C)$ be the value network of C; the following two properties are equivalent:*

- *C is consistent.*
- *There is a feasible flow in $N(C)$.*

Proof. **sketch of proof:** We can easily check that each tuple of $T(C)$ corresponds to a flow in $N(C)$ and conversely. □

From Corollary 3, we immediately have:

Proposition 3. *Let C be a consistent* GCC *and f be a feasible flow in $N(C)$. A value a of a variable x is not consistent with C if and only if $f_{ax} = 0$ and a and x do not belong to the same strongly connected component in $R(f)$.*

For our problem, a feasible flow can be computed in $O(nm)$, therefore, we have the same complexity for the check of the constraint consistency. Moreover, flow algorithms are incremental.

The search for strongly connected components can be done in $O(m + n + d)$ [146], thus a good complexity for computing arc consistency for a GCC is obtained.

Corollary 4. *Let C be a consistent* GCC *and f be a feasible flow in $N(C)$. Arc consistency for C can be established in $O(m + n + d)$.*

Here is a recapitulation of the complexities:

	Consistency	Arc consistency
From scratch	$O(nm)$	$O(m + n + d)$
After k modifications	$O(km)$	$O(m + n + d)$

3.2.4 Cardinality Matrix Constraint (CARD-MATRIX)

This constraint has been proposed by Régin and Gomes [123].

Cardinality matrix problems are the underlying structure of several real world problems such as rostering, sports scheduling , and timetabling. These are hard computational problems given their inherent combinatorial structure. The cardinality matrix constraint takes advantage of the intrinsic structure of the cardinality matrix problems. It uses a global cardinality constraint per row and per column and one cardinality (0,1)-matrix constraint per symbol. This latter constraint corresponds to solving a special case of a network flow problem, the transportation problem, which effectively captures the interactions between rows, columns, and symbols of cardinality matrix problems.

In order to show the advantage of this constraint, consider a restricted form of the cardinality matrix problems: the alldiff matrix problem [59]. In this case, each value has to be assigned at most once in each row and each column. The alldiff matrix characterizes the structure of several real world problems, such as design of scientific experiments or fiber optics routing. Consider the following example: a 6×6 matrix has to be filled with numbers ranging from 1 to 6 (this is a latin square problem). A classical model in CP consists of defining one variable per cell, each

variable can take a value from 1 to 6, and one ALLDIFF constraint per row and one ALLDIFF constraint per column. Now, consider the following situation:

		1	2		
		2	1		
		3	4		
		4	5		
•	•			•	•
•	•			•	•

In this case, the ALLDIFF constraints are only able to deduce that:

- only values 5 and 6 can be assigned to cells $(5, 3)$ and $(6, 3)$
- only values 3 and 6 can be assigned to cells $(5, 4)$ and $(6, 4)$.

However, with a careful study, we can see that the value 6 will be assigned either to $(5, 3)$ and $(6, 4)$ or to $(5, 4)$ and $(6, 3)$, which means that the other columns of rows 5 and 6 cannot take these values and therefore we can remove the value 6 from the domains of the corresponding variables (the ones with a • in the figure). The cardinality (0,1)-matrix automatically performs these inferences.

Definition 7. A **cardinality matrix** constraint is a constraint C defined on a Matrix $M = x[i, j]$ of variable s taking their values in a set V, and on two sets of cardinality variables $rowCard[i, j]$ and $colCard[i, j]$ and

$$\begin{aligned}\text{CARD-MATRIX}(M, V, rowCard, colCard) = \{&\tau \text{ s.t. } \tau \text{ is a tuple on } X(C)\\ &\text{and } \forall a_k \in V, \forall i \in Row(M) : \#(a_k, vars(i, *, M)) = rowCard[i, k]\\ &\text{and } \forall a_k \in V, \forall j \in Col(M) : \#(a_k, vars(*, j, M)) = colCard[j, k]\}\end{aligned}$$

If the matrix is form only by (0,1)-variable, then we say that we have a (0,1)-matrix.

Régin and Gomes proposed an AC filtering algorithm for matrix variables of the card-(0,1)-Matrix constraint by a similar method to the one used for GCCs[123]. A similar constraint, although expressed in a quite different way, with the same kind of algorithm to establish arc consistency, is given in [77].

Experimental results have shown that the CARD-MATRIX constraint outperforms standard constraint based formulations of cardinality matrix problems.

3.2.5 Global Cardinality Constraint with Costs (COST-GCC)

A global cardinality constraint with costs (COST-GCC) is the conjunction of a GCC constraint and a sum constraint:

Definition 8. A **cost function on a variable set** X is a function which associates with each value (x, a), $x \in X$ and $a \in D(x)$ an integer denoted by $cost(x, a)$.

Definition 9. A **global cardinality constraint with costs** is a constraint C associated with `cost` a cost function on $X(C)$, an integer H and in which each value $a_i \in D(X(C))$ is associated with two positive integers l_i and u_i; and defined by

$$\begin{aligned}\text{COST-GCC}(X, l, u, cost, H) = \{\tau \text{ s.t. } \tau \text{is a tuple on } X(C) \\ \text{and } \forall a_i \in D(X(C)) : l_i \leq \#(a_i, \tau) \leq u_i \\ \text{and } \Sigma_{i=1}^{|X(C)|} cost(var(C, i), \tau[i]) \leq H\}\end{aligned}$$

This constraint is used to model some preferences between assignments in resource allocation problems.
Note that there is no assumption made on the sign of costs.

The integration of costs within a constraint is quite important, especially to solve optimization problems, because it improves back-propagation, which is due to the modification of the objective variable. In other words, the domain of the variables can be reduced when the objective variable is modified. Caseau and Laburthe [38] have used an ALLDIFF constraint with costs, but only the consistency of the constraint has been checked, and no specific filtering has been used. The first given filtering algorithm comes from [49] and [50], and is based on reduced cost. A filtering algorithm establishing arc consistency has been given by Régin [114, 116]. The consistency of this constraint can be checked by searching for a minimum cost flow and arc consistency can be established in $O(|\Delta|S(m, n + d, \gamma))$ where $|\Delta|$ is the number of values that are taken by a variable in a tuple, and where $S(m, n + d, \gamma)$ is the complexity of the search for shortest paths from a node to every node in a graph with m arcs and n nodes with a maximal cost γ.

3.3 Balancing Constraints

Pesant and Régin introduced the notion of balancing constraints [100].

Many combinatorial problems require of their solutions that they achieve a certain balance of given features. Balance is often important in assignment problems or in problems with an assignment component. We give a few examples. In assembly line balancing, the workload of the line operators must be balanced. In rostering, we may talk of fairness instead of balance because of the human factor. Here, we want a fair distribution of weekends off or of night shifts among workers, for example. In vehicle routing, one dimension of the problem is to partition the customers into the different routes - balancing the number of customers served on each route, the quantity of goods delivered, or the time required to complete the route may be of interest.

We could describe the balance in the following way:

- The average value should be close to a given target, corresponding to the ideal value
- There should be no outliers, as they would correspond to an unbalanced situation
- Values should be grouped around the average value

Pesant and Régin claimed that statistics provide appropriate mathematical concepts to express this, and they proposed the first constraints based on statistic: the SPREAD constraint and bound consistency filtering algorithms associated with it. Roughly, the SPREAD constraint is defined on a set of numerical variables and combines the mean of these variables with the standard deviation. Schaus et al. noticed that this idea can be generalized to the concept of L_p norm [131–133].

Definition 10. Let $X = \{x_1, x_2, \ldots, x_n\}$ be a set of n variables, μ be a variable, L be a cost variable and the $L_p(X, \mu)$-norm defined as $L_p(X, \mu) = [\sum_{i=1}^{n} |x_i - \mu|^p)]^{\frac{1}{p}}$. The **balance** constraint is the constraint C defined on X, $\{\mu\}$ and L, and associated with a value of p such that C holds if and only if $L_p(X, \mu) = L$ and $\sum_{i=1}^{n} x_i = n\mu$ where the different norm are:

- $L_0 = |\{x \in X \text{ s.t. } x \neq \mu$ is the number of values different from the mean
- $L_1 = \sum_{x \in X} |x - \mu|$ is the sum of deviations from the mean
- $L_2 = \sum_{x \in X} (x - \mu)^2$ is the sum of square deviations from the mean
- $L_\infty = \max_{x \in X} |x - \mu|$ is the maximum deviation from the mean it is denoted by BALANCE(X, μ, L, p)

Note that the balancing constraint considers simultaneously the two sums.

None of these balance criteria subsumes the others. For instance, the minimization of L_1 does not imply in general a minimization of criterion L_2. This is illustrated on the following example. Assume a constraint problem with four solutions given in Fig. 5. The most balanced solution depends on the chosen norm. Each solution exhibits a mean of 100 but each one optimizes a different norm. Choosing the "best" criteria is an old question which has no definitive answer.

Pesant and Régin considered the case where $p = 2$ and named the constraint SPREAD. They gave a bound consistency filtering algorithm for the X variables in $O(n^2)$. Note that the SPREAD constraint in its general form considers μ and L (i.e. σ) as variables and not constants. Then, Schaus et al. proposed to simplify the algorithm when μ is a given constant [132]. They also derived some other filtering algorithms for L, X and μ. Having μ as a constant is quite frequent in practice. However, this is not always the case, for instance when X variables measures a

sol. num.	solution	L_0	L_1	L_2	L_∞
1	100 100 100 100 30 170	**2**	140	9800	70
2	60 80 100 100 120 140	4	**120**	4000	40
3	70 70 90 110 130 130	6	140	**3800**	30
4	71 71 71 129 129 129	6	174	5046	**29**

Fig. 5 Illustration showing that no balance criterion defined by the norm L_0, L_1, L_2 or L_∞ subsumes the others. The smallest norm is indicated in bold character. For example, solution 2 is the most balanced according to L_1

slack or an overflow, the ideal should be to have no overflow, so 0 for the mean, but we do not have negative x, because it has no practical meaning (an underflow does not compensate an overflow).

Next, Schaus et al. studied the DEVIATION constraint, that is the BALANCE constraint with $p = 1$ [131]. They proposed efficient bound consistency filtering algorithms (in $O(n)$) when considering the mean as a constant value. Further investigations and recent developments can be found in Schaus's PhD thesis [130].

3.4 Constraint Combination Based Constraints

3.4.1 Max-SAT Constraint (MAX-SAT)

A lot of work have been carried out in order to improve the computation of minimization of the number of violated constraints in an over-constrained problem (Max-CSP).

Several algorithms have been designed. First, Partial Forward Checking [51], which has been improved by PFC-DAC [81, 155], and then by PFC-MRDAC [80].[3] The major drawback of these algorithms is that there are ad-hoc algorithms based on a branch-and-bound procedure and they mainly consider only binary constraints. Therefore, their integration into a CP Solver is not easy. Thus, Régin et al. have proposed to define a constraint corresponding to this specific problem like any other global constraint [124].

Definition 11. Let $\mathcal{C} = \{C_i, i \in \{1, \ldots, m\}\}$ be a set of constraints, $cost(\mathcal{C})$ be a set of cost variables associated with the constraints of $\mathcal{C}$ and *unsat* a variable. A **Max-SAT constraint** is a constraint $ssc(\mathcal{C}, cost(\mathcal{C}), unsat)$ defined by the conjunction of the constraint

$$\text{unsat} = \sum_{C \in \mathcal{C}} \text{cost}(C)$$

and the set of disjunctive constraints

$$\{[(C \wedge (\text{cost}(C) = 0)) \vee (\overline{C} \wedge (\text{cost}(C) = 1))], C \in \mathcal{C}\}$$

It is denoted by MAX-SAT($\mathcal{C}$)

Then, Régin et al. integrated some classical algorithms defined for solving over-constrained problem as filtering algorithm of this global constraint. Let $P = (X, D, \mathcal{C})$ be a constraint network.

[3] This algorithm can be viewed as a generalization of the constructive disjunction in the case where several constraints must be satisfied and not only one.

Notation 1

- $v^*(P)$ *is the number of violated constraint of* P
- $v(P)$ *is any lower bound of* $v^*(P)$
- $v^*((x,a),P)$ *is the number of violated constraints of* P *when* $x = a$
- $v((x,a),P)$ *is any lower bound of* $v^*((x,a),P)$

Definition 12. Two sub-problems $Q1 = (X, \mathcal{D}, \mathcal{K})$ and $Q2 = (X, \mathcal{D}, \mathcal{L})$ of P are constraint disjoint if $\mathcal{K} \cap \mathcal{L} = \emptyset$

Theorem 2 (Régin et al.). *Given* $P = (X, \mathcal{D}, \mathcal{C})$*, a constraint network and* $\mathcal{Q}$ *et set of sub-problems of* P *that are pairwise constraint disjoint then:*

$$v^*(P) \geq \sum_{Q \in \mathcal{Q}} v^*(Q) \geq \sum_{Q \in \mathcal{Q}} v(Q)$$

Then we have two corollaries:

Corollary 5. *Let* obj *be a value. If* $\sum_{Q \in \mathcal{Q}} v(Q) > obj$*, then there is no solution of* P *with* $v^*(P) \leq obj$.

Corollary 6. *Let obj be a value and a be a value of a variable* x *involved in a sub-problem* Q *of* $\mathcal{Q}$*. If* $\sum_{R \in (\mathcal{Q}-Q)} v(R) + v((x,a),Q) > obj$*, then there is no solution of* P *with* $v^*((x,a),P) \leq obj$*.*

This corollary will permit to remove some inconsistent values.

The main issue is the computation of a set $\mathcal{Q}$ and the choice of $v(Q)$ for a given problem Q.

Régin et al. noted that PFC-MRDAC algorithm proposes, in fact, to build the set $\mathcal{Q}$ in the following way: we begin with the set of constraints $\mathcal{K} = \mathcal{C}$ and we order the variables (in any order). Then, we select the variable in that order. When a variable x is selected, we take all the constraints of $\mathcal{K}$ involving x in order to create a sub-problem denoted by $Q(x)$ and we remove from $\mathcal{K}$ these constraints before considering the next variable.

From the specific construction of $\mathcal{Q}$, we obtain a value of $v(Q)$ which is easy to compute: since each sub-problem is defined from a variable x which is involved in all the constraints of the sub-problem, we can compute for each value a of x the number of constraints violated if $x = a$, and we define $v(Q)$ as the value having the smallest number of violations. This information can be maintained efficiently.

Then, several improvements of this integration, and so also of PFC-MRDAC algorithm, have been proposed notably to deal efficiently with interval variables [102, 125]. At last, a new algorithm based on conflict set has been proposed [125]. A conflict set is a set of constraints which is inconsistent.

Some information about this constraint can also be found in Régin's HDR thesis [120].

3.4.2 Or and And Constraints (AND, OR)

Lhomme [86] studied the logical combination of constraints, sometimes called meta-constraints because it noticed that although these constraints are extremely useful for modelling problems, they have either poor filtering algorithms or are not very efficient.

First, Lhomme considered the constraint $\text{OR}(C_1, C_2)$ which is satisfied if $C1$ is satisfied or if C_2 is satisfied. He showed that the constructive disjunction (See [152] for more information) establishes arc consistency for this constraint, but constructive disjunction is complex to implement and not very efficient. Hence, he proposed another algorithm much more efficient. It is based on the following remarks:

- Let x be a variable involved in C_1 and in C_2. If we find a support for (x, a) on C_1, then it is useless to search for a support for (x, a) on C_2 because (x, a) satisfies the disjunction (i.e. has a support on at least one constraint)
- It is useless to search for supports for variables of $X(C_2)$ that are not in $X(C_1)$, because C_1 can be true, and in this case, all the values of these variables are consistent with the OR constraint. The same reasoning can be applied for variables of $X(C_1)$ that are not in $X(C_2)$.

Therefore, Lhomme established the following Proposition:

Proposition 4. *Let $C = \text{OR}(C_1, C_2)$ be the constraint equals to the disjunction between the constraint $C1$ and the constraint $C2$. Then, C is arc consistent if and only if the values of the variables of $(X(C_1) \cap X(C_2))$ are consistent either with $C1$ or with C_2.*

Lhomme also addressed the constraint equivalent to the conjunctions of two constraints C_1 and C_2. We have already mentioned the difference between establishing arc consistency for the constraints taken separately and for the constraint $C = \text{AND}(C_1, C_2)$. Lhomme proposed an algorithm based on the simultaneous search for support for the variables involved in both constraints. Unfortunately, such a search imposes the availability of some functions for each constraints which is rarely the case. However, this is the case for TABLE constraints given in extension and Lhomme gave an algorithm for them.

The principles given in this section can be easily generalized to combinations of several constraints and not only two.

3.5 Sequencing Constraints

Sequencing constraints are useful in rostering or car sequencing problems to constrain the number of time some values are taken by any group of k consecutive variables. For instance, they are used to model that on an assembly line, at most one car over three consecutive cars can have a sun roof. They are based on a conjunction of AMONG constraints, which are defined as follows.

3.5.1 Among Constraint (AMONG)

Definition 13. Given X a set of variables, l and u two integers with $l \leq u$ and V a set of values. The **among** constraint ensures that at least l variables of X and at most u will take a value in V, that is

$$\text{AMONG}(X, V, l, u) = \{t \mid t \text{ is a tuple of } X \text{ and } l \leq \sum_{a \in V} \#(a, \tau) \leq u\}$$

This constraint has been introduced in CHIP [17].

It is straightforward to design a filtering algorithm establishing arc consistency for this constraint. For instance, we can associate with each variable x_i of X a $(0, 1)$ variable y_i defined as follows: $y_i = 1$ if and only if $x_i = a$ with $a \in V$. Then, the constraint can be rewritten $l \leq \sum y_i \leq u$. Note that the AMONG constraint is sometimes directly defined in that way, that is by involving only this set Y of $(0, 1)$ variables.

The SEQUENCE constraint is a conjunction of gliding AMONG constraints.

3.5.2 Sequence Constraint (SEQUENCE)

Definition 14. Given X a set of variables, q, l and u three integers with $l \leq u$ and V a set of values. The **sequence** constraint holds if and only if for $1 \leq i \leq n_q + 1$ $\text{AMONG}(\{x_i, \ldots, x_{i+q-1}\}, V, l, u)$ holds. More precisely

$$\text{SEQUENCE}(X, V, q, l, u) = \{t \mid t \text{ is a tuple of } X \text{ and for each sequence } S \text{ of } q \text{ consecutive variables: } l \leq \sum_{v \in V} \#(v, t, S) \leq u\}$$

This constraint has been introduced in CHIP [17].

Several filtering algorithms have been proposed for this constraint. First, it is possible to use only the set of overlapping AMONG constraints, but this does not lead to efficient domain reductions. The CARD-PATH constraint of Beldiceanu and Carlsson can also be used [10]. However, the filtering algorithm does not establish arc consistency. Some pseudo polynomial algorithms for establishing arc consistency have been designed. Bessiere et al. [27] gave a domain consistency propagator that runs in $O(nqd^q)$ time. van Hoeve et al. [64] proposed an encoding into a REGULAR constraint which runs in $O(n2^q)$ time. The first strongly polynomial algorithm (in $O(n^3)$) establishing arc consistency has been proposed by van Hoeve et al. [64,65]. Then, several polynomial algorithms with different complexities have been introduced by Brand et al. [33] and, at last, a very nice model leading to the best filtering algorithm is described in [90]. We propose to describe quickly some of these algorithms because it is rare to obtain several algorithms while using different approaches, and this could be useful for some other constraints.

For the sake of clarity, we will use the two equivalent representations of the AMONG constraint. The one using the X variable set and the other using the Y variable set.

First, van Hoeve et al. [64] remarked that when l is equal to u then arc consistency can be established in linear time. In this case, in any solution x_i will be equal to x_{i+q} because for two among constraints we have $y_i + \ldots y_{i+q-1} = l$ and

$y_{i+1} + \ldots y_{i+q} = l$. So, by adding these new equality constraints and by using the filtering algorithm associated with each AMONG constraint, arc consistency will be established by the propagation mechanism.

Then, three strongly polynomial filtering algorithms establishing arc consistency have been proposed. There are based on different concepts: cumulative sum, difference constraint and flow.

FA Based on Cumulative Sum

A set of $n + 1$ new variables s_i are introduced. A variable s_i correspond to the sum of the y_j variables for $j = 1$ to i. The S variables are encoded as follows: $s_0 = 0$ and $s_i = y_i + s_{i-1}$. Then, the constraints $s_j \leq s_{j+q} - l$ and $s_{j+q} \leq s_j + u$ for $1 \leq j \leq n - q + 1$ are added to the model. Brand et al. [33] have proposed this model and shown that enforcing singleton bound consistency on these variables establishes arc consistency for the SEQUENCE constraint. In addition, they proved that the complexity of maintaining arc consistency on a branch of the tree search is in $O(n^3)$. This model is also a reformulation (and an improvement) of the first filtering algorithm establishing arc consistency in polynomial time of van Hoeve et al. [64, 65].

FA Based on Difference Constraint

This approach uses different constraints, that is constraints of the form $S \leq S' + d$, for encoding the SEQUENCE constraint. It has been proposed by Brand et al. [33] and it uses the S variables like in the previous approach, but the constraint $s_i = y_i + s_{i-1}$ is replaced by the two equivalent constraints $s_{i-1} \leq s_i \leq s_{i-1} + 1$ and $y_i \Leftrightarrow s_{i-1} \leq s_i - 1$. The consistency of a set of distance constraints can be checked by searching for the presence of negative cycle in a graph (see [42] or [39] Sect. 24.4). Thus, an AC filtering algorithm can be simply derived by using the current assignment of the Y variables in order to define the set DC of distance constraints. After checking the consistency of DC, the boundaries of the Y variables are explicitly tested. That is, if DC implies that $s_{i-1} \leq s_i - 1$ then $y_i = 1$ and if DC implies that $s_i \leq s_{i-1}$ then $y_i = 0$. The authors show that this FA can be maintained during the search with a complexity on a branch of the tree search in $O(n^2 \log(n))$.

FA Based on Flow

This clever approach has been proposed in [90]. The idea is to represent a SEQUENCE constraint by an integer linear programme formed by the AMONG constraints it contains. Then, the specific structure of the obtained matrix (it has the consecutive ones property for the columns) is exploited and the model is transformed into a network flow problem. Thus, the computation of some properties of

the flow problem, like the constant arc for all feasible flow, leads to an AC filtering algorithm. This approach has two main advantages: the problem is just transformed into a flow problem and so there is no need to write any specific algorithms; and the complexity is reduced. This FA, indeed, can be maintained during the search with a complexity on a branch of the tree search in $O(n^2)$. Maher et al. use an existing transformation from matrix having the consecutive ones property to a network flow problem. This transformation is explained in [2] and leads to a flow which is not really easy to understand. We propose here to try to directly explain the obtained flow.

Consider the following problem: 9 variables from x_1 to x_9, a sequence of width 4, $l = 1$, and $u = 3$, meaning that at least one variable and at most 3 variables of each sequence of 4 consecutive variables must be set to 1. Figure 6. The graph on which the flow is defined is built as follows:
First, we define the nodes:

- We create a source s and a sink t.
- We create as many W-nodes as there are complete sequences. For the example, we have 6 complete sequences, so we create 6 W-nodes: $w_1, \ldots, w_6$.

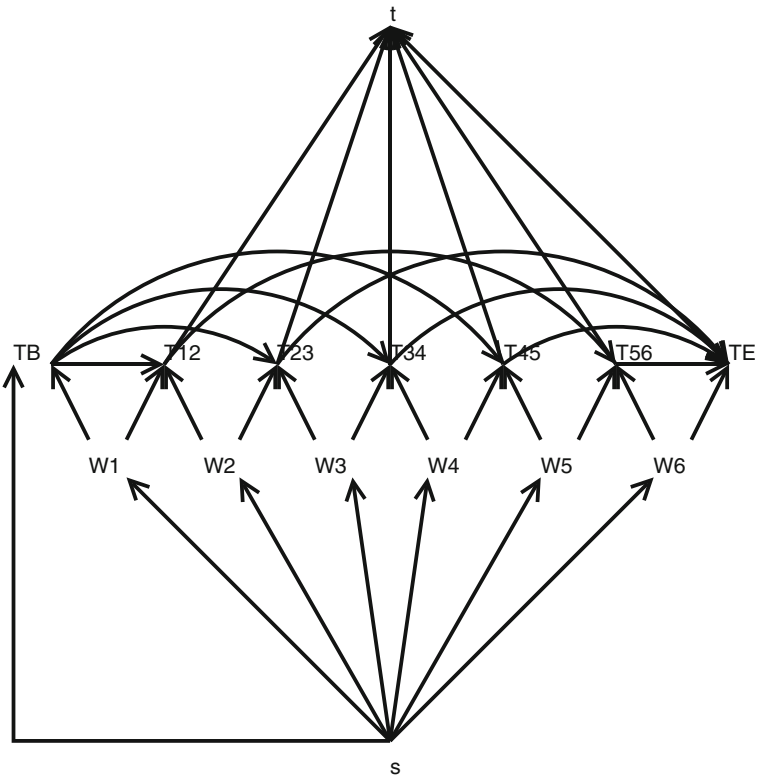


Fig. 6 The graph associated with a sequence problem

A W-node is a windows node and corresponds to a sequence. Node $W1$ represents the sequence x_1, x_2, x_3, x_4, node $W2$ the sequence x_2, x_3, x_4, x_5 etc.

- We create as many T-node as there are no empty intersection between consecutive W-nodes and we add 2 special T-node: TB and TE. A T-node is a transition node and represents what two W-nodes have in common. Node $T12$ corresponds to the W-nodes $W1$ and $W2$, node $T23$ corresponds to the W-nodes $W2$ and $W3$, etc. Node TB corresponds to the beginning of the sequence and node TE to the end of the sequence.

Then, we define the arcs:

- There is an arc from t to s.
- There is an arc from s to each W-node. The required quantity of flow (i.e. the flow must be equal to this value) in each arc is $[u - l]$ corresponding to the slack we have for sequence from the minimum (i.e. l).
- There is an arc from each T-node but TB to t. The required quantity of flow in each arc is $[u - l]$ corresponding to the slack we have for a sequence from the minimum (i.e. l), excepted for TE for which the required quantity of flow is u.
- There is an arc from s to TB. The required quantity of flow in this arc is l. The idea is that each sequence must have at least l units of flow. TB is used to receive this quantity and then to transmit it to the other T-nodes.
- There is an arc from any W-node to the two T-nodes associated with it. Node Wi is linked to T-node $T(i-1)i$ and node $Ti(i+1)$. $W1$ is linked to TB and $T12$, and $W6$ is linked to $T5$ and TE. All these arcs are (0,u-l) arcs (the flow traversing them has a value in $[0, u - l]$).
- There are arcs between T-nodes. Each T-node $Ti(i+1)$, excepted TB and TE, has one entering arc and one leaving arc. The entering arc represents the variable which is in Wi but not in $W(i+1)$, and the leaving arc represents the variable which is in $W(i+1)$ and not in Wi. For instance, the arc from $T12$ to $T56$ represents variable x_5. TB has 4 (the width) leaving arcs corresponding to the 4 first variables x_1, x_2, x_3, x_4 and TD has 4 entering arcs corresponding to the 4 last variables x_6, x_7, x_8, x_9. All these arc are (0,1) arcs and their flow value corresponds to the value of the variable in any solution.

Now, a feasible flow in this graph corresponds to a solution of the problem. The intuitive idea is that we send the minimum unit of flow TB and this quantity of flow will be propagated to the sequences thanks to arcs defined by x variables. The arcs leaving s and entering t ensure the flow conservation and express the fact that some sequences have more 1 than others. Then, arc consistency of the variables x is computed by searching for constant values for the arcs corresponding to these variables, which can be done thanks to Corollary 2 (See Flow Properties).

Figure 7 shows an example of feasible flow for some values of x variables. The arcs having a constant flow value equal to 0 have been removed.

Some experiments given in [33,90] show that the algorithms based on cumulative sums and the flows are the best in practice. The latter one seems to be more robust.

++Some generalizations or variations of the SEQUENCE constraint have also been studied.

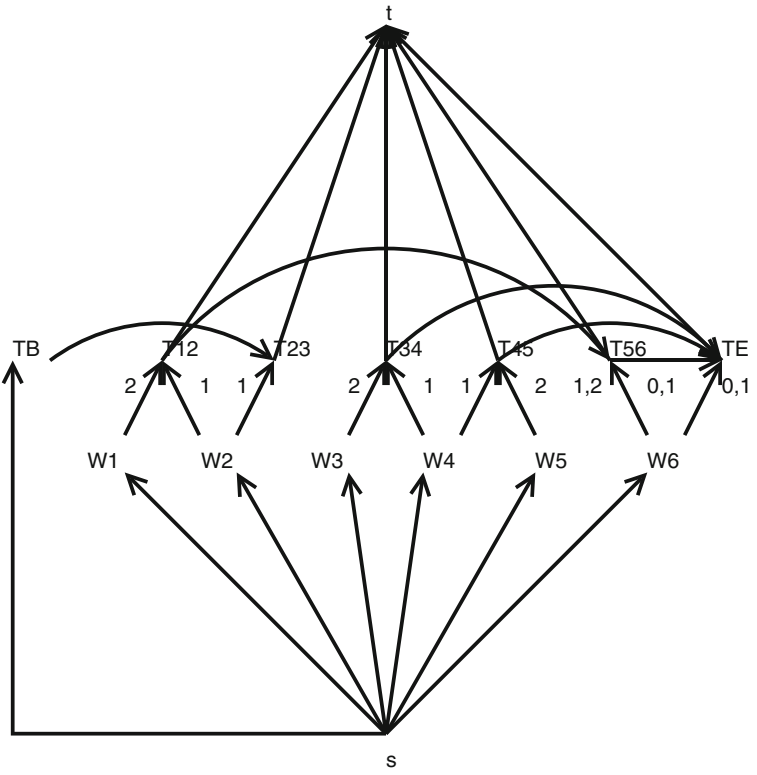


Fig. 7 A feasible flow when $x_1 = 0$, $x_2 = \{0, 1\}$, $x_3 = 0$, $x_4 = 0$, $x_5 = 1$, $x_6 = \{0, 1\}$, $x_7 = 1$, $x_8 = 1$, $x_9 = \{0, 1\}$. The arcs that cannot carry any unit of flow in any feasible flow have been removed. Number indicates the flow value. They are separated by comma when there are several possible flow values for other solutions

3.5.3 Generalized Sequence Constraint (GEN-SEQUENCE)

This constraint generalizes the SEQUENCE constraint by adding some other AMONG constraints. These AMONG constraints must be defined using the same set of values V than the SEQUENCE.

Definition 15. Given X an ordered set of variables, $\mathcal{Q}$ a set of ordered subsets of X (i.e subset of consecutive variables of X) where each subset Q_i is associated with two numbers l_i and u_i and V a set of values. The **generalized sequence constraint** holds if and only if for each $Q_i \in \mathcal{Q}$ AMONG(Q_i, V, l_i, u_i) holds. More precisely

$$\text{GEN-SEQUENCE}(X, V, \mathcal{Q}, \{l_i\}, \{u_i\}) = \{t \mid t \text{ is a tuple of } X \text{ and } \forall Q_i \in \mathcal{Q} \; l_i \leq \textstyle\sum_{v \in V} \#(v, t, Q_i) \leq u_i\}$$

This constraint has been proposed by van Hoeve et al. [64]. The authors gave an FA establishing arc consistency for it, whose complexity is in $O(n^4)$. This algorithm

is based on the same idea as the cumulative sum for the SEQUENCE constraint. However, Maher et al. [90] also considered this constraint in order to try to apply the modeling idea of representing the constraint by a flow. Sometimes, the obtained matrix satisfies the consecutive one property and the method can be applied and so a quadratic AC filtering algorithm exists. It is also possible that the matrix will satisfy the consecutive one property if the subset of X is reordered. Such a result can be obtained in polynomial time (see [90]). However, for some matrices, it will not be possible to obtained the desired property. In this case, an encoding based on difference can be used leading to an AC Filtering algorithm whose complexity is in $O(nm + n^2 \log(n))$ for any branch of the tree search, where m is the number of element of $\mathcal{Q}$.

3.5.4 Global Sequencing Constraint (GSC)

The global sequencing constraint (GSC) has been designed mainly to try to solve some car sequencing instances. It combines a SEQUENCE constraint and global cardinality constraints.

A global sequencing constraint C is specified in terms of an ordered set of variables X which take their values in D, some integers q, *min* and *max* and a given subset V of D. On one hand, a GSC constrains the number of variables in X instantiated to a value $v_i \in D$ to be in an interval $[l_i, u_i]$. On the other hand, a GSC constrains for each sequence S_j of q consecutive variables of $X(C)$, that at least *min* and at most *max* variables of S_j are instantiated to a value of V.

Definition 16. Given X an ordered set of variables, m, M, q three positive integers, a set of values D in which each value $a_i \in D$ is associated with two positive integers l_i and u_i, and a set of values V. The **global sequencing constraint** is defined by

$$\begin{aligned}\text{GSC}(X, D, \{l_i\}, \{u_i\}, V, m, M, q) = \{t \mid & t \text{ is a tuple of } X \\ & \text{and } \forall a_i \in D : l_i \leq \#(a_i, t) \leq u_i \\ & \text{and for each sequence } S \text{ of } q \text{ consecutive} \\ & \text{variables: } m \leq \textstyle\sum_{v_i \in V} \#(v_i, t, S) \leq M\}\end{aligned}$$

This constraint has been proposed by Régin [126].

It arises in car sequencing or in rostering problems. A filtering algorithm is described in [126]. It is based on the reformulation of the problem mainly using flows and it has been implemented in ILOG Solver. Thanks to it, some problems of the CSP-Lib have been closed, and a recent and nice experimental study of [65] shows that this constraint leads to good results for solving some car sequencing instances of the CSP-Lib. In fact, 12 problems are solved by a CP model only if it uses this constraint.

About sequencing constraints, some other combinations of AMONG constraints with or without cardinality constraints have been considered. For instance, Régin [121] studied several combinations of AMONG constraints. He mainly showed that in general a combination of AMONG constraints is an NP-Complete problem. Nevertheless, if the AMONG constraints are pairwise value disjoint (i.e. the set of values

associated with each AMONG constraint are disjoint), then it is possible to represent the set of AMONG constraints by a unique GCC and so to obtain a polynomial AC filtering algorithm. In addition, Régin proposed some kind of shaving or singleton arc consistency to improve the combination of AMONG and cardinality constraints. He applied his result to the inter-distance constraint.

At last, we need to mention two other variations of the sequencing constraints that have been recently considered: the multiple sequence and the SLIDING-SUM. The multiple sequence has been introduced in [33] and combines several SEQUENCE constraints provided that the values counted by each SEQUENCE are pairwise disjoint. Then, an AC filtering algorithm based on the REGULAR constraint has been proposed. The SLIDING-SUM has been introduced by Beldiceanu [9] and is a generalization of the SEQUENCE constraint to non (0,1) variables, that is the sum of variable is directly considered. An efficient bound consistency filtering algorithm has been proposed for this constraint in [90].

3.6 Distance Constraints

3.6.1 Inter-Distance Constraint (INTER-DISTANCE)

Régin [113] introduced, under the name "Global Minimum Constraint", a constraint defined on X a set of variables stating that for any pair of variable x and y of X the constraint $|x - y| \geq k$ is satisfied. This constraint is mentioned in [66].

Definition 17. An **inter-distance constraint** is a constraint C associated with an integer k and defined by

$$\text{INTER-DISTANCE}(k) = \{\tau \text{ s.t. } \tau \text{is a tuple of } X(C) \text{ and } \forall a_i, aj \in \tau : |a_i - a_j| \geq k\}$$

This constraint is present in frequency allocation problems or in scheduling problems in which tasks require p contiguous units of resource to be completed.

A filtering algorithm has been proposed for this constraint [113]. Note that there is a strong relation between this constraint and the SEQUENCE constraint. An $1/q$ SEQUENCE constraint constrains two variables assigned to the same value to be separated by at least $q - 1$ variables, with regard to the variable ordering. Here, we want to select the values taken by a set of variables such that all pairs of values are at least k units apart.

Then, a bound consistency algorithm has been proposed by Artiouchine and Baptiste [5, 6]. This algorithm runs in $O(n^3)$. It has been improved later by Quimper et al. [104] for running in $O(n^2)$.

3.6.2 Sum and Binary Inequalities Constraint (SUM-INEQ)

This constraint is the conjunction of a sum constraint and a set of distance constraints, that is constraints of the form $x_j - x_i \leq c$.

Definition 18. Let $SUM(X, y)$ be a sum constraint, and $\mathcal{I}_{neq}$ be a set of binary inequalities defined on X. The **sum and binary inequalities constraint** is a constraint C associated with $SUM(X, y)$ and $\mathcal{I}_{neq}$ defined by:

$$\begin{aligned}\text{SUM-INEQ}(X, y, \mathcal{I}_{neq}) = \{\tau \text{ s.t. } &\tau \text{is a tuple of } X \cup y \\ &\text{and } (\textstyle\sum_{i=1}^{|X|} \tau[i]) = \tau[y] \\ &\text{and the values of } \tau \text{ satisfy } \mathcal{I}_{neq}\}\end{aligned}$$

This constraint has been proposed by Régin and Rueher [127]. It is used to minimize the delays in scheduling applications.

Bound consistency can be computed in $O(n(m + n \log n))$, where m is the number of inequalities and n the number of variables. It is also instructive to remark that the bound consistency filtering algorithm still works when $y = \Sigma_{i=1}^{i=n} \alpha_i x_i$ where α_i are non-negative real numbers.

3.7 Geometric Constraints

The most famous geometric constraint is the DIFF-N constraint. We quote [17]: "The DIFF-N constraint was introduced in CHIP in order to handle multi-dimensional placement problems that occur in scheduling, cutting or geometrical placement problems. The intuitive idea is to extend the ALLDIFF constraint which works on a set of domain variables to a non-overlapping constraint between a set of objects defined in a n-dimensional space."

Definition 19. Consider R a set of multidirectional rectangles. Each multidirectional rectangle i is associated with two sets of variables $O_i = \{o_{i1}, \ldots, o_{in}\}$ and $L_i = \{l_{i1}, \ldots, l_{in}\}$. The variables of O_i represent the origin of the rectangle for every dimension, for instance the variable o_{ij} corresponds to the origin of the rectangle for the jth dimension. The variables of L_i represent the length of the rectangle for every dimension, for instance the variable l_{ij} represents the length of the rectangle for the j^{th} dimension. A **diff-n constraint** is a constraint C associated with a set R of multidirectional rectangles, such that:

$$\begin{aligned}\text{DIFF-N}(R) = \{\tau \text{ s.t. } &\tau \text{is a tuple of } X(C) \\ &\text{and } \forall i \in [1, m], \forall j \in [1, m], j \neq i, \exists k \in [1, n] \\ &\text{s.t. } \tau[o_{ik}] \geq \tau[o_{jk}] + \tau[l_{jk}] \text{ or } \tau[o_{jk}] \geq \tau[o_{ik}] + \tau[l_{ik}]\}\end{aligned}$$

This constraint is mainly used for packing problems. In [20], an $O(d)$ filtering algorithm for the non-overlapping constraint between two d-dimensional boxes and so a filtering algorithm for the non-overlapping constraint between two convex polygons are presented.

Some other geometric constraints, often based on the notion of non-overlapping objects, have been studied. Unfortunately, the papers are often difficult to understand. More information can be found in [1, 11, 20, 25].

3.8 Summation Based Constraints

Some variations of constraints based on the summation problems have been proposed. Trick proposed, under the name "knapsack constraint", a SUBSET-SUM constraint whose filtering is pseudo polynomial and based on dynamic programming [147,148]. This paper triggered some other researches such as Pesant's one on REGULAR constraints. Then, Shaw introduces another filtering algorithm which is polynomial but not characterized [140]. This means that we do not have a property defining the values that are removed by the algorithm. On the other hand, Fahle and Sellmann introduced a KNAPSACK constraint involving an objective to maximize (the profit) [48]. This constraint is closer to the original knapsack problem whereas the Trick's one is more related to the subset sum problem.

3.8.1 Subset-Sum Constraint (SUBSET-SUM)

The subset sum problem is: given a set of integers, does the sum of some non-empty subset equal exactly zero? For example, given the set $-7, -3, -2, 5, 8$, the answer is yes because the subset $-3, -2, 5$ sums to zero. The problem is NP-Complete. Trick proposed to consider the following variations: given a set of 0–1 variables $X = \{x_1, \ldots, x_n\}$ where each variable x_i is associated with a coefficient α_i and L and U tow bounds, find an assignment of variables such that $L \leq \sum_{x_i \in X} \alpha_i x_i \leq U$. For the sake of clarity, we will use the name SUBSET-SUM constraint for the Trick's knapsack constraint.

Definition 20. A **Subset-Sum constraint** is a constraint C defined on 0–1 variables and associated with a set of $n = |X(C)|$ coefficients : $A = \{\alpha_1, \ldots, \alpha_n\}$ and two bounds L and U such that

$$\text{SUBSET-SUM}(X, A, L, U) = \{\tau \text{ s.t. } \tau \text{is a tuple of } X(C) \text{ and } L \leq \Sigma_{i=1}^{n} \alpha_i \tau[i]) \leq U\}$$

Trick proposed to use the classical dynamic programming approach to check whether the constraint is consistent or not. We reproduce some parts of his presentation:

Define a function $f(i, b)$ equals to 1 if the variables $x_1, \ldots x_i$ can fill a knapsack of size b and 0 otherwise, with $i = 1 \ldots n$ and $b = 0 \ldots U$. We define then the dynamic programming recursion as follows:

- $f(0, 0) = 1$
- $f(i, b) = \max(f(i-1, b), f(i-1, b-\alpha_i))$

The second point means that there are two possibilities to have $f(i, b)$ equals to 1: either $f(i-1, b)$ is true (i.e. equals to 1) and if we set x_i to 0 then $f(i, b)$ will also be true, or $f(i-1, b-\alpha_i)$ is true and then by setting x_i to 1 we increasing $b - \alpha_i$ to b and $f(i, b)$ will also be true.

Then, Trick introduced the idea of visualizing these recursion equation as a network with one node for every $[i, b)$ and edges going from $(i-1, b)$ to (i, b') that is

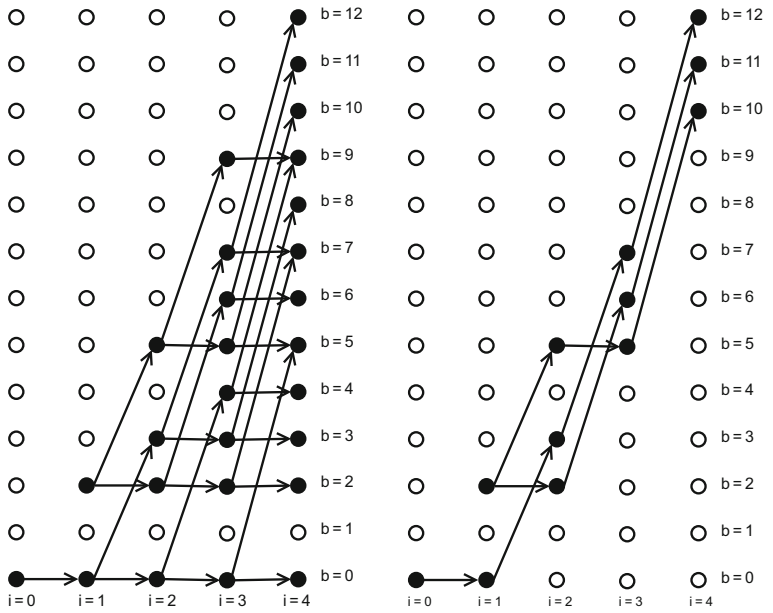


Fig. 8 On the *left*, the knapsack (or subset-sum) graph proposed by Trick [148] for the constraint $10 \leq 2x_1 + 3x_2 + 4x_3 + 5x_4 \leq 12$. On the *right*, the resulting graph after establishing arc consistency

between nodes with a 1 value of f (See Fig. 8: a horizontal edge from $(i-1, b)$ to (i, b) corresponds to the assignment $x_i = 0$ and an edge from $(i-1, b)$ to $(i, b+\alpha_i)$ corresponds to the assignment $x_i = 1$).

Trick proved that:

- The consistency of constraint is equivalent to the existence of a path from the node $(0, 0)$ to the nodes $(4, 10)$, $(4, 11)$ or $(4, 12)$.
- Any node which does not belong to such a path can be removed from the graph, and once all these nodes have been deleted then the value $(x_i, 0)$ is consistent if there exists a b, an arc from $(i-1, b)$ to (i, b), and the value $(x_i, 1)$ is consistent if there exists a b, and an arc from $(i-1, b)$ to $(i, b+\alpha_i)$.

Figure 8 gives an example of such pruning.

Trick gave an algorithm establishing arc consistency whose space and time complexity is in $O(nU^2)$, thus pseudo-polynomial.

We can show that there is no need of an extra filtering algorithm and that this complexity can be reduced. In fact, we can reformulate the problem with a set of binary constraints in a way similar as the one used by Beldiceanu et al. [15] for reformulating automaton based constraints. For $i = 1 \ldots n$, a binary constraint at the level i is simply defined by: $(S_i = S_{i-1}$ OR $S_i = S_{i-1} + \alpha_i)$. In addition, the constraints defining the initial and the final summation are added: $S_0 = 0$ and

$L \leq S_i \leq U$. This reformulation satisfies Corollary 1 (See Preliminaries section); therefore, establishing AC on this reformulation corresponds to establish AC on the SUBSET-SUM constraint. It is not difficult to maintain AC for each constraint of the form $(S_i = S_{i-1}$ OR $S_i = S_{i-1} + \alpha_i)$. The complexity depends on the size of the domain and the number of allowed combinations for each constraint. Clearly, each value v of S_i has at most 2 compatible values v and $v - \alpha_i$, so there are $2U$ allowed tuples per constraint. Thus, by Proposition 1 on the complexity for establishing arc consistency for TABLE constraint, we can establish and maintain AC for this constraint with a time complexity in $O(2 \times 2U) = O(U)$. Since there are n constraints, the overall complexity is $O(nU)$.

Note that this reformulation can be improved. For instance, we can add the constraint $S_i \leq \sum_{j=1}^{i} \alpha_i$.

We could also benefit from this representation and easily deal with non binary variables by adding some other OR parts into the binary constraints or by changing the reformulation in order to use ternary constraints of the form (S_{i-1}, X_i, S_i) where $S_i = X_i \times \alpha_i + S_{i-1}$ instead of binary constraints. The overall complexity is multiplied by d, that is the size of the domains.

At last, we need to mention that Shaw [140] gave another filtering algorithm for the SUBSET-SUM constraint. Unfortunately, his algorithm is too much complex to be included here.

3.8.2 Knapsack Constraint (KNAPSACK)

Fahle and Sellmann proposed to study the constraint corresponding to the classical knapsack problem [48].

The knapsack problem is defined as follows: Given a set of items, each with a weight and a profit, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total profit is as large as possible.

Definition 21. A **knapsack constraint** is a constraint C defined on 0–1 variables where x_i is the variable representing the belonging of item i to the knapsack, and associated with a set of $n = |X(C)|$ integer weights : $W = \{w_1, \ldots, w_n\}$, a set of n integer profits $P = \{p_1, \ldots, p_n\}$ and a capacity K and a lower bounds B such that

$$\text{KNAPSACK}(X, W, P, K, B) = \{\tau \text{ s.t. } \tau \text{is a tuple of } X(C) \text{ and } \Sigma_{i=1}^{n} w_i \tau[i] \leq K \text{ and } \Sigma_{i=1}^{n} p_i \tau[i] \leq B\}$$

The knapsack problem is NP-hard in general, therefore, Fahle and Sellmann do not propose to establish AC but give a weaker filtering algorithm having a low complexity. Then, Sellman in collaboration with some other researchers improved some aspects of the algorithm [73, 135, 137].

The algorithm is mainly based on a nice observation made by Dantzig [41]: Unlike the integer problem, the fractional problem is easy to solve:

- First, we arrange the items in non-decreasing order of efficiency, that is we assume that $p_1/w_1 \geq p_2/w_2 \geq \ldots \geq p_n/w_n$.

- Then, we select the most efficient item until doing so would exceed the capacity K. When this point is reached, we have reached the critical item denoted by s (and represented by the variable x_s) such that $\sum_{i=1}^{s-1} w_j \leq K$ and $\sum_{i=1}^{s-1} w_j + w_s > K$. If we put the maximum fraction of x_s that can fit into the knapsack ($K - \sum_{i=1}^{s-1} w_j$), then we obtain an optimal solution whose profit is $\hat{P} = \sum_{j=1}^{s-1} p_j + \frac{p_s}{w_s}(K - \sum_{i=1}^{s-1} w_j)$.

We will denote by relax(C), the fractional version of the KNAPSACK constraint C and by $\hat{P}$(relax(C)) the optimal profit of the relax(C). Fahle and Sellmann defined the following property:

Property 5. *Let i be an item and x_i the 0–1 variable associated with it.*

- *If $\hat{P}(relax(C \wedge (x_i = 0)) < B$, then $x_i = 1$ (because without x_i we cannot reach the minimum of the required profit) and i is named a mandatory item.*
- *If $\hat{P}(relax(C \wedge (x_i = 1)) < B$, then $x_i = 0$ (because by imposing item i we cannot reach the minimum of the required profit)[4] and i is named a forbidden item.*

In order to apply these rules as quickly as possible, Fahle and Sellmann identified all the items satisfying the previous property in linear time plus the time to sort the items by weight.

First, for each item i, they define s_i the critical item of $relax(C \wedge (x_i = 0))$. Then, they made two observations:

- If s is the critical item of $relax(C)$, then the items from 1 to $s-1$ are not forbidden and the items from $s+1$ to n are not mandatory.
- If, for two items i and j, we have $w_i \leq w_j$, then $s_i \leq s_j$.

Hence, if we traverse the items of $\{1, \ldots, s\}$ by non-decreasing weight, then all s_i items can be identified by a single linear scan of the items of $\{s, \ldots n\}$, because the search for the next critical item can begin at the location of the current critical item, and it always advance in one direction. If we constantly keep track of the sum of weights and the sum of profits of all items up to the current critical item, then we only need linear time to determine all mandatory elements.

Similarly, for each item i, we can define s_i the critical item of $relax(C \wedge (x_i = 1))$. And we can show that if we traverse the items in $\{s, \ldots, n\}$ by non-decreasing weight, each critical item is always to the left of the previous critical item, and we can identify the all forbidden elements with a single linear scan.

Sellmann [135] noticed that the same result can be obtained by sorting the items by non-increasing efficiency (i.e. $e_i = p_i/w_i$). This approach avoid needing to sort twice the items.

In addition, Katriel et al. [73] proved that if i and j are two items with $i \leq j \leq s$ and such that $e_i \geq e_j$ and $w_i \geq w_i$ and if i is not mandatory then j is not mandatory. They proposed a (complex) algorithm based on this idea.

[4] Note that imposing an item means that the problem is equivalent to the problem where the item is ignored and K becomes $K - w_i$ and B becomes $B - p_i$.

3.9 Packing Constraints

We propose to study different kinds of packing constraints, that is constraints which impose conditions on how items can be grouped together, for instance by pair or by limiting the number of consecutive variables having the same value.

3.9.1 Symmetric Alldiff Constraint (SYM-ALLDIFF)

The symmetric alldiff constraint constrains some entities to be grouped by pairs. It is a particular case of the ALLDIFF constraint, a case in which variables and values are defined from the same set S. That is, every variable represents an element e of S and its values represent the elements of S that are compatible with e. This constraint requires that all the values taken by the variables are different (similar to the classical ALLDIFF constraint) and that if the variable representing the element i is assigned to the value representing the element j, then the variable representing the element j is assigned to the value representing the element i.

Definition 22. Let X be a set of variables and σ be a one-to-one mapping from $X \cup D(X)$ to $X \cup D(X)$ such that

$\forall x \in X: \sigma(x) \in D(X); \forall a \in D(X): \sigma(a) \in X$ and $\sigma(x) = a \Leftrightarrow x = \sigma(a)$.

A **symmetric alldiff constraint** defined on X is a constraint C associated with σ such that:

$$\begin{aligned}\text{SYM-ALLDIFF}(X, \sigma) = \{&\tau \text{ s.t. } \tau \text{is a tuple on } X\\ &\text{and } \forall a \in D(X) : \#(a, \tau) = 1\\ &\text{and } a = \tau[index(C, x)] \Leftrightarrow \sigma(x) = \tau[index(C, \sigma(a))]\}\end{aligned}$$

This constraint has been proposed by Régin [115]. It is useful to be able to express certain items that should be grouped as pairs, for example in the problems of sports scheduling or rostering. Arc consistency can be established in $O(nm)$ after computing the consistency of the constraint which is equivalent to the search for a maximum matching in a non-bipartite graph, which can be performed in $O(\sqrt{n}m)$ by using the complex algorithm of [94].

In [115], another filtering algorithm is proposed. It is difficult to characterize it but its complexity is $O(m)$ per deletion. In this paper, it is also shown how the classical ALLDIFF constraint plus some additional constraints can be useful to solve the original problem. The comparison between this approach, the SYM-ALLDIFF constraint, and the ALLDIFF constraint has been carried out in [62].

3.9.2 Stretch Constraint (STRETCH)

This constraint has been proposed by Pesant [97]. It can be seen as the opposite of the SEQUENCE constraint. The STRETCH constraint aims to group the values by sequence of consecutive values, whereas the sequence is often used to obtain a homogeneous repartition of values.

A STRETCH constraint C is specified in terms of an ordered set of variables $X(C) = \{x_1, \ldots, x_p\}$ which take their values in $D(C) = \{v_1, \ldots, v_d\}$, and two set of integers $l = \{l_1, \ldots, l_d\}$ and $u = \{u_1, \ldots, u_d\}$, where every value v_i of $D(C)$ is associated with l_i the ith integer of L and u_i the ith integer of U. A STRETCH constraint states that if $x_j = v_i$, then x_j must belong to a sequence of consecutive variables that also take value v_i and the length of this sequence (the span of the stretch) must belong to the interval $[l_i, u_i]$.

Definition 23. A **stretch constraint** is a constraint C associated with a subset of values $V \subseteq D(C)$ in which each value $v_i \in D(C)$ is associated with two positive integers l_i and u_i and defined by

$$\begin{aligned}\text{STRETCH}(X, V, \{l_i\}, \{u_i\}) = \{t \text{ s.t. } & t \text{ is a tuple of } X(C) \\ & \text{and } \forall x_j \in [1 \ldots |X(C)|], (x_j = v_i \text{ and } v_i \in D(C)) \\ & \Leftrightarrow \exists p, q \text{ with } q \geq p, q - p + 1 \in [l_i, u_i] \\ & \text{s.t. } j \in [p, q] \text{ and } \forall k \in [p, q]\ x_k = v_i\}\end{aligned}$$

This constraint is used in rostering or in car sequencing problems (especially in the paint shop part).

A filtering algorithm has been proposed by Pesant [97]. The case of cyclic sequence (that is, the successor of the last variable is the first one) is also taken into account by this algorithm. Its complexity is in $O(m^2 \max(u) \max(l))$. Pesant also described some filtering algorithms for some variations of this constraint, notably one that deals with patterns and constrains the successions of patterns (that is some patterns cannot immediately follow some other patterns). An AC filtering algorithm based on dynamic programming and running in $O(nd^2)$, where n is the number of variables and d the number of values, is described in [61].

Note that this constraint can be easily represented by an automaton and so filtered by the techniques presented in the regular language based constraints section, notably by reformulating it.

3.9.3 k-diff Constraint (K-DIFF)

The K-DIFF constraint constrains the number of variables that are different to be greater than or equal to k.

Definition 24. A **k-diff constraint** is a constraint C associated with an integer k such that

$$\begin{aligned}\text{K-DIFF}(X, k) = \{\tau \text{ s.t. } & \tau \text{ is a tuple on } X(C) \text{ and} \\ & |\{a_i \in D(X(C)) \text{ s.t. } \#(a_i, \tau) \leq 1\}| \geq k\}\end{aligned}$$

This constraint has been proposed by Régin [111]. It is useful to model some parts of over-constrained problems where it corresponds to a relaxation of the ALLDIFF constraint.

A filtering algorithm establishing arc-consistency is detailed in [111]. Its complexity is the same as for the ALLDIFF constraint, because the filtering algorithm of

the ALLDIFF constraint is used when the cardinality of the maximum matching is equal to k. When this cardinality is strictly greater than k, we can prove that the constraint is arc consistent (see [111].)

3.9.4 Number of Distinct Values Constraint (NVALUE)

The number of distinct values constraint constrains the number of distinct values taken by a set of variables to be equal to another variable.

Definition 25. A **number of distinct values constraint** is a constraint C defined on a variable y and a set of variables X such that

$$\text{NVALUE}(X, y) = \{\tau \text{ s.t. } \tau \text{is a tuple on } X(C) \text{ and } |\{a_i \in D(X(C)) \text{ s.t. } \#(a_i, \tau) \leq 1\}| = \tau[y]\}$$

This constraint is quite useful for modeling some complex parts of problems.

A filtering algorithm based on the search of a lower bound of the dominating set problem [40] has been proposed by Beldiceanu [8]. When all the domains of the variables are intervals, this lead to an $O(n)$ algorithm if the intervals are already sorted.

3.9.5 Bin Packing Constraint (BIN-PACKING)

The bin packing problem is defined as follows: objects of different volumes must be packed into a finite number of bins of capacity V in a way that minimizes the number of bins used. Note that the problem can be viewed as the conjunction of two problems: for any bin, the problem to be solved is a subset sum problem and the goal is then to minimize the number of bins that are globally needed.

Shaw introduced the first constraint for one-dimensional bin packing [140] under the name of "Pack constraint". This constraint is mainly based on propagation rules incorporating knapsack-based reasoning. It also deals with lower bounds on the number of bins needed.

The more detailed and complete document about bin packing constraint is the PhD thesis of Schaus [130]. Some part of the following presentation is taken from his thesis.

We consider here a version which is more general than the classical bin packing because we accept bins with different capacities.

Definition 26. Let m be a set of bins, $L = \{l_1, \ldots, l_m\}$ be a set of loads (also named capacities) such that each bin i is associated with the load l_i, I be a set of n items, $S = \{s_1, \ldots, s_n\}$ be a set of sizes such that each item j is associated with the size s_j. A **bin packing constraint** is a constraint C defined on a set of n variables whose values express the bins in which the corresponding item may be placed, such that

$$\text{BIN-PACKING}(X, L, S, m) = \{\tau \text{ s.t. } \tau \text{ is a tuple of } X(C) \text{ and } \forall b = 1 \ldots m \ \Sigma_{i=1}^{n}((\tau[i] = b).s_i) \leq l_b\}$$

When the capacities (i.e. loads) are large, Sellmann proposes in [135] to palliate the pseudo-polynomial time by weakening the propagation strength by dividing down item sizes and bin capacities.

The filtering algorithm proposed by Shaw essentially works separately on each bin with a knapsack reasoning and detects non packable bins or non packable items into bins. The relaxation is that an item can be used in more than one bin.

Shaw also introduces a failure detection test based on fast bin-packing lower bound algorithms, because the general problem can be relaxed to the classical bin-packing problem with a fixed capacity (i.e. all bins have the same capacity).

Hence, bin-packing lower bounds can be very useful to detect quickly inconsistencies by comparing the lower bound to the number of available bins m. If the lower bound is larger than m in the reduced problem, then the constraint fails.

In order to be able to integrate the items that have been already packed into bins and the difference of capacities between items, Shaw proposed the following reduction:

- Take the maximum load $L_{\max}$ as capacity for all the bins
- The set of items to be packed is $U \cup \{a_1, \ldots, a_m\}$, where
  - U is the set of unassigned items
  - For each bin i, we create an item a_i for taking into account the already packed items into bin i and the fact that l_i may be smaller than $L_{\max}$. Let R_i be the sum of the size of the items already packed into i. The size of a_i is $R_i + L_{\max} - l_i$.

Then, Shaw uses the Martello and Toth lower bound denoted $\mathcal{L}_2$ [92] to compute a lower bound on the number of bins required. If this number is larger than m, then the constraint is not consistent. The bound $\mathcal{L}_2$ can be computed in linear time when the items are sorted by non increasing sizes. Therefore, the complexity is n plus the time to sort the a_i values.

Schaus proposed different other lower bounds based on different approaches. First, Schaus proposed to slightly modify the reformulation to the standard bin packing problem. Instead of considering for $L_{\max}$, the largest capacity of the bins, he proposed to considered the largest free space capacity of the bins (the size of the already assigned items are deduced from the maximum load of the bin). Then, he investigated the possibility to use the recent lower bound $\mathcal{L}_3$ of Labbé et al. [79].

At last and contrary to Shaw who proposed to work on each bin separately, Schaus proposed to consider the bins globally and to relax the belonging property and to accept to split an item among several bins. His idea is emphasized on the following example.

Consider 5 bins with capacity 5, and 11 items of size 1 and another of size 2 with the additional constraint that 9 items of size 1 and the item of size 2 can be placed only in the bins 4 and 5. Clearly, there is no solution because we need to put 9 items of size 1 and 1 of size 2, that is a size of 11 in two bins whose added capacity is only 10. Shaw's algorithm is unable to detect this inconsistency.

Schaus used a network flow to detect such inconsistencies. More information can be found in [130]. The algorithm is close to the one used to solve the preemptive

scheduling problem, hence this method is called filtering based on preemption relaxation. Experimental results confirm the improvement of Schaus's method over Shaw's one.

Shaus also considered a generalisation of the bin packing constraint which incorporates precedence constraints between items. A precedence constraint between items a and a' is satisfied if item a is placed in a bin B_i, and item a' in a bin B_j, with $i < j$. An original filtering algorithm dealing with that constraint is detailed in [134].

3.10 Graph Based Constraints

Some constraints are naturally defined as properties that a graph has to satisfy, or as problems of the graph theory. These constraints are named graph based constraints. They cause some problem of definition because it is often more convenient to define a graph variable (See [83] for the original introduction of the concept or [119] and [44] for a more detailed presentation) but graph variables are not classical CP variables.

3.10.1 Cycle Constraint (CYCLE)

We present here only the cycle/2 constraint. Here is the idea of this constraint [17]: "The cycle constraint was introduced in CHIP to tackle complex vehicle routing problems. The cycle/2 constraint can be seen as the problem of finding N distinct circuits in a directed graph in such a way that each node is visited exactly once. Initially, each domain variable x_i corresponds to the possible successors of the i^{th} node of the graph."

Definition 27. A **cycle constraint** is a constraint C associated with a positive integer n and defined on a set X of variables, such that:

$$\begin{aligned}\text{CYCLE}(X, n) = \{&\tau \text{ s.t. } \tau \text{is a tuple of } X(C)\\ &\text{and the graph defined from the arcs } (k, \tau[k])\\ &\text{has } n\text{connected components}\\ &\text{and every connected component is a cycle}\}\end{aligned}$$

This constraint is mentioned in the literature but no filtering algorithm is explicitly given. It is mainly used for vehicle routing problems or crew scheduling problems.

3.10.2 Path Constraint (PATH)

Some PATH constraints have been designed in existing solvers for a long time now. However, until recently, there were no publication about this constraint. In our

opinion, this comes from two facts. First, searching for a simple path from a node i to a node j which traverses a given node k is an NP-Complete problem! This means that it will be difficult to find a filtering algorithm able to find some mandatory nodes. Second, the relaxation of the notion of simple path for path leads to the simple filtering: let $d(i, j)$ be the minimum distance from node i to node j, then $d(i, k) + d(k, j)$ is a lower bound of the distance of a simple path from i to j which traverses the node k. This lower bound can easily be modelled by the constraint $d(i, k) + d(k, j) \leq d(i, j)$. This filtering has been used for solving a lot of problems, for instance network design [83].

Motivated by scheduling applications where it is often critical to evaluate the makespan or the earliest or latest completion time, Michel and Van Hentenryck [95] addressed the problem of maintaining the longest paths in directed acyclic graph (DAG). This problem is also known as Dynamic Heaviest Path [71]. The Heaviest Path problem or longest path problem is defined as follows: Given a DAG $G = (V, E)$ with a weight $w(e)$ for each edge e, compute for each node v the weight of the heaviest path from the source of G to v, where the weight of the path is the sum of the weight of the edges it contains. Then the Dynamic Heaviest Path problem or the Maintenance of Longest Path problem is to efficiently update this information when a small change is performed on G. Efficient means that the running time is proportional to the size of the portion of the graph that is affected by the change. There is no constraint definition here, but principles are close to the one used to maintain properties, which is a common task in CP, hence we mention these works. Several algorithms have been proposed and refined [71, 72, 95]. The best algorithm maintain the information in $O(\| \delta \| + |\delta| \log(|\delta|))$ for arc insertion and $O(\| \delta \|)$ for arc deletion, where $\| \delta \|$ and $|\delta|$ measure the change in the input and output. The same result has been obtained on Cyclic < 0 graphs, that is graph whose cycles have strictly negative lengths.

3.10.3 Path Partitioning Constraint (PATH-PARTITION)

Beldiceanu and Lorca [22] have proposed the path partitioning constraint which is defined as follows:

Definition 28. A **path partitioning constraint** is a constraint C defined on a digraph $D = (V, A)$, and associated with an integer k and a set $T \subseteq V$ of potential final nodes such that

$$\text{PATH-PARTITION}(X, k, T) = \{\tau \text{ such that } \tau \text{ is a tuple on } X(C) \text{ and the digraph defined by } \tau \text{ is a set of } k \text{ connected components such that each one is an elementary path that ends up in } T.\}$$

In general, the path partitioning problem is NP-Complete (even for $k = 2$). However, for some cases the problem becomes polynomial for instance for interval graph. It is also polynomial for acyclic digraphs. In this case, the problem can be

transformed into a flow problem, and it is possible to compute the minimum number of paths partitioning the digraph by computing a minimum feasible flow. The authors nicely exploited this idea to derive a filtering algorithm for this particular case. Then, they used the dominance theory to get a general necessary condition for the path partitioning constraint.

3.10.4 Shorter Path Constraint (SHORTER-PATH)

Sellmann defined the shorter path constraint [136, 139]. This constraint searches for paths whose length is smaller than a given threshold value.

Definition 29. A **shorter path constraint** is a constraint C defined on a graph G whose edges have a cost in W, and associated with an integer k and two nodes of G: a source s and a sink t

$$\text{SHORTER-PATH}(X, W, k, s, t) = \{\tau \text{ such that } \tau \text{ is a tuple on } X(C) \text{ and } \tau \text{defined a path from } s \text{to } t \text{ in } G \text{whose the sum of the edges is smaller than } k.\}$$

Sellmann proposed a relaxation of the constraint such that an efficient filtering algorithm can be designed. He also developed filtering for directed acyclic graph and general digraph with non-negative costs or graph that at least does not contain any negative weight cycle. An experimental study showed the advantage of this approach [53]. Unfortunately, it is not really easy to measure the difference of strengths between the new proposed filtering and the one we mentioned at the beginning of this section.

3.10.5 Tree Constraint (TREE)

The TREE constraint has been proposed by Beldiceanu et al. [18]. This constraint enforces the partitioning of a digraph into a set of vertex-disjoint anti-arborescences. An anti-arborescence is roughly a tree with the edges oriented in the opposite way: from the nodes to the root. A digraph A is an anti-arborescence with anti-root r if there exists a path from all vertices of A to r and the undirected graph associated with the digraph A is a tree.

Definition 30. A **tree constraint** is a constraint C defined on a digraph D, and associated with an integer k such that

$$\text{TREE}(X, k) = \{\tau \text{ such that } \tau \text{is a tuple on } X(C) \text{ and the digraph defined by } \tau \text{ is a set of } k \text{ vertex disjoint anti-arborescences}\}$$

Beldiceanu et al. [18] gave a linear consistency checking algorithm and an AC Filtering algorithm whose time complexity is in $O(nm)$ where n is the number of nodes of the digraph and m its number of arcs.

In another paper [19], the authors proposed to extend the original tree constraint with the following useful side constraints (we reproduce their presentation):

- Precedence constraints: a node u precedes a node v if there exists a directed path from u to v.
- Incomparability constraints: two nodes u and v are incomparable if there is no directed path from u to v or from v to u.
- Degree constraints that restrict the in-degrees of the nodes in the tree partition.
- Constraints on the number of proper trees, where a proper tree is a tree involving at least two nodes.

Combining the original problem with precedence or with incomparability constraints leads to an NP-Hard problem; therefore, the authors gave a set of necessary structural conditions combining the input graph with the graphs associated with these side constraints.

At last, two other variations of the tree constraint have been derived by Beldiceanu et al. [21] under the generic term of undirected forest:

- The RESOURCE-FOREST constraint. In this version, a subset of vertices are resource vertices and the constraint specifies that each tree in the forest must contain at least one resource vertex. They describe a hybrid-consistency algorithm that runs in $O(m+n)$ time for the resource-forest constraint and so improves the algorithm for the tree constraint.
- The PROPER-FOREST constraint. In this variant, there is no requirement about the containment of resource vertices, but the forest must contain only proper trees, i.e., trees that have at least two vertices each. They describe an $O(mn)$ hybrid-consistency algorithm.

3.10.6 Weighted Spanning Tree Constraint (WST)

The weighted spanning tree constraint (wst constraint) is a constraint defined on a graph G each of whose edges has an associated cost, and associated with a global cost K. This constraint states that there exists in G a spanning tree whose cost is at most K. This constraint has been introduced in a more general form by Dooms and Katriel [46]. Instead of considering the weighted spanning tree problem, they introduced the "Not-Too-Heavy Spanning Tree" constraint. This constraint is defined on undirected graph G and a tree T and it specifies that T is a spanning tree of G whose total weight is at most a given value I, where the edge weights are defined by a vector. The WST constraint is a simplified form of this constraint.

Definition 31. A **weighted spanning tree constraint** is a constraint C defined on a graph G, and associated with `cost` a cost function on the edge of G, and an integer K such that

$$\text{WST}(X, cost, K) = \{\tau \text{ such that } \tau \text{ is a tuple on } X(C) \text{ and the graph defined by } \tau \text{ is a tree whose cost is} \leq K\}$$

This kind of constraint does not often arise explicitly in real world applications, but it is used frequently as a lower bound of more complex problems such as Hamiltonian path or node covering problems. For instance, the minimum spanning tree is a well known bound of the travelling salesman problem.

It is straightforward to see that checking the consistency of this constraint is equivalent to finding a minimum spanning tree and to check if its cost is less than K. Moreover, arc consistency filtering algorithms are based on the computation for every edge e of the cost of the minimum spanning tree subject to the condition that the tree must contain e [46]. These two problems were solved for a long time. The search for a minimum spanning tree can be solved by several methods (Kruskal, Prim, etc.). The second problem is close of another problem called "Sensitivity Analysis of Minimum Spanning Trees" [145]. The best algorithms solve this problem in linear time. Unfortunately, they are quite complex to understand and to implement (see [43] or [91] for instance).

Régin proposed a simpler and easy to implement consistency checking and AC filtering algorithms for the WST constraint [122]. This algorithm is based on the creation of a new tree while running Kruskal's algorithm for computing a minimum spanning tree. Then, we find lowest common ancestors (LCA) in this tree by using the equivalence between the LCA and the range minimum query problem. A recent simple preprocessing leads to an $O(1)$ algorithm to find any LCA. The proposed algorithm is also fully incremental and try to avoid traversing all the edges each time a modification occurs. Its complexity is the same as the weighted spanning tree computation, that is linear plus the union-find operations.

Some variations of the weighted spanning tree constraint have been proposed.

For instance, Dooms and Katriel [45] introduced the Minimum spanning tree constraint, which is specified on two graph variables G and T and a vector W of scalar variables. The constraint is satisfied if T is a minimum spanning tree of G, where the edge weights are specified by the entries of W. They gave a bound consistency algorithm for all the variables.

On the other hand, the robust spanning tree problem with interval data has been addressed in [4]. This problem is defined as follows : given an undirected graph with interval edge costs, find a tree whose cost is as close as possible of that minimum spanning tree under any possible assignment of costs.

In conclusion of the Graph based Constraint section, we would like to mention some other works that have been carried out for some constraints like isomorphism, subgraph isomorphism or maximum clique. In fact, these algorithms are more dedicated to the resolution of a complex problem than to the filtering of a constraint corresponding to these problems. Hence, we do not detail them. Sorlin and Solnon have presented a filtering algorithm for the isomorphism problem [142, 143]. Zampelli et al. considered the subgraph isomorphism constraint [156]. This work improves Régin's algorithm [111]. At last, Régin defined a maximum clique constraint [118].

3.11 Order Based Constraints

3.11.1 Lexicographic Constraint (LEXICO)

The lexicographic ordering constraint $X \leq_{\text{lex}} Y$ over two ordered set of variables X and Y holds if the word defined by the assignment of X is lexicographically smaller than the word defined by the assignment of Y.

Definition 32. A **lexicographic ordering constraint** is a constraint C defined on two sets of ordered variables $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_n\}$ such that

$$\begin{aligned}\text{LEXICO} \leq (X, Y) = \{\tau \text{ s.t. } \tau \text{is a tuple on } X \cup Y \text{ s.t. either } \forall i \in [1 \ldots n]\ \tau[i] \leq \\ \tau[i + n]\} \text{ or } \exists j, 1 \leq j \leq n \text{with } \tau[j] < \tau[j + n] \\ \text{and } \forall i \in [1 \ldots j]\ \tau[i] = \tau[i]\}\end{aligned}$$

It is sometimes denoted by $X \leq_{\text{lex}} Y$.

A variation of this constraint is used to take into account the multidirectionnality in GAC-Schema [29]. Then, it has been formally defined by Frisch et al. [52] where a filtering algorithm is proposed. Carlsson and Beldiceanu showed that this constraint can be represented by an automaton [35], therefore with the reformulation of the automaton given in the section about formal based language constraints we have an efficient AC filtering algorithm.

A nice reformulation has been proposed by Quimper (Personal communication): we define a variable N whose value is the first index for which we will have $x_i < y_i$. Then for each i, we define the ternary constraint satisfying:

- $(N = i) \Rightarrow (x_i < y_i)$
- $(N < i) \Rightarrow (x_i = y_i)$
- $(N \leq i) \Rightarrow (x_i \leq y_i)$

Note that if $N > i$, then x_i and y_i are not constrained.

If all the constraints share only one variable N, then the bipartite constraint graph has no cycle and from Corollary 1, establishing arc consistency for this reformulation will establish arc consistency for the original constraint.

3.11.2 Sort Constraint (SORT)

This constraint has been proposed by Bleuzen–Guernalec and Colmerauer [32]: "A sortedness constraint expresses that an $n-$tuple $(y_1, \ldots, y_n)$ is equal to the $n-$tuple obtained by sorting in increasing order the terms of another $n-$tuple $(x_1, \ldots, x_n)$".

Definition 33. A **sort constraint** is a constraint C defined on two sets of variables $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_n\}$ such that

$$\begin{aligned}\text{SORT}(X, Y) = \{\tau \text{ s.t. } \tau \text{is a tuple on } X(C) \text{and } \exists f \text{ a permutation of } [1 \ldots n] \text{ s.t.} \\ \forall i \in [1 \ldots n]\ \tau[x_{f(i)}] = \tau[y_i]\}\end{aligned}$$

The best filtering algorithm establishing bound consistency has been proposed by Melhorn and Thiel [93]. Its running time is $O(n)$ plus the time required to sort the interval endpoints of the variables of X. If the interval endpoints are from an integer range of size $O(n^k)$ for some constant k, the algorithm runs in linear time, because this sort becomes linear.

A sort constraint involving 3 sets of variables has also been proposed by Zhou [157, 158]. The n added variables are used for making explicit a permutation linking the variables of X and those of Y. Well known difficult job shop scheduling problems have been solved thanks to this constraint.

3.12 Formal Language Based Constraints

Formal Language based Constraints are constraints defined from Automata or from Grammars. Recently, they have been intensively studied. They attracted a lot of researchers and this topic has certainly been the most active of the community in the last 5 years. However, the results that have been obtained are surprising because they tend to show that there is no need of specific algorithms for these constraints.

These constraints appeared 10 years after the graph based constraints which is also surprising because some computer scientists like kidding by saying that in computer science everything can be viewed from a graph theory or from the automaton theory.

3.12.1 Regular Language Based Constraints (REGULAR)

The explicit use of an automaton for representing a constraint and for deriving a filtering algorithm from this representation has been proposed by Carlsson and Beldiceanu [15, 34–36]. They aimed at finding a more efficient filtering algorithm for the lexicographic constraint. They were not the first to use an automaton in CP: Vempaty [154] introduced the idea of representing the solution set by a minimized deterministic finite automaton and Amilastre, in his Ph.D. Thesis [3], generalized this approach to non-deterministic automata and introduced heuristic to reduce their size. However, Carlsson and Beldiceanu were the first to design a filtering algorithm based on automata.

A bit afterwards and independently, Pesant [98, 99] introduced in a well written paper, the REGULAR constraint which ensures that the sequence of values taken by variables belongs to a given regular language.

We propose to study this constraint and the different filtering algorithms that have been associated with it.

First, we recall the definition of deterministic and non deterministic finite automata. This part is mainly inspired from [99].

A Deterministic Finite Automaton (DFA) is defined by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states.
- Σ is an alphabet, that is a set of symbols.
- $\delta : Q \times \Sigma \rightarrow Q$ is a partial transition function[5].
- q_0 is an initial state,
- $F \subseteq Q$ is the set of final (or accepting states).

Given an input string, the automaton starts in the initial state q_0 and processes the string one symbol at a time applying the transition function δ at each step to modify the current state. The string is accepted if and only if the last state reached belongs to the set of final states F. The language recognized by DFA's are precisely regular languages.

Thus, the definition of the regular membership constraint is immediate:

Definition 34. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a deterministic finite automaton. A **regular language membership constraint** is a constraint C associated with M such that

$$\text{REGULAR}(X, M) = \{\tau \text{ s.t. } \tau \text{is a tuple on } X(C) \text{and the sequence of values of } \tau \text{ belongs to the regular language recognized by } M\}$$

We reproduce Example 2 given in [99]. In rostering problems, the assignment of consecutive shifts must often follow certain patterns. Consider a sequence of 5 variables with $D(x_1) = \{a, b, c, o\}$, $D(x_2) = \{b, o\}$, $D(x_3) = \{a, c, o\}$, $D(x_4) = \{a, b, o\}$ and $D(x_5) = \{a\}$ subject to the following constraints : between a's and b's, a's and c's or b's and c's, there should be at least one o. In addition, the sequences a, o, c, b, o, a and c, o, b are forbidden. This problem can be represented by a finite automaton (See Fig. 9). Unfortunately, there is no explanation or help about the construction of the automata in any of the papers published on this topic.

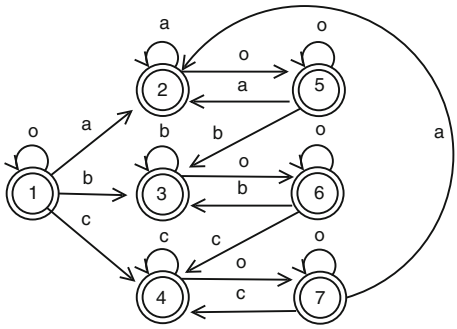


Fig. 9 A Deterministic Finite Automaton for a common pattern in rostering [99]. Integers are state. All states are final

[5] A partial function $\delta(q, x)$ does not have to be defined for any combination of $q \in Q$ and $x \in \Sigma$; and if $\delta(q, x)$ is defined and equal to q' then it does not exist another symbol y such that $\delta(q, y) = q'$.

Then, Pesant proposed a consistency checking and a filtering algorithm based on an idea similar as the one proposed by Trick for the KNAPSACK constraint [148]. A specific directed graph is built and the node that does not belong to some paths are deleted and this lead to domain reductions. For a constraint C, we will denote by $LD(C)$ this digraph. it is built as follows:

The digraph contains several layers. Each layer contains a different node for each state of the automaton. More precisely, if $\{q_0, q_1, \ldots, q_s\}$ are the states, then the layer i contains the nodes $\{q_0^i, q_1^i, \ldots, q_s^i\}$. If n variables are involved in the constraint, then there are $n+1$ layers. There are arcs only between nodes of consecutive layers. The arcs between layer i and layer $i+1$ correspond to the variable x_i. An arc from node q_j^i to node q_k^{i+1} is admissible for inclusion only if there exists some $v \in D(x_i)$ such that $\delta(q_j, v) = q_k$. The arc is labelled with the value v allowing the transition between two states. In the first layer, the only node with outgoing arcs is q_0^1 since q_0 is the only initial state. Figure 10 shows the layered digraph associated with previous example.

The computation of the consistency of the constraint C and the establishment of arc consistency correspond to path property in the layered digraph $LD(C)$. Pesant proved that

- The constraint is consistent if and only if there exists a path from q_0^1 to a node of the layer $n+1$
- A value (x_i, a) is consistent with C if and only if there is a path q_0^1 to a node of the layer $n+1$ which contain an arc from a node of layer i to a node of layer $i+1$ labelled by a.

The implementation of these properties can be done simply by removing all the nodes of $LD(C)$ which are not contained in any path from q_0^1 to a node of the layer $n+1$. The deletion of these nodes lead to the removal of arcs and so may lead to the disappearance of arcs labelled by a given value. In this case, this means that a

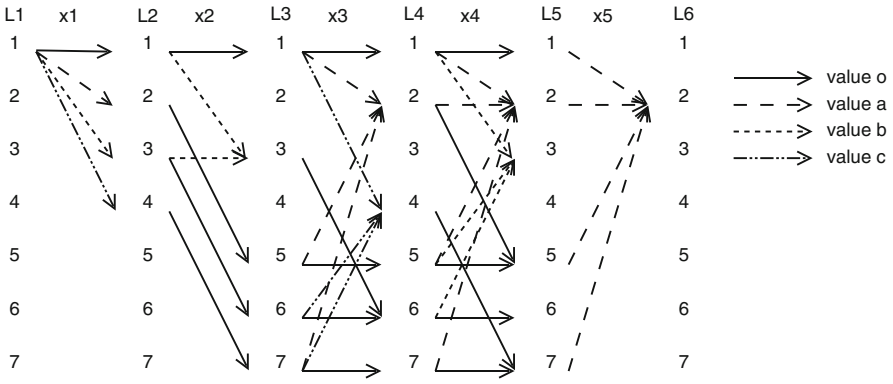


Fig. 10 The initial layered digraph associated with a deterministic finite automaton for a common pattern in rostering [99]. Li represents the nodes of the layer. For convenience, a node q_j^i, that is the node of the state q_j in the layer i is represented by the index j

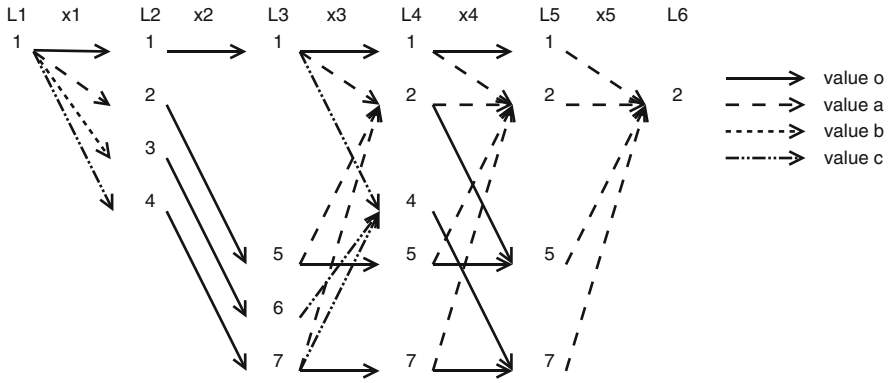


Fig. 11 The "after pruning" layered digraph associated with a deterministic finite automaton for a common pattern in rostering [99]. Li represents the nodes of the layer. For convenience, a node q^i_j, that is the node of the state q_j in the layer i is represented by the index j

value is no longer consistent with the constraint and can be deleted from its domain. Figure 11 is an example of such a deletion process for the previous example. For instance, if nodes $(L5, 3)$ and $(L5, 6)$ have no successor, then they will be deleted. Thus, node $(L4, 6)$ will be removed and also node $(L3, 3)$. Then, there is no longer any dotted edge for x_2 which means that value (x_2, b) is deleted.

Pesant proved that the identification of inconsistent values can be performed by 2 breadth first searches: one in the digraph starting from the node corresponding to the initial state and one in the transpose digraph starting from nodes corresponding to final states. Each node without any outgoing arc or any incoming arc is deleted. Thus, if n is the number of variables involved in C, d the number of symbols and s be the number of states, then the consistency and the arc consistency of C can be established in $O(nds)$ [99]. Pesant also proposed an incremental versions of the filtering algorithm by maintaining the layered digraph and by considering the deletion of the value of a variable.

At the same time,[6] Beldiceanu et al. proposed a nice reformulation of the problem [12, 15]. The main idea is to reformulate the automaton into transition constraints. A transition constraint is a constraint corresponding to the transition function. It involves 3 variables, two having for values the states and one having for value the symbols. In other word, the allowed combinations of values of a transition constraint is the set of 3-ary tuples (q_i, v, q_j) such that $\delta(q_i, v) = q_j$. We will denote by $T(\delta, x, y, z)$ such a constraint defined on the variable x, y and z. Then, instead of defining explicitly the graph like with Pesant's algorithm, only $n + 1$ transition constraints are defined. All the transition constraints are defined from the same set of tuples. Each transition constraint is defined on 2 state variables that is variables

[6] This is really at the same time because the two papers were presented during the same session at the same conference: CP'04.

whose domain is the set of possible states and one x variable containing symbols. The first transition constraint is $T(\delta, Q_1, x_1, Q_2)$ where Q_1 contains only the initial state, that is the state q_0, x_1 is the first variable and Q_2 is the variable representing the state that can be reached from q_0 by using the transition δ involving a symbol of x_1. The second transition constraint is $T(\delta, Q_2, x_2, Q_3)$ and so on until $T(\delta, Q_n, x_n, Q_{n+1})$ which is the last one and where Q_{n+1} contains only the final state of the automaton.

The transition constraint can be easily built from an automaton: each arc of the automaton corresponds to a tuple of the constraint. More precisely, if there is an arc (i.e a transition) from the state 3 to the state 6 with the symbol o then the triplet $(3, o, 6)$ is an allowed combination of the transition constraint and conversely. The following table contains all the triplets of the transition constraint corresponding to the automata of the previous example:

$(1, o, 1)$	$(1, a, 2)$	$(1, b, 3)$	$(1, c, 4)$	$(2, a, 2)$
$(2, o, 5)$	$(3, b, 3)$	$(3, o, 6)$	$(4, o, 7)$	$(4, c, 4)$
$(5, o, 5)$	$(5, a, 2)$	$(5, b, 3)$	$(6, o, 6)$	$(6, b, 3)$
$(6, c, 4)$	$(7, o, 7)$	$(7, c, 4)$	$(7, a, 2)$	

Thus, the reformulation replaces the REGULAR constraints by the constraints

$$T_1 = T(\delta, Q_1, x_1, Q_2), T_2 = T(\delta, Q_2, x_2, Q_3), T_3 = T(\delta, Q_3, x_3, Q_4)$$
$$T_4 = T(\delta, Q_4, x_4, Q_5), T_5 = T(\delta, Q_5, x_5, Q_6)$$
$$\text{with } Q_1 = 1.$$

The strong result of the paper of Beldiceanu et al. [12] is that the establishment of arc consistency for the reformulate problem is equivalent to the establishment of the arc consistency for the REGULAR constraint because the reformulated problem satisfies Corollary 1 of Preliminaries Section.

The establishment of arc consistency for a constraint of arity r with t allowed tuples and the maintenance of this arc consistency can be performed in $O(rt)$ (See Proposition 1 of TABLE constraint Section). Thus, for n transition constraints of arity 3, we can establish arc consistency in $O(n|\delta|)$ which is equivalent to $O(nds)$ with d symbols and s states because by definition of a deterministic finite automaton when the symbol and the state are given for δ, then there is only one result, so $O(|\delta|) = O(ds)$. The overall time complexity is exactly the same as the Pesant's algorithm.

In addition, Quimper et al. [106, 107] showed that in practice this method performs very well and better for some instances than the Pesant's approach. Furthermore, having a direct to access the transition constraints or to the state variables may be useful to model some others constraints easily. Either by changing the tuples of the transition constraints or by defining new constraints involving the state variables. For instance, the constraint $Max(N, \{x_1, \ldots, x_n\})$ which ensures that N is the maximum value taken by x_1 to x_n may be implemented by a set of ternary constraint $Q_{i+1} = max(x_i + Q_i)$ [107]. Hence, the reformulation seems to be definitely an interesting approach.

However, this approach also shows clearly the limit of such a model: each variable representing the symbols is involved in only one constraint and each state

variable is involved in at most 2 constraints. Therefore, it will be difficult to express some more complex constraints notably the one involving several variables in any order.

In order to improve the expressiveness of the REGULAR constraint Beldiceanu et al. [12, 15] proposed two improvements : the use of Non deterministic finite automata instead of deterministic finite automata DFA and the addition of counters. We will detail the first aspect and not the second because it leads to a more complex reformulation and the conditions for establishing arc consistency are also more complex. We encourage the reader to look at the paper of Beldiceanu et al. for more information.

Non deterministic Finite Automaton (NFA) differs from DFA only by the definition of δ the transition function. In a DFA, δ is a function which returns a state from a state and a symbol whereas in an NFA δ returns a set of state from a state and a symbol. NFA have the same power as DFA, in the sense that they recognize only regular languages, but they can do so with exponentially fewer states than a DFA.

The reformulation used to model a DFA can also be used to model an NFA. The transition constraints are changed in order to take into account the δ function of an NFA. This does not cause any particular problem because TABLE constraints does not make any assumption on the properties of the tuples it contains. However, the number of tuples of each constraint is increased from ds to ds^2 because for a given state and a given symbol we can have several states in an NFA. The total time complexity for establishing arc consistency is in $O(nds^2)$, that is a factor of s more than for a DFA. Since the number of state for an NFA may have exponentially fewer states than an equivalent DFA this can be worthwhile.

3.12.2 Context-Free Language Based Constraints(GRAMMAR)

Sellmann noticed that we can see any assignment of variables $x_1, \ldots x_n$ as a word $D(x_1) \ldots D(x_n)$ whose letters are the values assigned to the variables. Therefore, it is convenient to use formal languages to describe certain features that we would like our solution to exhibit. Since languages are recognized by grammars, he defined the GRAMMAR Constraint.

First, we recall the formal definition of a grammar. A grammar is a set of rules for forming strings in a formal language. These rules that make up the grammar describe how to form strings from the language's alphabet that are valid according to the language's syntax.

Definition 35. A grammar is a tuple (N, Σ, P, S) where

- N is a finite set of non-terminal symbols
- Σ a finite set of terminal symbols (the alphabet)
- S a start symbol
- P a set of production rules such that $P \subseteq (N \cup \Sigma)^* N (N \cup \sigma)^* \times (N \cup \sigma)^*$ We define by L_G the language given by G

We recall that * is the Kleene star operator : if V is a set of symbol, then V^* is the set of all string over symbol in V including the empty set ϵ.

Definition 36. Let $G = (N, \Sigma, P, S)$ be a grammar. A **grammar constraint** is a constraint C associated with G defined by

$$\text{GRAMMAR}(X, G) = \{\tau \text{ s.t. } \tau \text{is a tuple on } X(C) \text{ and the sequence of values of } \tau \text{is word of } L_G\}$$

Formal language theory is very rich and propose some categorization of languages. For instance, we have already seen regular languages. As we mentioned it, DFAs and NFAs recognize regular languages. Left regular grammars also generate exactly all regular languages. Hence, there is a direct one-to-one correspondence between the rules of a left regular grammar and those of a non-deterministic finite state automaton, such that the grammar generates exactly the language the automaton accepts.

Definition 37. A regular grammar is a formal grammar (N, Σ, P, S) where the rules of P are of the following forms

- $A \rightarrow a$, where A is a non-terminal in N and a is a terminal in Σ
- $A \rightarrow \epsilon$, where A is in N and ϵ is the empty string. and either
- $A \rightarrow Ba$, where A and B are in N and a is in Σ. In this case, the grammar is a left regular grammar or of the form
- $A \rightarrow aB$, where A and B are in N and a is in Σ. In this case, the grammar is a right regular grammar.

An example of a right regular grammar G with $N = \{S, A\}$, $\Sigma = \{a, b, c\}$, P consists of the following rules:
$S \rightarrow aS$
$S \rightarrow bA$
$A \rightarrow \epsilon$
$A \rightarrow cA$
and S is the start symbol. This grammar describes the same language as the regular expression $a * bc*$ (See Wikipedia).

It is possible to automatically build a finite automaton from a regular grammar. Let $G = (N, \Sigma, P, S)$ be a left regular grammar, then the automaton $A = (Q, \Sigma, \delta, q_0, F)$ equivalent to G is defined as follows:

- $Q = N \cup \{q_t\}$ where q_t is a new terminal state.
- $q_0 = S$
- The rules of P define δ and F:
  - if $P_i = A \rightarrow aB$ then $\delta(A, a) = B$
  - if $P_i = A \rightarrow a$ then $\delta(A, a) = q_t$
  - if $P_i = A \rightarrow \epsilon$ then $A \in F$

Similar rules can be defined if the grammar is right regular.

Such a transformation means that we can use the filtering algorithms (or the reformulations) designed for the DFA or the NFA to establish arc consistency for regular grammars. So for regular grammars, we already have interesting filtering algorithms.

However, grammars are more general than automata and there exist more complex and more powerful grammars. Sellmann proposed to investigate constraints based on grammars higher up in the Chomsky hierarchy [70, 138].

We recall the Chomsky's hierarchy. We reproduce here the presentation of [138]:

Definition 38. Let α and β be string of symbols and non-terminal; and $G = (N, \Sigma, P, S)$ be a grammar.

- If for all productions $(\alpha \rightarrow \beta) \in P$ we have β is at least as long as α, then the grammar is context sensitive also named Type-1 grammar;
- $P \subseteq N \times (N \cup \Sigma)^*$ then the grammar is context-free also named Type-2 grammar;
- $P \subseteq N \times (\Sigma^* N \cup \Sigma)^*$ then the grammar is regular also named Type-3 grammar.

Note that a context-free grammar is a grammar in which all the production rules are of the form $V \rightarrow w$, where V is a non terminal symbol and w a string of terminal and/or non-terminal. The "context-free" notion comes from the fact that a non-terminal V can always be replaced by w, without considering its context.

Unfortunately, it is PSPACE Complete to decide if a Type-1 grammar recognizes a given word. Since the Type-3 are equivalent to automaton for which filtering algorithms exist, Sellmann proposed to consider the context-free grammar.

The consistency algorithm and the filtering algorithm establishing arc consistency designed by Sellmann are mainly based on the Cocke–Younger–Kasami (CYK) algorithm that determines whether a string can be generated by a given context-free grammar and, if so, how it can be generated. The algorithm employs bottom-up parsing and dynamic programming.

The standard version of CYK operates on context-free grammars given in Chomsky normal form (CNF). Thus, Sellmann proposed to work with grammar under this form. The complexity of the algorithms is asymptotically the same as the CYK complexity that is $O(n^3|P|)$. Kadioglu and Sellmann improved the behaviour and the incremental aspect of the algorithm [69].

In parallel to the work of Sellmann and at the same time[7], Quimper and Walsh [106–109] proposed also a GRAMMAR constraint dedicated to context-free grammar and associated with two filtering algorithms : the first one based on CYK and the second one based on another parser written by Earley [47]. The algorithm based on the CYK parser is different from Sellmann's algorithm but has the same complexity. The second algorithm is original and has the same time complexity as the others: $O(n^3|P|)$.

[7] Once again it was exactly at the same time, because the two papers were presented at the same conference : CP'06

On the other hand, Katsirelos et al. [75] showed that it is possible to reformulate the GRAMMAR constraint into a REGULAR constraint. The transformation is currently quite complex but it is promising.

These algorithms are complex and we will not detail them in this chapter.

In conclusion about the formal language constraints, we note that it seems not so easy to define constraints via automata or grammars. The future will show whether it is really the case or not.

4 Filtering Algorithm Design

There are several ways to design a filtering algorithm associated with a constraint. However, for global constraints, we can identify different and important types of filtering algorithms:

- *Filtering algorithms based on a generic algorithm*: GENERIC constraints, TABLE constraints, REGULAR constraints, GRAMMAR constraints, etc. In this case, there is no new algorithm to write provided that an algorithm checking the consistency of the constraint is given or the list of allowed combinations is computed (TABLE constraint) or an automaton is designed (REGULAR constraint) or a grammar is defined (GRAMMAR constraint).
- *Filtering algorithms based on model reformulation*. There are several possibilities:
  - Either from the simultaneous presence of constraints the filtering algorithm consists of adding some new constraints,
  - Or a reformulation of the constraint is made, like for the REGULAR constraint or the SUBSET-SUM.
  - Or the constraint is remodelled as a flow like for the SEQUENCE constraint, or by as set of cardinality constraints like for the CARD-MATRIX constraint.
- *Filtering algorithms based on existing algorithms*. This idea is to be helped by existing results, like the ones based on dynamic programming (SUBSET-SUM) or flow (GCC).
- *Filtering algorithms based on ad-hoc algorithms*.

For the two first cases, there is no real new algorithm that is written.

We propose to discuss in more detail the constraint addition idea, the reuse of existing properties and the design of ad-hoc algorithms.

For convenience, we introduce the notion of pertinent filtering algorithm for a global constraint:

Definition 39. A filtering algorithm associated with $\mathcal{C} = \wedge\{C_1, C_2, \ldots, C_n\}$ is **pertinent** if it can remove more values than the propagation mechanism called on the network $(\cup_{C \in \mathcal{C}} X(C), \mathcal{D}_{X(\mathcal{C})}, \{C_1, C_2, \ldots, C_n\})$.

4.1 Algorithms Based on Constraints Addition

A simple way to obtain a pertinent filtering algorithm is to deduce from the simultaneous presence of constraints, some new constraints. In this case, the global constraint is replaced by a set of constraints that is a superset of the one defining the global constraint. That is, no new filtering algorithm is designed.

For instance, consider a set of 5 variables: $X = \{x_1, x_2, x_3, x_4, x_5\}$ with domains containing the integer values from 0 to 4; and four constraints ATLEAST$(X, 1, 1)$, ATLEAST$(X, 1, 2)$, ATLEAST$(X, 1, 3)$, and ATLEAST$(X, 1, 4)$ which mean that each value of $\{1, 2, 3, 4\}$ has to be taken at least one time by a variable of X in every solution.

An ATLEAST$(X, \#time, val)$ constraint is a local constraint. If such a constraint is considered individually, then the value *val* cannot be removed while it belongs to more than one domain of a variable of X. A filtering algorithm establishing arc consistency for this constraint consists of assigning a variable x to *val* if and only if x is the only one variable whose domain contains *val*.

Thus, after the assignments $x_1 = 0$, $x_2 = 0$, and $x_3 = 0$, no failure is detected. The domains of x_4 and x_5, indeed, remain the same because every value of $\{1, 2, 3, 4\}$ belongs to these two domains. Yet, there is obviously no solution including the previous assignments, because 4 values must be taken at least 1 time and only 2 variables can take them.

For this example, we can deduce another constraint by applying the following property: if 4 values must be taken at least 1 time by 5 variables, then the other values can be taken at most $5 - 4 = 1$, that is we have ATMOST$(x, 1, 0)$.

This idea can be generalized for a GCC(X, l, u). Let card(a_i) be a variable associated with each value a_i of $D(X)$ which counts the number of domains of X that contain a_i. We have $l_i \leq \text{card}(a_i) \leq u_i$. Then, we can simply deduce the constraint $\sum_{a_i \in D(X)} \text{card}(a_i) = |X|$; and each time the minimum or the maximum value of card(a_i) is modified, the values of l_i and u_i are accordingly modified and the GCC is modified.

This method is usually worthwhile because it is easy to implement. However, the difficulty is to find the constraints that can be deduced from the simultaneous presence of other constraints.

4.2 Filtering Algorithms Based on Existing Algorithms

The idea is to link the global constraints and some common properties of the graph theory and then to automatically derive filtering algorithm from these properties. It has mainly been proposed by Beldiceanu.

More precisely, the property that has to be satisfied by a global constraint may sometimes be expressed by some properties in graph theory. This is clear for some global constraints based on graph theory, like an assignment problem (see the ALLDIFF constraint for instance) or a TREE constraint for which the equivalent

properties defining a tree are well known and simple (see Chap. 3 in [26]): a tree is connected graph without cycle, or a tree is a connected graph with $n-1$ arcs, etc. Of course, the goal is to reduce the number of properties that are considered and try to factorize the results.

First, Beldiceanu proposed to describe global constraints in term of graph properties [7], but the goal, at that time, was mainly to try to express the constraints in given formalism and to organize the existing global constraints (this will lead to the well known catalogue of global constraints [9]). In this model, a constraint is represented by a graph whose nodes correspond to variables involved in the constraint and whose arcs correspond to primitives constraints. At the beginning, it is not known which of these primitives constraints will be respected. For instance, an NVALUE constraint imposes that a set of variables take at most n different values. The graph representing this constraint will have edges corresponding to binary constraints of equality, but we do not know which ones are going to be violated and which ones will be respected. At the end, the satisfied constraints have to respect some properties, for the NVALUE constraint, the number of connected component of the graph has to be equal to n. Thus, this method identifies the solutions of a global constraint to the sub-graphs of a unique initial digraph, which satisfy a set of properties defining the constraint. Then, Hanak [60] tried to exploit this description in order to derive automatically filtering algorithms. However, this is really from 2005 and 2006, that Beldiceanu proposed to consider the most common properties and to derive from them and from the initial digraph some boundaries about the possible sub-graphs that are solutions [16]. These boundaries provide the necessary conditions to the satisfiability of a lot of global constraints. Then, some filtering algorithms may be automatically derived from these properties [13, 14, 23, 24]: a filtering algorithm consists in the identification of the arcs of the initial digraph that belong (or not) to the sub-graphs corresponding to the solutions of the constraints. Therefore, a filtering algorithm removes some edges that do not satisfy some properties on the digraph (for instance on the absence of cycle) or imposes some edges to be in the digraph in order to satisfy some other properties (for instance, if the graph must be connected, any bridge will be imposed).

There is a relation between Beldiceanu's work and the notion of Graph Variable introduced during the Rococo project [83]. Graph variables have been presented in detail in [119]. Then, they have been more formalized in [44]. A graph variable is a variable that will be instantiated to a sub-graph of an initial graph while respecting some properties. The list of edges, the list of nodes, and the neighbourhood of each node, can be viewed as set variables and the filtering algorithm remove from the possible part of these sets or add to the required part of these sets some elements in order to respect some constraints. In addition, it is possible to define a condition for the existence of an arc (for instance, that an equality constraint exists between the variables corresponding to its extremities), therefore the two approaches are certainly close.

This method is quite interesting when the problem can be naturally expressed as a graph problem. This can lead to elegant solutions for designing filtering algorithms. Unfortunately, it is not obvious to represent some problems in graph theory (a sum

or a knapsack seem to be good examples) and this method did not bring any major result. Maybe, the research has been too much focused on the factorization of strong properties.

4.3 Dedicated Filtering Algorithms

The last method to design a pertinent filtering algorithm is to use the structure of the constraint in order to define some properties identifying that some values are not consistent with the global constraint.

The use of the structure of a constraint has four main advantages:

- The search for a support can be speeded up.
- Some inconsistent values can be identified without explicitly checking for every value whether it has a support or not.
- The call of the filtering algorithm, that is the need to check the consistency of some values, can be limited to some events that can be clearly identified.
- A better incrementality.

For instance, consider the constraint $(x < y)$, then:

- The search for a support for a value a of $D(x)$ is immediate because any value b of $D(y)$ such that $b > a$ is a support, so a is consistent with the constraint if $a < \max(D(y))$.
- We can immediately state that $\max(D(x)) < \max(D(y))$ and $\min(D(y)) > \min(D(x))$ which means that all values of $D(x)$ greater than or equal to $\max(D(y))$ and all values of $D(y)$ less than or equal to $\min(D(x))$ can be removed.
- Since the deletions of values of $D(y)$ depends only on $\max(D(y))$ and the deletions of values of $D(x)$ depends only on $\min(D(x))$, the filtering algorithm must be called only when $\max(D(y))$ or $\min(D(x))$ are modified. It is useless to call it for the other modifications.

A good example of such filtering algorithm is given in [96]. We propose here a simpler example for a well-known problem: the n-queens problem.

The n-queens problem involves placing n queens on a chess board in such a way that none of them can capture any other using the conventional moves allowed by a queen. In other words, the problem is to select n squares on a chessboard so that any pair of selected squares is neither aligned vertically, horizontally, nor diagonally.

This problem is usually modeled by using one variable per queen; the value of this variable represents the column in which the queen is set. If x_i represents the variable corresponding to queen i (that is the queen in row i), the constraints can be stated in the following way. For every pair (i, j), with $i \neq j$, $x_i \neq x_j$ guarantees that the columns are distinct; and $x_i + i \neq x_j + j$ and $x_i - i \neq x_j - j$ together guarantee that the diagonals are distinct.

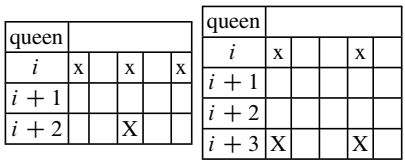


Fig. 12 Rules of the ad-hoc filtering algorithm for the n-queens problem

These relations are equivalent to defining an ALLDIFF constraint on the variables x_i, an ALLDIFF constraint on the variables $x_i + i$, and an ALLDIFF constraint on the variables $x_i - i$.

We propose to use a specific constraint that is defined on x_i and try to take into account the simultaneous presence of three ALLDIFF constraints. Consider a queen q: if there are more than three values in its domain, this queen cannot lead to the deletion of one value of another queen, because three directions are constrained (the column and the two diagonals) and so at least one value of queen q does not belong to one of these directions. Therefore, a first rule can be stated:

- While a queen has more than three values in its domain, it is useless to study the consequence of the deletion of one of its values because nothing can be deduced. From a careful study of the problem, we can deduce some other rules (see Fig. 12):
- If a queen i has 3 values $\{a, b, c\}$, with $a < b < c$ in its domain then the value b of queens $i - k$ and the value b of queen $i + k$ can be deleted if $b = a + k$ and $c = b + k$.
- If $D(x_i) = \{a, b\}$ with $a < b$, then the values a and b of queens $i - (b - a)$ and of queens $i + (b - a)$ can be deleted.
- If $D(x_i) = \{a\}$, then the value $a + j$ for all queens $i + j$, and the value $a - j$ for all queens $i - j$ can be deleted.

Therefore, a careful study of a constraint can lead to efficient filtering algorithms. This method is certainly the most promising way. However, it implies a lot of work. In [30], it is proposed to try to use first the general arc consistency algorithm in order to study if the development of a powerful filtering algorithm could be worthwhile for the considered problem. Using the solver itself then solves the consistency of the constraint.

5 Discussion

5.1 Incrementality and Amortized Complexity

Two points play an important part in the quality of a filtering algorithm: the incrementality and the amortized complexity. These points are linked together.

The incremental behaviour of a filtering algorithm is quite important in CP, because the algorithms are systematically called when a modification of a variable

involved in the constraint occurs. However, the algorithm should not be focus only on this aspect. Sometimes, the computation from scratch can be much more quicker. This point has been emphasized for general filtering algorithms based on the list of supported values of a value [31]. An adaptive algorithm has been proposed which outperforms both the non-incremental version and the purely incremental version. There are two possible ways to improve the incremental behaviour of the algorithm:

- The previous computations are taken into account when a new computation is made in order to avoid doing the same treatment twice. For instance, this is the idea behind the last support in some general filtering algorithms.
- The filtering algorithm is not systematically called after each modification. Some properties that cannot lead to any deletions are identified, and the filtering algorithm is called only when these properties are not satisfied. For instance, this is the case for the model we present to solve the n-queens problem.

When a filtering algorithm is incremental, we can expect to compute its amortized complexity. This is the complexity in regard to the number of deletions, or for one branch of the tree-search. This is why the complexity can be analyse after a certain number of modifications. The amortized complexity is often more accurate for filtering algorithm. Moreover, it can lead to new interesting algorithms that are not too systematic. For instance, there is a filtering algorithm for the symmetric alldiff constraint that is based on this idea. The filtering algorithm establishing arc consistency calls another algorithm A n times; therefore, its complexity is $n \times O(A)$. Another algorithm has been proposed in [115], which can be described as follows: pick a variable then run A, and let k be the number of deletions made by A. Then you can run A for k other variables. By proceeding like that, the complexity is $O(A)$ per deletions. Of course, the algorithm does not necessarily establish arc consistency but this may be a good compromise.

5.2 Incomplete Algorithms and Fixed-Point Property

Some global constraints correspond to NP-Complete problems. Hence, it is not possible to check polynomially the consistency of the constraint to establish arc consistency. Nevertheless, some filtering algorithms can be still proposed. This is the case for a lot of constraints: the DIFF-N constraint, the SEQUENCE constraint, the NVALUE constraint, the KNAPSACK constraint, the BIN-PACKING constraint and so on.

When the problem is NP-Complete, the filtering algorithm considers a relaxation, which is no longer difficult. Currently, the filtering algorithms associated with such constraints are independent of the definition of the problem. In other words, a propagation mechanism using them will reach a fixed-point. That is, the set of values that are deleted is independent from the ordering according to the constraints defined and from the ordering according to the filtering algorithms called. In order to guarantee

such a property, the filtering algorithm is based either on a set of properties that can be exactly computed (not approximated), or on a relaxation of the domains of the variables (that is, the domains are considered as ranges instead of as a set of enumerated values). The loss of the fixed-point property leads to several consequences: the set of values deleted by propagation will depend on the ordering along with the stated constraints and on the ordering along with the variables involved in a constraint. This means that the debugging will be a much more difficult task because fewer constraints can lead to more deleted values, and more constraints can lead to fewer deleted values.

In the future, we will certainly need filtering algorithms with which the fixed-point property of the propagation mechanism will be lost, because more domain-reductions could be done with such algorithms. For instance, suppose that a filtering algorithm is based on the removal of nodes in a graph that belong to a clique of size greater than k. Removing all the values that do not satisfy this property is an NP-Complete problem; therefore, the filtering algorithms will not be able to do it. However, some of these values can be removed, for instance by searching for one clique for every node (if a clique of size $\geq k$ is found, then the node is deleted else it remains in the graph). The drawback of this approach is that it will be difficult to guarantee that for a given node the graph will be traversed according to the same ordering of nodes. This problem is closed to the canonical representation of a graph; and currently this problem is unclassified: we do not know whether it is NP-Complete or not.

5.3 Identification of the Filtering

It is important to understand precisely the advantages and the drawbacks of some filtering algorithms, notably when the underlined problem of the constraint is an NP-Complete problem. In this case, we cannot establish arc consistency. Thus, a relaxation of the problem is considered and then some rules leading to domain reduction of the variables are defined. However, it is not really clear to figure out the filtering performance even for the relaxed problem.

It could be much more convenient if each constraint was associated with well defined filtering algorithm. For instance, if a constraint corresponds to an NP-Complete problem, then it could be interesting to show that the filtering algorithm established for this constraint is in fact an AC filtering algorithm for a specific relaxation of the constraint. It would help us a lot in the comparison of filtering algorithms.

5.4 Closure

In general, a filtering algorithm removes some values that do not satisfy a property. The question is "Should a filtering algorithm be closed with regard to this property?"

Consider the values deleted by the filtering algorithm. Then, the consequences of these new deletions can be:

- Taken into account by the same pass of the filtering algorithm.
- Or ignored by the same pass of the filtering algorithm.

In the first case, there is no need to call the filtering algorithm again, and in the second case, the filtering algorithm should be called again. When the filtering algorithm is good, usually the first solution is the good one, but when the filtering algorithm consists of calling another algorithm for every variable or every value, it is possible that any deletion calls the previous computations into question. Then, the risk is to have to check again and again the consistency of some values. It is also possible that the filtering algorithm internally manages a mechanism that is closed to the propagation mechanism of the solver, which is redundant.

In this case, it can be better to stop the filtering algorithm when some modifications occur in order to use the other filtering algorithms to further reduce the domains of the variable and to limit the number of useless calls.

5.5 Power of a Filtering Algorithm

Arc consistency is a strong property, but establishing it costs sometimes in practice. Thus, some researchers have proposed to use weaker properties in practice. That is, to let the user to choose which type of filtering algorithm should be associated with a constraint. In some commercial CP Solvers, such as ILOG-CP, the user is provided with such a possibility. Therefore, it is certainly interesting to develop some filtering algorithms establishing properties weaker than arc consistency. However, arc consistency has some advantages that must not be ignored:

- The establishing of arc consistency is much more robust. Sometimes, it is time consuming, but it is often the only way to design a good model. During the modelling phase, it is very useful to use strong filtering algorithms, even if, sometimes, some weaker filtering algorithms can be used to improve the time performance of the final model. It is rare to be able to solve some problems in a reasonable amount of time with filtering algorithms establishing properties weaker than arc consistency and not be able to solve these problems with a filtering algorithm establishing arc consistency.
- There is a room for the improvement of filtering algorithms. Most of the CP solvers were designed before the introduction of global constraints in CP. We could imagine that a solver especially designed to efficiently handle global constraints could lead to better performance. On the other hand, the behaviour of filtering algorithms could also be improved in practice, notably by identifying more quickly the cases where no deletion is possible.
- For binary CSPs,for a long time, it was considered that the Forward Checking algorithm (the filtering algorithms are triggered only when some variables are

instantiated) was the most efficient one, but several studies showed that the systematic call of filtering algorithms after every modification is worthwhile (for instance see [28]).

All industrial solver vendors aim to solve real world applications and claim that the use of strong filtering algorithms is often essential.

Thus, we think that the studies about filtering algorithms establishing properties weaker than arc consistency should take into account the previous points and mainly the second point. On the other hand, we think that it is really worthwhile to work on techniques stronger than arc consistency, such as singleton arc consistency which consists of studying the consequences of the assignments of every value to every variable.

6 Conclusion

Filtering algorithms are one of the main strengths of CP. In this chapter, we have presented several useful global constraints with references to the filtering algorithms associated with them. We have also detailed these filtering algorithms for some constraints. In addition, we have tried to identify several ways to design new filtering algorithms based on the existing work. At last, we have identified some problems that deserve to be addressed in the future.

References

1. Ågren M, Beldiceanu N, Carlsson M, Sbihi M, Truchet C, Zampelli S (2009) Six ways of integrating symmetries within non-overlapping constraints. In: CPAIOR'09, pp 11–25
2. Ahuja RK, Magnanti TL, Orlin JB (1993) Network flows. Prentice Hall, NJ
3. Amilhastre J (1999) Reprsentation par un automate d'ensemble de solutions de problme de satisfaction de contraintes. PhD thesis, University of Montpellier II
4. Aron I, Van Hentenryck P (2002) A constraint satisfaction approach to the robust spanning tree problem with interval data. In: Proceedings of UAI, pp 18–25
5. Artiouchine K, Baptiste P (2005) Inter-distance constraint: an extension of the all-different constraint for scheduling equal length jobs. In: CP, pp 62–76
6. Artiouchine K, Baptiste P (2007) Arc-b-consistency of the inter-distance constraint. Constraints 12(1):3–19
7. Beldiceanu N (2000) Global constraints as graph properties on a structured network of elementary constraints of the same type. In: Proceedings CP, pp 52–66
8. Beldiceanu N (2001) Pruning for the minimum constraint family and for the number of distinct values constraint family. In: Proceedings CP'01. Pathos, Cyprus, pp 211–224
9. Beldiceanu N (2005) Global constraint catalog. In: SICS technical report, pp T–2005–08
10. Beldiceanu N, Carlsson M (2001) Revisiting the cardinality operator and introducing the cardinality-path constraint family. In: Proceedings ICLP, vol 2237, pp 59–73
11. Beldiceanu N, Carlsson M (2001) Sweep as a generic pruning technique applied to the non-overlapping rectangles constraints. Proceedings CP'01, pp 377–391
12. Beldiceanu N, Carlsson M, Debruyne R, Petit T (2005) Reformulation of global constraints based on constraint checkers. Constraints 10(4):339–362

13. Beldiceanu N, Carlsson M, Demassey S, Petit T (2006) Filtrage bas sur des proprits de graphes. In: Proceedings of JFPC'06
14. Beldiceanu N, Carlsson M, Demassey S, Petit T (2006) Graph-based filtering. In: Proceedings of CP'06, pp 59–74
15. Beldiceanu N, Carlsson M, Petit T (2004) Deriving filtering algorithms from constraint checkers. In: CP'04, pp 107–122
16. Beldiceanu N, Carlsson M, Rampon J-X, Truchet C (2005) Graph invariants as necessary conditions for global constraints. In: Proceedings of CP'05, pp 92–106
17. Beldiceanu N, Contejean E (1994) Introducing global constraints in chip. Math Comput Model 20(12):97–123
18. Beldiceanu N, Flener P, Lorca X (2005) The tree constraint. In: Proceedings of CPAIOR05, pp 64–78
19. Beldiceanu N, Flener P, Lorca X (2008) Combining tree partitioning, precedence, and incomparability constraints. Constraints 13(4):459–489
20. Beldiceanu N, Guo Q, Thiel S (2001) Non-overlapping constraints between convex polytopes. In: Proceedings CP'01, Pathos, Cyprus, 2001, pp 392–407
21. Beldiceanu N, Katriel I, Lorca X (2006) Undirected forest constraints. In: CPAIOR'06, pp 29–43
22. Beldiceanu N, Lorca X (2007) Necessary condition for path partitioning constraints. In: CPAIOR'07, pp 141–154
23. Beldiceanu N, Petit T, Rochart G (2005) Bornes de caractristiques de graphes. In: Proceedings of JFPC'05
24. Beldiceanu N, Petit T, Rochart G (2005) Bounds of graph characteristics. In: Proceedings of CP'05, pp 742–746
25. Beldiceanu N, Carlsson M, Poder E, Sadek R, Truchet C (2007) A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In: CP'07, pp 180–194
26. Berge C (1970) Graphe et Hypergraphes. Dunod, Paris
27. Bessiere C, Hebrard E, Hnich B, Kiziltan Z, Quimper C-G, Walsh T (2007) Reformulating global constraints: the slide and regular constraints. In: Proceedings of SARA'07, pp 80–92
28. Bessière C, Régin J-C (1996) Mac and combined heuristics: two reasons to forsake fc (and cbj?) on hard problems. In: CP96, second international conference on principles and practice of constraint programming, Cambridge, USA, pp 61–75
29. Bessière C, Régin J-C (1997) Arc consistency for general constraint networks: preliminary results. In: Proceedings of IJCAI'97, Nagoya, pp 398–404
30. Bessière C, Régin J-C (1999) Enforcing arc consistency on global constraints by solving subproblems on the fly. In: Proceedings of CP'99, Alexandria, VA, USA, pp 103–117
31. Bessière C, Régin J-C (2001) Refining the basic constraint propagation algorithm. In: Proceedings of IJCAI'01, Seattle, WA, USA, pp 309–315
32. Bleuzen-Guernalec N, Colmerauer A (1997) Narrowing a 2n-block of sortings in o(nlog(n)). In: Proceedings of CP'97, Linz, Austria, pp 2–16
33. Brand S, Narodytska N, Quimper C-G, Stuckey P, Walsh T (2007) Encodings of the sequence constraint. In: Proceedings of CP 2007, pp 210–224
34. Carlsson M, Beldiceanu N (2002) Arc-consistency for a chain of lexicographic ordering constraints. Technical Report T2002:18, SICS
35. Carlsson M, Beldiceanu N (2002) Revisiting the lexicographic ordering constraint. Technical Report T2002:17, SICS
36. Carlsson M, Beldiceanu N (2004) From constraints to finite automata to filtering algorithms. In: European Symposium on Programming (ESOP'04), pp 94–108
37. Caseau Y, Guillo P-Y, Levenez E (1993) A deductive and object-oriented approach to a complex scheduling problem. In: Proceedings of DOOD'93
38. Caseau Y, Laburthe F (1997) Solving various weighted matching problems with constraints. In: Proceedings CP97, Austria, pp 17–31
39. Cormen TH, Leiserson CE, Rivest RL (1990) Introduction to algorithms. MIT Press, Cambridge

40. Damaschke P, Müller H, Kratsch D (1990) Domination in convex and chrodal bipartite graphs. Inform Process Lett 36:231–236
41. Dantzig G (1957) Discrete variable extremum problems. Oper Res 5:226–277
42. Dechter R, Meiri I, J Pearl (1991) Temporal constraint network. Artif Intell 49(1–3):61–95
43. Dixon B, Rauch M, Tarjan R (1992) Verification and sensitivity analysis of minimum spanning trees in linear time. SIAM J Comput 21(6):1184–1192
44. Dooms G, Deville Y, Dupont P (2005) Cp(graph): introducing a graph computation domain in constraint programming. In: Proceedings of CP'05
45. Dooms G, Katriel I (2006) The minimum spanning tree constraint. In: CP'06, pp 152–166
46. Dooms G, Katriel I (2007) The not-too-heavy spanning tree constraint. In: Proceedings of CPAIOR07, pp 59–70
47. Earley J (1970) An efficient context-free parsing algorithm. Commu ACM 2(13):94–102
48. Fahle T, Sellmann M (2002) Cost based filtering for the constrained knapsack problem. Ann Oper Res 115(1–4):73–93
49. Focacci F, Lodi A, Milano M (1999) Cost-based domain filtering. In: Proceedings CP'99, Alexandria, VA, USA, pp 189–203
50. Focacci F, Lodi A, Milano M (1999) Integration of cp and or methods for matching problems. In: Proceedings CP-AI-OR 99, Ferrara, Italy
51. Freuder E, Wallace R (1992) Partial constraint satisfaction. Artif Intell 58:21–70
52. Frisch A, Hnich B, Kiziltan Z, Miguel I, Walsh T (2002) Global constraints for lexicographic orderings. In: CP'02, pp 93–108
53. Gellermann T, Sellmann M, Wright R (2005) Shorter path constraints for the resource constrained shortest path problem. In: CPAIOR'05, pp 201–216
54. Gent I, Jefferson C, Miguel I, Nightingale P (2007) Data structures for generalised arc consistency for extensional constraints. In: Proceedings of AAAI'07, Vancouver, Canada, pp 191–197
55. Gervet C (1994) Conjunto: constraint logic programming with finite set domains. In: Proceedings ILPS-94
56. Gervet C (2006) Constraints over structured domains. In: Handbook of constraint programming. Elsevier, Amsterdam
57. Gervet C (2006) Programmation par Contraintes sur Domaines Ensemblistes. Habilitation à diriger des Recherches, Université de Nice-Sophia Antipolis
58. Gervet C, Van Hentenryck P (2006) Length-lex ordering for set csps. In: AAAI
59. Gomes C, Regin J-C (2003) The alldiff matrix. Technical report, Intelligent Information Institute – Cornell University
60. Hanak D (2003) Implementing global constraints as structured graphs of elementary constraints. Sci J Acta Cybern 16:241–258
61. Hellsten L, Pesant G, van Beek P (2004) A domain consistency algorithm for the stretch constraint. In: Proceedings of CP'04, pp 290–304
62. Henz M, Müller T, Thiel S (2003) Global constraints for round robin tournament scheduling. Eur J Oper Res 153(1):92–101
63. van Hoeve W-J, Katriel I (2006) Global constraints. In: Handbook of constraint programming. Elsevier, Amsterdam
64. van Hoeve W-J, Pesant G, Rousseau L-M, Sabharwal A (2006) Revisiting the sequence constraint. In: Proceedings of CP 2006, Nantes, France, pp 620–634
65. van Hoeve W-J, Pesant G, Rousseau L-M, Sabharwal A (2009) New filtering algorithms for combinations of among constraints. Constraints 14:273–292
66. ILOG (1999) ILOG Solver 4.4 User's manual. ILOG S.A
67. Janssen P, Vilarem M-C (1988) Problmes de satisfaction de contraintes: Techniques de rsolution et application la synthse de peptides. Technical Report 54, CRIM
68. Jégou P (1991) Contribution à l'Etude des Problèmes de Satisfaction de Contraintes: Algorithmes de Propagation et de Résolution, Propagation de Contraintes dans les Réseaux dynamiques. PhD thesis, Université de Montpellier II
69. Kadioglu S, Sellmann M (2008) Efficient context-free grammar constraints. In: AAAI-08, pp 310–316

70. Kadioglu S, Sellmann M (2009) Grammar constraints. Constraints 15(1):117–144
71. Katriel I (2004) Dynamic heaviest paths in dags with arbitrary edge weights. In: CPAIOR'04, pp 190–199
72. Katriel I, Michel L, Van Hentenryck P (2005) Maintaining longest paths incrementally. Constraints 10(2):159–183
73. Katriel I, Sellmann M, Upfal E, Van Hentenryck P (2007) Propagating knapsack constraints in sublinear time. In: AAAI-07, pp 231–236
74. Katriel I, Thiel S (2003) Fast bound consistency for the global cardinality constraint. In: Proceedings CP'03, Kinsale, Ireland, pp 437–451
75. Katsirelos G, Narodytska N, Walsh T (2009) Reformulating global grammar constraints. In: CPAIOR'09, pp 132–147
76. Katsirelos G, Walsh T (2007) A compression algorithm for large arity extensional constraints. In: Proceedings of CP'07, Providence, USA, pp 379–393
77. Kocjan W, Kreuger P (2004) Filtering methods for symmetric cardinality constraints. In: First international conference, CPAIOR 2004, Nice, France, pp 200–208
78. Kowalski R (1979) Algorithm = logic + control. Comm ACM 22(7):424–436
79. Labbé M, Laporte G, Martello S (2003) Upper bounds and algorithms for the maximum cardinality bin packing problem. Eur J Oper Res 149(3):490–498
80. Larrosa J, Meseguer P, Schiex T, Verfaillie G (1998) Reversible DAC and other improvements for solving Max-CSP. In: Proceedings AAAI, pp 347–352
81. Larrosa J, Meseguer P (1996) Exploiting the use of DAC in Max-CSP. In: Proceedings of CP'96
82. Lawler E (1976) Combinatorial optimization: networks and matroids. Holt, Rinehart and Winston
83. Le Pape C, Perron L, Régin J-C, Shaw P (2002) Robust and parallel solving of a network design problem. In: CP'02, Ithaca, NY, USA, pp 633–648
84. Leconte M (1996) A bounds-based reduction scheme for constraints of difference. In: Constraint-96, second international workshop on constraint-based reasoning, Key West, FL, USA
85. Lecoutre C, Szymanek R (2006) Generalized arc consistency for positive table constraints. In: Proceedings of CP'06, Providence, USA, pp 284–298
86. Lhomme O (2004) Arc-consistency filtering algorithms for logical combinations of constraints. In: Proceedings of CP-AI-OR'04, Nice, France
87. Lhomme O, Régin J-C (2005) A fast arc consistency algorithm for n-ary constraints. In: Proceedings of AAAI'05, Pittsburgh, USA, pp 405–410
88. Lopez-Ortiz A, Quimper C-G, Tromp J, van Beek P (2003) A fast and simple algorithm for bounds consistency of the alldifferent constraint. In: IJCAI'03, Acapulco, Mexico, pp 245–250
89. Maher M (2009) Open constraints in a boundable world. In: CPAIOR, pp 163–177
90. Maher M, Narodytska N, Quimper C-G, Walsh T (2008) Flow-based propagators for the sequence and related global constraints. In: Proceedings CP 2008, pp 159–174
91. Manku G (1994) An o(m + n log* n) algorithm for sensitivity analysis of minimum spanning trees. citeseer.ist.psu.edu/manku94om.html
92. Martello S, Toth P (1990) Knapsack problems. Wiley, New York
93. Melhorn K, Thiel S (2000) Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In: Proceedings of CP'00, Singapore, pp 306–319
94. Micali S, Vazirani VV (1980) An $O(\sqrt{|V|}|E|)$ algorithm for finding maximum matching in general graphs. In: Proceedings 21st FOCS, pp 17–27
95. Michel L, Van Hentenryck P (2003) Maintaining longest paths incrementally. In: CP'03, pp 540–554
96. Pachet F, Roy P (1999) Automatic generation of music programs. In: Proceedings of CP'99, Alexandria, VA, USA, pp 331–345
97. Pesant G (2001) A filtering algorithm for the stretch constraint. In: Proceedings CP'01, Pathos, Cyprus, pp 183–195

98. Pesant G (2003) A regular language membership constraint for sequence of variables. In: Workshop on modelling and reformulation constraint satisfaction problems, pp 110–119
99. Pesant G (2004) A regular language membership constraint for finite sequences of variables. In: Proceedins of CP'04, pp 482–495
100. Pesant G, Régin J-C (2005) Spread: a balancing constraint based on statistics. In: CP'05, pp 460–474
101. Petit T, Régin J-C, Bessière C (2001) Specific filtering algorithms for over-constrained problems. In: Proceedings CP'01, Pathos, Cyprus, pp 451–465
102. Petit T, Régin J-C, Bessière C (2002) Range-based algorithms for max-csp. In: Proceedings CP'02, Ithaca, NY, USA, pp 280–294
103. Puget J-F (1994) A c++ implementation of clp. Technical report, ILOG S.A
104. Quimper C-G, López-Ortiz A, Pesant G (2006) A quadratic propagator for the inter-distance constraint. In: AAAI-06
105. Quimper C-G, van Beek P, López-Ortiz A, Golynski A, Sadjad SB (2003) An efficient bounds consistency algorithm for the global cardinality constraint. In: Proceedings CP'03, Kinsale, Ireland, pp 600–614
106. Quimper C-G, Walsh T (2006) Global grammar constraints. In: CP'06, pp 751–755
107. Quimper C-G, Walsh T (2006) Global grammar constraints. Technical report, Waterloo University
108. Quimper C-G, Walsh T (2007) Decomposing global grammar constraints. In: CP'07, pp 590–604
109. Quimper C-G, Walsh T (2008) Decomposing global grammar constraints. In: NECTAR, AAAI-08, pp 1567–1570
110. Régin J-C (1994) A filtering algorithm for constraints of difference in CSPs. In: Proceedings AAAI-94, Seattle, Washington, pp 362–367
111. Régin J-C (1995) Développement d'outils algorithmiques pour l'Intelligence Artificielle. Application à la chimie organique. PhD thesis, Université de Montpellier II
112. Régin J-C (1996) Generalized arc consistency for global cardinality constraint. In: Proceedings AAAI-96, Portland, Oregon, pp 209–215
113. Régin J-C (1997) The global minimum distance constraint. Technical report, ILOG
114. Régin J-C (1999) Arc consistency for global cardinality with costs. In: Proceedings of CP'99, Alexandria, VA, USA, pp 390–404
115. Régin J-C (1999) The symmetric alldiff constraint. In: Proceedings of IJCAI'99, Stockholm, Sweden, pp 425–429
116. Régin J-C (2002) Cost based arc consistency for global cardinality constraints. Constraints 7(3-4):387–405
117. Régin J-C (2003) Global constraints and filtering algorithms. In: Milano M (ed) Constraints and integer programming combined Kluwer, Dordrecht
118. Régin J-C (2003) Using constraint programming to solve the maximum clique problem. In: CP'03, Kinsale, Ireland, pp 634–648
119. Régin J-C (2004) Modeling problems in constraint programming. In: Tutorial CP'04 Available at www.constraint-programming.com/people/regin/papers/modelincp.pdf
120. Régin J-C (2004) Modélisation et Contraintes globales en programmation par contraintes. Habilitation à diriger des Recherches, Université de Nice-Sophia Antipolis
121. Régin J-C (2005) Combination of among and cardinality constraints. In: Proceedings of CP-AI-OR'05
122. Régin J-C (2008) Simpler and incremental consistency checking and arc consistency filtering algorithms for the weighted spanning tree constraint. In: CPAIOR'08, pp 233–247
123. Régin J-C, Gomes C (2004) The cardinality matrix constraint. In: CP'04, Toronto, Canada, pp 572–587
124. Régin J-C, Petit T, Bessière C, Puget J-F (2000) An original constraint based approach for solving over constrained problems. In: Proceedings of CP'00, Singapore, pp 543–548
125. Régin J-C, Petit T, Bessière C, Puget J-F (2001) New lower bounds of constraint violations for over-constrained problems. In: Proceedings CP'01, Pathos, Cyprus, pp 332–345

126. Régin J-C, Puget J-F (1997) A filtering algorithm for global sequencing constraints. In: CP97: Third international conference on principles and practice of constraint programming, pp 32–46
127. Régin J-C, Rueher M (2000) A global constraint combining a sum constraint and difference constraints. In: Proceedings of CP'00, Singapore, pp 384–395
128. Sadler A, Gervet C (2004) Hybrid set domains to strengthen constraint propagation and reduce symmetries. In: CP, pp 604–618
129. Sadler A, Gervet C (2008) Enhancing set constraint solvers with lexicographic bounds. J Heuristics 14(1):23–67
130. Schaus P (2009) Solving balancing and bin-packing problems with constraint programming. PhD thesis, Université catholique de Louvain Louvain-la-Neuve
131. Schaus P, Deville Y, Dupont P, Régin J-C (2007) The deviation constraint. In: CPAIOR'07, pp 260–274
132. Schaus P, Deville Y, Dupont P, Régin J-C (2006) Simplification and extension of the spread constraint. In: CP'06, Workshop on constraint propagation and implementation, pp 72–92
133. Schaus P, Deville Y, Dupont P, Régin J-C (2007) Simplification and extension of the SPREAD constraint. In: Future and trends of constraint programming, ISTE, Washington DC, pp 95–99
134. Schaus P, Deville Y (2008) A global constraint for bin-packing with precedences: application to the assembly line balancing problem. In: AAAI-08, pp 369–374
135. Sellmann M (2003) Approximated consistency for knapsack constraints. In: CP'03, pp 679–693
136. Sellmann M (2003) Cost-based filtering for shorter path constraints. In: CP'03, pp 694–708
137. Sellmann M (2004) The practice of approximated consistency for knapsack constraints. In: AAAI-04, pp 179–184
138. Sellmann M (2006) The theory of grammar constraints. In: CP'06, pp 530–544
139. Sellmann M, Gellermann T, Wright R (2007) Cost-based filtering for shorter path constraints. Constraints 12(2):207–238
140. Shaw P (2004) A constraint for bin packing. In: CP'04, pp 648–662
141. Simonis H (1996) Problem classification scheme for finite domain constraint solving. In: CP96, Workshop on constraint programming applications: an inventory and taxonomy, Cambridge, USA, pp 1–26
142. Sorlin S, Solnon C (2004) A global constraint for graph isomorphism problems. In: CPAIOR'04, pp 287–302
143. Sorlin S, Solnon C (2008) A parametric filtering algorithm for the graph isomorphism problem. Constraints 13(4):518–537
144. Stergiou K, Walsh T (1999) The difference all-difference makes. In: Proceedings of IJCAI'99, Stockholm, Sweden, pp 414–419
145. Tarjan R (1982) Sensitivity analysis of minimum spanning trees and shortest path trees. Inform Process Lett 14(1):30–33
146. Tarjan RE (1983) Data structures and network algorithms. In: CBMS-NSF regional conference series in applied mathematics. SIAM, Philadelphia
147. Trick M (2001) A dynamic programming approach for consistency and propagation for knapsack constraints. In: Proceedings of CPAIOR'01
148. Trick M (2003) A dynamic programming approach for consistency and propagation for knapsack constraints. Ann Oper Res 118:73–84
149. Van Hentenryck P, Deville Y (1991) The cardinality operator: a new logical connective for constraint logic programming. In: Proceedings of ICLP-91, Paris, France, pp 745–759
150. Van Hentenryck P, Deville Y, Teng CM (1992) A generic arc-consistency algorithm and its specializations. Artif Intell 57:291–321
151. Van Hentenryck P, Michel L (2003) Control abstractions for local search. In: CP'03, pp 66–80
152. Van Hentenryck P, Saraswat V, Deville Y (1998) Design, implementation, and evaluation of the constraint language cc(fd). J Logic Program 37(1–3):139–164
153. Van Hentenryck P, Yip J, Gervet C, Dooms G (2008) Bound consistency for binary length-lex set constraints. In: AAAI, pp 375–380

154. Vempaty N (1992) Solving constraint satisfaction problems using finite state automata. In: AAAI-92, pp 453–458
155. Wallace R (1994) Directed arc consistency preprocessing as a strategy for maximal constraint satisfaction. In: Proceedings ECAI, pp 69–77
156. Zampelli S, Deville Y, Solnon C, Sorlin S, Dupont P (2007) Filtering for subgraph isomorphism. In: CP, pp 728–742
157. Zhou J (1996) A constraint program for solving the job-shop problem. In: Proceedings of CP'96, Cambridge, pp 510–524
158. Zhou J (1997) Computing smallest cartesian products of intervals: application to the jobshop scheduling problem. PhD thesis, Université de la Méditerranée, Marseille

Decomposition Techniques for Hybrid MILP/CP Models applied to Scheduling and Routing Problems

Pedro M. Castro, Ignacio E. Grossmann, and Louis-Martin Rousseau

Abstract This chapter provides a review of decomposition algorithms for models that are formulated as hybrid mixed-integer linear/constraint programming problems, such as logic Benders Decomposition and Constraint Programming-Based Column Generation. We first focus the decomposition techniques on single stage scheduling problems with parallel machines where the hybrid model provides a natural representation as the decisions decompose into assignment and sequencing decisions. We describe a general decomposition algorithm for the hybrid MILP/CP model in terms of a Benders decomposition scheme, as well as in terms of a branch and cut framework. We then consider Vehicle Routing and Crew Rostering applications to illustrate how a Hybrid Branch-and-Price method can be applied, and we discuss the different models that have been proposed in the literature.

1 Introduction

Many discrete/continuous optimization problems can be formulated either as mixed-integer linear programs (MILP) or as constrained programming (CP) problems. A number of authors have compared alternative MILP and CP based formulations for solving a variety of problems [38, 42]. Some of their main findings include the following:

- MILP based techniques are efficient when the LP relaxation is tight and the models have a structure that can be effectively exploited, as well as having objective functions involving several or many variables.
- CP based techniques are better suited for handling highly constrained discrete optimization problems that involve few variables in the objective function or the search of a feasible solution.

P.M. Castro (✉)
Unidade de Modelação e Optimização de Sistemas Energéticos, Laboratório Nacional de Energia e Geologia, 1649-038 Lisboa, Portugal
e-mail: pedro.castro@ineti.pt

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_4,
© Springer Science+Business Media, LLC 2011

Since the two approaches appear to have complementary strengths, in order to solve difficult problems that are not effectively solved by either of the two, several researchers have proposed models that integrate the two paradigms. The integration between MILP and CP can be achieved in two ways [42, 43, 77]:

(a) By combining MILP and CP constraints into one hybrid model. In this case, a hybrid algorithm that integrates constraint propagation with linear programming in a single search tree is also needed for the solution of the model [39, 68].
(b) By decomposing the original problem into two subproblems: one MILP and one CP subproblem. Each model is solved separately and information obtained while solving one subproblem is used for the solution of the other subproblem [9, 47, 72].

It is the major objective of this paper to provide a review of the decomposition approach based on MILP and CP subproblems and apply it to single stage scheduling problems with parallel machines and vehicle routing problems. This chapter relies heavily on the articles by Jain and Grossmann [47], Rousseau et al. [70], and Gualandi and Malucelli [31].

2 Literature Review

A number of papers have compared the performance of CP- and MILP-based approaches for solving a number of different problems, for example, the modified generalized assignment problem [20], the template design problem [62], the progressive party problem [74], and the change problem [38]. Properties of a number of different problems were considered by Darby-Dowman and Little [19], and their effect on the performance of CP and MILP approaches were presented. As discussed earlier, these papers showed that MILP is very efficient when the relaxation is tight, and the models have a structure that can be effectively exploited. CP works better for highly constrained discrete optimization problems where expressiveness of MILP is a major limitation.

Most of the attempts [39, 68] to integrate CP and MILP use constraint propagation along with linear programming in a single search tree to obtain bounds on the objective and to reduce the domains of the variables. In these approaches, a complete CP model and at the least a corresponding partial MILP model are required. This is because CP is a richer modeling tool and not all CP constraints may be easily reformulated as MILP constraints. These approaches in some sense perform redundant computations because a constraint-propagation problem and a simplex problem are solved at every node. For some problems, this may be justified because they are intractable for either of the two methods. Rodosek et al. [68] presented a systematic approach for transforming a CP model into a corresponding MILP model. However, automatic translation from a CP model to an MILP model may result in a poor

model involving numerous big-M constraints (poor LP relaxations). In this case, the advantage of performing "Global Reasoning" using LP relaxation is essentially lost. If automatic translation is not used, then the user has to model the problems for both approaches.

Hooker et al. [46] have argued that a new modeling paradigm may be required to perform efficient integration of MILP- and CP-based approaches. The modeling framework is motivated by the Mixed Logic/Linear modeling framework that was proposed by Hooker and Osorio [45]. Ottosson et al. [57] presented algorithms for solving such models. For a production-planning problem, they showed that the computational performance of the proposed method *vis-a-vis* pure MILP and CP approaches was significantly better. Bockmayr and Kasper [8] did an interesting analysis of CP and MILP approaches, and presented a unifying framework, Branch and Infer, that can be used to develop various integration strategies. They divide constraints for both MILP and CP into two different categories, primitive and nonprimitive. Primitive constraints are those for which there exists a polynomial-time solution algorithm, and nonprimitive constraints are those for which this is not true. The interesting aspect about this classification is that some of the primitive constraints in CP are nonprimitive in MILP and vice versa. They also discussed how nonprimitive constraints can be used to infer primitive constraints and the use of symbolic constraints for MILPs. Raman and Grossmann [64, 65] earlier modeled discrete/continuous optimization problems with disjunctions and symbolic constraints in the form of logic propositions. This model, which they denoted as a Generalized Disjunctive Program (GDP), can be converted all or in part into an MILP. They presented the idea of w-MIP representability, which is similar to the idea of primitive constraints. They showed that it is computationally efficient to transform w-MIP representable disjunctions into linear constraints and proposed a hybrid branch-and-bound algorithm that handles the non w-MIP representable disjunctions directly.

In general, it is not clear whether a general integration strategy will always perform better than either a CP or an MILP approach by itself. This is especially true for the cases where one of these methods is a very good tool to solve the problem at hand. However, it is usually possible to enhance the performance of one approach by borrowing some ideas from the other. For example, Raman and Grossmann [64] used logic cuts that were written as logic propositions to improve the performance of MILP models. Ideas on edge-finding [3, 11] that were used for guiding the search in MILPs to solve jobshop problems were exploited by Caseau and Laburthe [12,13] and Le Pape [49] to develop efficient inference engines for scheduling algorithms in CP. Furthermore, there are a number of similarities in some of the underlying ideas of both approaches. For example, probing and integer preprocessing in MILP is in some ways similar to constraint propagation. Chandru and Hooker [16] give an interesting operations-research perspective on consistency methods and logical inference. Also, Hooker [41] deals with the subject of MILP and CP integration in detail.

2.1 Hybrid MILP/CP Algorithms for Scheduling

Hybrid MILP/CP algorithms have been shown to be potentially better approaches than standalone MILP and CP models, particularly in scheduling problems. Scheduling can be viewed as involving two decisions: (a) job-machine assignment; (b) job sequencing on every machine, so as to meet some operational goal. Jain and Grossmann [47] have proposed a relaxed MILP master problem to find the assignments and CP feasibility subproblems to check if the assigned jobs can be sequenced on every unit. In the case of infeasible assignments, integer cuts are added to avoid infeasible solutions in subsequent iterations. The scheduling problem involved finding a minimum cost assignment of jobs with release and due dates to unrelated parallel machines involved in a single production stage, and orders of magnitude reduction in computational time were achieved by the decomposition algorithm.

The MILP and CP models are linked through the infeasibility cuts, so these are the most critical elements of the decomposition algorithm. On the one hand, they should be rigorous so that only infeasible assignments, and not feasible or even optimal solutions, are eliminated from the solution space. On the other hand, they should be of high quality so that fewer iterations are involved. Conceptually, hybrid MILP/CP algorithms can be applied to a wide variety of problems but are only practical if rigorous and strong cuts can be found.

Harjunkoski and Grossmann [37] extended the algorithm to multistage processes. They found that contrary to the single stage case, it was not possible to solve sequencing problems separately, one for each machine, since the stages and machines are connected. Two main complications arose. First, the sequencing problem needed to be solved simultaneously for every job and machine, resulting in large CP problems. Second, the infeasible CPs did not provide useful information about the reason for the infeasibility. While the same is true in single stage, the integer cuts that could be derived in a straightforward manner were even weaker since they involved all the machines. To yield better cuts, the CP feasibility subproblem was replaced by a minimization problem that measured the violation of the due dates through slack variables. However, the hybrid MILP/CP algorithm failed to find the optimal solution in some test problems, showing that the proposed cuts were not rigorous.

Following the same line of research, Maravelias and Grossmann [53] used the Shifting Bottleneck Procedure (SBP) [2, 5] for the solution of the multistage (job-shop) problem with fixed job-unit assignments. In its first stage, infeasibilities that are due to the assignments on a single machine are detected, leading to cuts similar to those of Jain and Grossmann [47]. However, since SBP is a heuristic algorithm, a feasible partial solution from the master problem may be found infeasible by the SBP. In such cases, the iterative algorithm will terminate with a suboptimal solution. Nevertheless, the algorithm did find the optimal solution in all ten test instances solved and exhibited a better performance than the hybrid MILP/CP scheme of Harjunkoski and Grossmann [37].

For the single stage problem with parallel machines, Maravelias and Grossmann [53] proposed a preprocessing algorithm that generates knapsack constraints and

cover cuts for certain subsets of jobs that can be added to the cut pool of the MILP master problem a priori. Further details are given later, including a computational performance of the hybrid MILP/CP algorithm with and without these cuts.

The same authors also addressed the more complex multipurpose plant structure, where a particular unit can handle jobs belonging to different production stages [52]. The underlying MILP scheduling model relies on a single continuous-time grid with event points rather than on sequencing variables [47]. Their hybrid MILP/CP algorithm consists on:

- A master MILP to select the type and number of jobs to be performed as well as the job-unit assignments
- A CP feasibility, when minimizing cost, or optimization subproblem to derive a feasible schedule (when maximizing profit or minimizing makespan). At each iteration, specialized integer cuts are added to the master problem to exclude infeasible or previously obtained assignments

All the cuts are rigorous to avoid cutting off feasible solutions. Overall, the computational results showed that for some classes of problems the algorithm was orders of magnitude faster than a standalone MILP model.

A hybrid algorithm MILP/CP for an industrial application consisting on a two-stage process with shared intermediate storage was developed by Timpe [75]. The master MILP model employs a discrete-time representation and is responsible for lot-sizing and the assignment component. Total cost minimization is used as the objective function. The CP model then tries to do the sequencing using makespan minimization as the objective function, avoiding objective functions consisting on large sums, for which CP solvers cannot deduce anything and propagation is poor. If unsuccessful, any infeasibility can be located very accurately so that the resulting cuts can be very efficient. Nevertheless, they are not valid cuts in the sense that they cutoff infeasible solutions only. While the quality of the resulting solutions was hard to estimate, with the lower bound provided by the relaxation of the MILP being rather weak, the algorithm could successfully generate good feasible solutions quickly, thus allowing the system to work in an environment very close to production.

The decomposition procedure of Jain and Grossmann [47] has been interpreted as a logic-based Benders decomposition [40]. Hooker [44] has shown that a Benders approach can succeed in a variety of planning and scheduling problems in which the Benders cuts are less obvious. Focus was set on single stage cumulative scheduling problems rather than disjunctive scheduling problems, where tasks must be run one at a time, and on two alternative objective functions to cost, makespan and tardiness minimization. By assuming the same release date for all jobs, relatively simple Benders cuts were derived, which can be further simplified when all the deadlines are the same. Unfortunately, no cuts were proposed for the general case. Overall, the Benders method was found to be several orders of magnitude faster than either CP or MILP. An important advantage is that it can be terminated early while still yielding a feasible solution and a lower bound on the optimal value. (This does not apply to the minimum cost problems, for which all intermediate solutions are infeasible).

The original problem of Jain and Grossmann [47] was more thoroughly tackled by Castro and Grossmann [14] and Sadykov and Wolsey [72].

Castro and Grossmann [14] have tested five alternative formulations to the hybrid MILP/CP model. Besides the standalone CP and MILP models that rely on the concept of general precedence, discrete and continuous-time MILP models with a single or multiple time grids were also involved. In particular, the discrete-time formulation was the best performer. For the most complex problem, the first master problem generated by the hybrid MILP/CP algorithm was more or less as difficult to solve as the full-space MILP from which it originated, with the disadvantage of not providing useful information. Thus, other alternatives should be considered in addition to improving the algorithm.

Sadykov and Wolsey [72] have proposed seven different algorithms ranging from pure MILP formulations to branch-and-cut and branch-and-price MILP/CP approaches. The results showed that it is important to develop as tight an MILP formulation as possible. When using an MILP/CP algorithm, it is important to tighten the IP formulation and also control its size through the addition of few strong cuts instead of many weak cuts, to reduce the number of feasibility tests and avoid increasing the solution time. Overall, instances with up to 54 jobs and nine machines were solved with an MILP/CP algorithm with either a tightened IP or a column generation algorithm with a combined MILP/CP algorithm for the subproblem.

For the simpler single machine problem, Sadykov [71] proposed a branch-and-check algorithm for the problem of minimizing the weighted number of late jobs subject to release dates. Five variants of the algorithm were studied, each using a different strategy to generate infeasibility cuts. Besides the weak (no-good) cuts of Jain and Grossmann [47], two other ways of generating cuts were proposed: (a) using the modified Carlier algorithm; (b) deriving generalized tightening inequalities (CP based cuts); with the former being more efficient in cases of sufficient speed of the modified Carlier algorithm.

Hybrid CP/MILP methods have also been reported in the literature. Correa et al. [18] have addressed the problem of dispatching and conflict free routing of automated guided vehicles (AGVs) in a flexible manufacturing system. The CP master problem determines both the assignment of the transportation requests to the vehicles and the expected times of the pick-up and the deliveries based on the shortest path routes, neglecting the possible conflicts. The MILP subproblem then tries to find the collision free routes satisfying the schedule. When no solution is found, logic cuts are generated and sent back to the master problem. Three reasons were given to switch the order of the models in the decomposition algorithm. The first is the existence of many nonlinear constraints in the scheduling part, which are better handled by CP. The second is that CP allowed designing a specific search strategy, where selection heuristics in combination with the ILOG's OPL Studio slice-based search, were implemented. The third is that for fixed assignments, the MILP subproblem has a very strong minimum cost flow problem structure and thus can be solved efficiently by the network simplex together with branch-and-bound.

A more complex hybrid MILP/CP algorithm with four model layers has been developed by Rasmussen and Trick [66]. The scope is sports scheduling, and the problem consists on designing a double round robin schedule with a minimum number of breaks. In the upper-most level, a CP model generates feasible home-away patterns. An IP model is then used to find a subset of patterns with a minimum number of consecutive home or away games. The next step is to check for feasibility, and necessary and sufficient conditions for a team allocation to exist, as well as necessary conditions for a game assignment to exist, are given. An MILP optimization model is involved in this process and Benders cuts are given for each of the conditions. If all the necessary conditions are met, a team allocation (to patterns) has already been found in the process. However, there can still be no feasible game assignment, so a CP model is used to check if a given subset of patterns can play the required number of mutual games. Overall, few iterations are involved and savings of several orders of magnitude in computational time were observed for hard instances when comparing to previous approaches.

Artigues et al. [1] propose exact hybrid methods based on integer linear programming and constraint programming for an integrated employee timetabling and job-shop scheduling problem. Each method investigates the use of a constraint programming (CP) formulation associated with a linear programming (LP) relaxation. Under a CP framework, the LP-relaxation is integrated into a global constraint using in addition reduced cost-based filtering techniques. The paper proposes two CP formulations of the problem yielding two different LP relaxations. The first formulation is based on a direct representation of the problem. The second formulation is based on a decomposition in intervals of possible operation starting times. The theoretical interest of the decomposition-based representation compared to the direct representation is shown through the analysis of dominant schedules. Computational experiments on a set of randomly generated instances confirm the superiority of the decomposition-based representation. In both cases, the hybrid methods outperforms pure constraint programming for employee cost minimization while it is not the case for makespan minimization. The experiments also investigate the interest of the proposed integrated method compared to a sequential approach and show its potential for multiobjective optimization.

Scheduling problems in the forest industry, which have received significant attention in the recent years, have contributed many challenging applications for optimization technologies. In this context, El Hachemi et al. [25] proposed a solution method based on constraint programming and mathematical programming for a log-truck scheduling problem. The problem consists of scheduling the transportation of logs between forest areas and woodmills, as well as routing the fleet of vehicles to satisfy these transportation requests. The objective is to minimize the total cost of the nonproductive activities such as the waiting time of trucks and forest log-loaders and the empty driven distance of vehicles. The authors propose a CP model to address the combined scheduling and routing problem and an integer programming model to deal with the optimization of deadheads. Both of these models are combined through the exchange of global constraints. The whole approach is validated on real industrial data.

2.2 Hybrid Column Generation Approaches

Constraint Programming-based column generation is a decomposition method that can model and solve very complex optimization problems. The general framework was first introduced in Junker et al. [48]. It has since been applied in areas such as airline crew scheduling [27, 73], vehicle routing [70], cutting-stock [26], and employee timetabling [35].

All these optimization problems may be decomposed in a natural way: They may be viewed as selecting a subset of individual patterns within a huge pool of possible and weighted patterns. The selected combination is the one with the lowest cost to fulfil some given global requirements. The selection problem can be formulated as an integer linear program with one column for each possible pattern and a corresponding integer variable representing the number of times the pattern should be selected. The design of the possible patterns is itself a hard constrained satisfaction problem and its solution set may be too large to be written out explicitly. Delayed column generation is then the only way to address such a formulation (see, for example, [17] for details on the approach). The LP-relaxation of the integer program, the *master problem*, is solved iteratively on a restricted set of columns. At each iteration, the *pricing problem* is to generate new entering columns, i.e., new possible patterns, which may improve the current solution of the master problem. The process ends when no such columns exist. The current solution is an optimal fractional solution of the initial linear program. To solve Integer Linear Programs, it is necessary to embed the column generation procedure into an enumeration search tree, namely a Branch-and-Price algorithm [6, 50].

In such approach, the pattern design subproblem is often solved several times. In each iteration, it is preferable to compute several solutions at once to limit the number of iterations of the column generation process. An optimization variant of the problem is often considered since the expected patterns (i.e., the most improving columns) are the ones with the most negative reduced costs in the master problem.

In routing, crew scheduling or employee timetabling applications, the rules defining the allowed individual patterns are often multiple and complex. Traditionally, they have been handled by dynamic programming techniques [22]. Instead, the use of a constraint programming solver to tackle the pricing problem adds flexibility to the whole solution procedure. For its modeling abilities, CP is more suited as rules are often prone to change.

Hence, CP-based column generation is an easily adaptable solution method: the problem decomposition makes the pattern design subproblem independent from the global optimization process, leaving the CP component alone to handle variations within the definition of the patterns. The recent introduction of both ergonomic and effective optimization constraints in the CP component can have a great impact on the success of this approach to solve various large-size optimization problems.

In the last decade, the CP-CG framework has been applied to several different applications. In a recent survey, Gualandi et al. [31] classifies these applications into four different types:

A: CP used as a simple black-box solver
B: CP-pricing solved by using optimization constraints
C: Method enhancements
D: Computational comparison among different approaches

Constraint Programming can generally also be used to generate the initial feasible solutions fed to the column generation algorithm, or can be embedded into a primal heuristic that constructs an integer solution from the optimal linear relaxation of the master problem. Therefore, the classification considers also when CP is used within the column generation algorithm. Gualandi and Malucelli distinguish four phases as follows:

I: CP used in the initialization phase
II: CP used to solve the pricing subproblem
III: CP used to heuristically construct an integer solution for the master problem from the linear relaxation solution
IV: CP used within a branch-and-price algorithm

Table 1 summarizes the types and phases classifications of the different applications of CP-based column generation.

In Sect. 4, we illustrate the application of CP-based column generation ot vehicle routing and crew scheduling problems. We present and compare the different models which have been used to solve these problems.

Table 1 Applications of CP-based column generation

Application	References	Type	Phase
Urban transit crew management	[79, 80]	A,D	II,IV
Airline planning	[32, 33]	A	I, III
	[34]		
Traveling tournament problem	[24]	A	II, IV
Two-dimensional bin packing	[61]	A	II,IV
Graph coloring	[30]	A, C	II, III, IV
Airline crew assignment	[48]	B	II, III
	[26]		
	[73]		
	[27]		
	[73]	C	II, III
Vehicle routing with time windows	[70]	B	II, IV
	[69]	C	II
Constrained cutting stock	[26]	B	II
Employee timetabling	[21]	B	II, IV
Grouping cabin crew	[36]	D	I
Wireless mesh networks	[10]	B, D	II
Multi-machine assignment scheduling	[72]	B,D	II,IV

3 Applications to Scheduling

The decomposition algorithms described in this section, which were proposed by Jain and Grossmann [47], were motivated by the work of Bockmayr and Kasper [8]. These algorithms are based on the premise that combinatorial problems may sometimes have some characteristics that are better suited for MILP and others that are better handled by CP. For these problems, pure MILP- and pure CP-based approaches may not perform well. As discussed earlier, most of the prior work on integrating the two approaches uses at least one of the models in complete form (usually CP).

3.1 Methodology

In the algorithms that we present, the problem is solved using relaxed MILP and CP feasibility models. Consider a problem, which when modeled as an MILP, has the following structure,

$$\text{(M1): min } c^T x \tag{1}$$

$$\text{s.t. } Ax + By + Cv \leq a \tag{2}$$

$$A'x + B'y + C'v \leq a' \tag{3}$$

$$x \in \{0,1\}^n, y \in \{0,1\}^m, v \in \Re^p \tag{4}$$

This is an optimization problem that has both continuous (v) and binary (x and y) variables, and only some of the binary variables (x) have non-zero objective-function coefficients. The constraint set can be divided into two subsets. In the first set of constraints (2), the polyhedral structure is represented efficiently in the MILP framework (e.g. assignment constraints) and has a significant impact on the LP relaxation. The second set of constraints (3), on the other hand, is assumed not to affect the LP relaxation significantly and is sometimes large in number because of the limited expressive power of MILP methods. The same problem can also be modeled as a CP. Note that more constructs are available in the CP framework to model the problem (e.g., logical constraints, disjunctions, all-different operator, etc.; Marriott and Stuckey [54]). For this reason, the MILP and CP models of the same problem may have different variable definitions and constraint structures. However, an equivalence can be established between the constraints and a complete labeling of variables can be derived in one framework from the values of variables in the other one. Let us assume that the equivalent CP model is

$$\text{(M2): min } f(\overline{x}) \tag{5}$$

$$\text{s.t. } G(\overline{x}, \overline{y}, \overline{v}) \leq 0 \tag{6}$$

$$\overline{x}, \overline{y}, \overline{v} \in \mathcal{D} \tag{7}$$

where $\overline{x}$, $\overline{y}$, and $\overline{v}$, are the CP variables. The domain of these variables $\mathcal{D}$ can be continuous, discrete, or boolean. Generally, these variables do not have a one-to-one correspondence with the MILP variables (x, y, v), although a mapping between the sets of variables x can be established. This is because these variables are needed to calculate the objective function. It may or may not be the case for the variables (y,v) and ($\overline{y}$,$\overline{v}$). Usually, equivalence can also be established between the sets of constraints.

Consider a class of problems with the above mentioned MILP and CP model structures. Furthermore, assume that it is difficult to solve this problem as an MILP because there is a large number of constraints in the constraint set (3) and finding feasible solutions for them is hard. Assume also that the broader expressive power of CP results in the smaller constraint set (6). Even though the constraint set in CP is much smaller, it may still not be efficient to solve the problem using CP because finding an optimal solution and proving optimality can be difficult for CP (lack of linear programming relaxations). Ideally, one would like to combine the strength of MILP to handle the optimization aspect of the problem by using LP relaxation and the power of CP to find feasible solutions by using better constraint formulations. To achieve this goal, Jain and Grossmann [47] proposed a hybrid model for this class of problems that can be solved using either a decomposition algorithm or a branch-and-bound algorithm. The main advantage of the proposed methods is that smaller LP and CP subproblems are solved. The hybrid model involves MILP constraints, CP constraints, and equivalence relations. The objective function in the hybrid model (M3) is the same as in the MILP model (M1). The constraints for this problem include MILP constraints (2), equivalence relations that relate MILP variables x to CP variables $\overline{x}$, and a reduced set of CP constraints that are derived from the CP constraint set (6) by assuming that the set of CP variables $\overline{x}$ is fixed.

$$\text{(M3)}: \min \; c^T x \tag{8}$$

$$\text{s.t. } Ax + By + Cv \leq a \tag{9}$$

$$x \Leftrightarrow \overline{x} \tag{10}$$

$$G(\overline{x}, \overline{y}, \overline{v}) \leq 0 \tag{11}$$

$$x \in \{0,1\}^n, y \in \{0,1\}^m, v \in \Re^p \tag{12}$$

$$\overline{x}, \overline{y}, \overline{v} \in \mathcal{D} \tag{13}$$

It should be noted that the hybrid model (M3) requires at least some variables (x) of the MILP model (M1) and all the variables ($\overline{x}, \overline{y}, \overline{v}$) of the CP model (M2). The values of the CP variables obtained using model (M3) will always satisfy all the constraints of the CP model (M2). Furthermore, the optimal solution for the problem at hand is given by the values of the CP variables ($\overline{x}, \overline{y}, \overline{v}$) obtained by solving the hybrid model (M3) to optimality. It should be noted that the values of the MILP variables (y,v) obtained from the model (M3) may not be valid for the Model (M1) because the model (M3) does not include the MILP constraint set (3).

Jain and Grossmann [47] proposed a decomposition algorithm to solve the hybrid MILP/CP model (M3). The basis for this algorithm is a relaxed MILP problem and a CP feasibility problem, both of which can be solved efficiently. The relaxed MILP model is used to obtain a solution that satisfies the constraint sets (9) and (12) and optimizes the objective function (8). The solution obtained for the relaxed MILP is used to derive a partial CP solution by using the equivalence relation (10). A CP feasibility model then verifies whether this solution can be extended to a full-space solution that satisfies the constraints (11) and (13) of the model. If the partial solution from the MILP can be extended, then the full-space solution obtained will also have the same value of the objective function. In this paper, we present two methods to search the solution space and obtain the optimal solution. Both of these ideas are essentially the same and use the same relaxed MILP and CP feasibility models. However, the difference lies in the order in which the CP subproblems are solved.

3.1.1 MILP/CP-Based Decomposition Method

This algorithm has some similarities to the Benders decomposition method [7]. The algorithm is summarized in Fig. 1. In this method, a *relaxed MILP model* of the problem, with (8) as the objective and (9) and (12) as constraints, is solved to

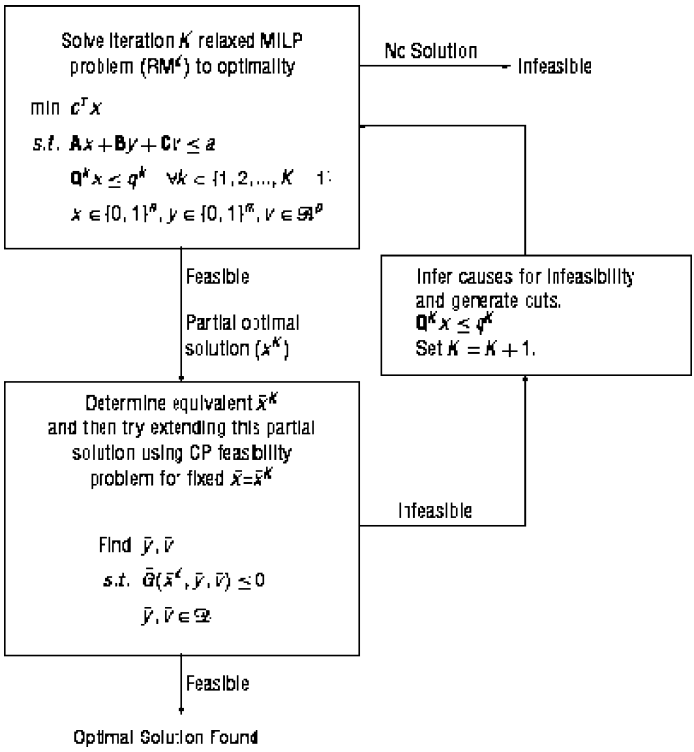


Fig. 1 Benders decomposition algorithm

optimality. Note that integrality constraints on the variable y can be dropped if that makes the relaxed MILP problem easier to solve. If there is no solution, then the original problem is infeasible. Otherwise, values of x are used to determine values of the equivalent CP variables $\overline{x}$. A CP feasibility problem then tries to extend this partial solution to a complete solution $\overline{x}, \overline{y}, \overline{v}$ that satisfies the constraints (11) and (13). If there exists a feasible solution, then this solution is the optimal solution of the problem and the search is terminated. Otherwise, the causes for infeasibility are inferred as cuts and added to the relaxed MILP model of the problem. These cuts could be general "no good" cuts Hooker et al. [46]. In the context of the model (M3), these cuts have the following form for iteration k,

$$\sum_{i \in T^k} x_i - \sum_{i \in F^k} x_i \leq B^k - 1, \; T^k = \{i | x_i^k = 1\}, F^k = \{i | x_i^k = 0\}, B^k = |T^k| \tag{14}$$

Here, $x^k = [x_1^k, x_2^k, ...]$ represents the optimal values of x in iteration k. These general "no good" cuts may be rather weak. For this reason, whenever possible stronger cuts ($Q^k x \leq q^k$) that exploit the special structure of the problem should be used. The problem-specific cuts should not only cut off the current partial solution, but also eliminate partial solutions with similar characteristics. Cuts are a means of communication between the relaxed MILP and CP feasibility models and play a very important role in the success of these methods. The entire procedure is repeated until the solution obtained using the relaxed MILP model can be extended to a full feasible solution, or the relaxed MILP problem becomes infeasible. Assuming that the general "no good" cuts (14) are used, we can establish the following convergence proof for the MILP/CP-based decomposition method [47]:

Theorem 1. *If the MILP/CP decomposition method is applied to solve problem (M3) with the cuts in (14), then the method converges to the optimal solution or proves infeasibility in a finite number of iterations.*

There are two ways in which this method can be implemented. The easier, but less efficient approach is to solve the relaxed MILP model from scratch whenever the cuts are updated. A more efficient approach will be to use a branch-and-cut algorithm where the relaxed MILP problem is considered at each node and updated at certain nodes (e.g., those that are integer feasible) with the corresponding cuts [63].

3.1.2 Branch-and-Cut Method

This algorithm extends the basic branch-and-bound (B&B) algorithm for mixed integer problems to solve problems that are represented using the hybrid MILP/CP model (M3). The idea is in principle straightforward, although it is non-trivial to implement. In the B&B algorithm, the current best integer solution is updated whenever an integer solution with an improved objective-function value is found. In the

proposed algorithm, an additional CP feasibility problem is solved to ensure that the integer solution obtained for the relaxed MILP problem can be extended in the full space. It is only in this case that the current best integer solution for the relaxed MILP problem is updated. If it cannot be extended, then the best current integer solution is not updated and cuts are added to the current and all other open nodes. The proposed branch-and-cut method involves solving a series of LP subproblems obtained by branching on the integer variables. An LP subproblem p for the proposed algorithm has the form,

$$\begin{aligned} \min\ & c^T x \\ \text{s.t.}\ & Ax + By + Cv \le a \\ & Qx \le q \\ & x_p^{LB} \le x \le x_p^{UB} \\ & y_p^{LB} \le y \le y_p^{UB} \\ & x \in \Re^n, y \in \Re^m, v \in \Re^p \\ & x_p^{LB},\ x_p^{UB},\ y_p^{LB},\ y_p^{UB} \in \{0,1\} \end{aligned}$$

The difference in various LP subproblems is only in the upper and lower bounds for all the integer variables. Also, the set of cuts is updated as the search progresses. The objective-function value for any feasible solution of the problem in the full space provides an upper bound (UB) of the objective function. Let P denote the set of LP subproblems to be solved. The branch-and-cut (B&C) method can be summarized as follows:

1. *Initialization.* $UB = \infty$, $P = \{p^0\}$. LP subproblem p^0 is generated by using the same lower and upper bounds on the integer variables as the original problem, and it does not include any cuts.
2. *Check if there are any more problems to be solved.* If $P = \emptyset$, then go to step 5; else go to 3.
3. *Fathom a Node.* Select and remove an LP subproblem p from the set P. The criterion for selecting an LP is also called a *node selection rule*. There are many different rules for selecting an LP, and they play a key role in the efficiency of the B&C algorithm [55]. Solve LP subproblem p.
   - If the LP is infeasible or the optimal value of the objective function (lower bound) is greater than UB, then go to step 2.
   - If any one of the integer variables does not have integral values, then go to step 4.
   - If the solution has integral values for all the integer variables, then determine the CP variables $\overline{x}$ using the values of LP variables x. For fixed $\overline{x}$ solve the following CP feasibility problem that tries to extend this partial solution.

$$\begin{aligned} \text{Find}\ & \overline{v}, \overline{y} \\ \text{s.t.}\ & G(\overline{x}, \overline{y}, \overline{v}) \le 0 \\ & \overline{y}, \overline{v} \in \mathcal{D} \end{aligned}$$

If the CP problem is feasible, then update UB; otherwise, add the cuts in (14) or infer the causes for infeasibility to generate tighter cuts and add them to all the LP subproblems belonging to set P by updating the set (Q, q). Go to step 2.

4. *Branch on a Variable.* Of all the variables that have been assigned non-integral values, select one according to some prespecified branching variable selection rule [55]. Let us denote this integer variable by z_p. Generate two LP subproblems p^1 and p^2 and add them to set P. The subproblem p^1 is generated by specifying the floor of the optimal value of z_p as the upper bound for z_p, and the subproblem p^2 is generated by specifying the ceiling of the optimal value of z_p as the lower bound of z_p. Go to step 2.
5. *Termination.* If $UB = \infty$, then the problem does not have a feasible solution or it is unbounded. Otherwise, the optimal solution corresponds to the current value UB.

For the proposed decomposition and B&C methods to be successful, it is of course very important to choose suitable relaxed MILP and CP feasibility models. Furthermore, rather than using the cuts in (14) inferring strong cuts, $Qx \leq q$, as causes for infeasibility of CP feasibility models can significantly reduce the number of problems to be solved. These cuts, however, are non-trivial and should be derived whenever possible for each class of problems at hand. It is also worth emphasizing that the proposed algorithms do not impose any restriction on the structure of the equivalence relation in (10). It can even be procedural. It should be noted that hybrid models similar to the ones presented in this section can also be solved using OPL [76]. In OPL, the solution algorithm for the hybrid model solves an LP subproblem involving all the linear constraints, as well as a constraint-propagation subproblem involving all the constraints at every node of the search tree. Usually, all the original CP constraints (6) are needed in the hybrid MILP/CP OPL model. Furthermore, the equivalence relations (10) must be written in closed form as equations, inequalities, or symbolic relations. Even though the algorithms presented in this section are limited to the MILP models that have only a subset of binary variables with non-zero coefficients in the objective function, they can in principle be generalized for MILP models that have both binary and continuous variables in the objective function. The partial solution will then involve both binary and continuous variables and the CP problem will extend this partial solution in the full space. However, in such a case, the biggest challenge is to derive effective cuts that exclude partial solutions that are feasible for the master problem but cannot be extended in the full space.

3.2 Single-Stage Scheduling Problem

Jain and Grossmann [47] considered a specific scheduling problem that falls into the class of those that can be tackled by a decomposition method. It consists of finding a least-cost schedule to process a set of orders $i \in I$ using a set of dissimilar

parallel machines $m \in M$. Processing of an order can only occur after its release date r_i and must be completed at the latest by its due date d_i. It is assumed that a particular order can be processed on any of the machines. The processing cost and the processing time of order i in machine m are given by $c_{i,m}$ and $p_{i,m}$, respectively.

3.2.1 Hybrid MILP/CP Model

The hybrid MILP/CP algorithm reported in Jain and Grossmann [47] has the following main components. The relaxed MILP master problem includes (15) as the objective function, (16)–(21) as the model constraints and (22) as the integer cuts. Set I_m^k gives the orders that were allocated to unit m in iteration k and for which no valid sequence could be found by solving the CP feasibility problem, see (23). Binary variables $X_{i,m}$ indicate that order i is assigned to unit m, while continuous variables Ts_i give the time at which order i starts to be processed.

$$\min \sum_{i \in I} \sum_{m \in M} c_{i,m} X_{i,m} \tag{15}$$

$$Ts_i \geq r_i \quad \forall i \in I \tag{16}$$

$$Ts_i \leq d_i - \sum_{m \in M} p_{i,m} X_{i,m} \quad \forall i \in I \tag{17}$$

$$\sum_{m \in M} X_{i,m} = 1 \quad \forall i \in I \tag{18}$$

$$\sum_{m \in M} p_{i,m} X_{i,m} \leq \max_i \{d_i\} - \min_i \{r_i\} \quad \forall m \in M \tag{19}$$

$$Ts_i \geq 0 \quad \forall i \in I \tag{20}$$

$$X_{i,m} \in \{0,1\} \quad \forall i \in I, m \in M \tag{21}$$

$$\sum_{i \in I_m^k} X_{i,m} \leq |I_m^k| - 1 \quad \forall m \in M, k = \{1,2,...K-1\} \tag{22}$$

$$I_m^k = \{i \in I : X_{i,m}^k = 1\} \quad \forall m \in M, k = \{1,2,...K-1\} \tag{23}$$

With the assignments $(X_{i,m}^k)$ from the master problem (iteration k) one can determine the variable superscript z_i by (24). Then, a total of m independent CP subproblems are generated to identify if a feasible schedule can be obtained.

Equations (25)–(28) are written using ILOG's OPL modeling language [76], valid up to OPL Studio 3.7.1. Note that in (28), t_m is the unary resource corresponding to machine m.

$$z_i = m \quad \forall i \in I, m \in M, X_{i,m}^k = 1 \tag{24}$$

$$i.start \geq r_i \quad \forall i \in I, X_{i,m}^k = 1 \tag{25}$$

$$i.start \leq d_i - p_{i,z_i} \quad \forall i \in I, X_{i,m}^k = 1 \tag{26}$$

$$i.duration = p_{i,z_i} \quad \forall i \in I, X_{i,m}^k = 1 \tag{27}$$

$$i \; requires \, t_{z_i} \quad \forall i \in I, X_{i,m}^k = 1 \tag{28}$$

Knapsack Constraints and Cover Cuts

Earlier hybrid methods have focused on the development of integer cuts during the iterative (or branch-and-cut) scheme [9, 47]. However, these tend be rather weak with many cuts being required for large problem sizes. Eventually, the problem becomes intractable and we end up with no solution since the first feasible schedule is also the optimal one. In order to widen the scope of the hybrid algorithm, Maravelias [51] has added a pre-processing stage that results in orders of magnitude savings in computational effort. It basically identifies infeasible assignments and generates constraints that are added to the cut-pool of the master problem, making it tighter.

The pre-processing algorithm of Maravelias [51] consists of three phases. The first two involve knapsack constraints and are given in Algorithms 1–2. The third adds cover cuts [4, 78] for pairs of jobs that are not examined by the procedures of Algorithms 1 and 2, see Algorithm 3.

Algorithm 1: Pre-processing algorithm of Maravelias [51] for single stage plants: Phase1

forall $(m \in M)$
  forall $(i, i' \in I : r_i \leq r_{i'} \wedge d_i \leq d_{i'})$
    $S = \text{union}(i'' \in I : r_i \leq r_{i''} \wedge d_{i''} \leq d_{i'})$
    If $\left(\sum_{i'' \in S} p_{i'',m} > d_{i'} - r_i \right)$ then
      Add Knapsack constraint $C1 := \sum_{i'' \in S} p_{i'',m} X_{i'',m} > d_{i'} - r_i$

Algorithm 2: Pre-processing algorithm of Maravelias [51] for single stage plants: Phase 2

forall $(m \in M)$
  forall $(i \in I)$
  S=union$(i' \in I : r_i \leq r_{i'} \wedge d_{i'} \leq d_i)$
    $A := \min_{i' \in S \setminus \{i\}}\{r_{i'}\} - r_i$; $B := d_i - \max_{i' \in S \setminus \{i\}}\{d_{i'}\}$
    C:=min$\{A, B\}$
    If $\left(\sum_{i' \in S} p_{i',m} > d_i - r_i - C\right)$ then
      Add Knapsack constraint $C2 := \sum_{i' \in S} p_{i',m} X_{i',m} > d_i - r_i - C$

Algorithm 3: Pre-processing algorithm of Maravelias [51] for single stage plants: Phase 3

forall $(m \in M)$
  forall $(i \in I)$
    If $\left(p_{i,m} + p_{i',m} > \max\{d_i - r_{i'},\ d_{i'} - r_i\}\right)$ then
      Add Knapsack constraint $C1 := X_{i,m} + X_{i',m} \leq 1$

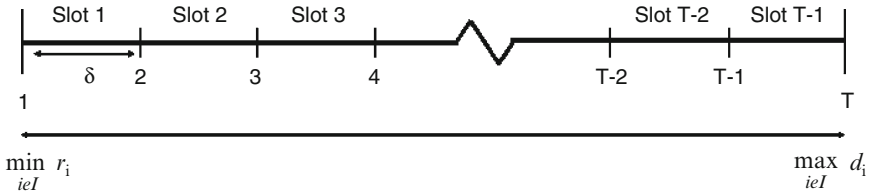


Fig. 2 Uniform time grid for discrete-time formulation

3.2.2 Discrete-Time MILP Model (DT)

An alternative to the decomposition algorithm is to use full-space models relying on a concept for event representation other than sequencing variables. Note that CP models for scheduling implicitly rely on general precedence sequencing variables [15]. The following discrete-time formulation [14] uses a single uniform time grid to keep track of the orders, see Fig. 2. It is a simplification of the general Resource-Task Network model of Pantelides [58]. If the problem data consist of integer values, setting the integer length $\delta = 1$ ensures accuracy. With this parameter and the maximum values for both release and due dates, one can easily specify the number of time points $t \in T$ in the grid.

The model is very simple and the resulting mathematical problem is very tight. There are only two sets of variables and constraints besides the objective function. Binary variables $N_{i,m,t}$ indicate that order i starts to be processed in unit m at time point t. Continuous variables $R_{m,t}$ then give the machines availability (they are equal to 1 if the unit is idle and 0 otherwise). Set $I_{t,m}$ gives the orders that can start to be processed in unit m at point t. Note that because of the release and due dates,

each order can only start on a subset of the time points in the grid. The objective function is given by (29). The resource balances for the machines are given by (30), while (31) ensures that every order must be processed in exactly one machine.

$$\min \sum_{t \in T} \sum_{m \in M} \sum_{i \in I_{t,m}} N_{i,m,t} c_{i,m} \tag{29}$$

$$R_{m,t} = \left(1|_{t=1} + R_{m,t-1}|_{t \neq 1}\right) - \sum_{i \in I_{t,m}} N_{i,m,t} + \sum_{i \in I_{t-p_{i,m},m}} N_{i,m,t-p_{i,m}} \quad \forall m \in M, t \in T \tag{30}$$

$$\sum_{m \in M} \sum_{t \in T} N_{i,m,t} = 1 \quad \forall i \in I \tag{31}$$

3.2.3 Continuous-Time MILP Model (CT)

Castro and Grossmann [14] have proposed a continuous-time formulation that uses a different time grid per machine. Contrary to the discrete-time grid, the location of the grid's time points are not known a priori but are going to be determined by the solver as part of the optimization. Furthermore, the number of time points to use cannot accurately be predicted. Since the solution space is very much dependent on the number of elements in set T and so is the computational effort, an iterative search heuristic procedure is typically employed. One just keeps incrementing the elements in T, one by one, up to the point the objective function stops improving. The corresponding nonuniform time grid is shown in Fig. 3, with continuous variables $T_{t,m}$ giving the time associated to point t of grid m. Note that the m time grids are totally independent.

It can be assumed without the loss of generality that all orders can be processed on a single time interval, whereas in the discrete-time model they typically span several intervals. This is the reason why (32) differs from (30). Besides (29) and (31) (with $I_{m,t} = I$), there are three other constraints. Equation 33 state that the

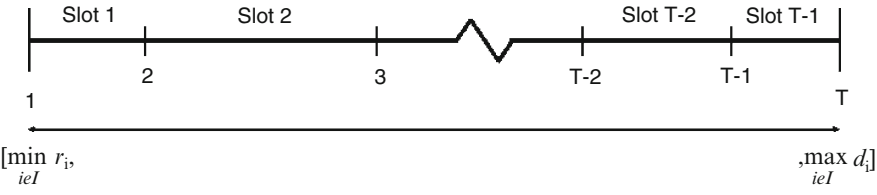


Fig. 3 Nonuniform time grid for continuous-time formulation

difference in time between two consecutive time points must be greater than the processing time of the order being processed in the machine. Equations (34) and (35) are the release and due date constraints, where the time horizon (H) should be set to the maximum due date.

$$R_{m,t} = \left(1|_{t=1} + R_{m,t-1}|_{t\neq 1}\right) - \sum_{i\in} N_{i,m,t} + \sum_{i\in I} N_{i,m,t-1} \quad \forall m \in M, t \in T \tag{32}$$

$$T_{t+1,m} - T_{t,m} \geq \sum_{i\in I} N_{i,m,t}\, p_{i,m} \quad \forall m \in M, t \in T \tag{33}$$

$$T_{t,m} \geq \sum_{i\in I} N_{i,m,t}\, r_i \quad \forall m \in M, t \in T \tag{34}$$

$$T_{t,m} \leq \sum_{i\in I} N_{i,m,t}\,(d_i - p_{i,m}) + H\left(1 - \sum_{i\in I} N_{i,m,t}\right) \quad \forall m \in M, t \in T \tag{35}$$

3.3 Computational Studies

In this section, the performance of the hybrid MILP/CP algorithm is evaluated, with and without the cuts proposed by Maravelias [51], and compared to that of the discrete- (DT) and continuous-time (CT) MILP models. Eight test problems (P3-P10) taken from Castro and Grossmann [14] are used for this purpose. The hybrid MILP/CP algorithms have been implemented in ILOG OPL Studio 3.7.1, which uses CPLEX 9.1 for the solution of the MILP master problems. The full-space MILP models have been implemented in GAMS and solved using both CPLEX 9.1 and CPLEX 11.1. They were all solved to optimality using default options, unless otherwise stated, by a laptop with an Intel Core2 Duo T9300 processor running at 2.5 GHz, 4 GB of RAM, and running Windows Vista Enterprise. The results are listed in Table 2.

Table 2 Computational results for hybrid MILP/CP algorithm (IT = iterations; IC = integer cuts; PH1−3 = Cuts from Phase 1 − 3)

		Jain and Grossmann [47]			Maravelias [51]					
Problem	Optimum	IT	IC	CPUs	CPUs	IT	IC	PH1	PH2	PH3
P3 (|I| = 15, |M| = 5)	116	11	16	0.16	0.12	1	0	262	39	12
P4 (|I| = 15, |M| = 5)	105	6	5	0.02	0.17	2	1	152	21	0
P5 (|I| = 20, |M| = 5)	159	29	62	1.09	0.44	1	0	536	56	12
P6 (|I| = 20, |M| = 5)	144	24	30	0.3	0.36	1	0	378	33	0
P7 (|I| = 25, |M| = 5)	51	1847	–	12400[a]	76.4	58	139	713	95	45
P8 (|I| = 25, |M| = 5)	54	451	–	3600[a]	85	27	69	717	97	64
P9 (|I| = 30, |M| = 5)	53	368	811	69.1	7.72	4	7	1020	106	11
P10 (|I| = 30, |M| = 5)	75	–	–	–	7200[b]	1	0	1043	120	65

[a]Interrupted, solution from relaxed MILP = 50.
[b]Interrupted during solution of first relaxed MILP, solution found = 73, best possible = 68.46.

With the exceptions of P4 and P6, the pre-processing algorithm of Maravelias [51] achieves large savings in computational time compared to the original cuts proposed by Jain and Grossmann [47]. Notice that there can be more than 1,000 cuts generated, which have the effect of leading to better assignments by the relaxed master MILP problem, which is measured by the requirement of fewer iterations and integer cuts. In this respect, P9 needs just 4 iterations as opposed to 368 by the original Jain and Grossmann [47] algorithm. The practical consequence is the ability to solve P7-8 and P11 in more or less one minute, with the latter approach being unable to find the solution in one-hour or more. Naturally, adding a large number of cuts makes the master problem harder to solve so if few iterations are involved it is better not to add them (only for the smaller instances).

Despite its very good performance, the approach of Maravelias [51] was unable to complete even the first iteration of P10 up to 2-h of computational time. However, the full-space formulations of the time grid based formulations are able to solve this problem relatively fast as seen in Table 3 where the number of orders and machines is displayed (|I|, |M|). Notice that the discrete-time formulation is very tight, with the integrality gap being equal to zero for P3 and P9. Because of this, and despite leading to one order of magnitude larger MILPs, it is able to solve all problems in less than 8 s, while its continuous-time counterpart is slightly worse, particularly in problem P10. It is interesting to note that these savings compared to the decomposition algorithm can be attributed to the great progress that has been achieved with MILP solvers (e.g., see the comparison between CPLEX 9.1 and 11.1 in Table 3, where the computational time of the former does not include the model generation time, which is somewhat important for the discrete-time formulation). Jain and Grossmann [47] found these solvers to be orders of magnitude slower in 2001 for an MILP model with general precedence sequencing variables. On the other hand, compared to the decomposition algorithm, the limitation of the full space methods is that one has to specify the number of time intervals for the discrete and continuous models (|T|). In Table 3, the values chosen yield the same objective value as the one found by the decomposition algorithm.

Table 3 Computational results for discrete- (DT) and continuous-time formulations (RMIP = solution from relaxed linear problem, DV = discrete variables, EQ = constraints)

	Castro and Grossmann [14] DT						Castro and Grossmann [14] CT					
Problem	|T|	RMIP	DV	EQ	CPUs[a]	CPUs[b]	|T|	RMIP	DV	EQ	CPUs[a]	CPUs[b]
P3	371	116	13320	1871	2.64	2.58	6	113.03	405	121	0.81	0.74
P4	371	104	16950	1871	1.31	4.19	8	104	405	121	0.05	0.41
P5	381	158.5	18790	1926	7.34	4.38	7	155.8	635	146	140	15.6
P6	381	142.94	23320	1926	6.52	7.21	9	143	845	186	15.8	0.59
P7	294	50.25	13548	1496	0.92	3.19	6	45.21	655	131	29.2	8.8
P8	294	53.71	13273	1496	1.9	3.92	6	48.13	655	131	27.3	18.8
P9	294	53	18063	1501	2.59	4.23	7	50	935	156	8.9	3.42
P10	294	73.85	16753	1501	5.64	7.36	7	69.72	935	156	4322	952

[a]CPLEX 9.1.
[b]CPLEX 11.1.

4 Applications to Vehicle Routing and Crew Scheduling

Vehicle Routing Problems (VRP) and Crew Scheduling Problems are widely present in today's industries since they are at the core of the transportation industry of both human and merchandise. They account for a significant portion of the operational cost of many organization such as airlines, freight carrier, or public transit companies.

The VRP can be described as follows: given a set of customers C, a set of vehicles V, and a depot d, find a set of routes of minimal length, starting and ending at d, such that each customer in C is visited by exactly one vehicle in V. Each customer having a specific demand, there are usually capacity constraints on the load that can be carried by a vehicle. In addition, there is a maximum amount of time that can be spent on the road. The time window variant of the problem (VRPTW) imposes the additional constraint that each customer c must be visited after time a_c and before time b_c. One can wait in case of early arrival, but late arrival is not permitted.

In a typical Crew Scheduling Problem, a set of trips has to be assigned to some available crews. The goal is to assign a subset of the trips, which represent segments of bus routes or flight legs, to each crew in such a way that no trip is left unassigned. As usual, not every possible assignment is allowed since a number of constraints must be observed. Additionally, a cost function has to be minimized. In almost all cases, the trips are defined as the nodes of a network whose arcs specify which connections are allowed and which ones are impossible. Crew rostering can be interpreted as vehicle routing where the capacity and time windows constraints are replaced by rules coming out of union agreements. This makes Crew Rostering Problems more complex to define formally. In the remainder of this chapter, we will use the notation of the VRP as its definition is more precise.

The Set Covering formulations of the VRP consist of selecting, among the possible sets of customers who can be visited by the same vehicle, the optimal set of routes. Letting r be a feasible route in the original graph (which contains N customers); R be the set of all possible routes r; c_r be the cost of visiting all the customers in r; $A = (a_{ir})$ be a Boolean matrix expressing the presence of a particular customer (denoted by index $i \in \{1..N\}$) in route r; and x_r a Boolean variable specifying whether the route r is chosen ($x_r = 1$) or not ($x_r = 0$), the Set Partitioning Problem is defined as (S):

$$\min \sum_{r \in R} c_r x_r$$

$$s.t \sum_{r \in R} a_{ir} x_r = 1 \ \forall i \in \{1..N\}$$

$$x \in \{0, 1\}^N$$

This formulation, however, poses some problems. First, since it is impractical to construct and store the set R because of its very large size, it is usual to work

with a partial set R' that is enriched iteratively by solving the *pricing problem*. Second, the Set Partitioning formulation is difficult to solve when R' is small and it allows negative dual values which can be problematic for the subproblem (a negative dual means a negative marginal cost to service a customer). That is why, in general, the following relaxed Set Covering formulation is used instead as a Master Problem (M):

$$\min \sum_{r \in R'} c_r x_r$$

$$s.t \sum_{r \in R'} a_{ir} x_r \geq 1 \ \forall i \in \{1..N\}$$

$$x \in [0,1]^N$$

To enrich R', it is necessary to find new routes that offer a better way to visit the customers they contain, that is, routes which present a negative reduced cost. The reduced cost of a route is calculated by replacing the cost of an arc (the distances between two customers) d_{ij} by the reduced cost of that arc $c_{ij} = d_{ij} - \lambda_i$ where λ_i is the dual value associated with customer i. The dual value associated with a customer can be interpreted as the marginal cost of visiting that customer in the current optimal solution (given for R'). The objective of the subproblem is then the identification of a negative reduced cost path, that is, a path for which the sum of the travelled distance is inferior to the sum of the marginal costs (dual values). Such a path represents a novel and better way to visit the customers it serves.

The optimal solution of (M) has been identified when there exists no more negative reduced cost path. This solution can, however, be fractional, since (M) is a relaxation of (S), and thus does not represent the optimal solution to (S) but rather a lower bound on it. If this is the case, it is necessary to start a branching scheme in order to identify an integer solution.

Most column generation methods make use of dynamic programming to solve the shortest path subproblem where the elementary constraint (i.e., the constraint imposing that a path does not go through the same node more than once) has been relaxed [23]. This method is very efficient. But since the problem allows negative weight on the arcs, the path produced may contain cycles (since negative cost cycles decrease the objective function). However, applications of column generation in Crew Scheduling generally present an acyclic subproblem graph (one dimension of the graph being *time*) which eliminates this problem. Since routing problems are cyclic by nature, the subproblem reduces to an elementary shortest path problem with resource constraints (ESPPRC). One has to find negative cost elementary paths from the depot to the depot, satisfying capacity and time constraints. A solution procedure based on dynamic programming for this problem is proposed in [28].

Constraint programming methods can not only identify elementary negative reduced cost paths by working on the smaller original cyclic graph, but also allow the addition of any form of constraints on the original problem (which is not the case with the dynamic programming approach). It is thus possible to deal with multiple

Table 4 Parameters of the set, position and arc based models

$N = 0..n$	The set of all customers
$0, n+1$	Copies of the depot
$N' = 1..n+1$	The set of all nodes except the initial depot
d_{ij}	Distance from node i to node j
t_{ij}	Travel time from node i to node j
a_i, b_i	Bounds of node i's time window
l_i	Load to take at node i
λ_i	Dual value associated with node i
C	Capacity of the vehicle
$c_{ij} = d_{ij} - \lambda_i$	Reduced cost to go from node i to node j

time windows, precedence constraints among visits or any logical implication satisfying special customer demands. In fact, the case of multiple time windows TSP was addressed successfully in [60].

The original motivation to use constraint programming-based column generation [48] to solve airline crew assignment problems was that some problems were too complex to be modeled easily by pure Operational Research (OR) methods. Thus, the use of constraint programming to solve the subproblem in a column generation approach provided both the decomposition power of column generation and the modeling flexibility of constraint programming.

We divide these CP models in three classes namely the *set-based* models, *position-based* model and *arc-based* models and illustrate these models on the pricing problem of CP-based column generation approach to the VRPTW, with the parameters of Table 4.

4.1 Set Based Model

To solve complex Crew Rostering Problems, Junker et al. [48] and Fahle et al. [27] propose to model the constrained shortest path problem in CP using a single set variable Y, which contains the nodes to be included in the negative reduced cost column. The value of a set variable is a set of integers selected from an initially given domain. The current domain of a set variable is defined by a lower and an upper bound, which are also called required set req(Y) and possible set pos(Y). The value of the set variable has to be a superset of req(Y) and a subset of pos(Y). Set variables replace an array of boolean variables and generally lead to more compact constraint models and more efficient and powerful propagation algorithms. The model thus looks extremely simple as all its complexity and efficiency are hidden in the structure of the underlying network and the global constraint *NegativeReducedCost*.

4.1.1 Variables

$Y \subseteq N$ The set of all clients in path to be generated

4.1.2 Objective

minimize z Reduced cost of column

4.1.3 Constraints

$NegativeReduceCost(Y, c, z)$	Y is a shortest path with $z < 0$.
$Y \in F$	Y is feasible.

The feasibility region F is formulated as a CP model using any combination of local and global constraints and possibly including additional variables. Since the Crew Rostering problem for which this approach was designed is by nature acyclic (the underlying network is time directed), it is easy to compute in polynomial time the shortest path covering nodes in Y.

A special constraint, *NegativeReduceCost*, is also introduced to improve pruning and efficiency of the overall method. This constraint, which ensures that the nodes in Y are part of a feasible path, also enforces bound consistency by solving a shortest path problem on both the required and possible sets of Y. An incremental implementation of the Shortest Path algorithm ensures that the filtering is done efficiently.

Numerical results based on data of a large European airline are presented in [48] and demonstrate the potential of the approach. However, set variables cannot be used to model Vehicle Routing Problem. Since these problems are defined over networks which contain cycles, the construction of a complete solution from the set of visits included in Y would require solving a TSP.

4.2 Position Based Model

A second model to the Crew Rostering Problem, simultaneously and independently proposed by Yunes et al. [79], is based on an array of finite domain variables $X_{p \in P} \in N$ which identify which node or task in N is to be performed by a bus driver in position $p \in P$. This model is quite straightforward, flexible and works without the addition of dedicated global constraints.

4.2.1 Variables

$X[p] \in N$	$\forall p \in P$	Identifies client visited in p.
$T[p] \in [\min_{i \in N}(a_i) .. \max_{i \in N}(b_i)]$	$\forall p \in P$	Starting time of service in p.
$L[p] \in [0..C]$	$\forall p \in P$	Load taken by the vehicle in p.

4.2.2 Objective

minimize $\sum_{i \in N} c_{X[l-1],\, X[l]}$	Reduced cost of the column

4.2.3 Constraints

$AllDifferent(X)$		Conservation of flow.
$T_{p-1} + t_{X_{p-1}, X_p} \leq T_p$	$\forall p \in N$	Time window constraints.
$L_{p-1} + l_{X_p} = L_p$	$\forall i \in N$	Capacity constraints.

The AllDifferent constraint imposes that all clients in the path cannot be visited more than once, thus making sure that the path generated is elementary.

In [80], the author mentioned that the hybrid column generation algorithms based on this model for solving these problems always performed better, when obtaining optimal solutions, than isolated CP or MIP approaches. All the proposed algorithms have been implemented and tested over real-world data obtained from the urban transit company serving the city of Belo Horizonte in Brazil.

The level of propagation achieved by such model is not sufficient to solve Vehicle Routing Problem which contains more than around 5 or 6 customers per routes. However, in a context where routes tend to naturally contain very few customers, the position-based model has the advantage of introducing much less variables than the arc-based model presented next.

4.3 Arc Based Model

To solve vehicle routing with time window with longer routes, Rousseau *et al.* [70] based their model on variables which identify the transitions between successive nodes (or trips). We present here the main component of this model.

4.3.1 Variables

$S_i \in N'$	$\forall i \in N$	Direct successor of node i.
$T_i \in [a_i, b_i]$	$\forall i \in N \cup \{n+1\}$	Time of visit of node i.
$L_i \in [0, C]$	$\forall i \in N \cup \{n+1\}$	Truck load before visit of node i.

4.3.2 Objective

$minimize \sum_{i \in N} c_{iS_i}$ Reduced cost of the column.

4.3.3 Constraints

$AllDifferent(S)$	(1) Conservation of flow.
$NoSubTour(S)$	(2) SubTour elimination constraint.
$T_i + t_{iS_i} \leq T_{S_i} \quad \forall i \in N$	(3) Time window constraints.
$L_i + l_i = L_{S_i} \quad \forall i \in N$	(4) Capacity constraints.

The S variable identifies the *direct successor* of each node of the graph, but for the sake of brevity we will refer to these variables only as *successor* variables. The nodes which are left out of the chosen path are represented with self loops and thus have their S_i value fixed to the value i. Constraint sets (3) and (4) enforce the respect of the capacity and time windows by propagating the information about the time and load when a new arc becomes known.

The *AllDifferent* (1) constraint is used to express conservation of flow in the network. The nature of the decision variable already enforces that each node has exactly one outgoing arc, but we also need to make sure it also has exactly one ingoing arc. To do so, it is necessary to insure that no two nodes have the same successor, which is the role of the *AllDifferent* constraint. This property is obtained by solving a matching problem in a bipartite graph and by maintaining arc consistency in an incremental fashion as described in [67].

The *NoSubTour* (2) constraint, illustrated in Fig. 4, is taken from the work of Pesant et al. [59]. For each chain of customers, we store the name of the first and last visit. When two chains are joined together (when a variable is fixed and a new arc is introduced), we take two actions. First, we update the information concerning the first and last visits of the new (larger) chain, and then, we remove the value of the first customer from the domain of the Successor variable of the last customer.

This model has never been tested in the context of Crew Rostering Problem, but it was used as a basis for the network reduction model proposed by [33] for Tail Assignment Problems.

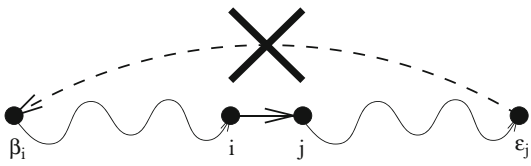


Fig. 4 NoSubTour constraint

4.3.4 Redundant Constraints

In order to improve solution time, [70] introduced redundant constraints, which do not modify the solution set but allow improved pruning and filtering. These are the CP equivalent of valid inequalities (cuts) usually added to MIP formulations in order to improve its linear relaxation.

TSPTW Constraints

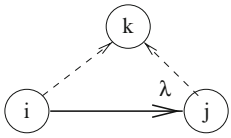
The constraints introduced in [59], which perform filtering based on time window narrowing, are included in the present method. These constraints maintain for each node (say i) the latest possible departure time and the node (say j) associated with this time. When the domain of S_i is modified, the constraint first verifies that j is still in the domain of S_i and if so performs no filtering.

Arc Elimination Constraints

Rousseau et al. [70] also introduced a new family of redundant constraints that can reduce the number of explored nodes of the search tree by reducing the number of arcs of the subproblem graph. The idea is to eliminate arcs which we know will not be present in the pricing problem optimal solution. Such a practice is known as cost-based filtering or optimization constraints (introduced in [29]) since it filters out feasible solutions but not optimal ones.

The proposed arc eliminating constraints can reduce the size of the original graph based on the following idea: if the dual value associated with a customer is not sufficiently large, it may then not be worth the trip to visit this customer. Again, these constraints are valid only if the triangular inequality holds for resource consumption. Otherwise, the visit of an intermediate customer could yield savings in some resources and thus allow the visit of extra customers. Since this inequality does not hold when $j = i$ (in the following equations), the constraint was not defined over self-loop.

The Arc Elimination constraint of type one (a second type of constraint is detailed in [70]) is defined as follows: given an arc (i, j), if for all other customers k that are elements of the domain of the successor variable of j (S_j), it is always cheaper to go directly from i to k (d_{ik}) than to travel through j $(d_{ij} + d_{jk} - \lambda_j)$, then the arc (i, j) can be eliminated from the subproblem graph since it will never be part of an optimal solution.



$\forall i \in N, \forall j \in S_i : j \neq i$ impose that $(\forall k \in S_j : k \neq i \neq j \quad (d_{ij} + d_{jk} - \lambda_j > d_{ik})) \Rightarrow S_i \neq j$

4.4 Solving the Integer Problem

The optimal solution to the master problem (M) is obtained once we have proven that no negative reduced cost path exits. Unfortunately, this solution is not always integral (therefore, not a solution to (S)) and a branching scheme is thus needed to close the integrality gap.

The *set-based* approach of Junker et al. [48] solves (S) approximately by applying a branch-and-bound algorithm on the set of obtained columns, once (M) has been solved to optimality. Yunes et al. [80], in their *position-based* model, implement a full branch-and-price algorithm that solve (S) exactly. Using ABACUS [56], they choose to branch on the original variable of (S), that is the variables associated to the columns. Constraint Programming is quite useful in this case since it allows to state that a given path is forbidden once its associated variable as been fixed to 0. In [70] *arc-based* model, the branching scheme uses branching variables $\{B_i \in N' | i \in N\}$, a set of successor variables similar to those used to describe the subproblem. Once the optimal solution of the master problem (M) has been found, the most fractional variable is chosen as the next branching (B) variable to be fixed. To do so, they first calculate the flow that traverses each arc $f_{ij} = \sum_{r \in R'} f_{ij}^r x_r$, where f_{ij}^r is a boolean value indicating whether j is the successor of i in route r. Then, for each customer i, they compute the number of positive flow outgoing arcs $o_i = \sum_{j \in \{1..N\}} (f_{ij} > 0)$. Finally, they select the B_i variable which is associated with the maximum value of o_i and a branch on the value j which maximizes f_{ij}. The whole branch-and-price algorithm is implemented as a CP model and solved with a CP solver in order to take advantage of the available tree search mechanisms.

5 Conclusions

This paper has provided a review of decomposition algorithms for models that are formulated as hybrid mixed-integer linear/constraint programming problems, focusing on single stage machine scheduling problems crew rostering and vehicle routing problems. A general decomposition algorithm for the hybrid MILP/CP model was first presented in terms of a Benders decomposition scheme and a branch and cut method. Computational results were presented to compare the hybrid model for the single stage scheduling problem, with full-space discrete and continuous-time MILP models. The computational results show that the decomposition algorithm is only competitive in small to medium scale problems, while the full space MILP models are significantly faster in the larger problems. This is largely attributed to the very significant advances that have been achieved with LP based branch and bound methods. Although these results are somewhat discouraging for the decomposition techniques, it should be noted that the branch and cut implementation is likely to be much faster, and could still hold promise compared to full space methods, which require specifying the number of intervals, a choice that is non-trivial, particularly in continuous-time models.

The Constraint Programming-Based Column Generation framework was then presented and illustrated on the Vehicle Routing and Crew Rostering Problems. This framework is flexible since it can handle not only resource based constraints but also constraints of almost any structure, while still providing acceptable performance on known benchmark problems. While the CP-based column generation framework has been introduced to handle modeling issues arising from complex constraints, it is now a valid option even for those problems having an NP-hard pricing subproblem as reported by Gualandi and Malucelli [31]. The framework is, however, more efficient when application dependent global constraints with cost are designed and implemented.

References

1. Artigues C, Gendreau M, Rousseau L-M, Vergnaud A (2009) Solving an integrated employee timetabling and jobshop scheduling problem via hybrid branch-and-bound. Comput Oper Res 36(8):2330–2340
2. Adams J, Balas E, Zawack D (1988) The shifting bottleneck procedure for job shop scheduling. Manage Sci 34(3):391–401
3. Applegate D, Cook B (1991) A computational study of the job shop scheduling problem. Oper Res Soc Am 3:149–156
4. Balas E (1975) Facets of the knapsack polytope. Math Program 8 (2), 146–164
5. Balas E, Vazacopoulos A (1998) Guided local search with shifting bottleneck for job shop scheduling. Manage Sci 44(2):262–275
6. Barnhart C, Johnson L, Nemhauser G, Savelsbergh M, Vance P (1998) Branch-and-Price: column generation for solving huge integer programs. Oper Res 46:316–329
7. Benders, J. F., 1962. Partitioning procedures for solving mixed-variables programming problems. Numer Math 4:238–252
8. Bockmayr A, Kasper T (1998) Branch-and-infer: a unifying framework for integer and finite domain constraint programming. INFORMS J Comput 287–300
9. Bockmayr A, Pisaruk N (2003) Detecting infeasibility and generating cuts for mixed integer programming using constraint programming. In: Proceedings CPAIOR'03. p 24
10. Capone A, Carello G, Filippini I, Gualandi S, Malucelli F (2010) Solving a resource allocation problem in wireless mesh networks: a comparison between a CP-based and a classical column generation. Networks 55:221–233
11. Carlier J, Pinson E (1989) An algorithm for solving the job-shop problem. Manage Sci 35: 164–176
12. Caseau Y, Laburthe F (1994). Improving clp scheduling with task intervals. In: Hentenryck PV (ed) Logic programming: proceedings of the *11th International Conference.* MIT, Cambridge, pp 369–383
13. Caseau Y, Laburthe F (1996) Cumulative scheduling with task intervals. In: Maher M (ed) Logic programming: proceedings of the *1996 Joint International Conference and Symposium*. MIT, Cambridge, pp 363–377
14. Castro PA, Grossmann IE (2006) An efficient MILP model for the short-term scheduling of single stage batch plants. Comput Chem Eng 30(6–7):1003–1018
15. Castro PM, Grossmann IE, Novais AQ (2006) Two new continuous-time models for the scheduling of multistage batch plants with sequence dependent changeovers. Ind Eng Chem Res 45(18):6210–6226
16. Chandru V, Hooker J (1999) Optimization methods for logical inference. Wiley, New York
17. Chvátal V (1983) Linear programming. Freeman, New York

18. Correa AI, Langevin A, Rousseau L-M (2007) Scheduling and routing of automated guided vehicles: a hybrid approach. Comput Oper Res 34(6):1688–1707
19. Darby-Dowman K, Little J (1998) Properties of some combinatorial optimization problems and their effect on the performance of integer programming and constraint logic programming. INFORMS J Comput 10:276–286
20. Darby-Dowman K, Little J, Mitra G, Zaffalon M (1997) Constraint logic programming and integer programming approaches and their collaboration in solving an assignment scheduling problem. Constraints 1:245–264
21. Demassey S, Pesant G, Rousseau L-M (2006) A cost-regular based hybrid column generation approach. Constraints 11(4):315–333
22. Desrosiers J, Dumas Y, Solomon MM, Soumis F (1995) Time constrained routing and scheduling. In: Ball MO, Magnanti TL, Monma CL, Nemhauser GL (eds) Network Routing, vol 8 of Handbooks in operations research and management science. North-Holland, Amsterdam, pp 35–139
23. Desrosiers J, Solomon MM, Soumis F (1993) Time constrained routing and scheduling. Handbooks of operations research and management science, vol 8. North-Holland, Amsterdam, pp 35–139
24. Easton K, Nemhauser GL, Trick MA (2002) Solving the travelling tournament problem: a combined inte- ger programming and constraint programming approach. In: Proceedings of practice and theory of automated timetabling. Lecture notes in computer science, vol 2740. Springer, Berlin, pp 100–112
25. El Hachemi N, Gendreau M, Rousseau L-M A hybrid constraint programming approach to the log-truck scheduling problem, Ann Oper Res DOI 10.1007/s10479-010-0698-x
26. Fahle T, Sellmann M (2002). Constraint programming based column generation with knapsack subproblems. Ann Oper Res 115:73–94
27. Fahle T, Junker U, Karisch SE, Kohl N, Sellmann M, Vaaben B (2002) Constraint programming based column generation for crew assignment. J Heuristics 8(1):59–81
28. Feillet D, Dejax P, Gendreau M, Gueguen C (2004) An exact algorithm for the elementary shortest path problem with resource constraints: application to some vehicle routing problems. Networks 44(3):216–229
29. Focacci F, Lodi A, Milano M (1999) Solving TSP through the Integration of OR and CP techniques. Electron Notes Discrete Math 1
30. Gualandi S (2008) Enhancing constraint programming-based column generation for integer programs. PhD thesis, Politecnico di Milano
31. Gualandi S, Malucelli F (2009) Constraint programming-based column generation. 4OR A Q J Oper Res 7(2):113–137
32. Grönkvist M (2004) A constraint programming model for tail assignment. In: Proceedings of integration of AI and OR techniques in CP for combinatorial optimization. Lecture notes in computer science, vol 3011. Springer, Berlin, pp 142–156
33. Grönkvist M (2006) Accelerating column generation for aircraft scheduling using constraint propagation. Comput OR 33(10):2918–2934
34. Gabteni S, Grönkvist M (2006) A hybrid column generation and constraint programming optimizer for the tail assignment problem. In: Proceedings of integration of AI and OR techniques in CP for combinatorial optimization. Lecture notes in computer science, vol 3990. Springer, Berlin, pp 89–103
35. Gendron B, Lebbah H, Pesant G (2005) Improving the cooperation between the master problem and the subproblem in constraint programming based column generation. In: Proceedings of 2th International Conference on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems – CPAIOR'05. LNCS, vol 3524. Springer, Berlin, 217–227
36. Hansen J, Liden T (2005) Group construction for airline cabin crew: comparing constraint programming with branch and price In: Proceedings of integration of AI and OR techniques in CP for combinatorial optimization. Lecture notes in computer science, vol 3524. Springer, Berlin, pp 228–242

37. Harjunkoski I, Grossmann I (2002) Decomposition techniques for multistage scheduling problems using mixed-integer and constraint programming methods. Comput Chem Eng 26(11):1533–1552
38. Heipcke S (1999). Comparing constraint programming and mathematical programming approaches to discrete optimisation – the change problem. J Oper Res Soc 50(6):581–595
39. Heipcke S (1999) An example of integrating constraint programming and mathematical programming. In: Electron Notes Discrete Math 1
40. Hooker J, Ottosson G (2003) Logic-based Benders decomposition. Math Program 96(1):33–60
41. Hooker JN (2000) Logic-based methods for optimization: combining optimization and constraint satisfaction. Wiley, New York
42. Hooker JN (2002) Logic, optimization, and constraint programming. INFORMS J Comput 14(4):295–321
43. Hooker JN (2007). Integrated methods for optimization. Springer, New York
44. Hooker JN (2007). Planning and scheduling by logic-based benders decomposition. Oper Res 55(3):588–602
45. Hooker JN, Osorio MA (1999) Mixed logic/linear programming. Discrete Appl Math 96–97 (395–442)
46. Hooker JN, Ottosson G, Thorsteinsson ES, Kim HJ (1999) On integrating constraint propagation and linear programming for combinatorial optimization. In: Proceedings of the sixteenth national conference on artificial intelligence (AAAI-99), AAAI. The AAAI/MIT, Cambridge, pp 136–141
47. Jain V, Grossmann IE (2001) Algorithms for hybrid MILPCP models for a class of optimization problems. INFORMS J Comp 13(4):258–276
48. Junker U, Karisch SE, Kohl N, Vaaben B, Fahle T, Sellmann M (1999) A framework for constraint programming based column generation. In: Principles and practice of constraint programming, Lecture notes in computer science, pp 261–274
49. Le Pape C (1994) Implementation of resource constraints in ILOG schedule: a library for the development of constraintbased scheduling systems. Intell Sys Eng 3:55–66
50. Lübbecke ME (2005) Dual variable based fathoming in dynamic programs for column generation. Eur J Oper Res 162(1):122–125
51. Maravelias CT (2006) A decomposition framework for the scheduling of single- and multistage processes. Comput Chem Eng 30(3):407–420
52. Maravelias CT, Grossmann IE (2004). A hybrid MILPCP decomposition approach for the continuous time scheduling of multipurpose batch plants. Comput Chem Eng 28(10):1921–1949
53. Maravelias CT, Grossmann IE (2004). Using MILP and CP for the scheduling of batch chemical processes. In: Proceedings CPAIOR'04. pp 1–20
54. Marriott K, Stuckey PJ (1998). Programming with constraints. MIT, Cambridge
55. Nemhauser GL, Wolsey LA (1988). Integer and combinatorial optimization. Wiley, New York
56. OREAS GmbH (1999) ABACUS: a Branch-And-CUt system, Ver. 2.3. User's Guide and Reference Manual.
57. Ottosson G, Thorsteinsson ES, Hooker J (2002) Mixed global constraints and inference in hybrid CLP-IP solvers. Ann Math Artif Intell 34(4):271–290
58. Pantelides CC (1994). Unified frameworks for the optimal process planning and scheduling. In: Second conference on foundations of computer aided operations. Cache Publications, New York, p 253
59. Pesant G, Gendreau M, Potvin J-Y, Rousseau J-M (1998) An exact constraint logic programming algorithm for the traveling salesman problem with time windows. Transp Sci 32:12–29
60. Pesant G, Gendreau M, Potvin J-Y, Rousseau J-M (1999) On the flexibility of constraint programming models: from single to multiple time windows for the traveling salesman problem. Eur J Oper Res 117:253–263
61. Pisinger D, Sigurd M (2007) Using decomposition techniques and constraint programming for solving the two-dimensional bin-packing problem. J Comput 19(1):36–51
62. Proll L, Smith B (1998) Integer linear programming and constraint logic programming approaches to a template design problem. INFORMS J Comput 10:265–275

63. Quesada I, Grossmann IE (1992) An LP/NLP based branch-andbound algorithm for convex MINLP optimization problems. Comput Chem Eng 16:937–947
64. Raman R, Grossmann IE (1993) Symbolic integration of logic in MILP branch and bound techniques for the synthesisof process networks. Ann Oper Res 42:169–191
65. Raman R, Grossmann IE (1994) Modelling and computational techniques for logic based integer programming. Comput Chem Eng 18: 563–578
66. Rasmussen RV, Trick MA (2007) A Benders approach for the constrained minimum break problem. Eur J Oper Res 177(1):198–213
67. Régin J-C (1994). A filtering algorithm for constraints of difference in CSPs. In: Proceedings of the twelfth national conference on artificial intelligence (AAAI-94), pp 362–367
68. Rodosek R, Wallace M, Hajian M (1999) A new approach to integrating mixed integer programming and constraint logic programming. Ann Oper Res 86:63–87
69. Rousseau L-M (2004) Stabilization issues for constraint programming based column generation. In: Proceedings of integration of AI and OR techniques in CP for combinatorial optimization. Lecture notes in computer science, vol 3011. Springer, Berlin, pp 402–408
70. Rousseau L-M, Gendreau M, Pesant G, Focacci F (2004) Solving VRPTWs with constraint programming based column generation. Ann Oper Res 130(1):199–216
71. Sadykov R (2008). A branch-and-check algorithm for minimizing the weighted number of late jobs on a single machine with release dates. Eur J Oper Res 189(3):1284–1304
72. Sadykov R, Wolsey L (2006) Integer programming and constraint programming in solving a multimachine assignment scheduling problem with deadlines and release dates. INFORMS J Comput 18(2):209–217
73. Sellmann M, Zervoudakis K, Stamatopoulos P, Fahle T (2002) Crew assignment via constraint programming: integrating column generation and heuristic tree search. Ann Oper Res 115(1):207–225
74. Smith BM, Brailsford SC, Hubbard PM, Williams HP (1996) The progressive party problem: Integer linear programming and constraint programming compared. Constraints 1(1/2): 119–138
75. Timpe C (2002) Solving planning and scheduling problems with combined integer and constraint programming. OR Spectr 24(4):431–448
76. Van Hentenryck P (1999) The OPL optimization programming language. MIT, Cambridge
77. Van Hentenryck P (2002). Constraint and integer programming in OPL. INFORMS J Comput 14(4):345–372
78. Wolsey LA (1975) Faces for a linear inequality in 0-1 variables. Math Program 8(2):165–178
79. Yunes TH, Moura AV, de Souza CC (2009) Solving very large crew scheduling problems to optimality. In: Proceedings of ACM symposium on applied computing, vol 1. ACM, New York, pp 446–451
80. Yunes TH, Mour AV, de Souza CC (2005) Hybrid column generation approaches for urban transit crew management problems. Transp Sci 39(2):273–288

Hybrid Solving Techniques

Tobias Achterberg and Andrea Lodi

Abstract Hybrid methods have always been one of the most intriguing directions in these 10–15 years spent in creating and enhancing the relationship between constraint programming and operations research. Three main hybridization contexts have been explored: hybrid modeling, hybrid solving (algorithmic methods) and hybrid software tools. In this chapter we concentrate on the algorithmic side of the hybridization.

1 Introduction

Since the mid Nineties and with the creation of the CPAIOR venue, a great importance has been given to the communication between communities, namely those of constraint programming (CP) and operations research (OR), that had evolved in a rather separate way but looking at very related, almost coincident questions. In that context, the development of "hybrids" has of course been one the most intriguing issues, the rationale being that the combination of the best features of both CP and OR would have led to stronger methods.

It is hard to say if such an integration goal has been accomplished within the first 10–15 years of our surveying. For sure, there is now a much deeper understanding among the players and such an understanding gave rise to very interesting work on rather all domains in which hybrids could be devised. Specifically, three main hybridization directions have been explored: hybrid modeling, hybrid solving (algorithmic methods), and hybrid software tools.

T. Achterberg (✉)
IBM, CPLEX Optimization Schoenaicher Str. 220, 71032 Boeblingen, Germany
e-mail: achterberg@de.ibm.com

A. Lodi (✉)
DEIS, University of Bologna, Viale Risorgimento 2, 40136 Bologna, Italy
e-mail: andrea.lodi@unibo.it

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_5,
© Springer Science+Business Media, LLC 2011

In this chapter, we concentrate on the algorithmic side of the hybridization, and we survey some of the milestones in such an integration process. Of course, the chapter is strongly connected to second chapter on "Hybrid Modeling", to Chapter "Decomposition Techniques for Hybrid MILP/CP Models Applied to Scheduling and Routing Problems" on decomposition methods, and to Chapter "What is Autonomous Search?" on software tools supporting integration. In particular,

- In the second Chapter "Hybrid Modeling", Hooker discusses the idea that, in order to obtain an effective hybrid algorithm, one should start with a hybrid model which allows the solver to integrate CP and OR in a way that exploits the model structure (creating a sort of positive loop). The most relevant examples of this approach are discussed by Hooker [14].
- In Chapter "Decomposition Techniques for Hybrid MILP/CP Models Applied to Scheduling and Routing Problems", Castro and Grossmann show that a clever decomposition in the model between a part that mostly benefits from CP techniques and another one, instead of having a (mixed) integer programming, (M)IP, flavor can lead to extremely good results in several applied contexts including scheduling. Relevant examples of this approach are, e.g., Jain and Grossmann [15] and Sadykov and Wolsey [30].

The algorithmic hybridizations discussed in this chapter are somehow in between the two approaches described in Chapters "Hybrid Modeling" and "Decomposition Techniques for Hybrid MILP/CP Models Applied to Scheduling and Routing Problems". On the one hand, there is no explicit decomposition and the model remains unique but the integration is certainly looser than that described by Hooker in the sense that the modeling aspect is mostly associated with either CP or MIP. Thus, the techniques described here combine different solving technologies within a unique algorithm driven by either a CP or an MIP model depending on the focus the hybrid framework should have.

For the reasons discussed above, this chapter should be seen as complementary to Chapters "Hybrid Modeling" and "Decomposition Techniques for Hybrid MILP/CP Models Applied to Scheduling and Routing Problems". Thus, a "passionate" (of hybridization) reader most certainly needs to look at the three chapters together to get the full picture. As mentioned, Chapter "What is Autonomous Search?" completes the reading by providing a detailed overview on the software environments using or developing hybrid algorithms.

We assume throughout the chapter the basics as well as the classical terminology of both CP and OR and especially MIP. We present some less standard concepts in detail whenever needed.

The chapter is split into two parts. In the first part, we discuss algorithmic approaches in which OR (mostly IP) components are used within a CP framework to deal with those aspects in which constraint programming has been traditionally weak(er). This is discussed in Sect. 2. In the second part, we present more or less the opposite direction, i.e., a fine integration of CP concepts within an MIP framework. This is discussed in Sect. 3. Finally, some conclusions are drawn in Sect. 4.

2 Exploiting OR Techniques within CP

CP modeling is based on the so-called *global constraints*[1], i.e., higher level structured collections of simpler constraints representing fundamental structures of the overall problem. Modeling by global constraints is very flexible, compact, and shows immediately which are the main characteristics of the problem at hand. In addition, with each global constraint, there is an associated *propagation* algorithm that prunes variable-value assignments which are proven infeasible.

Propagation algorithms have been the first, *ante litteram*, form of integration of OR techniques, and in particular *graph theoretic* algorithms, in CP. A classical example is that of the most famous among the global constraints, the ALLDIFF. The ALLDIFF constraint is defined on a set of variables $[X_1, \ldots, X_n]$ ranging over finite domains $[D_1, \ldots, D_n]$. It is satisfied if and only if each variable is assigned a different value. The structure of the ALLDIFF constraint has been recognized to be the one of a *flow* in a graph and polynomial time propagation algorithms to enforce arc-consistency have been designed by Régin [27] by exploiting such a structure.

Although the roots of the propagation algorithm for the ALLDIFF are clearly in the OR context, this does not strictly qualify as a form of hybridization. Indeed, finding an effective and efficient algorithm to prune the solution space by removing values from variables' domains is intrinsically in the CP framework, actually it is what CP does.

The hybridization we will discuss in the following are instead specifically designed to take advantage of the strength of an approach to recover the weakness of another. That is sometimes achieved by exploiting an OR technique with a rather different flavor.

It is common sense that a crucial reason to use CP is the aforementioned flexibility of its modeling paradigm and, indeed, many of the hybrid approaches exploit CP on the modeling side. Of course, however, the first problem to be considered, once a CP model has been selected and a mathematical programming based algorithm has to be integrated, is the mapping of such a model (or part of it) into a traditional MIP one.

To deal with such a mapping, almost all authors have followed the seminal path disclosed by Rodošek, Wallace and Hajian [29] of associating a binary variable x_{ij} with each discrete choice j of the CP variable X_i in the (finite) domain D_i. Then,

$$x_{ij} = 1 \Leftrightarrow X_i = j \quad \text{and} \quad \sum_{j \in D_i} x_{ij} = 1. \tag{1}$$

In the rest of the section, we discuss three basic hybridization approaches (Sects. 2.1–2.3) that appeared in the literature around the same period at the end

[1] Chapter "Global Constraints: A Survey" of the book is devoted to this topic.

of the nineties and have, as a common denominator, the solution of linear representations of the CP model. In Sect. 2.4, a couple of more specialized examples of integration will be presented.

Finally, note that the role of OR and MIP within CP has been extensively discussed by Milano et al. [21] and Milano and Wallace [23].

2.1 Interaction Between CP & MIP Solvers

A first formal approach in which both a CP and a MIP solver were extensively exploited is due to Rodošek, Wallace and Hajian [29] and Rodošek and Wallace [28]. The model is expressed by using the CP paradigm and the basic scheme implies a linearization of the constraints by means of the mapping (1) at each node of the search tree. The main idea is indeed that performing both a "local" constraint propagation by CP and a "global" propagation by solving the continuous relaxation of the problem, one can obtain a stronger reduction of the search space. Specifically,

- *Local.* CP is in charge of the search decisions, i.e., to construct a feasible solution and to perform local propagation based on global constraints.
- *Global.* The associated continuous relaxation is obtained by dropping the integrality requirement on the binary variables defined by the mapping (1) and solved by the simplex algorithm (more generally, a general-purpose *linear programming*, LP, solver). This is used as a global propagation to detect the infeasibility of a node, i.e., a partial solution which cannot be completed, so as to stop the search at high(er) tree levels.

The rationale behind such a hybridization mechanism is that the two propagations are inherently different as shown by a slight adaptation of an example given in [28]. From the one side, given the following simple constraint on the domain variables X and Y

$$X : 0..3,\ Y : 0..1,\ X + 2Y = 3, \tag{2}$$

a (local) constraint propagation immediately detects that values 0 and 2 can be removed from the domain of variable X because they do not give rise to feasible complete assignments. The continuous relaxation of the associated MIP can also exclude 0 from the domain of X, it cannot exclude 2.

On the other hand, constraint propagation is unable to detect any inconsistency in the constraints

$$X : 1..10, Y : 1..10, Z : 0..1, 2X + 2Y + Z \geq 23, X + Y \leq 11, X + Y + 10Z \leq 12, \tag{3}$$

while the continuous relaxation of the associated MIP is infeasible.

It is easy to see that in the approach described above the main focus remains on the infeasibility detection as in the classical CP tradition. In other words, if there exists a *penalty* function ranking feasible solutions, the associated optimization problem is reduced to a sequence of feasibility problems where one requires to

improve the current incumbent solution. More precisely, if the penalty function to be minimized is written as $\sum_{i=1}^{n} C_i X_i$ and the current incumbent solution has value *best*, then the problem formulation is enhanced through the constraint

$$\sum_{i=1}^{n} C_i X_i \leq best - \varepsilon, \tag{4}$$

where ε is a small positive value[2].

Unfortunately, the continuous relaxation of constraint (4) (after mapping) usually does not (globally) propagate very much and this results in a rather weak exploitation of the penalty function to abort the search at high levels of the tree.

2.2 Linearizing Global Constraints

In the previous section, we have already presented the mapping which leads to the linearization of the global constraints of the CP model in order to obtain the associated MIP representation. However, the linearization problem in its full generality has been considered by Refalo [26] whose contribution is discussed in detail in this section.

From the one side, Refalo [26] noted that solving the continuous relaxation provides a lower bound (by again considering minimization problems) that can be used to prune the search space (in a weak way, see previous section) but also a relaxed solution which, in turn, can be very useful to guide the search toward optimal solutions. This idea has been used for example by Milano and van Hoeve [22] to produce a ranking of the variables in order to first explore promising parts of the search space and quickly improve the incumbent solution[3].

On the other hand, Refalo [26] noted that an MIP representation of a global constraint is not restricted to the linear constraints that are required to make such a linearization valid. In other words, the idea is to generate *cutting planes* with the aim of improving the linear representation/relaxation of a global constraint on the fly, i.e., when the current relaxed solution violates them. This approach, which is standard in MIP *branch-and-cut* algorithms, has the advantage of improving the current representation only when needed while keeping the relaxation under control in terms of size. From a CP perspective, the integration of a cutting plane generation phase is very clean in the sense that the linear representation is used not only to detect the infeasibility of a partial solution (a node in the tree) but also to propagate new constraints. Focacci, Lodi and Milano [9] discussed several ways of using

[2] One is allowed to use $\varepsilon = 1$ if all variables can only assume integer values and the C coefficients are integer as well.

[3] It must be noted that in [22] the authors solve the continuous relaxation of one of the global constraints by a combinatorial algorithm and not by a general-purpose LP solver.

cutting planes in CP global constraints ranging from a straightforward integration to more sophisticated ones involving the use of a Lagrangean relaxation of the cuts. (The use of Lagrangean relaxation in conjunction with cuts will be discussed in more detail in Sect. 2.4.)

Before ending the section, we would like to emphasize that, building upon the work in [28, 29], the contribution of Refalo's work [26] has been giving a detailed and systematic way of producing and updating linear representations of global constraints and extending the use of the information associated with them.

It is also worth noting that the goal of collecting useful pieces of information from linear representations does not necessarily require the reformulation of all constraints to produce a complete MIP. Indeed, as shown for example in [22], even a *local* reformulation of single global constraints can be very effective both for propagation and to guide the search. In addition, the special structure generally associated with global constraints allows *polyhedral*[4] cutting planes to be generated.

2.3 Cost-Based Domain Filtering

Focacci, Lodi and Milano [8, 11] looked at the linearization of a global constraint from a slightly different perspective. Again, the idea relies on having a linear relaxation of the global constraint itself. However, the propagation due to the objective function is used not only in the weak (and already too expensive) traditional way to remove partial solutions, but also to filter variable-value assignments through the use of *reduced costs*. In linear programming theory, the reduced cost of a variable x_{ij} is the change in the current value of the objective function due to an increase of 1 unit in the value of x_{ij}. In other words, the reduced costs are a *gradient function* measuring the variable-value assignment cost and, because of this, they are particularly suitable for traditional CP domain propagation. More precisely, let *best* be the value of the incumbent solution, and let LB be the optimal value of the linear relaxation of the global constraint at hand (at the current node) representing a lower bound on the overall objective function. Moreover, let $\bar{c}_{ij}$ be the reduced cost of the binary variable x_{ij}. Then,

$$LB + \bar{c}_{ij} \geq best \Rightarrow X_i \neq j, \tag{5}$$

i.e., value j can be eliminated from the domain of variable X_i in case the reduced cost of the associated binary variable x_{ij} would make the current lower bound value greater than or equal to the incumbent value.

[4] A cutting plane is often referred to as polyhedral when its separation is based on the knowledge of the structure of the underlying polyhedron. That is in opposition to the so-called general-purpose cutting planes, like Chvátal–Gomory cuts, whose derivation only relies on algebraic properties like integrality.

This technique has been well-known[5] in MIP for decades but, in general, it is extensively applied only at the root node of the search tree with the rationale that variable fixing in the internal nodes is "implicitly" achieved[6] through more sophisticated techniques. In the CP context, instead, the technique called *Cost-based Filtering* has a partially different flavor. First, CP relies on filtering variable-value assignments as its main tool. Second, as already mentioned, traditionally the impact of the objective function is very limited. Third, the reduced cost computation is somehow *incremental*, in the sense that a re-optimization of the linear relaxation generally does not require re-calculation of the reduced costs from scratch.

The third aspect of the method being incremental has been particularly stressed in [8] where the authors used the *path* constraint to model the famous *Traveling Salesman Problem* (TSP) where a traveling salesman is looking for the shortest tour visiting n cities exactly once. In this case, the linear relaxation considered was the *Assignment Problem* which is (a) solvable in $O(n^3)$ time through a combinatorial algorithm, (b) integer valued, and (c) fully incremental in the sense that any re-optimization after a branching decision has been taken requires only $O(n^2)$ time. In the most general case, i.e., when the linear relaxation is solved by a general-purpose LP solver, such a very clean incremental re-computation is not possible but if the dual simplex algorithm is used, the number of iterations (so-called *pivots*) is small in practice.

Finally, we like to stress that the Cost-based Filtering technique is fully general, the only requirements being (a) the availability of a gradient function like the reduced costs and (b) the fact that solving the relaxation is indeed giving a valid lower bound on the overall objective function.

2.4 Specialized OR Methods within Hybrid Algorithms

In the Sects. 2.1–2.3, we have surveyed three fully general techniques that integrate OR elements within a CP framework. The common denominator of those techniques is the solution of a linear representation of either the entire or part of the problem modeled within a CP paradigm. However, the way in which the information provided by the solution of these associated linear programs is algorithmically exploited differs in the described frameworks.

In the following, we describe two specialized OR methods which have been used within hybrid algorithms of the type described above.

Lagrangean Relaxation of Cuts. As already mentioned in Sect. 2.2, there are several ways of using cutting plane generation within CP once a linear representation of a

[5] It is often referred to as *reduced cost fixing*.

[6] An implicit fixing is achieved because no branching will be performed on the binary variable associated with the variable-value assignment because its value will be moved toward integrality, e.g., by tightening the relaxation through cutting planes.

global constraint has been obtained. Of course, the cuts can be generated, as often done by MIP solvers, at the root node of the search tree so as to tighten the initial relaxation or can be separated at any node of the tree. However, in the CP context in which the number of explored search nodes is in general very high, the addition of too many cuts can result in a slow algorithm in which the computational effort associated with any node is too high, mainly because of the size of the resulting LP[7].

One idea introduced by Focacci, Lodi and Milano [9] is the one of relaxing the cuts (at given reference points) in a Lagrangean fashion, i.e., by dualizing them in the objective function through appropriate penalties. More precisely, if a valid cutting plane, say $\alpha^T x \leq \alpha_0$ has been generated, its Lagrangean relaxation (or dualization) is obtained by putting it into the objective function as

$$\min \sum_{i=1}^{n} \sum_{j \in D_i} c_{ij} x_{ij} + \lambda(\alpha^T x - \alpha_0), \tag{6}$$

where $\sum_{i=1}^{n} \sum_{j \in D_i} c_{ij} x_{ij}$ is the linearization of the objective function (4) and $\lambda \geq 0$ is the penalty term. In other words, if the cut is violated by a given solution x^*, then $\alpha^T x^* - \alpha_0 > 0$ and it gives a positive contribution to the objective function, i.e., penalizes it proportionally to λ.

This idea has been effectively used by Focacci, Lodi and Milano [10] for solving the time constrained variant of the TSP in which each city must be visited within a specific *time window*. In particular, the cuts separated at the root node can be dualized into the objective function by selecting optimal penalty values λ's, one per each cut and precisely the dual value associated with that cut in the optimal LP solution[8].

The effect of such a relaxation is to keep the size of the linear representation "small" and at the same time being able to deal with a much tighter relaxation of the *path* constraint which has been very useful in [10] to solve large TSP with time windows instances. (Some additional tricks have been used to make the relaxation effective such as removing, *purging*, the cuts when they are no more useful, see [11] for details.)

Finally, note that once a group of constraints (cuts in this case) has been relaxed in a Lagrangean fashion, the remaining linear program sometimes shows a structure that can be exploited by solving it with a combinatorial algorithm, which is generally more efficient than a general-purpose technique. This is true in [11] where, once the cuts have been dualized, the remaining relaxation is the Assignment Problem (see Sect. 2.3 above).

Additive Bounding. The additive bounding procedure has been proposed by Fischetti and Toth [7] as an effective technique for computing bounds for combinatorial optimization problems. Intuitively, the approach consists of solving a sequence of relaxations of a given problem, each producing an improved bound.

[7] Note that in MIP such a problem exists as well but it is less crucial since in the branch-and-cut algorithm branching is not conceived as the primary tool like it is instead in CP.

[8] The interested reader is referred to Nemhauser and Wolsey [25] for a detailed treatment of Lagrangean relaxation.

More precisely, we suppose we have a set of d bounding procedures $\mathscr{B}_1, \ldots, \mathscr{B}_d$ for our problem, which is expressed in minimization form and with a cost vector c. In addition, we assume that every bounding procedure $\mathscr{B}_h$ returns a lower bound value LB_h and a reduced cost vector $\bar{c}^h$. It is not difficult to prove that in case the bounding procedures $\mathscr{B}_1, \ldots, \mathscr{B}_d$ are applied in sequence, and procedure $\mathscr{B}_h$ receives on input instead of the original cost vector the reduced cost vector returned by $\mathscr{B}_{h-1}$, i.e., $\bar{c}^{h-1}$, then

$$LB = \sum_{h=1}^{d} LB_h \tag{7}$$

is a valid lower bound for the overall problem.

The additive bounding approach has been disregarded in the last 15 years by the OR community due to the increasing availability of reliable and fast LP solvers which allow to solve the full LP relaxation in "one shot." However, Lodi, Milano and Rousseau [17] revisited it in the CP context by showing its interest in particular in conjunction with the enumeration strategy called *Limited Discrepancy Search* (LDS, see Harvey and Ginsberg [13]). Again, one of the reasons why the additive bounding idea is particularly interesting for CP is that enumeration is a strong ingredient of CP, thus getting a cheap improvement in the bound without the price of solving large LPs is appealing.

LDS is one of the most well-known search strategies used in CP. A discrepancy is a branching decision which does not follow the suggestion of the heuristic[9] and the basic idea of LDS is that the search space is explored by fixing the amount of discrepancies one is allowed to accept. In other words, the solution space is split into slices with fixed discrepancy k and it is explored for increasing values of k itself. In particular, the variant of LDS used in [17] is called *Decomposition Based Search* (DBS, see Milano and van Hoeve [22]) and splits the domain of each CP variable X_i into a *bad* set B_i and a *good* one G_i. Thus, a discrepancy corresponds to assign a value $j \in B_i$ to variable X_i.

It is not hard to see that the condition that fixes the cardinality of the discrepancy set to k can be easily expressed as the linear constraint

$$\sum_{i=1}^{n} \sum_{j \in B_i} x_{ij} = k, \tag{8}$$

where of course $\sum_{j \in B_i} x_{ij} \leq 1$ must hold as well.

The main observation in [17] is that imposing constraint (8) during search enables the definition and solution of an additional relaxation which can be used in conjunction with any combinatorial relaxation associated with a global constraint in additive fashion. Specifically, Lodi, Milano and Rousseau [17] use the Assignment Problem relaxation of the ALLDIFF constraint as a primary bounding procedure and

[9] Recall that in CP the rule that selects the next variable-value assignment to be instantiated is called *heuristic*.

the resulting reduced cost vector is used to "feed" the simple relaxation defined by the discrepancy constraint (8). Such a bound improvement has been proven very useful to solve difficult instances of asymmetric TSP and Resource Constrained Assignment Problems (see, [17] for details).

More sophisticated additive algorithms are also described in [17] for the special case in which the ALLDIFF constraint is used. However, the additive bounding technique is a general bounding approach having the two major characteristics of (a) establishing a non-trivial link between search and bound and (b) providing a sometimes non-negligible improvement without the cost of solving large LPs.

3 Exploiting CP Techniques within MIP

As mentioned in Sect. 2, it is common to use an LP relaxation inside a CP solver as an additional propagation engine and to direct the search. From a high-level perspective, a typical branch-and-cut MIP solver is just a special case of such an LP based CP solver, namely one that supports only linear constraints and integrality conditions. Just like a CP solver, an MIP solver traverses a search tree that represents a recursively defined decomposition of the search space, and it performs domain propagation at the nodes to reduce the size of the tree. In particular, it solves the LP relaxations to provide bounds on the objective function.

From a practical point of view, however, state-of-the-art MIP and CP solvers are very different regarding the concepts that are emphasized in the implementations. The most important aspect of a CP solver is its propagation engine. Each individual propagation algorithm has to be specifically tailored toward the class of constraints that is addressed by the propagator, and the CP framework has to support the efficient interaction between multiple propagation algorithms. The communication between the different constraint types is mainly done via the domains of the variables, and this communication must be extremely fast in order to obtain a high throughput of search tree nodes. An example of a specifically tailored propagation algorithm can be found in SAT solvers, which can also be seen as a very special case of CP solvers: Moskewicz et al. [24] discovered that to propagate SAT clauses it suffices to track the bounds of only two of the involved variables. This so-called *two watched literals scheme* turned out to be a major advancement in SAT solving technology as it greatly improves the performance of the SAT propagation engine.

Although an MIP solver also employs domain propagation on the linear constraints, the central object in the solving process clearly is the LP relaxation, and the search process is primarily guided by the LP solutions and the dual bounds that are provided by the LP. A lot of tricks are employed to speed up the LP solves, for example, by recording warm start bases to be able to resolve LP relaxations of child nodes in few iterations and by storing dual norms to obtain faster steepest edge pricing in the dual simplex solver. Typical MIP branching rules [3] are based on primal and dual information from the LP relaxation and try to increase the dual objective

bound that is provided by the LP. Many of the primal heuristics [4] to find feasible solutions take the LP solution as a starting point and try to turn it into an integer feasible solution.

It is common knowledge in the MIP community that an MIP model can usually be solved better if its LP relaxation is close to the convex hull of integer feasible points. Therefore, one of the central topics in MIP research is to derive cutting planes to tighten the LP relaxation, see, for example, [16] for a survey and [32] for implementation details. Efficient MIP solvers need very efficient implementations of several cutting plane procedures and good heuristic procedures to control the interaction among them. The same holds true for primal heuristics and branching rules. In contrast, efficient domain propagation is not that crucial for an MIP solver, as most of its run time is consumed in the LP solves.

CP has very rich and expressive modeling capabilities. In fact, one can view a CP solver as a relatively simple framework that is extended by a whole library of constraint specific algorithms. The expressiveness of the modeling language that a CP solver supports is directly connected to the domain propagation algorithms that the solver implements. On the other hand, an MIP solver is limited to linear constraints and integrality. This limitation is exploited in today's solvers to achieve a very high performance for this special case.

The question arises as to how one should combine CP and MIP solving technologies. The answer to this question, in our view, strongly depends on the focus that the hybrid technology should have. In theory, there is no difference; in both cases, one will end up with a CP solver that solves LP relaxations. In practice, the tradeoff to be made is between expressiveness and performance.

In this section, we focus on a performance oriented approach, represented by the CP/MIP framework SCIP [1,2], which tries to extend a tightly integrated MIP solver in order to incorporate some of the expressiveness of CP into the final hybrid system. Note that the solver performance comes with the downside of more complicated interfaces between the different solver components, as the amount of information they have to share is relatively large.

3.1 Design Concepts

Our approach for building a hybrid CP/MIP solver is based on a core framework that provides the necessary infrastructure and plugins which implement the semantics of the constraints and provide additional functionality to speed up the solving process. The core algorithms of CP and MIP solvers that have to be efficiently supported by the framework are the following:

Presolving. The given problem instance is transformed into an equivalent (usually smaller) problem instance. The transformation has to be stored such that one can easily *crush* a vector of the original space into a vector of the transformed space, and *uncrush* a transformed vector into a corresponding original vector.

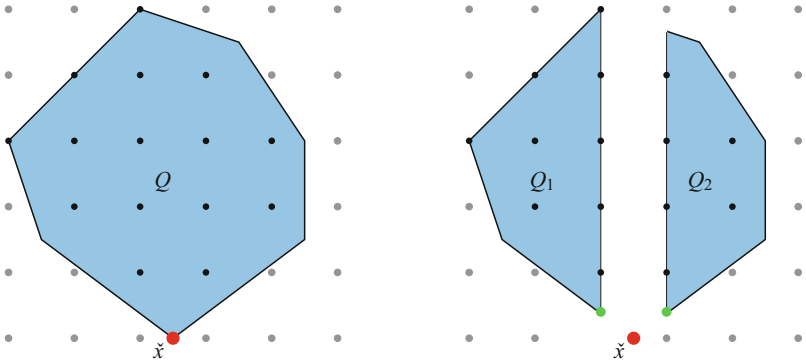


Fig. 1 LP based branching on a single fractional variable

Branch-and-bound. The problem instance is successively divided into subproblems, usually by splitting the domain of a variable into two disjoint parts (branching), see Fig. 1. The dissection of a subproblem ends if it is infeasible, an optimal solution for the subproblem can be identified, or if it can be proven that no better solution than the currently best known one can be contained in the subproblem (bounding).

Relaxation. A relaxation of the problem at hand is obtained by dropping some of the constraints. The relaxation should be chosen such that it can be solved efficiently. Its solution provides a dual bound on the objective value which can be used for the bounding step in branch-and-bound. Moreover, the primal solution to the relaxation can be used to guide the branch-and-bound search, and implications found for the relaxation are valid for the original problem as well.

Typically, MIP solvers use the LP relaxation, which is obtained by dropping the integrality restrictions of the variables. One example of implications derived from the LP are reduced cost fixings [25], see Sect. 2.3. In a hybrid system, the LP relaxation arises as the intersection of linear relaxations of the individual constraints in the problem instance. In SCIP, it is also possible to use alternative relaxations such as NLP or SDP, but the support for the LP relaxation is far more elaborate and efficient.

Cutting plane separation. After having solved the LP relaxation of a subproblem, it is possible to exploit the integrality restrictions in order to tighten the relaxation and thereby improve the bound obtained. This is achieved by adding linear inequalities that are valid for all integer feasible solutions of the problem but violated by the current LP solution, see Fig. 2. This approach of tightening a *relaxation* of a general problem, ideally until the convex hull of all feasible solutions has been obtained, can be seen as an extension to the approach of Crowder, Johnson, and Padberg [5].

Domain propagation. After having tightened a variable's domain in the branching step of branch-and-bound, domain propagation infers additional domain reductions

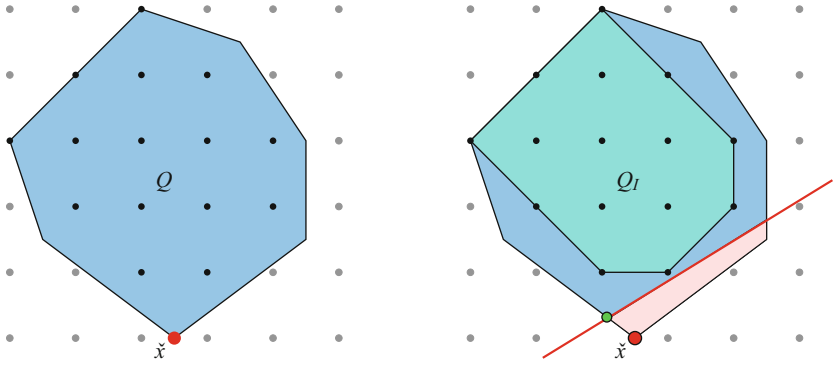


Fig. 2 A cutting plane separates the LP solution $\check{x}$ from the convex hull Q_I of integer points of Q

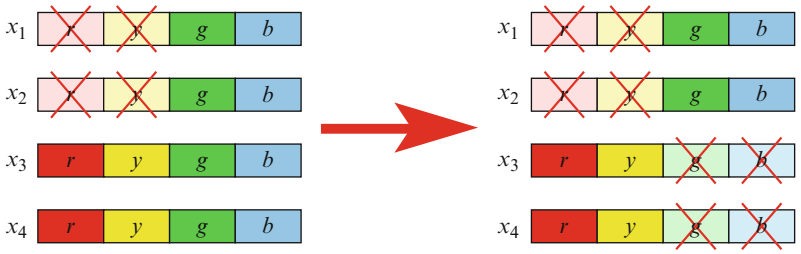


Fig. 3 Domain propagation on an ALLDIFF constraint. In the current subproblem on the left hand side, the values *red* and *yellow* are not available for variables x_1 and x_2 (for example, due to branching). The propagation algorithm detects that the values *green* and *blue* can be ruled out for the variables x_3 and x_4

on the variables by inspecting the individual constraints and the current domains of the involved variables. Figure 3 illustrates domain propagation on the example of the ALLDIFF constraint, which imposes that all variables of the constraint take pairwise different values.

One can view domain propagation as finding implications for another relaxation of the problem, namely the relaxation that arises by only keeping one constraint and the integrality conditions.

Conflict analysis. Infeasible subproblems are analyzed to identify the reason of the infeasibility and to extract additional valid constraints. These can be applied later during the search to prune the search tree and to improve the subproblems' relaxations.

The most important plugins in SCIP are the *constraint handlers*. They define the semantics of the constraint types that are supported by the solver. This concept is similar to typical constraint programming libraries, in which each class of constraints is represented by a set of constraint specific algorithms. These algorithms are accessed through an abstract interface.

In terms of the interface, the main difference between different CP frameworks is the amount of information a constraint handler has to provide. In this regard, there is a tradeoff between ease of implementation and performance of the resulting solver. The more information a constraint handler provides about the structure of the constraints, the more conclusions the other components of the solver can derive. Consequently, the search space will usually be traversed in a more efficient way and a smaller number of branch-and-bound nodes are needed to solve the problem.

In the extreme case, a constraint handler is just an oracle which inspects a given solution vector and attributes it to be feasible for its constraints or infeasible. Without providing further information to the framework, this leads to an almost complete enumeration of all vectors of the search space, because apart from bounding, no parts of the search space can be pruned. The other extreme case can be found in an MIP solver: here, all components of the solver have perfect information about the constraint structure and can exploit this knowledge at every place in the algorithm.

SCIP tries to exploit as much structural information as possible within a reasonable application programming interface (API). Of course, this comes at the cost of a more complex constraint handler interface and a pretty high learning investment for new users. On the other hand, this interface leads to high performing code, as can be seen in Hans Mittelmann's MIP solver comparison in Fig. 4. In the following, we will discuss some of the design decisions in SCIP and how they impact the ability to exploit certain structure in the solver. Additionally, we highlight the infrastructure that SCIP provides to support an efficient implementation of plugin algorithms such as cutting plane separation or domain propagation, and we provide a brief overview of the main steps in the branch-and-bound solution process as implemented in SCIP.

3.2 Interfacing to the Framework

The plugin concept of SCIP facilitates the implementation of self-contained solver components that can be employed by a user to solve his particular hybrid model. For example, if some user implemented a constraint handler for a certain class of constraints, he could make this plugin publicly available in order to enable all other

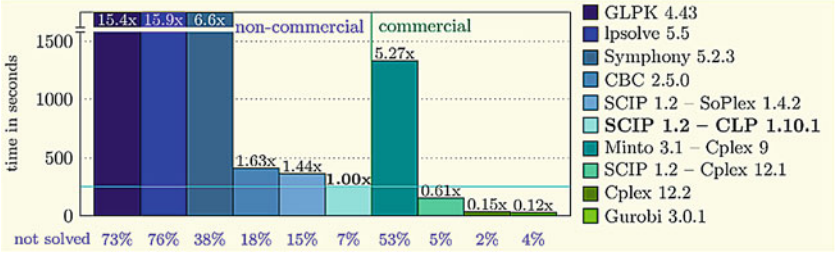


Fig. 4 Geometric mean of results taken from the homepage of Hans Mittelmann (July 29, 2010). Unsolved or failed instances are accounted for with the time limit of 2 h. Figure from scip.zib.de

SCIP users to use such constraints in their models. Since all MIP specific components that come with the SCIP distribution are implemented as plugins, more general applications can immediately benefit from these components.

Such a modular code design has the consequence that all communication between the plugins has to pass through a unified interface defined by the framework. In SCIP, this communication interface is mainly based on the MIP relaxation of the problem. The actual semantics of the various (potentially non-linear) constraints in the model is hidden from all plugins except the responsible constraint handler.

An important aspect of this information hiding is the loss of the dual view, i.e., the column-based representation of the problem. By looking at the columns of the constraint system $Ax \leq b$, a typical MIP solver knows exactly how the feasibility of the constraints is affected if the value of a particular variable changes. Such data are, for instance, used for dual presolving reductions such as the identification of parallel and dominated columns and for symmetry detection.

Therefore, SCIP and other CP solvers with an open constraint interface suffer from some handicaps when compared to specialized MIP solvers. To attenuate this drawback, SCIP demands from the constraint handlers to provide a limited amount of dual information to the framework, namely the number of constraints that may block the increase or decrease of each variable. This information suffices to enable the most important dual presolving operations such as dual fixing and dual bound reduction and helps guide primal heuristics and branching, see Achterberg [1]. Nevertheless, it is not enough to fully support dual algorithms that–like parallel column identification and symmetry detection–need to know precisely the coefficients in the constraints.

The requirement to provide this (limited) dual information has the obvious drawback that the development of a constraint handler is slightly more complicated. One cannot restrict the focus on the primal view of the problem, which is typically easier to understand. Moreover, forgetting to provide the dual information essentially means to remove a dual constraint from the problem and thus may lead to invalid presolve reductions or dual domain propagations that cut off the optimal solution.

The loss of dual information arises from decomposing the problem formulation into individual constraints. An additional remedy to this issue is provided by the constraint handler approach itself. Other branch-and-cut frameworks such as ABACUS [31] treat each individual constraint as an isolated object. In contrast, a constraint handler in SCIP, which manages *all* constraints of a certain type, can still perform operations on multiple constraints. In particular, if the limited dual information reveal that some variables are contained only in a single class of constraints, then the corresponding constraint handler can apply dual presolving methods on these variables. Some dual methods are already incorporated into the linear constraint handler of SCIP 1.2. Others, such as symmetry detection and parallel column aggregation, are not yet available.

On the primal side, the plugin concept with its information hiding has also a disadvantage compared to MIP solvers, namely, that it is impossible to consider multiple constraints at the same time to derive conclusions such as presolving reductions, domain propagations, or cutting planes. Again, this drawback is mitigated by the

constraint handler approach in which each constraint handler can look at all constraints of the corresponding type at once. For example, the linear constraint handler exploits this possibility so as to detect redundant constraints or to add equations to other linear constraints in order to reduce the number of non-zero coefficients in the constraint matrix. Another tool to partly address this issue is given by some of the infrastructure components in SCIP, namely the implication graph, the clique table, and the MIP relaxation (that is the LP relaxation and the integrality information). These data structures capture global relationships between the variables that can be exploited in the various plugins.

Finally, a more subtle drawback of the plugin approach of SCIP is its memory footprint and poor locality. An MIP solver can store the whole problem matrix A in a single memory block. In contrast, the problem data in SCIP are stored locally in the constraint handlers, which means that the data are distributed across the memory address range. This usually leads to a degradation in the cache usage of the CPU and consequently to a performance loss. Additionally, since SCIP manages an MIP relaxation of the problem and additionally employs a black-box LP solver as subalgorithm to manage the LP relaxation, most of the problem data are copied several times. This yields a significant increase in the memory consumption. Moreover, treating the LP solver as a black box and passing information through an interface layer abandons the opportunity for runtime improvements, namely to take certain shortcuts and to exploit integrality during the LP solves.

3.3 Infrastructure Provided by the Framework

As indicated in the previous section, the abstraction of constraints and the information hiding that comes with it limit the amount of global problem structure that the solver can exploit. One task of the framework of a hybrid solver is to provide infrastructure to facilitate the communication between the plugins, in particular the different constraint handlers. SCIP offers a variety of such infrastructure components, each of which can be seen as a relaxation of the problem:

Dual information. As described above, constraint handlers have to provide a limited amount of dual information about their constraints, namely the number of constraints that prevent each variable to be increased or decreased arbitrarily. These *variable locks* basically say how many constraints can potentially be violated by shifting a variable upward or downward. They are exploited for dual presolving and dual domain propagation as well as in primal heuristics and branching rules.

Implication graph. The implication graph stores logical implications on binary variables $x \in \{0, 1\}$ of the form

$$x = 0 \to y \leq b, \quad x = 0 \to y \geq b, \quad x = 1 \to y \leq b, \quad \text{or} \quad x = 1 \to y \geq b. \tag{9}$$

Such implications are, for example, used by the complemented mixed integer rounding cut separator [18, 19] to derive stronger cutting planes.

Clique table. The clique table $\mathcal{Q}$ stores relations between binary variables $x_j \in \{0, 1\}$, $j \in Q \in \mathcal{Q}$, namely, set packing constraints

$$\sum_{j \in Q} x_j \leq 1.$$

These cliques may contain both the positive literal x_j and the negative literal $\bar{x}_j$ of a binary variable. Thereby, the clique table is a compact representation of implications (9) between binary variables.

Variable aggregation graph. The variable aggregation graph stores equality relations between variables of the form

$$y_i = \sum_{j=1}^{n} a_{ij} x_j + b_i.$$

These are found during the presolving phase of the algorithm and used to remove variables y_i from the problem instance by replacing their occurrences with the corresponding affine linear sum of variables x_j. Constraint handlers can check the variable aggregation graph and optionally perform the corresponding substitution in their own constraint data structures. This could lead to further presolving reductions and problem simplifications. Sometimes, however, performing such a substitution is not possible because it would violate the structure of the constraint. For example, in the ALLDIFF(y_1, y_2, y_3) constraint, we cannot substitute, say, $y_1 := 2x + 3$ because the ALLDIFF(y_1, y_2, y_3) cannot be translated into an equivalent ALLDIFF constraint on x, y_2, and y_3. Instead, the original variables are kept in the constraint and the variable aggregation graph is used as an efficient and transparent way to automatically perform the necessary *crushing* and *uncrushing* operations to map original variables to active variables of the presolved model.

MIP relaxation. The most important infrastructure component to facilitate communication between plugins is the MIP relaxation of the problem, which consists of a linearization of the constraints plus the bounds and integrality information for the variables. The MIP relaxation is populated by the constraint handlers and the cutting plane separators. It is automatically updated whenever the search process moves to a different subproblem in the search tree.

The MIP relaxation is used extensively during the search process: Primal heuristics search for feasible solutions to the MIP relaxation and hope that these are also feasible for the full model; Cutting plane separators add violated inequalities to tighten the LP relaxation by exploiting the integrality information; Reduced cost fixing exploits dual LP information and the integrality restrictions; Branching rules to split the problem into subproblems are guided by the primal LP solution and the dual LP bound; The dual LP bound is used in the bounding step of branch-and-bound; Conflict analysis converts the dual ray of an infeasible LP relaxation into a globally valid conflict constraint that can help prune the search tree later on.

All of these components are independent of the abstract constraints in the model and only use the information provided by the MIP relaxation. Therefore, if the model to be solved has a tight MIP relaxation, it is likely that the solving process can greatly benefit from these MIP-specific SCIP plugins.

In addition to the "communication infrastructure," a hybrid framework like SCIP has to provide a "bookkeeping infrastructure" that helps ease the implementation of constraint handlers and improve the performance of the code. In this regard, SCIP offers the following main concepts:

Search tree. The search tree can be easily defined in the branching rules by methods to create children of the currently active search node. The node selection rule defines the ordering of the open nodes in the node priority queue. Individual nodes and whole sub-trees can be marked to be pruned or reevaluated by domain propagation, which is exploited in the so-called *non-chronological backtracking* [20] that SCIP applies automatically after having derived conflict constraints.

Event mechanism. An efficient domain propagation needs to process the constraints and variables in a sparse fashion: only those constraints need to be evaluated again for which the data of at least one of the involved variables have been modified since the last inspection of the constraint. This process is supported by the event mechanism in SCIP: a constraint can "watch" the status of variables and be informed whenever the status (for example, the bounds of the variable) changes. The list of constraints that watch the status of a variable has to be dynamically adjustable to facilitate efficient propagation schemes such as the two-watched literals scheme [24]. Using this scheme, each set covering or bound disjunction constraint only needs to watch two of the involved variables, but this pair of "watched literals" has to be changed over time.

Besides watching the status of variables, events can also be used to be informed about new feasible solutions that have been discovered, about the switching of subproblems in the search tree, or about the solving of an LP relaxation.

Solution pool. The solution pool of SCIP collects all feasible solutions that have been discovered during the search process. From a practical perspective, these suboptimal solutions can be useful since often the model does not capture all aspects of the real-world problem and therefore a sub-optimal solution might be useful for the decision makers. From a solver perspective, the solutions in the solution pool can be used, for example, to guide primal heuristics like crossover [4] or RINS [6].

Cut pool. The cut pool stores cutting planes that have been separated by constraint handlers or cutting plane separators. It is used to collect cuts that are non-trivial to derive and for which it is therefore more efficient to keep them in the pool instead of having to find them again by a separation algorithm. Often, not all available cuts are simultaneously present in the LP relaxation. The pool provides an efficient way to add these cuts dynamically to the LP if they are violated by the current LP solution.

Separation storage and cut filtering. During each round of cutting plane separation, the cutting planes produced by the separation algorithms are collected in the *separation storage*. At the end of each round, the set of cutting planes is automatically filtered in order to reduce the number of cuts that enter the LP relaxation [1]. The goal of the cut filtering is to obtain an almost orthogonal set of cuts, each of them having a large Euclidean violation with respect to the current LP solution.

Pricing storage. The pricing storage is the analogous to the separation storage for new columns found in a branch-and-price algorithm. It collects new columns produced by pricing algorithms before they enter the LP relaxation.

3.4 Solution Process

We conclude the section by giving a brief overview of how SCIP solves a given constraint programming instance and which role the user plugins play in this process.

The solution process of SCIP, as sketched in Fig. 5, is controlled by the framework. The user methods, in particular the constraint specific algorithms, are incorporated into the control flow by means of plugins. At each step in the solution process, the callback methods of the applicable plugins are executed to perform the required tasks. The plugins interact with the infrastructure of SCIP to return the requested information to the main solution algorithm of the framework.

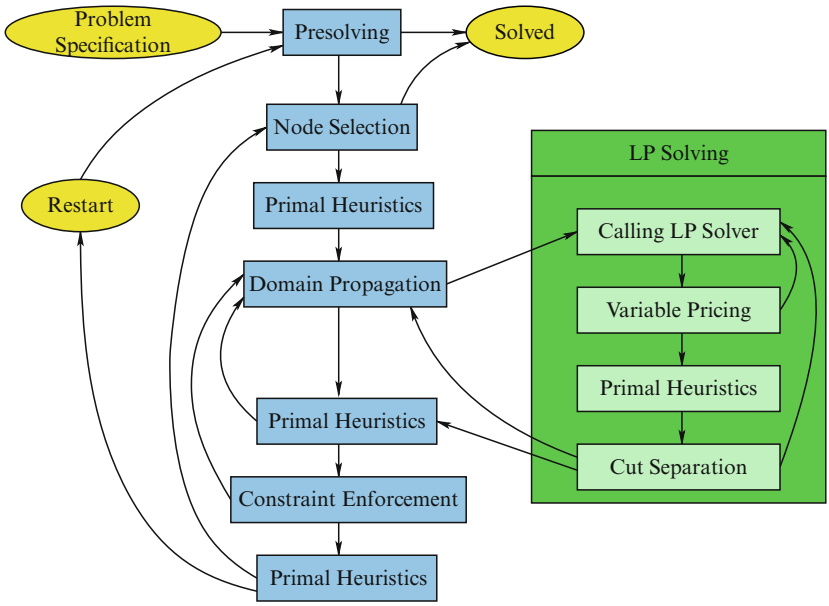


Fig. 5 Flow chart of the solution process in SCIP

After the user has specified his CP problem instance in terms of variables and constraints, the presolver plugins and the presolving methods of the constraint handlers are called to simplify the problem instance. Then, the actual search commences by creating the root node of the search tree and by selecting the root as the first node to be processed. Primal heuristics are called at various places in the solving loop, and each heuristic plugin can specify when it should be called. For example, heuristics like farthest insert for the TSP that do not depend on an LP solution can be called before the LP relaxation is solved. Very fast heuristics such as rounding are best to be called inside the cutting plane loop, whereas more time consuming heuristics such as diving should only be called once at the end of the node processing cycle.

After a node has been selected and applicable heuristics have been called, domain propagation is applied by calling the domain propagation plugins and the domain propagation methods of the constraint handlers. If specified in the parameter settings, the next step is to solve one or more relaxations of the problem, with the LP relaxation being directly supported by the framework.

The LP solving loop consists of an inner pricing loop in which the pricer plugins produce additional variables, and an outer cutting plane loop in which the cut separators and the cut callbacks of the constraint handlers add cutting planes to the LP relaxation. Cutting plane separators, in particular reduced cost fixing, can tighten the bounds of the variables, which triggers another call to the domain propagators in order to infer further domain reductions.

Eventually, no more improvements of the relaxation can be found and the constraint enforcement is executed. If the relaxation of the current node becomes infeasible during the process, the node can be pruned and another node is selected from the search tree for processing. Otherwise, the constraint handlers have to check the final solution of the relaxation for feasibility. If the solution is feasible for all constraints, a new incumbent has been found and the node can be pruned. Otherwise, the constraint handlers have the options to add further cutting planes or domain reductions, or to conduct a branching. In particular, if there are integer variables with fractional LP value, the integrality constraint handler calls the branching rule plugins to split the problem into subproblems. Finally, a new unprocessed node is selected from the search tree by the current node selector plugin and the process is iterated. If no unprocessed node is left, the algorithm terminates.

After processing a node, there is also the option to trigger a restart, which is an idea originating from the SAT community (see, e.g., [12]). Restarting means to exploit collected knowledge such as incumbents, cutting planes, and variable fixings in a subsequent repeated presolving step and to restart the tree search from scratch. Our experiments with SCIP indicate that in the MIP context, restarts should only be applied at the root node, and only if a certain fraction of the variables have been fixed while processing the root node. This is in contrast to SAT solvers, which usually perform periodic restarts throughout the whole solution process.

4 Conclusions

We have reviewed the algorithmic side of the hybridization between constraint programming and operations research (more precisely mixed integer programming) by discussing some of the milestones in such a process.

The implicit question is of course how one should combine CP and MIP solving technologies. The answer to this question, in our view, strongly depends on the focus that the hybrid framework should have. In theory, there is no difference: in both cases one will end up with a CP solver that solves LP relaxations. In practice, the tradeoff to be made is between expressiveness and performance.

More precisely, when the expressiveness of the model has very high importance, e.g., on the applied contexts in which a basic model can be iteratively complicated by the addition of heterogeneous side constraints, the algorithmic approaches in which OR (mostly IP) components are used within a CP framework tend to be preferable. This has been described in Sect. 2. On the other hand, in order to obtain stronger performance, the reverse order is preferable, i.e., the extension of a tightly integrated MIP solver to incorporate some of the expressiveness of CP into the final hybrid system. This has been discussed in Sect. 3.

Acknowledgments We are grateful to the editors Michela Milano and Pascal Van Hentenryck for their support and patience. We are indebted to an anonymous referee for a very careful reading and useful suggestions.

References

1. Achterberg T (2007) Constraint integer programming. PhD thesis, Technische Universität Berlin. http://opus.kobv.de/tuberlin/volltexte/2007/1611/
2. Achterberg T (2009) SCIP: solving constraint integer programs. Math Program Comput 1(1):1–41
3. Achterberg T, Koch T, Martin A (2005) Branching rules revisited. Oper Res Lett 33:42–54
4. Berthold T (2006) Primal heuristics for mixed integer programs. Master's thesis, Technische Universität Berlin
5. Crowder H, Johnson EL, Padberg MW (1983) Solving large scale zero-one linear programming problems. Oper Res 31:803–834
6. Danna E, Rothberg E, Le Pape C (2005) Exploring relaxation induced neighborhoods to improve MIP solutions. Math Program 102(1):71–90
7. Fischetti M, Toth P (1989) An additive bounding procedure for combinatorial optimization problems. Oper Res 37:319–328
8. Focacci F, Lodi A, Milano M (1999) Cost-based domain filtering. In: Jaffar J (ed) Principles and practice of constraint programming – CP99. Lecture notes in computer science, vol 1713. Springer, New York, pp 189–203
9. Focacci F, Lodi A, Milano M (2000) Cutting planes in constraint programming: an hybrid approach. In: Dechter R (ed) Principles and practice of constraint programming – CP00. Lecture notes in computer science, vol 1894. Springer, Berlin, pp 187–201
10. Focacci F, Lodi A, Milano M (2002) A hybrid exact algorithm for the TSPTW. INFORMS J Comput 14:403–417
11. Focacci F, Lodi A, Milano M (2002) Optimization-oriented global constraints. Constraints 7:351–365

12. Gomes C, Selman B, Kautz H (1998) Boosting combinatorial search through randomization. In: Proceedings of the fifteenth national conference on artificial intelligence (AAAI-98)
13. Harvey W, Ginsberg M (1995) Limited discrepancy search. In: Proceedings of the 14th IJCAI. San Francisco, CA, Morgan Kaufmann, pp 607–615
14. Hooker JN (2007) Integrated methods for optimization. Springer, Berlin
15. Jain V, Grossmann IE (2001) Algorithms for hybrid MILP/CP models for a class of optimization problems. INFORMS J Comput 13:258–276
16. Klar A (2006) Cutting planes in mixed integer programming. Master's thesis, Technische Universität Berlin
17. Lodi A, Milano M, Rousseau L-M (2006) Discrepancy-based additive bounding procedures. INFORMS J Comput 18:480–493
18. Marchand H (1998) A polyhedral study of the mixed knapsack set and its use to solve mixed integer programs. PhD thesis, Faculté des Sciences Appliquées, Université catholique de Louvain
19. Marchand H, Wolsey LA (2001) Aggregation and mixed integer rounding to solve MIPs. Oper Res 49(3):363–371
20. Marques-Silva JP, Sakallah KA (1999) GRASP: a search algorithm for propositional satisfiability. IEEE Trans Comput 48:506–521
21. Milano M, Ottosson G, Refalo P, Thorsteinsson ES (2002) The role of integer programming techniques in constraint programming's global constraints. INFORMS J Comput 14:387–402
22. Milano M, van Hoeve WJ (2002) Reduced cost-based ranking for generating promising subproblems. In: Van Hentenryck P (ed) Principles and practice of constraint programming – CP02. Lecture notes in computer science, vol 2470. Springer, Berlin, pp 1–16
23. Milano M, Wallace M (2006) Integrating operations research in constraint programming. 4OR 4:175–219
24. Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S (2001) Chaff: engineering an efficient SAT solver. In: Proceedings of the design automation conference
25. Nemhauser GL, Wolsey LA (1988) Integer and combinatorial optimization. Wiley, New York
26. Refalo P (2000) Linear formulation of constraint programming models and hybrid solvers. In: Dechter R (ed) Principles and practice of constraint programming – CP00. Lecture notes in computer science, vol 1894. Springer, London, pp 369–383
27. Régin JC (1994) A filtering algorithm for constraints of difference in CSPs. In: Hayes-Roth B, Korf R (eds) Proceedings of the national conference on artificial intelligence – AAAI94, pp 362–367
28. Rodošek R, Wallace M (1998) A generic model and hybrid algorithm for hoist scheduling problems. In: Maher MJ, Puget J-F (eds) Principles and practice of constraint programming – CP98. Lecture notes in computer science, vol 1520. Springer, London, pp 385–399
29. Rodošek R, Wallace MG, Hajian MT (1999) A new approach to integrating mixed integer programming with constraint logic programming. Ann Oper Res 86:63–87
30. Sadykov R, Wolsey LA (2006) Integer programming and constraint programming in solving a multi-machine assignment scheduling problem with deadlines and release dates. INFORMS J Comput 18:209–217
31. Thienel S (1995) ABACUS – A Branch-and-Cut System. PhD thesis, Institut für Informatik, Universität zu Köln
32. Wolter K (2006) Implementation of cutting plane separators for mixed integer programs. Master's thesis, Technische Universität Berlin

Over-Constrained Problems

Willem-Jan van Hoeve

Abstract Over-constrained problems are ubiquitous in real-world applications. In constraint programming, over-constrained problems can be modeled and solved using *soft constraints*. Soft constraints, as opposed to hard constraints, are allowed to be violated, and the goal is to find a solution that minimizes the total amount of violation. In this chapter, an overview of recent developments in solution methods for over-constrained problems using constraint programming is presented, with an emphasis on soft global constraints.

1 Introduction

In the context of constraint programming, combinatorial optimization problems are modeled using variables and constraints over subsets of these variables. When the constraints in a model do not allow any solution to the problem, we say that the problem is *over-constrained*. Unfortunately, most combinatorial problems found in real-world applications are essentially over-constrained. Practitioners typically circumvent this inherent difficulty when *modeling* the problem by ignoring certain aspects of the problem. The resulting model, that hopefully allows a solution, then serves as a relaxation of the original problem.

Instead of removing constraints, one may wish to slightly modify (some of) the constraints, thereby maintaining a model that is as close as possible to the original problem description. A natural way to modify constraints in an over-constrained setting is to allow some constraints to be (partly) violated. In constraint programming, constraints that are allowed to be violated are called *soft constraints*. Solving the original problem then amounts to finding a solution that minimizes the overall cost of violation, or to optimize the original objective function given a threshold value on the total amount of violation that is acceptable.

W.-J. van Hoeve
Tepper School of Business, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA, USA
e-mail: vanhoeve@andrew.cmu.edu

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_6,
© Springer Science+Business Media, LLC 2011

This chapter gives an overview of techniques to handle over-constrained problems in the context of constraint programming. Following the nature of this collection, the focus will be on recent developments that are most relevant to CPAIOR, over (roughly) the last 10 years. Interestingly, in 1998, the first paper appeared that marked the start of the recent research efforts that will be discussed in this chapter, that of *soft global constraints*.

1.1 A Brief Historical Overview

We start by presenting a brief overview of soft constraints and over-constrained problems in constraint programming. The most influential early works on soft constraints are the framework for Constraint Hierarchies by Borning et al. [17], and the Partial-CSP framework by Freuder and Wallace [26]. The latter includes the *Max-CSP* framework that aims to maximize the number of satisfied constraints. Since in this framework each constraint is either violated or satisfied, the objective is equivalent to minimizing the number of violated constraints. It has been extended to the *Weighted CSP* framework by Larrosa [42] and Larrosa and Schiex [43], associating a degree of violation (not just a Boolean value) to each constraint and minimizing the sum of all weighted violations. The *Possibilistic-CSP* framework in [76] associates a preference to each constraint (a real value between 0 and 1) representing its importance. The objective of the framework is the hierarchical satisfaction of the most important constraints, i.e., the minimization of the highest preference level for a violated constraint. The *Fuzzy-CSP* framework in [22, 24], and [72] is somewhat similar to the Possibilistic-CSP but here a preference is associated to each tuple of each constraint. A preference value of 0 means the constraint is highly violated and 1 stands for satisfaction. The objective is the maximization of the smallest preference value induced by a variable assignment. The last two frameworks are different from the previous ones since the aggregation operator is a *min*/*max* function instead of addition. With valued-CSPs [77] and semi-rings [14], it is possible to encode Max-CSP, weighted CSPs, Fuzzy CSPs, and Possibilistic CSPs.

Even though the above approaches allow to model a wide range of over-constrained problems, certain aspects arising in practical problems cannot be represented, as argued by Petit, Régin and Bessière [61]. First, it is important to distinguish hard constraints that must always be satisfied (for example, due to physical restrictions) and soft constraints, that are allowed to be violated. All above frameworks, except for Max-CSP, allow to model this distinction. However, in most practical problems, not all soft constraints are equally important. Instead, they are usually subject to certain rules, such as "if constraint c_1 is violated, then c_2 cannot be violated," or "if constraint c_3 is violated, a new constraint c_4 becomes active". Rules of this nature cannot be modeled using the above frameworks, which was one of the main motivations to introduce the *meta-constraint* framework by Petit et al. [61]; see also Petit [59]. In this framework, a cost variable is associated to each soft constraint, representing the degree of violation for that constraint. If the cost variable is 0, the constraint is satisfied. By posting meta-constraints on these

cost variables, we can easily model additional rules and preferences among the soft constraints. For example, if z_i represents the cost variable of soft constraint c_i (for $i = 1, 2, 3, 4$), the above rules can be modeled as $(z_1 > 0) \rightarrow (z_2 = 0)$, and $(z_3 > 0) \rightarrow c_4$, respectively. In addition, Petit et al. [61] show that the meta-constraint framework can be used to model the Max-CSP, Weighted CSP, Possibilistic CSP, and Fuzzy CSP frameworks in a straightforward manner.

An important aspect of the meta-constraint framework is that it allows to propagate information from one (soft) constraint to the other through the domains of the cost variables using domain filtering algorithms. This can be done even for *global constraints* (and soft global constraints) that encapsulate a particular combinatorial structure on an arbitrary number of variables. The first such filtering algorithm was given by Baptiste [5], while Petit et al. [62] introduce soft global constraints in the context of their meta-constraint framework. Since then, several papers have appeared that present filtering algorithms for soft global constraints, many of which use methods from operations research (e.g., matchings and network flows), or computer science (e.g., formal languages). Therefore, the developments in the area of soft global constraints are an exemplary illustration for the successful integration of CP, AI, and OR over the last 10 years.

1.2 Outline

The main focus of this chapter is on soft global constraints. We first introduce basic constraint programming concepts in Sect. 2. Then, in Sect. 3, we introduce soft constraints and show how they can be treated as hard optimization constraints using the meta-constraint framework of Petit et al. [61]. Section 4 presents soft global constraints: We discuss in detail the soft `alldifferent` constraint, the soft *global cardinality* constraint, and the soft `regular` constraint. This section also provides a comprehensive overview of other soft global constraints that have appeared in the literature. Section 5 discusses constraint-based local search and shows the parallel between soft global constraints and constraint-based local search. Finally, we present a conclusion and an outlook in Sect. 6.

2 Constraint Programming

We first introduce basic constraint programming concepts. For more information on constraint programming, we refer to the books by Apt [4], Dechter [21], and Rossi et al. [71]. For more information on global constraints, we refer to [35, 67], and Chapter "Global Constraints: A Survey" of this collection.

Let x be a variable. The *domain* of x, denoted by $D(x)$, is a set of values that can be assigned to x. In this chapter, we only consider variables with *finite* domains. For a set of variables X, we denote $D(X) = \bigcup_{x \in X} D(x)$.

A *constraint* C on a set of variables $X = \{x_1, x_2, \ldots, x_k\}$ is defined as a subset of the Cartesian product of the domains of the variables in X, i.e., $C \subseteq D(x_1) \times D(x_2) \times \cdots \times D(x_k)$. A tuple $(d_1, \ldots, d_k) \in C$ is called a *solution* to C. We also say that the tuple *satisfies* C. A value $d \in D(x_i)$ for some $i = 1, \ldots, k$ is *inconsistent* with respect to C if it does not belong to a tuple of C, otherwise it is *consistent*. C is *inconsistent* if it does not contain a solution. Otherwise, C is called *consistent*. A constraint is called a *binary constraint* if it is defined on two variables. If it is defined on an arbitrary number of variables, we call it a *global constraint*.

A *constraint satisfaction problem*, or a *CSP*, is defined by a finite set of variables $\mathscr{X} = \{x_1, x_2, \ldots, x_n\}$ with respective domains $\mathscr{D} = \{D(x_1), D(x_2), \ldots, D(x_n)\}$, together with a finite set of constraints $\mathscr{C}$, each on a subset of $\mathscr{X}$. This is written as $P = (\mathscr{X}, \mathscr{D}, \mathscr{C})$. The goal is to find an assignment $x_i = d_i$ with $d_i \in D(x_i)$ for $i = 1, \ldots, n$, such that all constraints are satisfied. This assignment is called a *solution to the CSP*. A *constraint optimization problem*, or *COP*, is a CSP $(\mathscr{X}, \mathscr{D}, \mathscr{C})$ together with an objective function $f : D(x_1) \times \cdots \times D(x_n) \to \mathbb{R}$ that has to be optimized. This is written as $P = (\mathscr{X}, \mathscr{D}, \mathscr{C}, f)$. A variable assignment is a solution to a COP if it is a solution to its associated CSP. An *optimal* solution to a COP is a solution that optimizes the objective function. In this chapter, we assume that the objective function is to be minimized, unless stated otherwise.

The solution process of constraint programming interleaves *constraint propagation* and *search*. The search process essentially consists of enumerating all possible variable-value combinations, until we find a solution or prove that none exists. We say that this process constructs a *search tree*. To reduce the exponential number of combinations, *domain filtering* and *constraint propagation* is applied at each node of the search tree. A *domain filtering algorithm* operates on an individual constraint. Given a constraint, and the current domains of the variables in its scope, a domain filtering algorithm removes domain values that do not belong to a solution to the constraint. Since variables usually participate in several constraints, the updated domains are propagated to the other constraints, whose domain filtering algorithms in effect become active. This process of constraint propagation is repeated for all constraints until no more domain values can be removed, or a domain becomes empty.

In order to be effective, domain filtering algorithms should be computationally efficient, because they are applied many times during the solution process. Further, they should remove as many inconsistent values as possible. If a domain filtering algorithm for a constraint C removes *all* inconsistent values from the domains with respect to C, we say that it makes C *domain consistent*.[1] In other words, all remaining domain values participate in at least one solution to C. More formally:

Definition 1 (Domain consistency). A constraint C on the variables $x_1, \ldots, x_k$ is called *domain consistent* if for each variable x_i and each value $d_i \in D(x_i)$ $(i = 1, \ldots, k)$, there exist a value $d_j \in D(x_j)$ for all $j \neq i$ such that $(d_1, \ldots, d_k) \in C$.

[1] In the literature, domain consistency is also referred to as *hyper-arc consistency* or *generalized arc consistency*.

In practice, one usually tries to develop filtering algorithms that separate the check for consistency and the actual domain filtering. That is, we would like to avoid applying the algorithm that performs the consistency check for each individual variable-value pair. Moreover, one typically tries to design incremental algorithms that re-use data structures and partial solutions from one filtering event to the next, instead of applying the filtering algorithm from scratch every time it is invoked.

In the context of constraint optimization problems, we define *optimization constraints* in the following way. Let variable z represent the value of the objective function $f(X)$ to be minimized, where $X = \{x_1, x_2, \ldots, x_n\}$ is a set of variables. The corresponding "optimization constraint" can then be defined as

$$C(X, z, f) = \{(d_1, \ldots, d_n, d) | d_i \in D(x_i), d \in D(z), f(d_1, \ldots, d_n) \leq d\}. \quad (1)$$

In other words, C allows only those tuples in the Cartesian product of variables in X that have an objective function smaller than the maximum value of z (we assume z is to be minimized). An optimization constraint is different from a standard inequality constraint mainly because its right-hand side (the value representing the current best solution) will change during the search for a solution. Note that in this definition, we add the function f as an argument to C for syntactical convenience.

It should be noted that we intentionally define z to be not *equal* to $f(X)$ in (1). The reason for this is that the relation $f(X) \leq z$ allows us to establish domain consistency on several optimization constraints efficiently. In particular, it implies that we can filter the domains of variables in X with respect to $\max D(z)$, and to potentially increase $\min D(z)$ with respect to X. If we would have used the relation $f(X) = z$ in (1) instead, the task of establishing domain consistency becomes NP-complete for general optimization constraints.

In some cases, the objective function aggregates several sub-functions, e.g., $z = z_1 + z_2 + \cdots + z_k$, where $z_i = f_i(X_i)$, and X_i is a set of variables, for $i = 1, \ldots, k$. We apply the concept of optimization constraint to each of these variables z_i and functions f_i correspondingly.

3 From Soft Constraints to Hard Optimization Constraints

So far, all constraints in a given CSP or COP are defined as hard constraints that must always be satisfied. We next focus on *soft constraints* that are allowed to be violated. When a soft constraint is violated, we assume that we can measure to what degree it is violated, and that we wish to minimize the overall amount of violation. As discussed in Sect. 1.1, there exist several frameworks to handle soft constraints, and we will focus on the meta-constraint framework introduced by Petit et al. [61], Petit [59].

The meta-constraint framework of Petit et al. [61] for over-constrained problems works as follows. With each soft constraint, we associate a particular measure of violation, and a "cost" variable that represents this violation. As we will see later, the eventual effectiveness of a soft constraint depends heavily on the measure

of violation that is applied. We then transform each soft constraint into a hard optimization constraint and minimize an aggregation function on the cost variables. The aggregation function can, for example, be a weighted sum, or a weighted maximum, of the cost variables. In addition, we can post meta-constraints over the cost variables to model preferences among soft constraints, or more complex relationships, as indicated in Sect. 1.1.

Let us first consider a small motivating example, taken from Petit et al. [62], to illustrate the application and potential of this framework.

Example 1. Consider the constraint $x \leq y$ where x and y are variables with respective domains specified by the intervals $D(x) = [9000, 10000]$ and $D(y) = [0, 20000]$. We soften this constraint by introducing a cost variable z, representing the amount of violation for the constraint. In this case, we let z represent the gap between x and y if the constraint is not satisfied, that is, z represents $\max\{0, x - y\}$. Suppose the maximum amount of violation is 5, i.e., $D(z) = [0, 5]$. This allows us to deduce that $D(y) = [8995, 20000]$, based on the relation $x - y \leq 5$. We can use the semantics of this constraint to obtain the updated domain efficiently by only comparing the bounds of the variables. If we would not exploit the semantics but instead list and check all possible variable-value combinations, reducing $D(y)$ would take at least $|D(x)| \cdot 8995$ checks.

The example above demonstrates how we can exploit the semantics of a constraint to design efficient filtering algorithms for soft constraints. Moreover, it shows that we can perform "back-propagation" from the cost variable to filter the domains of the other variables. This is crucial to make soft global constraints (and optimization constraints in general) effective in practice [5, 25].

We next formally introduce violation measures and the transformation of soft constraints into hard optimization constraints, following the notation of van Hoeve et al. [37].

Definition 2 (Violation measure). A *violation measure* of a constraint $C(x_1, \ldots, x_n)$ is a function $\mu : D(x_1) \times \cdots \times D(x_n) \to \mathbb{R}_+$ such that $\mu(d_1, \ldots, d_n) = 0$ if and only if $(d_1, \ldots, d_n) \in C$.

Definition 3 (Constraint softening). Let z be a variable with finite domain $D(z)$ and $C(x_1, \ldots, x_n)$ a constraint with a violation measure μ. Then,

$$\textit{soft-}C(x_1, \ldots, x_n, z, \mu) = \{(d_1, \ldots, d_n, d) \mid d_i \in D(x_i), d \in D(z), \\ \mu(d_1, \ldots, d_n) \leq d\}$$

is the soft version of C with respect to μ.

In the definition of *soft-*C, z is the cost variable that represents the measure of violation of C; $\max D(z)$ represents the maximum amount of violation that is allowed for C, given the current state of the solution process. Note that *soft-*C is an optimization constraint, since we assume that z is to be minimized.

In addition to Definition 2, we usually require that the violation measure allows us to "back-propagate" the domain of the cost variable z to the domains of the other

variables efficiently when we apply Definition 3. That is, we need to be able to remove inconsistent domain values from $D(x_1), \ldots, D(x_n)$, based on $D(z)$. The violation measures discussed in this chapter possess that property.

For most global constraints, there exist several natural ways to evaluate the degree to which it is violated, and these are usually not equivalent. Two general measures are the *variable-based* violation measure and the *decomposition-based* violation measure, both introduced by Petit et al. [62].

Definition 4 (Variable-based violation measure). Let C be a constraint on the variables $x_1, \ldots, x_n$ and let $d_1, \ldots, d_n$ be an instantiation of variables such that $d_i \in D(x_i)$ for $i = 1, \ldots, n$. The *variable-based violation measure* μ_{var} of C is the minimum number of variables that need to change their value in order to satisfy C.

For the decomposition-based violation measure, we make use of the binary decomposition of a constraint [20].

Definition 5 (Binary decomposition). Let C be a constraint on the variables $x_1, \ldots, x_n$. A *binary decomposition* of C is a minimal set of binary constraints $C_{\text{dec}} = \{C_1, \ldots, C_k\}$ (for integer $k > 0$) on the variables $x_1, \ldots, x_n$ such that the solution set of C equals the solution set of $\bigcap_{i=1}^{k} C_i$.

Note that we can extend the definition of binary decomposition by defining the constraints in C_{dec} on arbitrary variables, such that the solution set of $\bigwedge_{i=1}^{k} C_i$ is mapped to the solution set of C and vice versa as proposed in [70].

Definition 6 (Decomposition-based violation measure). [2] Let C be a constraint on the variables $x_1, \ldots, x_n$ for which a binary decomposition C_{dec} exists and let $d_1, \ldots, d_n$ be an instantiation of variables such that $d_i \in D(x_i)$ for $i = 1, \ldots, n$. The *decomposition-based violation measure* μ_{dec} of C is the number of violated constraints in C_{dec}.

Example 2. The `alldifferent` constraint specifies that a given set of variables take pairwise different values. Consider the following over-constrained CSP:

$$x_1 \in \{a, b\}, x_2 \in \{a, b\}, x_3 \in \{a, b\}, x_4 \in \{b, c\},$$
$$\texttt{alldifferent}(x_1, x_2, x_3, x_4).$$

The following table shows the value of μ_{var} and μ_{dec} for a number of different variable assignments:

(x_1, x_2, x_3, x_4)	μ_{var}	μ_{val}
(a, a, b, c)	1	1
(a, a, b, b)	2	2
(a, a, a, b)	2	3
(b, b, b, b)	3	6

[2] In [62], the decomposition-based violation measure is referred to as *primal graph based violation cost*.

The table shows that μ_{dec} can be more distinctive than μ_{var}. For example, the assignments (a, a, b, b) and (a, a, a, b) are equivalent with respect to μ_{var}, while μ_{dec} is able to distinguish them.

Next, we convert the `alldifferent` constraint into a `soft-alldifferent` constraint and introduce a variable z that measures its violation. For the sake of this example, we assume that its domain is $D(z) = \{0, 1, 2\}$:

$$x_1 \in \{a, b\}, x_2 \in \{a, b\}, x_3 \in \{a, b\}, x_4 \in \{b, c\}, z \in \{0, 1, 2\}$$
$$\texttt{soft-alldifferent}(x_1, x_2, x_3, x_4, z, \mu).$$

We can choose μ to be any measure of violation, for example, μ_{dec} or μ_{var}. This choice impacts the solution space; the assignment (a, a, a, b) is allowed by μ_{var} since its violation value is 2, but not by μ_{dec} because its violation value of 3 is higher than the maximum of $D(z)$.

The variable-based and decomposition-based violation measures can be viewed as "combinatorial violation measures," as they are based on the combinatorial structure of the global constraint. Other violation measures were introduced by Beldiceanu and Petit [10]. For example, they introduce the *refined* variable-based violation measure, that applies the variable-based violation measure to a specific subset of variables only. Furthermore, they introduce the *object-based* violation measure, that can be applied to high-level modeling objects such as activities in a scheduling context. Finally, they propose specific violation measures based on the *graph properties*-representation of global constraints [7].

In addition to these general violation measures, alternative measures exist for specific constraints. For example, van Hoeve et al. [37] introduce the *value-based* violation measure for the global cardinality constraint and the *edit-based* violation measure for the `regular` constraint.

After we have assigned a violation measure to each soft constraint, we can recast our problem as follows. Consider a CSP of the form $P = (X, D, C)$. Suppose, we partition the constraint set C into a subset of hard constraints C_{hard} and a subset of constraints to be softened C_{soft}. We soften each constraint $c_i \in C_{\text{soft}}$ using the violation measure it has been assigned and a cost variable z_i $(i = 1, \ldots, |C_{\text{soft}}|)$ representing this measure. We choose an aggregation function $f : D(z_1) \times \cdots \times D(z_{|C_{\text{soft}}|}) \rightarrow \mathbb{R}$ over the cost variables to represent the overall violation to be minimized. Then, we transform the CSP into the COP $\tilde{P} = (\tilde{X}, \tilde{D}, \tilde{C}, f)$ where $\tilde{X} = X \cup \{z_1, \ldots, z_{|C_{\text{soft}}|}\}$, $\tilde{D}$ contains their corresponding domains, and $\tilde{C}$ contains C_{hard} and the softened version of each constraint in C_{soft}. Note that if our initial problem P is a COP rather than a CSP, we need to define an objective function that balances the original objective and the aggregation of the cost variables.

4 Soft Global Constraints

In this section, we present several soft global constraints, together with, for some of them, detailed filtering algorithms establishing domain consistency. We will consider in detail the soft `alldifferent` constraint in Sect. 4.3, the soft global

Table 1 Best worst-case time complexity for three hard global constraints on n variables and their soft counterparts. Here, "consistency check" denotes the time complexity to verify that the constraint is consistent, while "domain consistency" denotes the additional time complexity to make the constraint domain consistent, given at least one solution. Each algorithm is based on a graph with m arcs

Constraint	Violation measure	Consistency check	Domain consistency	References
`alldifferent`		$O(m\sqrt{n})$	$O(m)$	[65]
`soft-alldifferent`	Variable-based	$O(m\sqrt{n})$	$O(m)$	[62]
`soft-alldifferent`	Decomposition-based	$O(mn)$	$O(m)$	[34]
`gcc`		$O(m\sqrt{n})$	$O(m)$	[64]
`soft-gcc`	Variable-based	$O(m\sqrt{n})$	$O(m)$	[82]
`soft-gcc`	Value-based	$O(m\sqrt{n})$	$O(m)$	[82]
`regular`		$O(m)$	$O(m)$	[57]
`soft-regular`	Variable-based	$O(m)$	$O(m)$	[37]
`soft-regular`	Edit-based	$O(m)$	$O(m)$	[37]

cardinality constraint in Sect. 4.6, and the soft `regular` constraint in Sect. 4.7. An interesting observation for these soft global constraints is that the corresponding filtering algorithms establish domain consistency in the same worst-case time complexity as their hard counterparts, as shown in Table 1. Finally, in Sect. 4.8, an overview of other soft global constraints will be presented.

Some of the presented filtering algorithms rely on matching theory or network flow theory. We present below the basic definitions that we will use in this chapter. For more information we refer to Schrijver [78] and Ahuja et al. [3].

4.1 Matchings

Let $G = (V, E)$ be a graph with vertex set V and edge set E. A *matching* $M \subseteq E$ is a subset of edges such that no two edges in M are incident to a common vertex. A vertex that is incident to an edge in M is said to be *covered* by M. A vertex that is not incident to any edge in M is called an M*-free* vertex. A *maximum matching* or *maximum-size matching* is a matching in G of maximum size.

Let $c : V \to \mathbb{N}$ be a "capacity" function on the vertices of G. A *capacitated matching* $M \subseteq E$ is a subset of edges such that each vertex $v \in V$ is incident to at most $c(v)$ edges in M. Note that a capacitated matching is equivalent to a "normal" matching if $c(v) = 1$ for all $v \in V$. A *maximum (capacitated) matching* in a vertex-capacitated graph is a capacitated matching of maximum size.

4.2 Network Flows

Let $D = (V, A)$ be a directed graph (or network) and let $s, t \in V$ represent the "source" and the "sink," respectively. An arc $a \in A$ from u to v will also be represented as (u, v).

A function $f : A \to \mathbb{R}$ is called a *flow from s to t*, or an *$s-t$ flow*, if

$$\begin{aligned} &(i)\ f(u,v) \geq 0 && \text{for each } (u,v) \in A, \\ &(ii)\ \sum_{u:(u,v)\in A} f(u,v) = \sum_{w:(v,w)\in A} f(v,w) && \text{for each } v \in V \setminus \{s,t\}. \end{aligned} \tag{2}$$

Property (2)(*ii*) ensures *flow conservation*, i.e., for a vertex $v \neq s,t$, the amount of flow entering v is equal to the amount of flow leaving v.

The *value* of an $s-t$ flow f is defined as

$$\text{value}(f) = \sum_{v:(s,v)\in A} f(s,v) - \sum_{u:(u,s)\in A} f(u,s).$$

In other words, the value of a flow is the net amount of flow leaving s, which by flow conservation must be equal to the net amount of flow entering t.

In a flow network, each arc $a \in A$ has an associated "demand" $d(a)$ and "capacity" $c(a)$, such that $0 \leq d(a) \leq c(a)$. We say that a flow f is *feasible* in the network if $d(a) \leq f(a) \leq c(a)$ for every $a \in A$. If the demand d and capacity c are integer-valued, it can be shown that if there exists a feasible flow, there also exists an *integer* feasible flow in D.

Let $w\colon A \to \mathbb{R}$ be a "weight" (or "cost") function on the arcs. We define the weight of a directed path P as $\text{weight}(P) = \sum_{a\in P} w(a)$. Similarly for a directed circuit, the weight of a flow f is defined as

$$\text{weight}(f) = \sum_{a\in A} w(a) f(a).$$

A feasible flow f is called a *minimum-weight flow* if $\text{weight}(f) \leq \text{weight}(f')$ for any feasible flow f'.

Let f be an $s-t$ flow in G. The *residual graph* of f (with respect to c and d) is defined as $D_f = (V, A_f)$, where the arc set A_f is defined as follows. For all arcs, $a = (u,v) \in A$:

- If $f(a) < c(a)$, then $(u,v) \in A_f$ with residual demand $\max\{d(a) - f(a), 0\}$, residual capacity $c(a) - f(a)$, and residual weight $w(a)$
- If $f(a) > d(a)$, then $(v,u) \in A_f$ with residual demand 0, residual capacity $f(a) - d(a)$, and residual weight $-w(a)$

4.3 Soft Alldifferent Constraint

The `alldifferent` constraint on a set of variables specifies that all variables should take pairwise different values. Here we consider two measures of violation to soften the `alldifferent` constraint: The variable-based violation measure μ_{var} and

the decomposition-based violation measure μ_{dec}. For $\texttt{alldifferent}(x_1, \ldots, x_n)$, we have

$$\mu_{\text{var}}(x_1, \ldots, x_n) = \sum_{d \in D(X)} \max\left(|\{i \mid x_i = d\}| - 1, 0\right),$$
$$\mu_{\text{dec}}(x_1, \ldots, x_n) = \left|\left\{(i, j) \mid x_i = x_j, \text{ for } i < j\right\}\right|.$$

If we apply Definition 3 to the `alldifferent` constraint using the measures μ_{var} and μ_{dec}, we obtain $\texttt{soft-alldifferent}(x_1, \ldots, x_n, z, \mu_{\text{var}})$ and $\texttt{soft-alldifferent}(x_1, \ldots, x_n, z, \mu_{\text{dec}})$. Each of the violation measures μ_{var} and μ_{dec} gives rise to a different domain consistency filtering algorithm for `soft-alldifferent`.

4.4 Variable-Based Violation Measure

A domain consistency filtering algorithm for the variable-based `soft-alldifferent` constraint was presented by Petit et al. [62]. It makes use of bipartite matchings.

Throughout this section, let X be a set of variables. The *value graph* of X is a bipartite graph $\mathcal{G}(X) = (V, E)$ where $V = X \cup D(X)$ and $E = \{(x, d) \mid x \in X, d \in D(x)\}$ [44]. It was first observed by Régin [65] that a solution to $\texttt{alldifferent}(X)$ is equivalent to a matching covering X in the corresponding value graph. For the variable-based `soft-alldifferent` constraint, we can exploit the correspondence with bipartite matchings in a similar way.

Lemma 1 (Petit et al. [62]). *Let M be a maximum-size matching in the value graph $\mathcal{G}(X)$. For $\texttt{alldifferent}(X)$, the minimum value of $\mu_{\text{var}}(X)$ is equal to $|X| - |M|$.*

Theorem 1 (Petit et al. [62]). *The constraint $\texttt{soft-alldifferent}(X, z, \mu_{\text{var}})$ is domain consistent if and only if*

(i) All edges in the value graph $\mathcal{G}(X)$ belong to a matching M in $\mathcal{G}(X)$ with $|X| - |M| \leq \max D(z)$

(ii) $\min D(z) \geq |X| - |M|$, where M is a maximum-size matching in $\mathcal{G}(X)$.

We can apply Theorem 1 to establish domain consistency for $\texttt{soft-alldifferent}(x_1, \ldots, x_n, z, \mu_{\text{var}})$ as follows. First, we compute a maximum matching M in the value graph. This can be done in $O(m\sqrt{n})$ time [32], where m is the number of edges in the graph. We then distinguish the following cases:

- If $n - |M| > \max D(z)$, the constraint is inconsistent.
- If $n - |M| < \max D(z)$, the constraint is consistent, and moreover all domain values are consistent. Namely, if we change the value of any variable, the violation increases with at most 1 unit.
- If $n - |M| = \max D(z)$, the constraint is consistent, and only those domain values $d \in D(x)$ whose corresponding edge (x, d) belongs to a maximum matching

are consistent. We can identify all consistent domain values in the same way as for the hard `alldifferent` constraint. That is, we direct the edges in M from X to $D(X)$, and edges not in M from $D(X)$ to X. Then, an edge belongs to any maximum matching if and only if it belongs to M, or it belongs to a path starting from an M-free vertex, or it belongs to a strongly connected component. All these edges can be identified, and the corresponding domain values can be removed, in $O(m)$ time [65, 80].

Finally, we can update $\min D(z)$ to be the maximum of its current value and $n - |M|$.

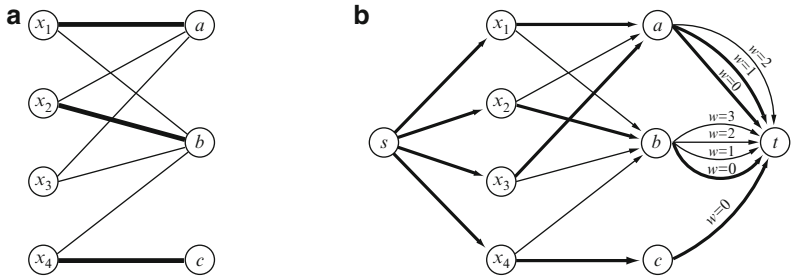
The algorithm above separates the check for consistency and the actual domain filtering. Moreover, it can be implemented to behave incrementally; after k domain changes, a new matching can be found in $O(\min\{km, \sqrt{n}m\})$ time, by re-using the previous matching.

Example 3. Consider the following CSP:

$$x_1 \in \{a, b\}, x_2 \in \{a, b\}, x_3 \in \{a, b\}, x_4 \in \{b, c\}, z \in \{0, 1\},$$
$$\texttt{soft-alldifferent}(x_1, x_2, x_3, x_4, z, \mu_{\text{var}}).$$

The corresponding value graph is depicted in Fig. 1a. The bold edges indicate a maximum-size matching, covering three variables. Hence, the minimum value of μ_{var} is $4 - 3 = 1$, which is equal to $\max D(z)$. This allows us to remove edge (x_4, b), as it does not belong to a matching of size 3. Also, note that we can remove value 0 from $D(z)$, since it does not belong to any solution.

An alternative domain consistency algorithm for the variable-based `soft-alldifferent` constraint was given by van Hoeve et al. [37], based on the correspondence to a minimum-weight network flow. In that work, additional arcs are introduced to the network whose weights reflect the violation measure. A similar approach is presented in the next section, for the decomposition-based `soft-alldifferent` constraint.



Maximum matching in the value graph — Minimum-weight flow in the value network

Fig. 1 Graph representation for the `soft-alldifferent` constraint. In figure (**a**), the value graph for the variable-based `soft-alldifferent` is depicted; bold edges form a maximum matching. In figure (**b**), the extended value network for the decomposition-based `soft-alldifferent` is presented. For all arcs, the capacity is 1. For some arcs, the weight w is given. For all other arcs, the weight is 0

4.5 Decomposition-Based Violation Measure

A first filtering algorithm for the decomposition-based `soft-alldifferent` constraint was given by Petit et al. [62]. It does not necessarily establish domain consistency, and runs in $O(m^2 n\sqrt{n})$ time, where n is the number of variables and m is the sum of the cardinalities of their domains. A domain consistency filtering algorithm was given in van Hoeve [34], running in $O(mn)$ time. Here, we present the latter algorithm.

The filtering algorithm for the decomposition-based `soft-alldifferent` constraint by van Hoeve [34] exploits the correspondence with a minimum-weight network flow. Let us first introduce the network representation of the hard `alldifferent` constraint, which can be viewed as an extension of the value graph. For a set of variables X, we define the *value network* of X as a directed graph $\mathscr{D}(X) = (V, A)$, with vertex set $V = X \cup D(X) \cup \{s, t\}$, and arc set $A = A_s \cup A_X \cup A_t$, where

$$\begin{aligned} A_s &= \{(s, x) \mid x \in X\}, \\ A_X &= \{(x, d) \mid x \in X, d \in D(x)\}, \\ A_t &= \{(d, t) \mid d \in D(X)\}, \end{aligned}$$

with "capacity" function $c(a) = 1$ for all $a \in A$. An integer flow f of value $|X|$ in $\mathscr{D}(X)$ corresponds to a solution to the constraint `alldifferent`(X); the solution is formed by assigning $x = d$ for all arcs $a = (x, d) \in A_X$ with $f(a) = 1$. Moreover, those arcs form a maximum-size matching in the graph induced by A_X (i.e., the value graph).

If the `alldifferent` constraint cannot be satisfied, there does not exist a flow of value $|X|$ in the value network. Therefore, for the `soft-alldifferent` constraint, we adapt the value network in such a way that a flow of value $|X|$ becomes possible, and moreover represents a variable assignment whose violation measure is exactly the cost of the network flow. This is done as follows.

In the graph $\mathscr{D}(X) = (V, A)$, we replace the arc set A_t by $\tilde{A}_t = \{(d, t) \mid d \in D(x), x \in X\}$, with capacity $c(a) = 1$ for all arcs $a \in \tilde{A}_t$. Note that $\tilde{A}_t$ contains parallel arcs if two or more variables share a domain value. If there are k parallel arcs (d, t) between some $d \in D(X)$ and t, we distinguish them by numbering the arcs as $(d, t)_0, (d, t)_1, \ldots, (d, t)_{k-1}$ in a fixed but arbitrary way. One can view the arcs $(d, t)_0$ to be the original arc set A_t.

We next apply a "cost" function $w : A \to \mathbb{N}$ as follows. If $a \in \tilde{A}_t$, i.e., $a = (d, t)_i$ for some $d \in D(X)$ and integer i, we define $w(a) = i$. Otherwise, $w(a) = 0$. Let the resulting digraph be denoted by $\mathscr{D}_{\text{dec}}(X)$. We have the following result.

Lemma 2 (van Hoeve [34]). *Let X be a set of variables, and let f be an integer $s - t$ flow of value $|X|$ in $\mathscr{D}_{\text{dec}}(X)$. Let $\bar{X}$ be the variable assignment $\{x = d \mid (x, d) \in A_X, f(x, d) = 1\}$. For* `alldifferent`$(X)$, $\mu_{\text{dec}}(\bar{X}) = \text{weight}(f)$.

Example 4. For the problem in Example 2, the extended value network $\mathscr{D}_{\text{dec}}(X)$ is presented in Fig. 1b. Bold arcs indicate a minimum-weight flow of weight 1, corresponding to the variable assignment $x_1 = a, x_2 = b, x_3 = a, x_4 = c$. Indeed, this assignment violates one not-equal constraint, $x_1 \neq x_3$.

To illustrate how the cost structure of $\mathscr{D}_{\text{dec}}$ represents the decomposition-based violation measure, suppose we were to assign all variables to value b. Then, there are three units of flow that need to use an arc in $\tilde{A}_t$ with positive cost, while one unit of flow can use the arc in $\tilde{A}_t$ without violation cost. Indeed, for the first variable assigned to b, say x_1, there is no violated binary constraint and the corresponding unit of flow may use the arc without violation cost. The second variable assigned to b, say x_2, violates one binary constraint, namely $x_1 \neq x_2$. Indeed it uses the arc with the next lowest possible cost, i.e., 1. The following variable assigned to b, say x_3, violates two binary constraints (involving x_1 and x_2), which corresponds to using the arc with cost 2. Finally, the fourth variable assigned to b, x_4, violates three binary constraints and uses the arc with cost 3. Together, they exactly constitute the decomposition-based violation of value 6.

Theorem 2 (van Hoeve [34]). *The constraint* $\texttt{soft-alldifferent}(X, z, \mu_{\text{dec}})$ *is domain consistent if and only if*

(*i*) *For every arc* $a \in A_X$ *there exists an integer feasible* $s-t$ *flow* f *of value* $|X|$ *in* $\mathscr{D}_{\text{dec}}(X)$ *with* $f(a) = 1$ *and* $\text{weight}(f) \leq \max D(z)$.

(*ii*) $\min D(z) \geq \text{weight}(f)$ *for a feasible minimum-weight* $s-t$ *flow* f *of value* $|X|$ *in* $\mathscr{D}_{\text{dec}}$.

We can apply Theorem 2 to establish domain consistency for $\texttt{soft-alldifferent}(X, z, \mu_{\text{dec}})$ as follows. We first compute a minimum-weight flow f in $\mathscr{D}_{\text{dec}}$. Since the only positive costs are on arcs in $\tilde{A}_t$, this can be done in $O(mn)$ time, where m is the number of arcs in the graph, and n is the number of variables in X [34]. If $\text{weight}(f) > \max D(z)$, we know that the constraint is inconsistent.

Consistent domain values $d \in D(x)$ for $x \in X$ correspond to arcs $a = (x, d) \in A_X$ for which there exists a flow g with $g(a) = 1$, $\text{value}(g) = |X|$ and $\text{weight}(g) \leq \max D(z)$. To identify these arcs, we apply a theorem from flow theory stating that a minimum-weight flow g with $g(a) = 1$ can be found by "re-routing" the flow f through a shortest directed cycle C containing the arc a in the residual graph of f. Then $\text{weight}(g) = \text{weight}(f) + \text{weight}(C)$. In other words, for each arc $a = (x, d)$ with $f(a) = 0$, we need to compute a shortest $d - x$ path in the residual graph. If the weight of this path exceeds $\max D(z) - \text{weight}(f)$, the value $d \in D(x)$ is inconsistent.

In order to find the shortest $d - x$ paths, we first consider the strongly connected components in the graph induced by A_X. For all arcs (x, d) in these components, the shortest $d - x$ path will remain within the component and has cost 0; indeed, if the path would visit t, the cost cannot decrease since f is a minimum-weight flow.

We next consider all arcs (x, d) between two strongly connected components. Observe that we can assume that the shortest $d - x$ path must visit t *exactly once*. Therefore, we can split the path into two parts: The shortest $d - t$ path and the shortest $t - x$ path. Now, all vertices inside a strongly connected component have the same shortest distance to t, and also the same shortest distance from t (possibly visiting other strongly connected components). Therefore, we can contract

the strongly connected components in the graph induced by A_X and use the resulting acyclic "component graph." Since the algorithm to compute the strongly connected components also provides the topological order and inverse topological order of the component graph, we can apply these to efficiently compute the shortest distance to and from t for every component. Hence, a shortest $d - x$ path is the shortest path from the component to which d belongs to t, plus the path from t to the component to which x belongs. All these computations can be done in $O(m)$ time [19, 80].

Finally, we update $\min D(z) = \text{weight}(f)$ if $\min D(z) < \text{weight}(f)$. Again, this algorithm separates the consistency check from the actual domain filtering. Moreover, the algorithm can be implemented to behave incrementally. After k domain changes, we can re-compute a minimum-weight flow in $O(km)$ time.

4.6 Soft Global Cardinality Constraint

The *global cardinality constraint* (`gcc`) was introduced by Régin [66]. It is defined on a set of variables and specifies for each value in the union of their domains an upper and lower bound to the number of variables that are assigned to this value.

Throughout this section, let $X = \{x_1, \ldots, x_n\}$ be a set of variables and let $l_d, u_d \in \mathbb{N}$ with $l_d \leq u_d$ for all $d \in D(X)$.

Definition 7 (Global cardinality constraint).

$$\begin{aligned}\texttt{gcc}(X, l, u) = \{(d_1, \ldots, d_n) \mid\ & d_i \in D(x_i)\ \forall i \in \{1, \ldots, n\}, \\ & l_d \leq |\{i \mid d_i = d\}| \leq u_d\ \forall d \in D(X)\}.\end{aligned}$$

The `gcc` is a generalization of the `alldifferent` constraint; if we set $l_d = 0$ and $u_d = 1$ for all $d \in D(X)$, the `gcc` is equal to the `alldifferent` constraint.

In order to define measures of violation for the `gcc`, it is convenient to introduce for each domain value a "shortage" function $s : D(x_1) \times \cdots \times D(x_n) \times D(X) \to \mathbb{N}$ and an "excess" function $e : D(x_1) \times \cdots \times D(x_n) \times D(X) \to \mathbb{N}$ as follows [37]:

$$s(X, d) = \begin{cases} l_d - |\{x \mid x \in X, x = d\}| & \text{if } |\{x \mid x \in X, x = d\}| \leq l_d, \\ 0 & \text{otherwise,} \end{cases}$$

$$e(X, d) = \begin{cases} |\{x \mid x \in X, x = d\}| - u_d & \text{if } |\{x \mid x \in X, x = d\}| \geq u_d, \\ 0 & \text{otherwise.} \end{cases}$$

For $\texttt{gcc}(X, l, u)$, the variable-based violation measure μ_{var} can then be expressed in terms of the shortage and excess functions:

$$\mu_{\text{var}}(X) = \max\left(\sum_{d \in D(X)} s(X, d), \sum_{d \in D(X)} e(X, d)\right)$$

provided that

$$\sum_{d \in D(X)} l_d \leq |X| \leq \sum_{d \in D(X)} u_d. \tag{3}$$

Note that if condition (3) does not hold, there is no variable assignment that satisfies the gcc, and μ_{var} cannot be applied. Therefore, van Hoeve et al. [37] introduced the following violation measure for the gcc, which can also be applied when assumption (3) does not hold.

Definition 8 (Value-based violation measure). For $\texttt{gcc}(X, l, u)$, the *value-based violation measure* is

$$\mu_{\text{val}}(X) = \sum_{d \in D(X)} (s(X, d) + e(X, d)).$$

Example 5. Consider the over-constrained CSP

$$x_1 \in \{1, 2\}, x_2 \in \{1\}, x_3 \in \{1, 2\}, x_4 \in \{1\},$$
$$\texttt{gcc}(x_1, x_2, x_3, x_4, [1, 3], [2, 5]).$$

That is, value 1 must be taken between 1 and 2 times, while value 2 must be taken between 3 and 5 times. The violation measures for all possible tuples are:

(x_1, x_2, x_3, x_4)	$\sum_{d \in D(X)} s(X, d)$	$\sum_{d \in D(X)} e(X, d)$	μ_{var}	μ_{val}
(1, 1, 1, 1)	3	2	3	5
(2, 1, 1, 1)	2	1	2	3
(1, 1, 2, 1)	2	1	2	3
(2, 1, 2, 1)	1	0	1	1

For both the variable-based and value-based violation measures for the soft-gcc constraint, van Hoeve et al. [37] present domain consistency filtering algorithms, running in $O(n(m + n \log n))$ and $O((n + k)(m + n \log n))$ time respectively, where n is the number of variables, m is the sum of the cardinalities of the variable domains, and k is the cardinality of the union of the variable domains. Their algorithms are based on an extension of the value network for the decomposition-based soft-alldifferent constraint. They apply the same concept of adding "violation arcs" to allow feasible flows with cost equal to the corresponding variable assignment. A more efficient approach based on matching theory, running in $O(m\sqrt{n})$ time, was proposed by Zanarini [82], and we describe their method below.

4.6.1 Two Capacitated Matchings

Similar to the method proposed by Petit et al. [62], the approach taken by Zanarini et al. [82] for the soft-gcc uses the value graph representation. Recall from

Sect. 4.3 that for a set of variables X, the value graph of X is a bipartite graph $\mathcal{G}(X) = (V, E)$ where $V = X \cup D(X)$ and $E = \{(x, d) \mid x \in X, d \in D(x)\}$. For the `soft-gcc`, the goal is to find two *capacitated* maximum matchings, one minimizing the shortage function and one minimizing the excess function. These matchings can then be used to measure the overall violation cost.

For a constraint $\texttt{gcc}(X, l, u)$, we define two vertex-capacitated value graphs $\mathcal{G}_\text{e}(X)$ and $\mathcal{G}_\text{s}(X)$, by extending the value graph with a "capacity" function $c : V \to \mathbb{N}$ on its vertices. For both $\mathcal{G}_\text{e}(X)$ and $\mathcal{G}_\text{s}(X)$, we define $c(x) = 1$ for each vertex $x \in X$. For the vertices $d \in D(X)$, we define $c(d) = l_d$ for $\mathcal{G}_\text{s}(X)$ and $c(d) = u_d$ $\mathcal{G}_\text{e}(X)$. We will slightly abuse terminology and refer to a capacitated matching as simply a matching.

We first focus on minimizing the *excess* function. Let M_e be a maximum matching in the value graph $\mathcal{G}_\text{e}$. If $|M_\text{e}| = |X|$, the edges in M_e correspond to a partial assignment satisfying the upper capacities u of the values in the `gcc`. If $|M_\text{e}| < |X|$, exactly $|X| - |M_\text{e}|$ variables must be assigned to a saturated value, which equals the total excess for all domain values, i.e., $\sum_{d \in D(X)} e(X, d) = |X| - |M_\text{e}|$.

Analogously for the *shortage* function, let M_s be a maximum matching in the value graph $\mathcal{G}_\text{s}$. Edges in M_s correspond to a partial assignment satisfying the lower capacities l of the values in the `gcc`. If $|M_\text{s}| < \sum_{d \in D(X)} l_d$, one or more values have not enough variables assigned to them. In fact, the difference corresponds to the total shortage of all domain values, i.e., $\sum_{d \in D(X)} s(X, d) = \sum_{d \in D(X)} l_d - |M_\text{s}|$.

4.6.2 Variable-Based Violation Measure

We can characterize domain consistency for the variable-based `soft-gcc` as follows.

Theorem 3 (Zanarini et al. [82]). *The constraint* $\texttt{soft-gcc}(X, l, u, z, \mu_\text{var})$ *is domain consistent if and only if* $\min D(z) \geq \max \left\{ |X| - |M_\text{e}|, \sum_{d \in D(X)} l_d - |M_\text{s}| \right\}$, *and either*

(*i*) $\max \left\{ |X| - |M_\text{e}|, \sum_{d \in D(X)} l_d - |M_\text{s}| \right\} < \max D(z)$, *or*

(*ii*) $|X| - |M_\text{e}| = \max D(z)$ *and* $\sum_{d \in D(X)} l_d - |M_\text{s}| < \max D(z)$, *and all edges in* G_e *belong to a maximum matching, or*

(*iii*) $|X| - |M_\text{e}| < \max D(z)$ *and* $\sum_{d \in D(X)} l_d - |M_\text{s}| = \max D(z)$, *and all edges in* G_s *belong to a maximum matching, or*

(*iv*) $|X| - |M_\text{e}| = \sum_{d \in D(X)} l_d - |M_\text{s}| = \max D(z)$, *and all edges in* $\mathcal{G}_\text{e}$ *and* $\mathcal{G}_\text{s}$ *belong to a maximum matching.*

To establish domain consistency algorithmically, we first compute maximum matchings M_e and M_s in the value graphs $\mathcal{G}_\text{e}$ and $\mathcal{G}_\text{s}$, respectively. An algorithm to compute such capacitated matchings was given by Quimper et al. [64]. It is a generalization of the Hopcroft–Karp algorithm and, similar to the Hopcroft–Karp

algorithm, runs in $O(m\sqrt{n})$, where $n = |X|$ and m is the number edges in the value graph. If $\max\left\{|X| - |M_e|, \sum_{d \in D(X)} l_d - |M_s|\right\} > \max D(z)$, we know the constraint is inconsistent.

Next, we filter the inconsistent edges and corresponding domain values. Using the cardinalities of M_e and M_s, we can easily determine which of the four cases of Theorem 3 applies. In case (*ii*), (*iii*), or (*iv*), we can identify all edges that do not belong to a maximum matching in $O(m)$ time, similar to the approach for the variable-based `soft-alldifferent` constraint in Sect. 4.4.

Once again, the algorithm separates the check for consistency and the actual domain filtering, and it can be implemented to behave incrementally.

Notice that Theorem 3 is an extension of Theorem 1 for the variable-based `soft-alldifferent` constraint by Petit et al. [62]. In fact, when the upper bounds u_d are 1 for all $d \in D(X)$, the two filtering algorithms are equivalent.

Beldiceanu and Petit [10] discuss the variable-based violation measure for a different version of the `soft-gcc`. Their version considers the parameters l and u to be variables instead of constants. Hence, the variable-based violation measure becomes a rather poor measure, as we trivially can change l and u to satisfy the `gcc`. For this reason, they introduce the refined variable-based violation measure, and apply it to their version of the `soft-gcc` by restricting the violation measure to the set of variables X, which corresponds to the `soft-gcc` described above. Beldiceanu and Petit [10] do not provide a filtering algorithm, however.

4.6.3 Value-Based Violation Measure

For the value-based `soft-gcc`, domain consistency can be characterized as follows.

Theorem 4 (Zanarini et al. [82]). *The constraint* `soft-gcc`$(X, l, u, z, \mu_{\text{val}})$ *is domain consistent if and only if* $\min D(z) \geq |X| - |M_e| + \sum_{d \in D(X)} l_d - |M_s|$, *and either*

(*i*) $|X| - |M_e| + \sum_{d \in D(X)} l_d - |M_s| < \max D(z)$, *or*

(*ii*) $|X| + |M_e| = \max D(z) - 1$ *and all edges belong to a maximum matching in at least one of* G_e *or* G_s, *or*

(*iii*) $|X| + |M_e| = \max D(z)$ *and all edges belong to a maximum matching in both* G_e *and* G_s.

The filtering algorithm for the value-based `soft-gcc` proceeds similar to the algorithm for the variable-based `soft-gcc`. We first need to compute maximum matchings M_e and M_s, again in $O(m\sqrt{n})$ time, which allows us to perform the consistency check. We then remove all edges and corresponding domain values in $O(m)$ time if we are in cases (*ii*) and (*iii*).

4.7 Soft Regular Constraint

The `regular` constraint was introduced by Pesant [57] (related concepts were introduced by Beldiceanu [11]). It is defined on a fixed-length sequence of finite-domain variables and it states that the corresponding sequence of values taken by these variables belongs to a given regular language. Particular instances of the `regular` constraint can, for example, be applied in rostering problems or sequencing problems.

Before we introduce the `regular` constraint, we need the following definitions [33]. A *deterministic finite automaton* (DFA) is described by a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final (or accepting) states. Given an input string, the automaton starts in the initial state q_0 and processes the string one symbol at the time, applying the transition function δ at each step to update the current state. The string is *accepted* if and only if the last state reached belongs to the set of final states F. Strings processed by M that are accepted are said to belong to the language defined by M, denoted by $L(M)$. For example, with M depicted in Fig. 2, strings $aaabaa$ and cc belong to $L(M)$ but not $aacbba$. The languages recognized by DFAs are precisely regular languages.

Given an ordered sequence of variables $X = x_1, x_2, \ldots, x_n$ with respective finite domains $D(x_1), D(x_2), \ldots, D(x_n) \subseteq \Sigma$, there is a natural interpretation of the set of possible instantiations of X, i.e., $D(x_1) \times D(x_2) \times \cdots \times D(x_n)$, as a subset of all strings of length n over Σ.

Definition 9 (Regular language membership constraint). Let $M = (Q, \Sigma, \delta, q_0, F)$ denote a DFA and let $X = x_1, x_2, \ldots, x_n$ be a sequence of variables with respective finite domains $D(x_1), D(x_2), \ldots, D(x_n) \subseteq \Sigma$. Then

$$\texttt{regular}(X, M) = \{(d_1, \ldots, d_n) \mid d_i \in D(x_i), d_1 d_2 \cdots d_n \in L(M)\}.$$

Here, we consider two measures of violation for the `regular` constraint: The variable-based violation measure μ_{var} and the *edit-based* violation measure μ_{edit} that was introduced by van Hoeve et al. [37].

Let s_1 and s_2 be two strings of the same length. The *Hamming distance* $H(s_1, s_2)$ is the number of positions in which they differ. Associating with a tuple $(d_1, d_2, \ldots, d_n)$ the string $d_1 d_2 \cdots d_n$, the variable-based violation measure can be expressed in terms of the Hamming distance:

$$\mu_{\text{var}}(X) = \min\{H(D, X) \mid D = d_1 \cdots d_n \in L(M)\}.$$

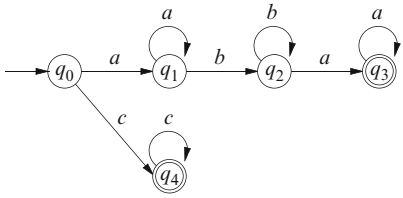


Fig. 2 A representation of a DFA with each state shown as a *circle*, final states as a *double circle*, and transitions as arcs

Another distance function that is often used for comparing two strings is the following. Again, let s_1 and s_2 be two strings of the same length. The *edit distance* $E(s_1, s_2)$ is the smallest number of insertions, deletions, and substitutions required to change one string into another. It captures the fact that two strings that are identical except for one extra or missing symbol should be considered close to one another. The edit distance is probably a better way to measure violations of a `regular` constraint than the Hamming distance. Consider, for example, a regular language in which strings alternate between pairs of a's and b's, for example, "*aabbaabbaa*" belongs to this language. The string "*abbaabbaab*" does not belong to the language, and the minimum Hamming distance, i.e., to any string of the same length that belongs to the language, is 5 (that is, the length of the string divided by 2) since changing either the first a to a b or the first b to an a has a domino effect. On the other hand, the minimum edit distance of the same string is 2, since we can insert an a at the beginning and remove a b at the end. In this case, the edit distance reflects the number of incomplete pairs whereas the Hamming distance is proportional to the length of the string rather than to the amount of violation.

Definition 10 (Edit-based violation measure). For $\texttt{regular}(X, M)$, the *edit-based violation measure* is

$$\mu_{\text{edit}}(X) = \min\{E(D, X) \mid D = d_1 \cdots d_n \in L(M)\}.$$

Example 6. Consider the CSP

$$x_1 \in \{a, b, c\}, x_2 \in \{a, b, c\}, x_3 \in \{a, b, c\}, x_4 \in \{a, b, c\},$$
$$\texttt{regular}(x_1, x_2, x_3, x_4, M)$$

with DFA M as in Fig. 2. We have $\mu_{\text{var}}(c, a, a, b) = 3$, because we need to change the value of at least 3 variables; corresponding valid strings with Hamming distance 3 are, for example, $aaba$ or $cccc$. On the other hand, we have $\mu_{\text{edit}}(c, a, a, b) = 2$, because we can delete the value c at the front and add the value a at the end, thus obtaining the valid string $aaba$.

A graph representation for the `regular` constraint was presented by Pesant [57]. Recall that $M = (Q, \Sigma, \delta, q_0, F)$.

Theorem 5 (Pesant [57]). *A solution to* $\texttt{regular}(X, M)$ *corresponds to an* $s - t$ *path in the digraph* $\mathscr{R} = (V, A)$ *with vertex set*

$$V = V_1 \cup V_2 \cup \cdots \cup V_{n+1} \cup \{s, t\}$$
$$\text{and arc set} \quad A = A_s \cup A_1 \cup A_2 \cup \cdots \cup A_n \cup A_t,$$

$$\begin{aligned}
\text{where } V_i &= \{q_k^i \mid q_k \in Q\} \text{ for } i = 1, \ldots, n+1,\\
\text{and } A_s &= \{(s, q_0^1\},\\
A_i &= \{(q_k^i, q_l^{i+1}) \mid \delta(q_k, d) = q_l \text{ for } d \in D(x_i)\} \text{ for } i = 1, \ldots, n,\\
A_t &= \{(q_k^{n+1}, t) \mid q_k \in F\}.
\end{aligned}$$

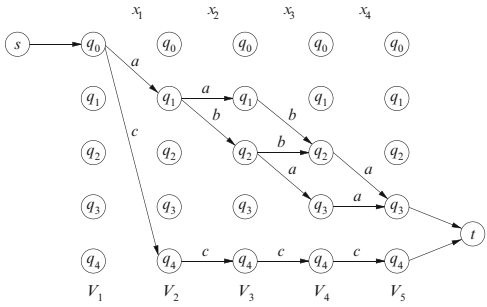


Fig. 3 Graph representation for the `regular` constraint of Example 6, after filtering inconsistent arcs

Theorem 5 can be applied to filter the `regular` constraint to domain consistency, by removing all arcs (and corresponding domain values) that do not belong to an $s - t$ path in $\mathscr{R}$. For the `regular` constraint in Example 6, Fig. 3 gives the corresponding graph representation, after filtering inconsistent arcs. Observe that the filtering algorithm has correctly removed domain value b from $D(x_1)$ and $D(x_4)$.

Whenever the `regular` constraint cannot be satisfied there does not exist an $s - t$ path in $\mathscr{R}$. Therefore, for the `soft-regular` constraint, van Hoeve et al. [37] extend the digraph $\mathscr{R}$ in such a way that an $s - t$ path always exist and has a cost corresponding to the respective measure of violation. For both the variable-based and the edit-based `soft-regular` constraint, again particular weighted "violation arcs" are added to $\mathscr{R}$ to make this possible.

4.7.1 Variable-Based Violation Measure

For the variable-based `soft-regular` constraint, we add the following violation arcs to the graph $\mathscr{R}$ of Theorem 5:

$$A_{\text{sub}} = \{(q_k^i, q_l^{i+1}) \mid \delta(q_k, d) = q_l \text{ for some } d \in \Sigma,\ i = 1, \ldots, n\}.$$

We next apply a "cost" function $w : A \to \mathbb{N}$ as follows. For all arcs $a \in A$, $w(a) = 1$ if $a \in A_{\text{sub}}$ and $w(a) = 0$ otherwise. Let the resulting digraph be denoted by $\mathscr{R}_{\text{var}}$ (see Fig. 4 for an illustration on Example 6).

The input automaton of this constraint specifies the allowed transitions from state to state according to different values. The objective here, in counting the minimum number of substitutions, is to make these transitions value independent. Therefore, the violation arcs in A_{sub} are added between two states (q_k^i, q_l^{i+1}) if there already exists at least one valid arc between them. This means that an $s - t$ path using a violation arc is in fact a solution where a variable takes a value outside its domain. The number of such variables thus constitutes a minimum on the number of variables which need to change value.

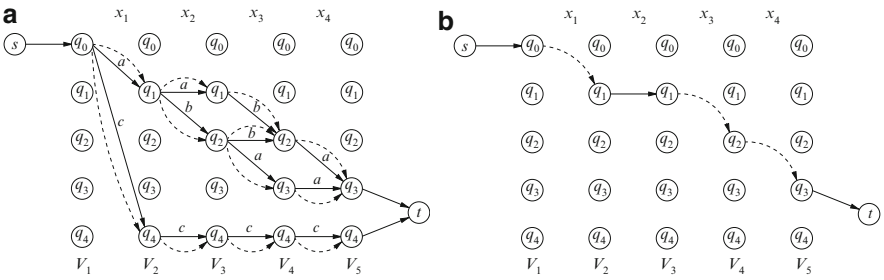


Fig. 4 (**a**) Graph representation for the variable-based `soft-regular` constraint. *Dashed arcs* indicate the inserted weighted arcs with weight 1. (**b**) Example: arcs and associated path used in solution $x_1 = c, x_2 = a, x_3 = a, x_4 = b$ of weight 3, corresponding to three substitutions from valid string *aaba*

Theorem 6 (vanHoeve et al. [37]). *The constraint* `soft-regular`$(X, M, z, \mu_{\text{var}})$ *is domain consistent if and only if*

(*i*) *Every arc* $a \in A_1 \cup \cdots \cup A_n$ *belongs to an* $s-t$ *path* P *in* $\mathscr{R}_{\text{var}}$ *with* $\text{weight}(P) \leq \max D(z)$

(*ii*) $\min D(z) \geq \text{weight}(P)$ *for a minimum-weight* $s-t$ *path in* $\mathscr{R}_{\text{var}}$

The filtering algorithm must ensure that all arcs corresponding to a variable-value assignment are on an $s-t$ path with cost smaller than $\max D(z)$. Computing shortest paths from the initial state in the first layer to every other node and from every node to a final state in the last layer can be done in $O(n\,|\delta|)$ time through topological sorts because of the special structure of the graph (it is acyclic), as observed by Pesant [57]. Here $|\delta|$ denotes the number of transitions in the corresponding DFA. Hence, the algorithm runs in $O(m)$ time, where m is the number of arcs in the graph. The computation can also be made incremental in the same way as proposed by Pesant [57].

A similar filtering algorithm for the variable-based `soft-regular` constraint was proposed by Beldiceanu [11]. That filtering algorithm does not necessarily achieve domain consistency, however.

4.7.2 Edit-Based Violation Measure

For the edit-based `soft-regular` constraint, we add the following violation arcs to the graph $\mathscr{R}$ representing the `regular` constraint. As in the previous section, we add A_{sub} to allow the substitution of a value. To allow deletions and insertions, we add violation arcs

$$A_{\text{del}} = \{(q_k^i, q_k^{i+1}) \mid i = 1, \ldots, n\}\} \setminus A$$

and $\quad A_{\text{ins}} = \{(q_k^i, q_l^i) \mid \delta(q_k, d) = q_l \text{ for some } d \in \Sigma, k \neq l, i = 1, \ldots, n+1\}.$

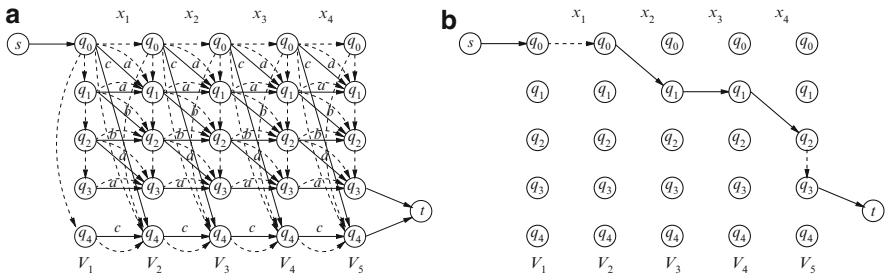


Fig. 5 (**a**) Graph representation for the edit-based `soft-regular` constraint. *Dashed arcs* indicate the inserted weighted arcs with weight 1. (**b**) Example: arcs and associated path used in solution $x_1 = c, x_2 = a, x_3 = a, x_4 = b$ of weight 2, corresponding to one deletion (c in position 1) and one insertion (a in position 4) from valid string $aaba$

We extend the cost function w of the previous section such that $w(a) = 1$ if $a \in A_{\text{del}}$ or $a \in A_{\text{ins}}$. Let the resulting digraph be denoted by $\mathscr{R}_{\text{edit}}$ (see Fig. 5 for an illustration on Example 6).

Deletions are modeled with the arcs introduced in A_{del}, which link equivalent states of successive layers. The intuition is that by using such an arc it is possible to remain at a given state and simply ignore the value taken by the corresponding variable. The arcs in A_{ins} allow a path to make more than one transition at any given layer. Since a layer corresponds to a variable and a transition is made on a symbol of the string, this is equivalent to *inserting* one or more symbols. Of course, one has to make sure only to allow transitions defined by the automaton.

Theorem 7 (vanHoeve et al. [37]). *The constraint* `soft-regular`$(X, M, z, \mu_{\text{edit}})$ *is domain consistent if and only if*

(*i*) *Every arc* $a \in A_1 \cup \cdots \cup A_n$ *belongs to an* $s - t$ *path* P *in* $\mathscr{R}_{\text{edit}}$ *with* $\text{weight}(P) \leq \max D(z)$

(*ii*) $\min D(z) \geq \text{weight}(P)$ *for a minimum-weight* $s - t$ *path* P *in* $\mathscr{R}_{\text{edit}}$

For the filtering algorithm, we proceed slightly different from the variable-based `soft-regular` constraint because the structure of the graph is not the same: Arcs within a layer may form (positive weight) directed circuits. We compute once and for all the smallest cumulative weight to go from q_k^i to q_l^i for every pair of nodes and record it in a table. This can be done through breadth-first-search from each node since every arc considered has unit weight. Notice that every layer has the same "insertion" arcs – we may preprocess one layer and use the result for all of them. In all, this initial step requires $\Theta(|Q|\,|\delta|)$ time. Then, we can proceed as before through topological sort with table lookups, in $O(n\,|\delta|)$ time. The overall time complexity is therefore $O((n + |Q|)\,|\delta|) = O(m)$, where m is the number of arcs in the graph. The last step follows from $|Q| \leq n$, because otherwise some states would be unreachable.

4.8 Other Soft Global Constraints

We next present, in brief, a comprehensive[3] overview of other soft global constraints that have appeared in the literature.

4.8.1 Soft Cumulative Constraint

The `cumulative` constraint can be applied to model and solve resource constraints that appear for example in scheduling and packing problems [2]. It is defined on a set of 'activities,' each of which has an associated variable representing the starting time, a given fixed duration, a given fixed time window in which it can be executed, and a given fixed amount of resource consumption. For example, when scheduling jobs on machines such that any two jobs cannot overlap, jobs correspond to activities, machines represent the (unary) resource, and each job has a unary resource consumption.

For the restricted version of the `cumulative` constraint on unary resources, Baptiste et al. [5] consider the soft version in which the number of late activities (i.e., that are completed after their associated deadline) is to be minimized. They provide a filtering algorithm that is able to identify that some activities must be on time, while others must be late. It is the first soft global constraint with an associated filtering algorithm reported in the literature.

Petit and Poder [60] propose a version of the `soft-cumulative` constraint that aims to minimize the amount of over-load of the resource, while enforcing the time windows for the activities as hard constraints. They present a filtering algorithm for the variable-based violation measure on this constraint. Petit and Poder [60] also provide an experimental comparison between their soft global `cumulative` constraint and the Valued-CSP approach (see Sect. 1.1) on over-constrained scheduling problems, showing the computational advantage of the soft `cumulative` constraint.

4.8.2 Soft Precedence Constraint

Lesaint et al. [46] introduce the `soft-precedence` constraint. It groups together hard precedence constraints and (weighted) soft precedence constraints among certain objects. In the telecommunication application that motivates their work, the objects correspond to features in a call-control feature subscription configuration problem. The `soft-precedence` constraint states that all hard precedence constraints be respected, while the total weight of respected soft precedence constraints is equal to a given value. Achieving domain consistency on `soft-precedence` is NP-hard, and therefore Lesaint et al. [46] propose filtering rules based on lower and upper bounds to the problem.

[3] Comprehensive to the best of our knowledge.

4.8.3 Soft Constraints for a Timetabling Application

Cambazard et al. [18] present three soft global constraints that are applied to solve a particular problem class from the 2007 International Timetabling competition. The three soft global constraints are problem-specific; their purpose is to derive and exploit good bounds for this particular problem class.

4.8.4 Soft Balancing Constraints

Balancing constraints appear in many combinatorial problems, such as fairly distributing workloads (or shifts) over employees, or generating spatially balanced experimental designs. Because a perfect balance is generally not possible, it is natural to soften the balancing constraint and minimize the induced cost of violation, as proposed by [73], following earlier work by Pesant and Régin [58].

For a set of variables $X = \{x_1, \ldots, x_n\}$ and a given fixed sum s, Schaus [73] defines as a measure of violation for the balancing constraint the L_p-norm of $(X - s/n)$, assuming that $\sum_{i=1}^{n} x_i = s$. The L_p-norm of $(X - s/n)$ is defined as

$$\|X - s/n\|_p = \left(\sum_{i=1}^{n} |x_i - s/n| \right)^{\frac{1}{p}},$$

with $p \geq 0$. Schaus [73] then introduces the constraint `soft-balance`(X, s, z, L_p) that holds if and only if $\sum_{i=1}^{n} x_i = s$ and $\|X - s/n\|_p \leq z$.

Different values of p lead to different realizations of the `soft-balance` constraint. For example, for L_0, we measure the number of different values from the mean, while for L_1, we sum the deviations from the mean. For L_2, we sum the squared deviations from the mean (which is equivalent to the variance). Finally, for L_∞, we measure the maximum deviation from the mean.

When the L_1-norm is applied, the resulting `soft-balance` constraint corresponds to the `deviation` constraint introduced by Schaus et al. [75]. Bound consistency filtering algorithms for the `deviation` constraint were given by Schaus et al. [74].

When the L_2-norm is applied, the resulting `soft-balance` constraint corresponds to the `spread` constraint, introduced by Pesant and Régin [58]. The `spread` constraint is more general however, as it allows to represent the mean and standard deviation as (continuous) variables. Pesant and Régin [58] also provide filtering algorithms for the `spread` constraint.

4.8.5 Soft Same Constraint

The `same` constraint is defined on two sequences of variables of equal length and states that the variables in one sequence use the same values as the variables in the other sequence. It can be applied to timetabling problems and pairing problems.

van Hoeve et al. [37] present a domain consistency filtering algorithm for the variable-based soft `same` constraint. Similar to the algorithms for the decomposition-based `soft-alldifferent` and `soft-regular` constraints presented before, it is based on the addition of "violation arcs" to a network flow representation of the problem.

4.8.6 Soft All-Equal Constraint

The ALLEQUAL constraint states that a given set of variables should all be assigned an equal value. The `soft-`ALLEQUAL constraint was introduced by Hebrard et al. [30] as the inverse of the (decomposition-based) `soft-alldifferent` constraint. Hebrard et al. [30] show that finding a solution to the decomposition-based `soft-`ALLEQUAL constraint is NP-complete. Therefore, they propose to filter the constraint using an approximation algorithm, which can be implemented to run in linear amortized time.

Hebrard et al. [31] study the relationship between the `soft-`ALLEQUAL constraint and the `soft-alldifferent` constraint in more detail. They consider variants of the two constraints by combining the variable-based violation measure and the decomposition-based violation measure with the minimization objective and the maximization objective, respectively. In particular, they show that bounds consistency on the minimization-version of the decomposition-based `soft-`ALLEQUAL constraint can be established in polynomial time.

A related soft global constraint, named SIMILAR, was proposed by Hebrard et al. [29] to bound similarities between (partial) solutions, for example, based on the Hamming distance.

4.8.7 Soft Sequence Constraint

The `sequence` constraint was introduced as a global constraint by Beldiceanu and Contejean [9]. It is defined on an ordered sequence of variables X, a fixed number q, a fixed set of domain values S, and fixed lower and upper bounds l and u. It states that for every subsequence of q consecutive variables, the number of variables taking a value from S must be between l and u. The `sequence` constraint can be applied to model problems such as car sequencing or nurse rostering [38, 39].

The `soft-sequence` constraint was studied by Maher et al. [49]. For each subsequence of q consecutive variables, they apply a violation measure that represents the deviation from the lower bound l or upper bound u. The violation measure for the `soft-sequence` is the sum of the violations for all subsequences. Maher et al. [49] present a domain consistency filtering algorithm for this `soft-sequence` constraint based on a particular minimum-weight network flow representation.

4.8.8 Soft Slide Constraint

The `slide` constraint was introduced by Bessiere et al. [13]. It is an extension of the `sequence` constraint, as well as a special case of the `cardinality_path` constraint [8]. The `slide` constraint allows to "slide" any constraint over an ordered sequence of variables, similar to the `sequence` constraint. Additionally, it allows to slide the particular constraint over more than one sequence of variables. Bessiere et al. [12] show how the edit-based and variable-based `soft-slide` constraints can be reformulated in terms of hard `slide` constraints using sequences of additional variables. The `slide` constraint can similarly be applied to encode the variable-based and edit-based `soft-regular` constraints.

4.8.9 Soft Context-Free Grammar Constraint

The context-free grammar constraint (CFG) is an extension of the `regular` constraint; it restricts an ordered sequence of variables to belong to a context-free grammar [63, 79]. The `soft-`CFG constraint was presented by Katsirelos et al. [41] as a special case of the weighted context-free grammar constraint. They propose domain consistency filtering algorithms for the variable-based (or Hamming-based) and edit-based versions of the `soft-`CFG constraint.

4.8.10 Σ-Alldifferent, Σ-Gcc, and Σ-Regular Constraints

The Σ-`alldifferent` constraint was introduced by Métivier et al. [50] as a variation of the `soft-alldifferent` constraint. In the `softalldifferent` constraint as discussed in Sect. 4.3, all variables and all not-equal constraints are equally important. In order to be able to model preferences among variables and constraints, the Σ-`alldifferent` constraint allows to associate a weight to variables and not-equal constraints. These weights have to be taken into account when evaluating the amount of violation of the constraint.

For the variable-based Σ-`alldifferent` constraint, a weight is associated to each variable, and the goal is to find an assignment whose total weighted violation is within the allowed bound defined by the cost variable. Métivier et al. [50] present a domain consistency filtering algorithm based on a weighted network flow representation.

Similarly, for the decomposition-based Σ-`alldifferent` constraint, a weight is associated to each not-equal constraint. For this constraint, achieving domain consistency is NP-hard, however. Therefore, Métivier et al. [50] propose filtering algorithms based on relaxations of the constraint.

[51] present a filtering algorithm for the decomposition-based `soft-gcc` with preferences (the Σ-`gcc` constraint), and for a distance-based `soft-regular`

constraint with preferences (the Σ-regular constraint). The Σ-gcc and other (soft) global constraints are applied by Métivier et al. [52] to model and solve nurse rostering problems.

4.8.11 Soft Global Constraints for Weighted CSPs

Lee and Leung [45] consider soft global constraints in the context of the weighted CSP framework, where costs are associated to the tuples of variable assignments (see Sect. 1.1). In particular, Lee and Leung [45] study the extension of the flow-based soft global constraints of van Hoeve et al. [37] in a weighted CSP setting. They show that the direct application of the flow-based filtering algorithms of van Hoeve et al. [37] can enforce so-called $\varnothing$-inverse consistency in weighted CSPs. Lee and Leung [45] further show how to modify the flow-based algorithms to achieve stronger forms of consistency in weighted CSPs.

4.8.12 Soft Global Constraints for Preference Modeling

Joseph et al. [40] study soft global constraints for preference modeling in the context of multi-criteria decision support and social choice theory. Their underlying model applies several objective functions and binary preference relations. They apply soft global constraints to build hierarchical preference models.

4.8.13 Global Constraint for Max-CSP

The Max-CSP framework aims to maximize the number of satisfied constraints, or equivalently minimize the number of violated constraints (see also Sect. 1.1). Because Max-CSP problems can occur as a subproblems of real-world applications, Régin et al. [68,69] propose to encapsulate the Max-CSP problem as a single global constraint, which can be applied as a soft global constraint. Régin et al. [68] propose a filtering algorithm based on a lower bound on the number of constraint violations, for example, using 'conflict sets'. A conflict set is a set of constraints that leads to a contradiction. For example, the set of constraints $\{x < y, y < z, z < x\}$ is a conflict set, and we can infer that at least one constraint in this set must be violated in any solution. Régin et al. [69] provide new lower bounds based on conflict-sets, where the constraints in the Max-CSP subproblem can be of any arity.

4.8.14 Soft Open Global Constraints

Traditionally, a (global) constraint has a fixed scope of variables on which it is defined. Many practical applications require the scope of a constraint to be less rigid, however. For example, suppose we need to execute a set of activities on different machines, such that on each machine no two activities overlap. Assuming

unit processing times, we can model the non-overlapping requirement using an `alldifferent` constraint on the starting time variables of the activities for each machine. However, the scope of each such `alldifferent` constraint is unknown until we have assigned the activities to the machines. Constraints of this nature are called *open* constraints [6, 23, 36]. During the search for a solution, variables can be added to, or removed from, the scope of an open constraint dynamically.

Maher [47] considers soft open global constraints and investigates when a filtering algorithm for the closed version of a constraint is sound for the open version. The property of *contractibility* introduced by Maher [48] can be used for this purpose. Maher [47] shows that the contractibility of a soft constraint is independent on the contractibility of the associated hard constraint and relies solely on the violation measure that is applied. He further shows that the decomposition-based violation measure and various versions of the edit-based violation measure lead to contractible soft open global constraints. For such soft open global constraints, one can therefore safely apply the existing filtering algorithm for the closed version of the constraint in an open setting. Maher [47] presents a corresponding filtering algorithm for the open `soft-regular` constraint under a weighted edit-based violation measure, building on the existing algorithm for the `soft-regular` constraint by van Hoeve et al. [37].

5 Constraint-Based Local Search

Local search methods provide an alternative to complete systematic search methods (such as constraint programming) for solving combinatorial problems [1, 81]. Conceptually, local search iteratively moves from one solution to a neighboring one, with the aim of improving the objective function. Therefore, local search algorithm are based on a definition of a neighborhood and cost evaluation functions. In many cases, local search can quickly find solutions of good quality, but in general, it is not able to prove optimality of a solution. Local search is a natural approach to solve over-constrained problems, where the objective is to minimize some specified measure of violation of the problem.

In the literature, local search algorithms have been largely described using low-level concepts close to the actual computer implementation. The first modeling language for local search was Localizer [54, 55], which offered a generic and reusable way of implementing different local search methods. In *constraint-based* local search, the aim is to model the problem at hand using constraints and objectives to which then any (suitable) local search can be applied. One of the earliest of such general approaches was developed by Galinier and Hao [27], see also [28]. In that work, a library of constraints is presented that can be used to model a problem. To each constraint, a penalty function is associated that is used in the evaluation function of the Tabu Search engine underlying the system. A similar approach was taken by Michel and Van Hentenryck [53] for the system *Comet*, and by Bohlin [15, 16] for the system *Composer*.

Essential to constraint-based local search is that the solution method can be derived from the constraints of the problem. That is, the definition of neighborhoods as well as the evaluation functions can be based on the combinatorial properties of the constraints. Also global constraints can be used for this purpose. For example, Nareyek [56] applies global constraints to define improvement heuristics for scheduling problems.

The evaluation functions (or penalty functions) for constraints in local search are closely related to the violation measures for soft global constraints in constraint programming. For example, in the system *Comet*, to each constraint, a measure of violation is associated similar to those that are used to define soft global constraints. In constraint-based local search, the violation measures play a different role, however. Instead of filtering variable domains, they are applied to compute a "gradient" with respect to the violation measure. That is, for each variable-value pair, we can define the additional amount of violation if we were to assign this variable to the value.

In the context of constraint-based local search, Van Hentenryck and Michel [81] present violation measures for several global constraints, including `alldifferent`, `atmost`, `atleast`, multi-knapsack, `sequence`, systems of not-equal constraints, and arbitrary (weighted) constraint systems.

6 Conclusion and Outlook

In this chapter, we have presented an overview of techniques to handle over-constrained problems in constraint programming. The main focus has been on recent developments in the area of soft global constraints. Starting from initial works by Baptiste et al. [5], Petit et al. [61], and especially Petit et al. [62], the field of soft global constraint has developed into a mature and established approach to modeling and solving over-constrained problems in constraint programming.

We have presented detailed filtering algorithms for the `soft-alldifferent` constraint, the `soft-gcc` constraint, and the `soft-regular` constraint. In addition, we have given a comprehensive overview of other soft global constraints that have been studied in the literature. The techniques for handling soft global constraints are often based on methods from graph theory, network flows, and regular languages, which reflect the synergy between constraint programming, operations research, and artificial intelligence; the focus of this collection.

Several soft global constraints that appeared in the literature have been applied successfully to solve practical (over-constrained) problems, as we have seen in Sect. 4.8. Nevertheless, so far no commercial constraint programming solver offers soft global constraints as part of their product. Given the increasing interest of the research community as well as the growing number of successful applications, it would be highly desirable if soft global constraints were added to these commercial solvers.

Many research challenges remain in this area. Perhaps the most important one is the issue of aggregating effectively different soft global constraints. It is likely that a weighted sum of the associated cost variables is not the most effective aggregation. Other approaches, such as minimizing the maximum over all cost variables, or applying a (soft) balancing constraint to the cost variables [73], appear to be more promising.

Finally, as was discussed in Sect. 5, the violation measures for soft global constraints are closely related to constraint-based evaluation functions in local search. Therefore, integrating local search and constraint programming based on soft global constraints appears to be an interesting and promising avenue for future research.

Acknowledgements As parts of this chapter are based on the paper [37], I wish to thank Gilles Pesant and Louis-Martin Rousseau.

References

1. Aarts E, Lenstra JK (eds) (2003) Local search in combinatorial optimization. Princeton University Press, Princeton
2. Aggoun A, Beldiceanu N (1993) Extending CHIP in order to solve complex scheduling and placement problems. Math Comput Model 17(7):57–73
3. Ahuja RK, Magnanti TL, Orlin JB (1993) Network flows Prentice Hall, New Jersey
4. Apt KR (2003) Principles of constraint programming. Cambridge University Press, Cambridge
5. Baptiste P, Le Pape C, Péridy L (1998) Global constraints for partial CSPs: a case-study of resource and due date constraints. In: Maher MJ, Puget J-F (eds) Proceedings of the fourth international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 1520. Springer, Berlin, pp 87–101
6. Barták R (2003) Dynamic global constraints in backtracking based environments. Ann Oper Res 118(1–4):101–119
7. Beldiceanu N (2000) Global constraints as graph properties on a structured network of elementary constraints of the same type. In: Dechter R (ed) Proceedings of the sixth international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 1894. Springer, Berlin, pp 52–66
8. Beldiceanu N, Carlsson M (2001) Revisiting the cardinality operator and introducing the cardinality-path constraint family. In: Codognet P (ed) Proceedings of the 17th international conference on logic programming (ICLP). Lecture notes in computer science, vol 2237. Springer, Berlin, pp 59–73
9. Beldiceanu N, Contejean E (1994) Introducing global constraints in CHIP. Math Comput Model 20(12):97–123
10. Beldiceanu N, Petit T (2004) Cost evaluation of soft global constraints. In: Régin J-C, Rueher M (eds) Proceedings of the first international conference on the integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR). Lecture notes in computer science, vol 3011. Springer, Heidelberg, pp 80–95
11. Beldiceanu N, Carlsson M, Petit T (2004) Deriving filtering algorithms from constraint checkers. In: Wallace M (ed) Proceedings of the tenth international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 3258. Springer, Berlin, pp 107–122
12. Bessiere C, Hebrard E, Hnich B, Kiziltan Z, Quimper C-G, Walsh T (2007) Reformulating global constraints: the slide and regular constraints. In: Miguel I, Ruml W (eds) Proceedings of 7th international symposium on abstraction, reformulation, and approximation (SARA). Lecture notes in computer science, vol 4612. Springer, Berlin, pp 80–92

13. Bessiere C, Hebrard E, Hnich B, Kiziltan Z, Walsh T (2008) SLIDE: a useful special case of the CARDPATH constraint. In: Ghallab M, Spyropoulos CD, Fakotakis N, Avouris NM, (eds) Proceedings of the 18th European conference on artificial intelligence (ECAI). IOS, Amsterdam, pp 475–479
14. Bistarelli S, Montanari U, Rossi F (1997) Semiring-based constraint satisfaction and optimization. J ACM 44(2):201–236
15. Bohlin M (2004) Design and implementation of a graph-based constraint constraint model for local search. PhD thesis, Mälardalen University, Licentiate Thesis No. 27
16. Bohlin M (2005) A local search system for solving constraint problems. In: Seipel D, Hanus M, Geske U, Bartenstein O (eds) Applications of declarative programming and knowledge management. Lecture notes in artificial intelligence, vol 3392. Springer, Berlin, pp 166–184
17. Borning A, Duisberg R, Freeman-Benson B, Kramer A, Woolf M (1987) Constraint hierarchies. In: Proceedings of the ACM conference on object-oriented programming systems, languages, and applications (OOPSLA), pp 48–60
18. Cambazard H, Hebrard E, O'Sullivan B, Papadopoulos A (2008) Local search and constraint programming for the post enrolment-based course timetabling problem. In: Proceedings of the 7th international conference on the practice and theory of automated timetabling (PATAT)
19. Cormen TH, Leiserson CE, Rivest RL, Stein C (2001) Introduction to algorithms – second edition. MIT, Cambridge
20. Dechter R (1990) On the expressiveness of networks with hidden variables. In: Proceedings of the 8th national conference on artificial intelligence (AAAI). AAAI/MIT, Cambridge, pp 555–562
21. Dechter R (2003) Constraint processing Morgan Kaufmann, San Mateo
22. Dubois D, Fargier H, Prade H (1993) The calculus of fuzzy restrictions as a basis for flexible constraint satisfaction. In: Proceedings of the second IEEE international conference on fuzzy systems, vol 2, pp 1131–1136
23. Faltings B, Macho-Gonzalez S (2002) Open constraint satisfaction. In: Van Hentenryck P (ed) Proceedings of the 8th international conference on principles and practice of constraint programming (CP 2002). Lecture notes in computer science, vol 2470. Springer, Heidelberg, pp 356–370
24. Fargier H, Lang J, Schiex T (1993) Selecting preferred solutions in fuzzy constraint satisfaction problems. In: Proceedings of the first European congress on fuzzy and intelligent technologies
25. Focacci F, Lodi A, Milano M (2002) Optimization-oriented global constraints. Constraints 7(3):351–365
26. Freuder EC, Wallace RJ (1992) Partial constraint satisfaction. Artif Intell 58(1–3):21–70
27. Galinier P, Hao JK (2000) A general approach for constraint solving by local search. In: Proceedings of the second international workshop on integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR)
28. Galinier P, Hao JK (2004) A general approach for constraint solving by local search. J Math Model Algorithm 3(1):73–88
29. Hebrard E, O'Sullivan B, Walsh T (2007) Distance constraints in constraint satisfaction. In: Veloso MM (ed) Proceedings of the twentieth international joint conference on artificial intelligence (IJCAI), pp 106–111. Available online at http://ijcai.org/.
30. Hebrard E, O'Sullivan B, Razgon I (2008) A soft constraint of equality: complexity and approximability. In: Stuckey PJ (ed) Proceedings of the 14th international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 5202. Springer, Berlin, pp 358–371
31. Hebrard E, Marx D, O'Sullivan B, Razgon I (2009) Constraints of difference and equality: a complete taxonomic characterization. In: Gent IP (ed) Proceedings of the 15th international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 5732. Springer, Berlin, pp 424–438
32. Hopcroft JE, Karp RM (1973) An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. SIAM J Comput 2(4):225–231
33. Hopcroft JE, Ullman JD (1979) Introduction to automata theory, languages, and computation. Addison-Wesley, Reading

34. van Hoeve W-J (2004) A hyper-arc consistency algorithm for the soft alldifferent constraint. In: Wallace M (ed) Proceedings of the tenth international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 3258. Springer, Berlin, pp 679–689
35. van Hoeve W-J, Katriel I (2006) Global constraints. In: Rossi F, Van Beek P, Walsh T (eds) Handbook of constraint programming, chapter 6. Elsevier, New York
36. van Hoeve W-J, Régin J-C (2006) Open constraints in a closed world. In: Beck JC, Smith BM (eds) Proceedings of the third international conference on the integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR). Lecture notes in computer science, vol 3990. Springer, Heidelberg , pp 244–257
37. van Hoeve W-J, Pesant G, Rousseau L-M (2006) On global warming:flow-based soft global constraints. J Heuristics 12(4):347–373
38. van Hoeve W-J, Pesant G, Rousseau L-M , Sabharwal A (2006) Revisiting the sequence constraint. In: Benhamou F (ed) Proceedings of the twelfth international conference on principles and practice of constraint programming (CP). Lecture Notes in Computer Science, vol 4204. Springer, Berlin, pp 620–634
39. van Hoeve W-J, Pesant G, Rousseau L-M, Sabharwal A (2009) New filtering algorithms for combinations of among constraints. Constraints 14:273–292
40. Joseph R-R, Chan P, Hiroux M, Weil G (2007) Decision-support with preference constraints. Eur J Oper Res 177(3):1469–1494
41. Katsirelos G, Narodytska N, Walsh T (2008) The weighted CFG constraint. In: Perron L, Trick MA (eds) Proceedings of the 5th international conference on the integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR). Lecture notes in computer science, vol 5015. Springer, Heidelberg, pp 323–327
42. Larrosa J (2002) Node and arc consistency in weighted CSP. In: Dechter R, Kearns M, Sutton R (eds) Proceedings of the eighteenth national conference on artificial intelligence. AAAI, Canada, pp 48–53
43. Larrosa J, Schiex T (2003) In the quest of the best form of local consistency for weighted CSP. In: Gottlob G, Walsh T (eds) Proceedings of the eighteenth international joint conference on artificial intelligence. Morgan Kaufmann, San Francisco, pp 239–244
44. Lauriere J-L (1978) A language and a program for stating and solving combinatorial problems. Artif intell 10(1):29–127
45. Lee JHM, KL Leung (2009). Towards efficient consistency enforcement for global constraints in weighted constraint satisfaction. In: Boutilier C (ed) Proceedings of the twenty-first international joint conference on artificial intelligence (IJCAI), pp 559–565. Available online at http://ijcai.org/.
46. Lesaint D, Mehta D, O'Sullivan B, Quesada L, Wilson N (2009) A soft global precedence constraint. In: Boutilier C (ed) Proceedings of the twenty-first international joint conference on artificial intelligence (IJCAI), pp 566–571. Available online at http://ijcai.org/.
47. Maher MJ (2009) SOGgy constraints: soft open global constraints. In: Gent IP (ed) Proceedings of the 15th international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 5732. Springer, Heidelberg , pp 584–591
48. Maher MJ (2009) Open contractible global constraints. In: Boutilier C (ed) Proceedings of the twenty-first international joint conference on artificial intelligence (IJCAI), pp 578–583. Available online at http://ijcai.org/
49. Maher MJ, Narodytska N, Quimper C-G, Walsh T (2008) Flow-based propagators for the SEQUENCE and related global constraints. In: Stuckey PJ (ed) Proceedings of the 14th international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 5202. Springer, Heidelberg , pp 159–174
50. Métivier J-P, Boizumault P, Loudni S (2007) Σ-alldifferent: softening alldifferent in weighted CSPs. In: Proceedings of the 19th IEEE international conference on tools with artificial Intelligence (ICTAI), IEEE, pp 223–230
51. Métivier J-P, Boizumault P, Loudni S (2009) Softening Gcc and regular with preferences. In: Proceedings of the 2009 ACM symposium on applied computing (SAC) ACM, pp 1392–1396

52. Métivier J-P, Boizumault P, Loudni S (2009) Solving nurse rostering problems using soft global constraints. In: Gent IP (ed) Proceedings of the 15th international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 5732. Springer, Berlin, pp 73–87
53. Michel L, Van Hentenryck P (2002) A constraint-based architecture for local search. In: Proceedings of the ACM conference on object-oriented programming systems, languages, and applications (OOPSLA), pp 101–110
54. Michel L, Van Hentenryck P (2000) Localizer. Constraints 5:43–84
55. Michel L, Van Hentenryck P (1997) Localizer: a modeling language for local search. In: Smolka G (ed) Proceedings of the third international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 1330. Springer, Berlin, pp 237–251
56. Nareyek A (2001) Using global constraints for local search. In: Constraint programming and large scale discrete optimization: DIMACS workshop constraint programming and large scale discrete optimization, September 14–17, 1998, DIMACS Center. DIMACS series in discrete mathematics and theoretical computer science, vol 54. American Mathematical Society, pp 9–28
57. Pesant G (2004) A regular language membership constraint for finite sequences of variables. In: Wallace M (ed) Proceedings of the tenth international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 3258. Springer, Berlin, pp 482–495
58. Pesant G, Régin J-C (2005) Spread: a balancing constraint based on statistics. In: van Beek P (ed) Proceedings of the 11th international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 3709. Springer, Berlin, pp 460–474
59. Petit T (2002) Modélisation et Algorithmes de Résolution de Problèmes Sur-Contraints. PhD thesis, Université Montpellier II. In French
60. Petit T, Poder E (2008) Global propagation of practicability constraints. In: Perron L, Trick MA (eds) Proceedings of the 5th international conference on the integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR). Lecture notes in computer science, vol 5015. Springer, Berlin, pp 361–366
61. Petit T, Régin J-C, Bessière C (2000) Meta constraints on violations for over constrained problems. In: Proceedings of the 12th IEEE international conference on tools with artificial intelligence (ICTAI). IEEE, pp 358–365
62. Petit T, Régin J-C, Bessière C (2001) Specific filtering algorithms for over-constrained problems. In: Walsh T (ed) Proceedings of the seventh international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 2239. Springer, Berlin, pp 451–463
63. Quimper C-G, Walsh T (2006) Decomposing global grammar constraints. In: Benhamou F (ed) Proceedings of the twelfth international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 4204. Springer, Heidelberg, pp 751–755
64. Quimper C-G, López-Ortiz A, van Beek P, Golynski P (2004) Improved algorithms for the global cardinality constraint. In: Wallace M (ed) Proceedings of the tenth international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 3258. Springer, New York, pp 542–556
65. Régin J-C (1994) A filtering algorithm for constraints of difference in CSPs. In: Proceedings of the twelfth national conference on artificial intelligence (AAAI), vol 1. AAAI, pp 362–367
66. Régin J-C (1996) Generalized arc consistency for global cardinality constraint. In: Proceedings of the thirteenth national conference on artificial intelligence and eighth innovative applications of artificial intelligence conference (AAAI/IAAI), vol 1. AAAI/MIT, pp 209–215
67. Régin J-C (2003) Global constraints and filtering algorithms. In: Milano M (ed) Constraint and integer programming – toward a unified methodology. Operations research/computer science interfaces, chapter 4, vol 27. Kluwer Academic, Dordrecht

68. Régin J-C, Petit T, Bessière C, Puget J-F (2000) An original constraint based approach for solving over constrained problems. In: Dechter R (ed) Proceedings of the sixth international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 1894. Springer, Berlin, pp 543–548
69. Régin J-C, Petit T, Bessière C, Puget J-F (2001) New lower bounds of constraint violations for over-constrained problems. In: Walsh T (ed) Proceedings of the seventh international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 2239. Springer, Berlin, pp 332–345
70. Rossi F, Petrie C, Dhar V (1990) On the equivalence of constraint satisfaction problems. In: Proceedings of the 9th European conference on artificial intelligence (ECAI), pp 550–556
71. Rossi F, Van Beek P, Walsh T (eds) (2006) Handbook of constraint programming. Elsevier, Amsterdam
72. Ruttkay Z (1994) Fuzzy constraint satisfaction. In: Proceedings of the first IEEE conference on evolutionary computing, pp 542–547
73. Schaus P (2009) Solving balancing and Bin-packing problems with constraint programming. PhD thesis, Université catholique de Louvain
74. Schaus P, Deville Y, Dupont P (2007) Bound-consistent deviation constraint. In: Bessiere C (ed) Proceedings of the 13th international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 4741. Springer, Berlin, pp 620–634
75. Schaus P, Deville Y, Dupont P, Régin J-C (2007) The deviation constraint. In: Van Hentenryck P, Wolsey LA (eds) Proceedings of the 4th international conference on the integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR). Lecture notes in computer science, vol 4510. Springer, Berlin, pp 260–274
76. Schiex T (1992) Possibilistic constraint satisfaction problems or "How to handle soft constraints?". In: Dubois D, Wellman MP (eds) Proceedings of the eighth annual conference on uncertainty in artificial intelligence. Morgan Kaufmann, San Francisco, pp 268–275
77. Schiex T, Fargier H, Verfaillie G (1995) Valued constraint satisfaction problems: hard and easy problems. In: Proceedings of the fourteenth international joint conference on artificial intelligence. Morgan Kaufmann, San Francisco, pp 631–639
78. Schrijver A (2003) Combinatorial optimization – polyhedra and efficiency. Springer, Berlin
79. Sellmann M (2006) The theory of grammar constraints. In: Benhamou F (ed) Proceedings of the twelfth international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 4204. Springer, Heidelberg , pp 530–544
80. Tarjan R (1972) Depth-first search and linear graph algorithms. SIAM J Comput 1:146–160
81. Van Hentenryck P, Michel L (2005) Constraint-based local search. MIT, Cambridge
82. Zanarini A, Milano M, Pesant G (2006) Improved Algorithm for the Soft Global Cardinality Constraint. In: Beck JC, Smith BM (eds) Proceedings of the third international conference on the integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CPAIOR). Lecture notes in computer science, vol 3990. Springer, New York, pp 288–299

A Survey on CP-AI-OR Hybrids for Decision Making Under Uncertainty

Brahim Hnich, Roberto Rossi, S. Armagan Tarim, and Steven Prestwich

Abstract In this survey, we focus on problems of decision making under uncertainty. First, we clarify the meaning of the word "uncertainty" and we describe the general structure of problems that fall into this class. Second, we provide a list of problems from the Constraint Programming, Artificial Intelligence, and Operations Research literatures in which uncertainty plays a role. Third, we survey existing modeling frameworks that provide facilities for handling uncertainty. A number of general purpose and specialized hybrid solution methods are surveyed, which deal with the problems in the list provided. These approaches are categorized into three main classes: stochastic reasoning-based, reformulation-based, and sample-based. Finally, we provide a classification for other related approaches and frameworks in the literature.

1 Introduction

In this work, we survey problems in which we are required to make decisions under uncertainty, and we categorize existing hybrid techniques in Constraint Programming (CP), Artificial Intelligence (AI), and Operations Research (OR) for dealing with them. The word *uncertainty* is used to characterize the existence, in these problems, of uncontrollable or "random" variables,[1] which cannot be influenced by the decision maker. In addition to these random variables, problems also comprise controllable or "decision" variables, to which a value from given domains has to be assigned. More specifically, a problem classified as *deterministic* with respect to the degree of uncertainty does not include random variables, while a *stochastic* problem does.

[1] Alternatively, in the literature, these variables are also denoted as "stochastic".

B. Hnich (✉)
Faculty of computer science, Izmir University of Economics, Izmir, Turkey
e-mail: brahim.hnich@ieu.edu.tr

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_7,
© Springer Science+Business Media, LLC 2011

Random variables are typically employed to model factors such as the customer demand for a certain product, the crop yield of a given piece of land during a year, the arrival rate of orders at a reservation center, and so forth. A continuous or discrete domain of possible values that can be observed is associated with each random variable. A probabilistic measure – typically a probability distribution – over such a domain is assumed to be available in order to fully quantify the likelihood of each value (respectively, range of values in the continuous case) that appears in the domain.

The decision making process comprises one or more subsequent *decision stages*. In a decision stage, a decision is taken by the decision maker who assigns a value to each controllable variable related to this decision stage of the problem and, subsequently, the uncontrollable variables related to this stage are observed, and their realized values become known to the decision maker.

It should be noted that, in this work, we do not consider situations in which the decision maker has the power to modify the probability distribution of a given random variable by using his decisions. Random variables are therefore fully uncontrollable. To clarify, this means that a situation in which the decision maker has the option of launching a marketing campaign to affect the distribution of customer demands will not be considered.

This work is structured as follows: in Sect. 2, we employ a motivating example and a well established OR modeling framework – Stochastic Programming – in order to illustrate key aspects associated with the process of modeling problems of decision making under uncertainty; in Sect. 3, we provide a list of relevant problems from the literature on hybrid approaches for decision making under uncertainty and, for each problem, we also provide a short description and a reference to the work in which such a problem has been proposed and tackled; in Sect. 4, we introduce frameworks, from AI and from CP that aim to model problems of decision making under uncertainty; in Sect. 5, we classify existing hybrid approaches for tackling problems of decision making under uncertainty into three classes: in the first class (Sect. 6), we identify general and special purpose approaches that perform "stochastic reasoning"; in the second class (Sect. 7), we list approaches, general and special purpose, that use reformulation; and in the third class (Sect. 8), we categorize approximate techniques based on a variety of strategies employing sampling; finally, in Sect. 9, we point out connections with other related works, and in Sect. 10, we draw conclusions.

2 Decision Making Under Uncertainty

Several interesting real world problems can be classified as "stochastic". In this section, we use a variant of the Stochastic Knapsack Problem (SKP) discussed in [34] as a running example to demonstrate ideas and concepts related to stochastic problems.

2.1 Single-Stage Stochastic Knapsack

A subset of k items must be chosen, given a knapsack of size c into which to fit the items. Each item i, if included in the knapsack, brings a deterministic profit r_i. The size ω_i of each item is stochastic, and it is not known at the time the decision has to be made. Nevertheless, we assume that the decision maker knows the probability mass function PMF(ω_i) [31], for each $i = 1, \ldots, k$. A per unit penalty cost p has to be paid for exceeding the capacity of the knapsack. Furthermore, the probability of the plan not exceeding the capacity of the knapsack should be greater than or equal to a given threshold θ. The objective is to find the knapsack that maximizes the expected profit.

We now discuss Stochastic Programming, which is one of the most well known modeling approaches in OR for problems of decision making under uncertainty, such as the SKP. We arbitrarily chose to employ such a framework to introduce the key concepts of decision making under uncertainty. In the next sections, the following frameworks will be also introduced: Stochastic Boolean Satisfiability, Probabilistic Constraint Satisfaction Problems, Event-Driven Probabilistic Constraint Programming, and Stochastic Constraint Programming.

Stochastic Programming (SP) [11, 32] is a well established technique often used for modeling problems of decision making under uncertainty. A *Stochastic Program* typically comprises a set of decision variables defined over continuous or discrete domains, a set of random variables also defined over continuous or discrete domains and, for each random variable, the respective probability density function (PDF) if continuous or probability mass function (PMF) if discrete. Decision and random variables are partitioned into decision stages. Within a decision stage, first, all the associated decision variables are assigned values; and second, all the associated random variables are observed. A set of constraints is usually enforced over decision and random variables in the model. These constraints may be *hard*, that is they should always be met regardless of the values that are observed for the random variables, or they may be *chance-constraints* [15]. Chance-constraints are constraints that should be satisfied with a probability exceeding a given threshold. If the problem is an optimization one, it may minimize/maximize an objective function defined over some expressions on possible realizations (for example, maximize the worst case performance of the stochastic system under control, or minimize the difference between the maximum and minimum values a performance measure may take to increase the robustness of a system) or some probabilistic measure – such as expectation or variance – of decision and random variables in the model.

To clarify these concepts, we now introduce a Stochastic Programming model for the single-stage SKP (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 available capacity c is not sufficient. Control actions that are performed after the uncertainty is resolved – such

Objective:

$$\max \left\{ \sum_{i=1}^{k} r_i X_i - p\mathbb{E}\left[\sum_{i=1}^{k} \omega_i X_i - c \right]^+ \right\}$$

Subject to:

$$\Pr\left\{ \sum_{i=1}^{k} \omega_i X_i \leq c \right\} \geq \theta$$

Decision variables:

$$X_i \in \{0, 1\} \qquad \forall i \in 1, \ldots, k$$

Random variables:

$$\omega_i \rightarrow \text{item } i \text{ weight} \quad \forall i \in 1, \ldots, k$$

Stage structure:

$$V_1 = \{X_1, \ldots, X_k\}$$
$$S_1 = \{\omega_1, \ldots, \omega_k\}$$
$$L = [\langle V_1, S_1 \rangle]$$

Fig. 1 A Stochastic Programming formulation for the single-stage SKP. Note that $[y]^+ = \max\{y, 0\}$ and $\mathbb{E}$ denotes the expected value operator

as buying additional capacity at a high cost – are called, in SP, "recourse actions". The only chance-constraint in the model ensures that the capacity c is not exceeded with a probability of at least θ. There is only a single decision stage in the model. Decision stages define how uncertainty unfolds in the decision making process. In other words, what the alternation should be between decisions and random variable observations. In a decision stage $\langle V_i, S_i \rangle$, first we assign values to all the decision variables in the set V_i, then we observe the realized values for all the random variables in the set S_i. More specifically, in the single decision stage $\langle V_1, S_1 \rangle$ of the SKP, first, we select all the objects that should be inserted into the knapsack, that is, we assign a value to every decision variable $X_i \in V_1$, $\forall i \in 1, \ldots, k$; second, we observe the realized weight $\omega_i \in S_1$ for every object $i \in 1, \ldots, k$.

We now introduce a numerical example for the single-stage SKP.

Example 1. Consider $k = 5$ items whose item rewards r_i are $\{16, 16, 16, 5, 25\}$. The discrete PMFs for the weight ω_i of item $i = 1, \ldots, 5$ are, respectively: $\text{PMF}(\omega_1) = \{10(0.5), 8(0.5)\}$, $\text{PMF}(\omega_2) = \{9(0.5), 12(0.5)\}$, $\text{PMF}(\omega_3) = \{8(0.5), 13(0.5)\}$, $\text{PMF}(\omega_4) = \{4(0.5), 6(0.5)\}$, $\text{PMF}(\omega_5) = \{12(0.5), 15(0.5)\}$.

The figures in parenthesis represent the probability that an item takes a certain weight. The other problem parameters are $c = 30$, $p = 2$ and $\theta = 0.6$.

As discussed, the problem has a single decision stage. This means that every decision has to be taken in a proactive way, before any of the random variables is observed. Therefore, the optimal solution can be expressed as a simple assignment for the decision variables X_i, $\forall i \in 1, \ldots, k$. More specifically, the optimal solution for Example 1 proactively selects items $\{1, 4, 5\}$ and achieves an expected profit of 45.75. Such a solution can be validated using a scenario tree, as shown in Fig. 2. This tree considers every possible future realization for the random variables ω_i, $\forall i \in 1, \ldots, k$. Since every random variable in the problem takes each of the possible values in its domain with uniform probability, all the paths in the scenario tree are likely to be equal. Therefore, it is easy to compute the expected profit of such an assignment and the expected additional capacity required. By plugging these values

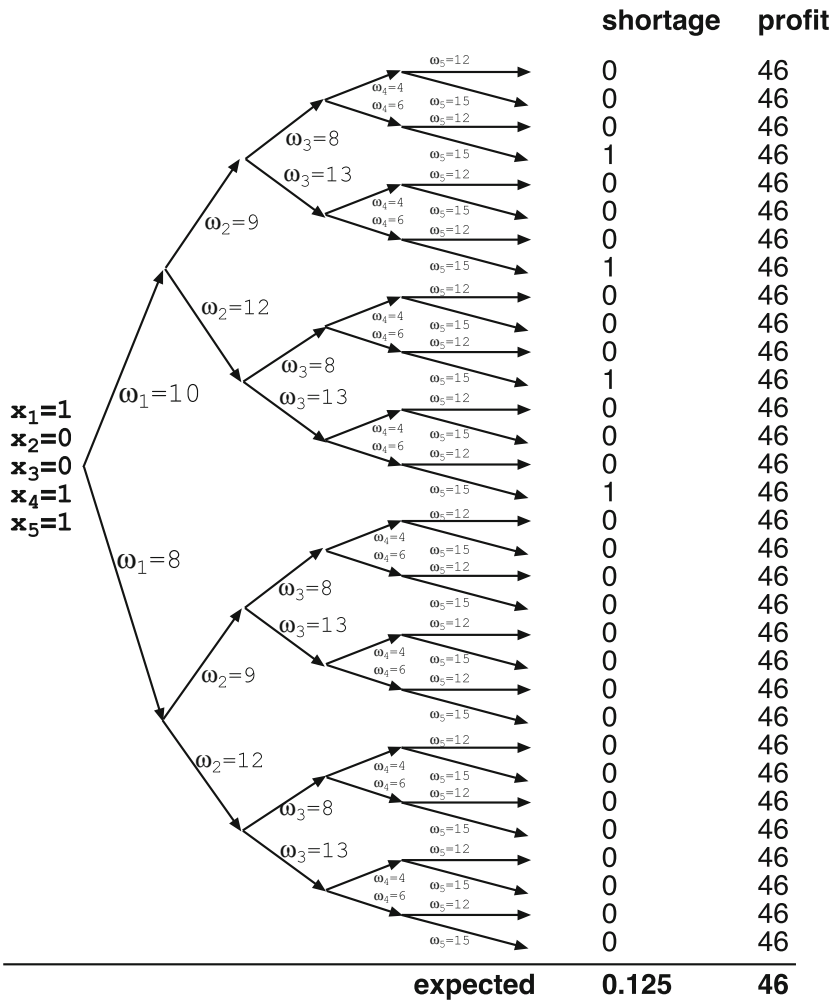


Fig. 2 Scenario tree representing the solution of the single-stage SKP in Example 1

into the objective function, the profit associated with this solution can be easily obtained (i.e., $46 - 2 \cdot 0.125 = 45.75$). Finally, it can be easily verified that the chance-constraint in the model is also satisfied by this solution. In fact, a shortage is observed only in 4 out of 32 scenarios; therefore, the chance constraint is satisfied, in this solution, with probability $0.875 \geq \theta = 0.6$.

The problem discussed in the former paragraphs only comprises a single decision stage. However, in general, stochastic programs may comprise multiple decision stages, that is a sequence of decisions and observations. In order to clarify this, we slightly modify the SKP presented above in such a way as to allow for multiple decision stages. Therefore, we introduce the multi-stage SKP.

2.2 Multi-Stage Stochastic Knapsack

The single-stage problem description and assumptions are valid here with the exception that the items are considered sequentially, starting from item 1 up to item k. In other words, first we take the decision of whether inserting or not a given object into the knapsack, then we immediately observe its weight, which is a random variable, before any further item is taken into account.

A stochastic programming model for the multi-stage SKP is shown in Fig. 3. The model is similar to the one presented in Fig. 1, but the structure of the objective function is different. In this new model, expectation ($\mathbb{E}_{\omega_i}$) and $\max_{X_i}$ operators are nested and parameterized each by, respectively, the random variable ω_i over which the expectation is computed and the decision variable X_i that should be assigned in order to maximize the objective function value. This means, in practice, that an object may be selected or not, depending on the realized weights for previous objects. The stage structure is also different, because now the problem comprises multiple decision stages that alternate decisions and observations according to the arrival sequence of the objects.

We refer, once more, to the Example 1 presented above. The numerical data introduced there can be used to obtain an instance of the multi-stage SKP. As discussed, the problem now has multiple decision stages. This means that decisions are taken in a dynamic way, and they are alternated with observations for random variables. Therefore, the optimal solution is now expressed by using a *solution tree*. A solution tree encodes full information on how to act at a certain decision stage, when some random variables have been already observed. More specifically, the optimal solution tree for the instance of the multi-stage SKP defined by the data in Example 1 achieves an expected profit of 47.75 and it is shown in Fig. 4. To clarify: at the root node, no uncertainty has been unfolded. The optimal solution tree in Fig. 4 shows

Objective:

$$\max_{X_1}\{r_1 X_1 + \mathbb{E}_{\omega_1}\{\max_{X_2} r_2 X_2 + \mathbb{E}_{\omega_2}\{\ldots\{\max_{X_{k-1}} r_{k-1} X_{k-1} + \mathbb{E}_{\omega_k}\{\max_{X_k} r_k X_k + p[\textstyle\sum_{i=1}^{k} \omega_i X_i - c]^+\}\}\ldots\}\}\}$$

Subject to:

$$\Pr\left\{\sum_{i=1}^{k} \omega_i X_i \leq c\right\} \geq \theta$$

Decision variables:

$$X_i \in \{0, 1\} \qquad \forall i \in 1, \ldots, k$$

Random variables:

$$\omega_i \rightarrow \text{item } i \text{ weight} \quad \forall i \in 1, \ldots, k$$

Stage structure:

$$V_i = \{X_i\}\ \forall i \in 1, \ldots, k$$
$$S_i = \{\omega_i\}\ \forall i \in 1, \ldots, k$$
$$L = [\langle V_1, S_1\rangle, \langle V_2, S_2\rangle, \ldots, \langle V_k, S_k\rangle]$$

Fig. 3 Stochastic programming formulation for the multi-stage SKP

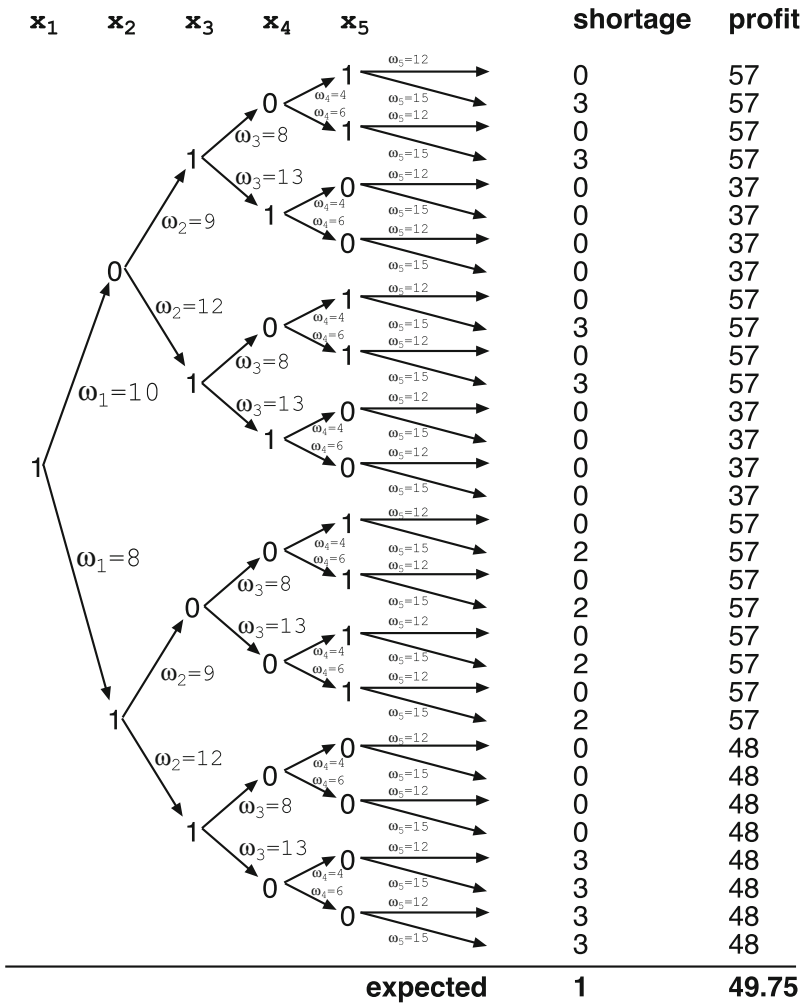


Fig. 4 Solution tree for the multi-stage SKP in Example 1

that it is always optimal to take item 1 in the knapsack. Nevertheless, depending on the observed value for the weight of item 1, two alternative decisions may be optimal: not taking item 2 if the observed weight for item 1 is 10; or taking item 2 if the observed weight for item 1 is 8. To reiterate, since every random variable in the problem takes each of the possible values in its domain with uniform probability, all the paths in the solution tree are likely to be equal. Therefore, it is easy to compute the expected profit of such an assignment and the expected additional capacity required. By plugging these values into the objective function, the profit associated with this solution can be easily obtained. Finally, it can be also easily

verified that the chance-constraint in the model is also satisfied by this solution. In fact, a shortage is observed only in 12 out of 32 scenarios, therefore the chance constraint is satisfied, in this solution, with probability $0.625 \geq \theta = 0.6$.

In this section, we discussed the SKP; in Sect. 3, we provide a further list of problems from the literature discussing hybrid approaches to decision making under uncertainty.

3 A Collection of Stochastic Problems

In this section, we provide a list of nine other problems of decision making under uncertainty for which hybrid approaches have been proposed in the literature. This list is comprehensive in the sense that it contains representative problems for each hybrid CP-AI-OR approach for decision making under uncertainty surveyed in this work.
The problems are:

- Stochastic queueing control problem [8, 74, 75]
- Scheduling Conditional Task Graphs [40]
- Stochastic reservation [5]
- Job shop scheduling (JSP) with probabilistic durations [3]
- Two-stage stochastic matching problem [33]
- Production/inventory management [78]
- Stochastic template design [52, 71]
- Scheduling internal audit activities [60]
- Stochastic sequencing with release times and deadlines [57]

For each of these problems, we provide a textual description. The reader may refer to the respective works where these problems were first introduced to obtain a more detailed description. In Sect. 5, we discuss and classify the hybrid solution methods proposed for modeling and solving these problems.

3.1 Stochastic Queueing Control Problem

In a facility with front room and back room operations, the aim is to switch workers between the rooms in order to cope with changing customer demand. Customer arrival and service time are stochastic and the decision maker seeks a policy for switching workers such that the expected customer waiting time is minimized, while the staff in the back room remains sufficient to perform all work. The problem was originally proposed and analyzed in [8]. Terekhov and Beck investigated it in [74, 75].

3.2 Scheduling Conditional Task Graphs

This is the problem, discussed in [40], of scheduling conditional task graphs in the presence of unary and cumulative resources, minimizing the expected makespan. Conditional task graphs are directed acyclic graphs containing activities linked by precedence relations. Some of the activities represent branches. At run time, only one of the successors of a branch is chosen for execution, depending on the occurrence of a condition labeling the corresponding arc. Since the truth or the falsity of those conditions is not known a priori, the problem is stochastic. Therefore, all the possible future scenarios must be taken into account while constructing the schedule.

3.3 Stochastic Reservation

This problem, introduced in [5], is a particular application of the stochastic multi-knapsack problem. A travel agency may aim at optimizing the reservation of holiday centers during a specific week with various groups in the presence of stochastic demands and cancellations. The requests are coming according a given probability distribution and they are characterized by the size of the group and the price the group is willing to pay. The requests cannot specify the holiday center. However, the travel agency, if it accepts a request, must inform the group of its destination and must commit to it. Groups can also cancel the requests at no cost. Finally, the agency may overbook the centers, in which case the additional load is accommodated in hotels at a fixed cost.

3.4 Job Shop Scheduling with Probabilistic Duration

This problem was originally proposed in [3]. The problem is a classic Job Shop Scheduling (JSP) (see [23], p. 242) in which the objective is to find the minimum makespan. In contrast to the classic formulation for the JSP presented in [23] the authors assume, in this case, that the job durations are probabilistic. The objective is therefore accordingly modified to account for uncertainty: the authors search for a proactive plan, consisting of a partial order among activities and of resource-activity allocations, which attains the lowest possible makespan with probability greater or equal to a given threshold.

3.5 Two-Stage Stochastic Matching Problem

We consider the minimum cost maximum bipartite matching problem discussed in [33]. The task is to buy edges of a bipartite graph which together contain

a maximum-cardinality matching in the graph. The problem is formulated as a two-stage stochastic program with recourse, therefore edges can be bought either during the first stage, or with a recourse action after uncertainty has been resolved. There are two possible variants of this problem. In the first, the uncertainty is in the second stage edge-costs, that is, the cost of an edge can either increase or decrease in the second stage. In the second variant, all edges become more expensive in the second stage, but the set of nodes that must be matched is unknown. This problem can model real-life stochastic integral planning problems such as commodity trading, reservation systems, and scheduling under uncertainty.

3.6 Production/Inventory Management

Uncertainty plays a major role in production and inventory management. In this simplified production/inventory planning example, there are a single product, a single stocking point, production capacity constraints, service level constraints, and a stochastic demand. The objective is to find a replenishment plan associated with the minimum expected total cost. The cost components taken into account are inventory holding costs and fixed replenishment (or setup) costs. The optimal plan gives the timing of the replenishments as well as the order quantities, which depend upon the previously realized demand. This production/inventory management problem has been investigated in [71, 78]. In [69], the authors investigate the same problems under the assumption that the production capacity constraints are relaxed.

3.7 Stochastic Template Design

The deterministic template design problem (prob002 in CSPLib[2]) is described as follows. We are given a set of variations of a design, with a common shape and size and such that the number of required pressings of each variation is known. The problem is to design a set of templates, with a common capacity to which each must be filled, by assigning one or more instances of a variation to each template. A design should be chosen that minimizes the total number of runs of the templates required to satisfy the number of pressings required for each variation. As an example, the variations might be for cartons for different flavors of cat food, such as fish or chicken, where ten thousand fish cartons and twenty thousand chicken cartons must be printed. The problem would then be to design a set of templates by assigning a number of fish and/or chicken designs to each template such that a minimal number of runs of the templates is required to print all thirty thousand cartons.

[2] http://www.csplib.org

Proll and Smith [55] address this problem by fixing the number of templates and minimizing the total number of pressings. In the stochastic version of the problem [52], the demand for each variation is uncertain. In compliance with production/inventory theory, the authors incorporate two conventional cost components: scrap cost, incurred for each template that is produced in excess of the realized demand, and shortage cost, incurred for each unit of demand not fulfilled. The objective is then to minimize the expected total cost.

3.8 Scheduling Internal Audit Activities

Based on costs and benefits that change over time, the focus of the internal audit scheduling problem is how often to conduct an internal audit on an auditable unit. Auditable units are the units upon which internal control procedures are applied in order to safeguard assets and assure the reliability of information flows. The problem, originally introduced in [60], can be stated as follows. We consider a planning horizon comprising of N time periods. We are given a set of M audit units over which random losses may accrue over time. Losses in each period are assumed to have a known PMF that could easily be estimated from available historical data. The distribution of losses may vary from period to period, that is, it is non-stationary. Losses at different periods are assumed to be independent. Auditing is a time-consuming task, and the auditing team is given a strict deadline for performing an audit. Specifically, an audit must be completed in T time periods. Therefore after T periods, the accrued losses drop to zero. If a team has already started auditing a unit at a given time period, then no other audit can be initiated during this period for the given audit team. The timing of audits are fixed once and for all at the beginning of the planning horizon and cannot be changed thereafter, even if it is suspected that certain auditable units have accrued unexpected losses. The objective is to find the optimal audit schedule while respecting the maximum loss criteria. That is, the invariant audit cost (i.e., fixed audit costs incurred each time an audit is conducted) and expected total discounted audit losses (i.e., cumulative losses accrued at the end of each period) are minimized by satisfying a minimum probability α that the losses will not exceed a predetermined level (allowed maximum loss) in any given audit period for any auditable unit.

3.9 Stochastic Sequencing with Release Times and Deadlines

The problem, introduced in [57], consists in finding an optimal schedule to process a set of orders using a set of parallel machines. The objective is to minimize the expected total tardiness of the plan. Processing an order can only begin after its release date and should be completed at the latest by a given due date for such an order. An order can be processed on any of the machines. The processing time of a

given order, when processed on a certain machine, is a random variable. A solution for this problem consists in an assignment for the jobs on the machines and in a total order between jobs on the same machine. A job will be processed on its release date if no other previous job is still processing, or as soon as the previous job terminates.

4 Frameworks for Decision Making Under Uncertainty in CP and AI

In Sect. 2, we introduced SP, a well established OR framework for decision making under uncertainty. In this section, we introduce other existing frameworks for decision making under uncertainty from AI and CP. Stochastic Boolean Satisfiability extends a well established AI modeling framework, Propositional Satisfiability, by considering uncertainty. Probabilistic CSP, Event-Driven Probabilistic Constraint Programming, and Stochastic Constraint Programming set the scene for dealing with uncertainty in CP. Where appropriate, we describe connections and similarities among these different frameworks.

4.1 Stochastic Boolean Satisfiability

The Boolean Satisfiability (SAT) community have investigated problems involving uncertainty, with the *Stochastic Satisfiability* (SSAT) framework. SSAT aims to combine features of logic and probability theory and has been applied to probabilistic planning, belief networks, and trust management. We base our discussion on a recent survey [43].

4.1.1 Definitions

The SAT problem is to determine whether a Boolean expression has a satisfying labeling (set of truth assignments). The problems are usually expressed in conjunctive normal form (CNF): a conjunction of clauses $c_1 \wedge \ldots \wedge c_m$ where each clause c is a disjunction of literals $l_1 \vee \ldots \vee l_n$ and each literal l is either a Boolean variable v or its negation $\bar{v}$. A Boolean variable can be labeled true (T) or false (F). Many constraint problems can be SAT-encoded (modeled as a SAT problem) and vice-versa. In fact any SAT problem can be viewed as a Constraint Satisfaction Problem (CSP) with binary domains and non-binary constraints via the *non-binary encoding* [77]: for example, a clause $a \vee b \vee \bar{c}$ corresponds to the constraint (or conflict) preventing the assignments $\{a \leftarrow F, b \leftarrow F, c \leftarrow T\}$. The SSAT terminology is somewhat different than that of SP but there are many correspondences.

An SSAT problem $\Phi = Q_1 v_1 \ldots Q_n v_n \phi$ is specified by:

- A *prefix* $\Phi = Q_1 v_1 \ldots Q_n v_n$ that orders the Boolean variables $v_1 \ldots v_n$ of the problem and *quantifies* them. Each variable v_i is quantified by its quantifier Q_i either as *existential* ($\exists$) or *randomized* ($\mathfrak{R}$)
- A *matrix* ϕ: a Boolean formula containing the variables, usually in CNF

An existential variable is a standard SAT variable (corresponding to a decision variable in SP), while a randomized variable v_i is a Boolean variable that is true with associated probability π_i (corresponding to a random variable in SP). Sequences of similarly quantified variables may be grouped together into (existential or randomized) *blocks*, and an SP stage corresponds to an existential block followed by a randomized block. The values of existential variables may be contingent on the values of (existential or randomized) variables earlier in the prefix, so an SSAT solution takes the form of an *assignment tree* (corresponding to the solution tree in SP) specifying an assignment to each existential variable for each possible instantiation of the randomised variables preceding it in the prefix. An *optimal assignment tree* is one that yields the maximum probability of satisfaction; alternatively, the decision version of SSAT asks whether the probability of satisfaction exceeds a threshold θ.

SSAT is simpler than a Stochastic Program in three ways: the variable domains are Boolean only (as in SAT), the constraints (clauses) are of a fixed type (as in SAT), and no distinction is made between scenarios in which different clauses are violated. The latter means that SSAT is akin to a stochastic program with a single chance-constraint.

4.1.2 Restrictions and Generalizations

Some special cases have been identified in the literature: if all variables are randomized then we have a MAJSAT problem; if the prefix has only an existential block followed by a randomized block, then we have an E-MAJSAT problem; and if each block contains a single variable, then we have an Alternating SSAT (ASSAT) problem. SSAT has also been extended by the addition of *universal* quantifiers ($\forall$) to give Extended SSAT (XSSAT). A formula $\forall v \phi$ must be true for both $v = T$ and $v = F$. XSSAT subsumes Quantified Boolean Formulae (QBF), which is the archetypal PSPACE-complete problem: QBF is XSSAT without randomized quantifiers.

4.2 Probabilistic Constraint Satisfaction Problems

The Probabilistic CSP framework, proposed in [19], is an extension of the CSP framework [1] that deals with some decision problems under uncertainty. This extension relies on a differentiation between the agent-controllable decision variables and the uncontrollable parameters whose values depend on the occurrence of uncertain events. The uncertainty on the values of the parameters is assumed to be given under the form of a probability distribution.

4.2.1 Definitions

A probabilistic CSP is a CSP equipped with a partition between (controllable) decision variables and (uncontrollable) parameters, and a probability distribution over the possible values of the parameters. More specifically, the authors define a Probabilistic CSP as a 6-tuple $\mathcal{P} = \langle \Lambda, W, X, D, \mathcal{C}, \text{pr} \rangle$, where $\Lambda = \{\lambda_1, \ldots, \lambda_p\}$ is a set of parameters; $W = W_1 \times \cdots \times W_p$, where W_i is the domain of λ_i; $X = \{x_1, \ldots, x_n\}$ is a set of decision variables; $D = D_1 \times \ldots \times D_n$, where D_i is the domain of x_i; $\mathcal{C}$ is a set of constraints, each of them involving at least one decision variable; and $\text{pr} : W \rightarrow [0, 1]$ is a probability distribution over the parameter assignments. Constraints are defined as in classical CSP. A complete assignment of the parameters (i.e., of the decision variables) is called a "world" (i.e., a "decision").

The authors consider successively two assumptions concerning the agents awareness of the parameter values at the time the decision must imperatively be made.

- "No more knowledge": the agent will never learn anything new before the deadline for making a decision; all it will ever know is already encoded by the probability distribution.
- "Complete knowledge": the actual parameters will be completely revealed before the deadline is reached (possibly, just before), so that it useful to the agent to compute off-line a ready-to-use conditional decision, that the agent will be able to instantiate on-line, as soon as it knows what the actual parameters are.

For the first case, a solution is an unconditional decision that is most likely to be feasible according to world probabilities. For the second case, a solution provides a set of decisions with their conditions of applicability – i.e., under which world(s) a given decision should be used – together with the likelihood of occurrence of these conditions, which also follows from world probabilities.

4.3 Event-Driven Probabilistic Constraint Programming

In Event-driven Probabilistic Constraint Programming (EDP-CP), which is an extension of the Probabilistic CSP framework, some of the constraints can be designated by the user as *event constraints*. The user's objective is to maximize his/her chances of realizing these "events". In each world – as defined in the Probabilistic CSP framework – events are subject to certain pre-requisite constraints and to certain conditions. If a pre-requisite is unsatisfied in a given world, then the event is also classed as unsatisfied in that world; and if a condition is unsatisfied in a world, then the event is classed as satisfied in that scenario. Intuitively, this means that in EDP-CP it is possible to express the fact that the feasibility of certain event constraints may depend on the satisfaction of other constraints (denoted as "prerequisite constraints") under certain "conditions". In order to model such situations, a new meta-constraint – the *dependency meta-constraint* – is introduced.

4.3.1 Definitions

An EDP-CP is a 9-tuple $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \Lambda, \mathcal{W}, \mathcal{E}, \mathcal{C}, \mathcal{H}, \Psi, \Pr \rangle$ where:

- $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of decision variables.
- $\mathcal{D} = D_1 \times \ldots \times D_n$, where D_i is the domain of X_i.
- $\Lambda = \{\lambda_1, \ldots, \lambda_l\}$ is a set of uncertain parameters.
- $\mathcal{W} = W_1 \times \ldots \times W_l$, where W_i the domain of λ_i.
- $\mathcal{E} = \{e_1, \ldots, e_m\}$ is a set of event constraints. Each e_i may either be probabilistic (involving a subset of $\mathcal{X}$ and a subset of Λ) or deterministic (involving only a subset of $\mathcal{X}$).
- $\mathcal{C} = \{c_1, \ldots, c_o\}$ is a set of dependency meta-constraints. For each dependency meta-constraint c_i : DEPENDENCY(e, p, f) we have $e \in \mathcal{E}$, where p may be either a probabilistic or a deterministic prerequisite constraint, and f is a deterministic condition constraint.
- $\mathcal{H} = \{\langle_\infty, \ldots, h_p\}$ is a set of hard constraints. Each h_i may either be probabilistic (involving a subset of $\mathcal{X}$ and a subset of Λ) or deterministic (involving only a subset of $\mathcal{X}$).
- Ψ is any expression involving the event realization measures on the event constraints in $\mathcal{E}$.
- $\Pr : \mathcal{W} \rightarrow [0, 1]$ is a probability distribution over uncertain parameters.

An optimal solution to an EDP-CP $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \Lambda, \mathcal{W}, \mathcal{E}, \mathcal{C}, \mathcal{H}, \Psi, \Pr \rangle$ is any assignment S to the decision variables such that:

1. The hard constraints are satisfied in each possible world.
2. There exists no other assignment satisfying all the hard constraints with a strictly better value for Ψ, according to the DEPENDENCY constraints introduced in the model.

4.3.2 Relations to Other Frameworks

The Event-driven Probabilistic Constraint Programming (EDP-CP) framework, proposed in [67], extends both the Probabilistic CSP framework [19] and the Dependent-chance Programming framework [38]. In contrast to probabilistic CSP, which treats all probabilistic constraints uniformly, EDP-CP distinguishes between event, prerequisite, condition, and hard constraints. Furthermore, in Dependent-chance Programming a feasible solution satisfies all event constraints, while in EDP-CP such a requirement is relaxed. This gives the decision-maker more flexibility in modeling. Finally, the notion of constraint dependency introduced in [67] comprises condition constraints, in addition to the event and prerequisite constraints. As the authors remark, constraint dependency without condition constraints does not guarantee optimal plans since in certain instances common variables may take values which break the link between two dependent constraints.

4.4 Stochastic Constraint Programming

Stochastic Constraint Programming (SCP) was first introduced in [78] in order to model combinatorial decision problems involving uncertainty and probability. According to Walsh, SCP combines together the best features of CP (i.e., global constraints, search heuristics, filtering strategies, etc.), of SP (expressiveness in representing problems involving random variables), and of Stochastic Satisfiability.

4.4.1 Definitions

An m-stage Stochastic Constraint Satisfaction Problem (SCSP) is defined, according to [78], as a 7-tuple $\langle V, S, D, P, C, \theta, L\rangle$[3], where V is a set of decision variables and S is a set of random variables, D is a function mapping each element of V and each element of S to a domain of potential values. In what follows, we assume that both decision and random variable domains are finite. P is a function mapping each element of S to a probability distribution for its associated domain. C is a set of chance-constraints over a non-empty subset of decision variables and a subset of random variables. θ is a function mapping each chance-constraint $h \in C$ to θ_h which 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. $L = [\langle V_1, S_1\rangle, \ldots, \langle V_i, S_i\rangle, \ldots, \langle V_m, S_m\rangle]$ is a list of *decision stages* such that each $V_i \subseteq V$, each $S_i \subseteq S$, the V_i form a partition of V, and the S_i form a partition of S.

To solve an m-stage SCSP 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, ..., 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 SCSP is represented by means of a *policy tree*. 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. The policy tree, in fact, corresponds to the solution tree adopted in SP.

Let $\mathcal{S}$ denote the space of policy trees representing all the solutions of a SCSP. 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 a set $\widehat{S} \subseteq S$ of random variables (edges of the policy tree) and over a set $\widehat{V} \subseteq V$ of the decision variables (nodes in the policy tree). A *stochastic constraint optimization problem* (SCOP) is then defined in general as $\max_{s\in\mathcal{S}} f(s)$.

[3] The original formulation, proposed in [78], does not directly encode the stage structure in the tuple and actually defines a SCSP as a 6-tuple; consequently the stage structure is given separately. We believe that a more adequate formulation is the one proposed in [30], that explicitly encodes the stage structure as a part of the tuple, giving a 7-tuple.

Fig. 5 Stochastic Constraint Programming formulation for the single-stage SKP

Objective:

$$\max \left\{ \sum_{i=1}^{k} r_i X_i - p\mathbb{E}\left[\sum_{i=1}^{k} \omega_i X_i - c \right]^+ \right\}$$

$\langle V, S, D, P, C, \theta, L \rangle$**:**

$V = \{X_1, \ldots, X_k\}$

$S = \{\omega_1, \ldots, \omega_k\}$

$D = \{X_1, \ldots, X_k \in \{0, 1\}, D(\omega_1), \ldots, D(\omega_k)\}$

$P = \{PDF(\omega_1), \ldots, PDF(\omega_k)\}$

$C = \left\{ \Pr\left\{ \sum_{i=1}^{k} \omega_i X_i \leq c \right\} \geq \theta \right\}$

$L = [(\{X_1, \ldots, X_k\}, \{\omega_1, \ldots, \omega_k\})]$

Unlike SP, SCP offers a richer modeling language which supports chance-constraints over global, nonlinear, and logical constraints in addition to linear ones.

It is easy to reformulate the running example discussed in Sect. 2 (SKP) as a single-stage SCOP, the respective model is given in Fig. 5. As in the SP model, in the SCP model, we have sets of decision and random variables with their respective domains. For the random variables, the respective PMF is specified. There is a chance-constraint with an associated threshold θ. In fact, the SCOP in Fig. 5 fully captures the structure of the stochastic program in Fig. 1.

5 A Classification of Existing Approaches

In previous sections, we stressed the fact that this survey is centered on "uncertainty," and we also clarified the precise meaning we associate with the term uncertainty. Other literature surveys tend to merge uncertainty with other concepts; in Sect. 9, we will briefly discuss related works in these different areas, and the reader may refer to these surveys for more details. Furthermore, there exist surveys that are more explicitly focused on pure AI [10] or OR [62] techniques, but little attention has been dedicated so far to hybrid techniques.

In this section, we propose a classification for existing hybrid approaches and frameworks that blend CP, AI, and OR for decision making under uncertainty. The integration of CP, AI, and OR techniques for decision making under uncertainty is a relatively young research area. We propose to classify existing approaches in the literature within three main classes (Fig. 6).

- The first class comprises those approaches that perform some form of "stochastic reasoning" by using dedicated – general or special purpose – search procedures, filtering algorithms, neural networks, genetic algorithms, etc.
- The second class, in contrast, includes approaches that exploit reformulation – once again employing either a specialized analytical derivation for a given problem or general purpose techniques – in order to produce a deterministic model that can be solved using existing solvers.

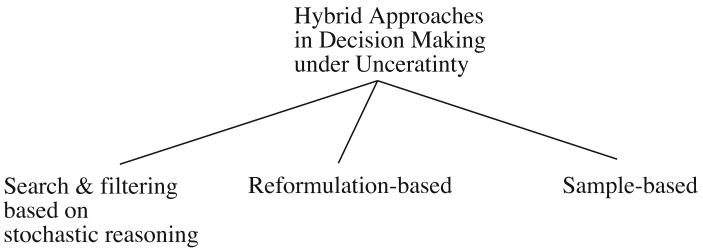


Fig. 6 A classification of hybrid approaches in CP-AI-OR for decision making under uncertainty

- Finally, the third class comprises incomplete approaches that exploit sampling in order to attain a near-optimal solution for problems of optimization under uncertainty. We believe that approaches based on sampling are particularly attractive and deserve a dedicated class. In fact, a high level of complexity is a typical trait of decision problems involving uncertainty, therefore it seems that the only feasible way of tackling many of these problems consists in developing effective approximation strategies.

Before discussing further this classification, it is worth mentioning that we believe it would be impractical to list all existing applications of hybrid methods from CP, AI, and OR in decision making under uncertainty. For this reason, we aim rather to classify the different strategies – and not the specific applications – adopted in the literature for solving this class of problems using hybrid approaches. Nevertheless, for each strategy mentioned in this section, we will report some of the respective applications.

In Sect. 6, we will discuss approaches performing "stochastic reasoning"; in Sect. 7, we will discuss approaches that exploit reformulation; and finally in Sect. 8, we will discuss incomplete approaches that exploit sampling.

6 Approaches Based on Stochastic Reasoning

In this section, we will analyze existing approaches that perform some sort of "stochastic reasoning" by using dedicated – general or special purpose – techniques. These techniques take several different forms: search procedures, filtering algorithms, neural networks, genetic algorithms, etc.

First, we shall distinguish between *general purpose* and *problem specific* strategies (Fig. 7).

General purpose strategies aim to develop frameworks that provide modeling and solving facilities to handle generic problems of decision making under uncertainty. The modeling frameworks proposed in the literature typically aggregate concepts from different domains, for instance *global constraints* from CP, *chance-constraints*

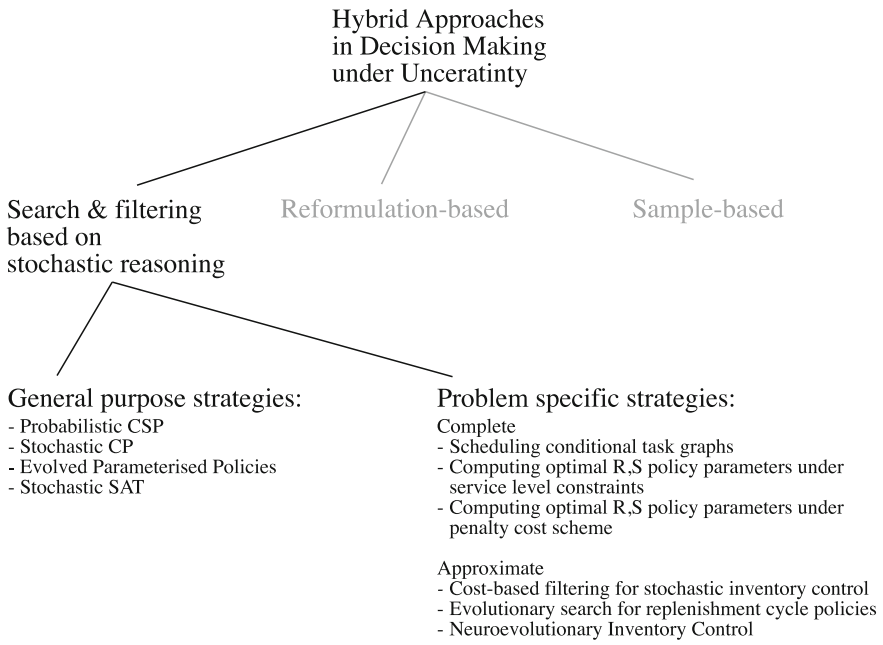


Fig. 7 A classification of hybrid approaches in CP-AI-OR for decision making under uncertainty: approaches based on stochastic reasoning

and *random variables* from SP (OR). These frameworks exploit well established AI strategies, such as forward checking procedures and genetic algorithms in the solution process.

Problem specific strategies typically develop specialized reasoning algorithms that, during the search, are able to perform inference by exploiting the specific structure of the problem. For instance, a typical approach is to encapsulate the reasoning within a dedicated global constraint that prunes decision variable domains according to the underlying stochastic reasoning.

In addition, both general purpose and problem specific strategies may be complete or heuristic. We shall now discuss in more detail these two different classes of approaches based on stochastic reasoning by providing pointers to works in the literature.

6.1 General Purpose Strategies

We survey four different general purpose strategies for modeling and solving different classes of problems of decision making under uncertainty. These are Probabilistic CSP, Stochastic CP, Evolving Parameterized Policies, and Stochastic SAT.

6.1.1 Probabilistic CSP

One of the first general purpose frameworks for modeling uncertainty in CP is the Probabilistic CSP [19]. In the Probabilistic CSP, a distinction is made between *controllable* and *uncontrollable* variables which correspond, respectively, to decision and random variables in SP. As in SP, a PDF is associated with each uncontrollable variable. The authors discuss two different settings. Under the first of these settings, for each of the possible realizations that may be observed for the uncontrollable variables, the best decision is determined. This strategy corresponds to the wait-and-see policy in SP ([32], p. 8) and it presents a posterior analysis. The second setting simply corresponds to a conventional single stage stochastic program where an optimal decision has to be taken before observing the realized values for the uncontrollable variables. The optimal decision, in this second case, is the one that guarantees the maximum likelihood to result feasible with respect to the given PDFs for the uncontrollable variables.

The authors propose two algorithms for solving Probabilistic CSPs. The first algorithm, used for solving problems formulated under the first setting discussed, borrows ideas from solution methods developed in for solving Dynamic CSPs [18] and, in particular, reuses a procedure proposed in [21]. The second proposed algorithm consists of a depth first branch and bound algorithm and of a forward checking procedure. These are employed to solve problems formulated under the second setting discussed.

6.1.2 Stochastic Constraint Programming

The Probabilistic CSP represents the first attempt to include random variables, and thus uncertainty, within the CP framework. Nevertheless, only in [78] a clear link is established between CP and SP with the introduction of SCP. We have already discussed in detail SCP as a modeling framework in Sect. 4.4. In [78], Walsh discusses the complexity of Stochastic CSPs, and proposes a number of complete algorithms and of approximation procedures for solving them. Namely, a backtracking algorithm and a forward checking procedure are proposed, which resemble those proposed in [19] for Probabilistic CSPs. Nevertheless, we want to underscore the fact that the key difference between a Probabilistic CSP and a Stochastic CSP is that the former does not handle multiple decision stages.

In [2] Balafoutis et al. build on the SCP framework introduced in [78], they correct a flaw in the original forward checking procedure for Stochastic CSPs and they also extend this procedure in order to better take advantage of probabilities and thus to achieve stronger pruning. In addition, *arc-consistency* is defined for Stochastic CSPs and an arc-consistency algorithm able to handle constraint of any arity is introduced. Tests are carried on random binary Stochastic CSPs formulated as single and multi-stage problems.

In [13], Bordeaux and Samulowitz investigate two extensions to the original SCP framework. First, they investigate situations in which variables are not ordered

sequentially, corresponding to situations in which the future can follow different branches; they show that minor modifications allow the framework to deal with nonsequential forms. Second, they investigate how to extend the framework in such a way as to incorporate multi-objective decision making. An algorithm is proposed, which solves multi-objective stochastic constraint programs in polynomial space.

Global chance-constraints – which we discussed in Sect. 4.4 – were introduced first in [58], and they bring together the reasoning power of global constraints from CP and the expressive power of chance-constraints from SP. A general purpose approach for filtering global chance-constraints is proposed in [30]. This approach is able to reuse existing propagators available for the respective deterministic global constraint which corresponds to a given global chance-constraint when all the random variables are replaced by constant parameters. In addition, in [57], Rossi et al. discuss some possible strategies to perform cost-based filtering for certain classes of Stochastic COPs. These strategies exploit well-known inequalities borrowed from SP and used to compute valid bounds for any given Stochastic COP that respects some mild assumptions. Examples are given for a simplified version of the SKP previously discussed and for the stochastic sequencing problem discussed in Sect. 3.9.

6.1.3 Evolved Parameterized Policies

Inspired by the success of machine learning methods for stochastic and adversarial problems, a recent approach to Stochastic CSPs/COPs called *Evolved Parameterised Policies* (EPP) is described in [53]. Instead of representing a policy explicitly in a Stochastic Constraint Program, an attempt is made to find a rule that decides, at each decision stage, which domain value to assign to the decision variable(s) at that stage. The quality of a rule can be determined by constructing the corresponding policy tree and observing the satisfaction probability of each chance constraint (and the value of the objective function if there is one). Evolutionary or other nonsystematic search algorithms can be used to explore the space of rules.

EPP treats a Stochastic CSP/COP problem as an unconstrained noisy optimization problem with at worst the same number of (real-valued) variables. This allows a drastic compression of the policy tree into a small set of numbers, and this compression together with the use of evolutionary search makes EPP scalable to large multi-stage Stochastic CSPs/COPs. It has the drawback that only policies of a relatively simple form can be discovered, but it results much more robust than a scenario-based approach on a set of random multi-stage problems [53]. Moreover, arbitrarily complex rules could be discovered by using artificial neural networks instead of these simple functions, a *neuroevolutionary* approach that has been successfully applied to many problems in control [24, 29, 64].

6.1.4 Stochastic SAT

Another general purpose framework for modeling and solving a well established class of problems under uncertainty in AI – and especially in planning under

uncertainty – is Stochastic SAT. We introduced the modeling framework in Sect. 4.1. Current SSAT algorithms fall into three classes: *systematic*, *approximation*, and *non-systematic*.

The systematic algorithms are based on the standard SAT backtracking algorithm – the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [16, 17] – and correspond roughly to some current SCSP algorithms. The first such algorithms were described in [36], in particular, the `evalssat` algorithm for XSSAT which formed the basis for future systematic SSAT algorithms. `evalssat` did not use branching heuristics as in current SAT and CSP solvers, though [36] also used some restricted branching heuristics, but assigned variables in the order specified by the prefix. However, it did use SAT-based techniques (unit propagation and pure variable elimination) and reasoning on the probability threshold θ to prune the search tree. The policy-based SCSP algorithm of [78] is essentially `evalssat` with forward checking. Systematic algorithms have also been devised for special cases of XSSAT. MAXPLAN [45], ZANDER [46] and DC-SSAT [44] use special techniques for planning problems modelled as XSSAT problems.

The `sampleevalssat` approximation algorithm uses random sampling to select paths, then uses SAT techniques to search the restricted tree to maximize θ. The `APPSSAT` algorithm [42] considers scenarios in decreasing order of probability to construct a partial tree for the special case of planning problems modeled as SSAT problems.

The `randevalssat` algorithm [36] is based on the `sampleevalssat` algorithm mentioned above, but applies stochastic local search to the existential variables in a random set of scenarios, thus it is non-systematic. Other ways of applying local search were described in [41], including periodically restarting `randevalssat` with different sampled scenarios, an approach used by the WALKSSAT algorithm [79].

6.2 Problem Specific Strategies

In the previous section, we discussed general purpose solution methods that bring together CP, AI, and OR techniques for decision making under uncertainty. We will now discuss some special purpose approaches proposed in the literature that perform stochastic reasoning on specific problems.

6.2.1 Scheduling Conditional Task Graphs

The work of [40] describes a complete, special purpose approach that concerns the problem – discussed in Sect. 3.2 – of scheduling conditional task graphs. Similarly to the approach in [56], the authors propose an analytical formulation of the stochastic objective function, in this case based on the task graph analysis, and a conditional constraint able to handle such a formulation efficiently. The authors

show the benefit of such an approach by comparing the results with a deterministic model, which disregards uncertainty, and with a scenario-based formulation [71] that requires an exponential number of scenarios to fully represent the stochastic objective function.

6.2.2 Computing Optimal R,S Policy Parameters Under Service Level Constraints

Another special purpose strategy is presented in [58], and proposes a dedicated global chance-constraint for computing replenishment cycle inventory policy parameters under service level constraints. More specifically, the problem considered in this work is the production/inventory problem described in Sect. 3.6. Computing optimal replenishment cycle policy parameters for such a problem is a complex task [69]. By using a dedicated global chance-constraint, the authors were able to perform the complex stochastic reasoning required to compute optimal replenishment cycle policy parameters. Such a complete algorithm performs a numerical integration step in order to compute the real service level provided in each period by a given set of policy parameters and the associated expected total cost.

6.2.3 Computing Optimal R,S Policy Parameters Under a Penalty Cost Scheme

Similarly, a dedicated global constraint has been proposed in [56] in order to solve to optimality the problem of computing optimal replenishment cycle policy parameters under a penalty cost scheme. Such a problem has been investigated in [70], but in this work, the authors could only solve the problem in a heuristic way, by employing a piecewise linear approximation of the convex cost function in the problem in order to build up a deterministic equivalent MIP model. In [56], the authors were able to embed a closed-form non-linear analytical expression for such a convex cost function within a global constraint, thus obtaining a complete model able to compute optimal replenishment cycle policy parameters.

6.2.4 Cost-based Filtering for Stochastic Inventory Control

The work in [68] has a different flavor. In this case, the underlying model is the deterministic equivalent CP formulation proposed in [73] for computing near-optimal replenishment cycle policy parameters under service level constraints. The CP formulation was originally proposed as a reformulation of the MIP model in [69]. Such a reformulation showed significant benefits in terms of efficiency. The authors, in [68], propose three independent cost-based filtering strategies that perform stochastic reasoning and that are able to significantly speed up the search when applied to the original CP model in [73].

6.2.5 Evolutionary Search for Replenishment Cycle Policies

A recent application of a genetic algorithm to a multi-stage optimization problem in inventory control is described in [51]. Each chromosome represents a replenishment cycle policy plan as a list of order-up-to levels, with a level of 0 representing no order, and the fitness of a chromosome is averaged over a large number of scenarios. This approach is enhanced in [50] by hybridizing the genetic algorithm with the SARSA temporal difference learning algorithm [61]. This is shown to greatly improve the performance of genetic search for replenishment cycle policies, both with and without order capacity constraints.

6.2.6 Neuroevolutionary Inventory Control

One may evolve an artificial neural network to optimally control an agent in an uncertain environment. The network inputs represent the environment and its outputs the actions to be taken. This combination of evolutionary search and neural networks is called *neuroevolution*. A recent paper [54] applies neuroevolution to find optimal or near-optimal plans in inventory control, following no special policy. The problems are multi-stage and involve multi-echelon systems (they have more than one stocking point). Such problems have no known optimal policy and rapidly become too large for exact solution. The inputs to the network are the current stock levels and the outputs are the order quantities.

7 Reformulation-Based Approaches

In this section, we will analyze existing approaches that are based on a reformulation that produces a deterministic model, which can be solved using an existing solver.

Once more, we shall distinguish between *general purpose* and *problem specific* strategies (Fig. 8).

Hybrid general purpose reformulation strategies have recently appeared especially at the borderline between CP and OR. These typically take the form of a high level language – such as Stochastic OPL – used to formulate the problem under uncertainty, and of a general purpose compiler that can handle the high level stochastic model and produce a compiled deterministic equivalent one. Often, the compilation relies on a well known technique in SP: *scenario-based modeling*. In addition, due to the complexity of stochastic programs in general, approximation strategies are often proposed in concert with these general purpose frameworks in order to make the size of the compiled model manageable.

In contrast, problem specific strategies aim to fully exploit the structure of the problem in order to produce a deterministic – and possibly equivalent – model that can be handled efficiently by existing solver. In many cases, in order to obtain a model that is manageable by existing solvers, it is necessary to introduce

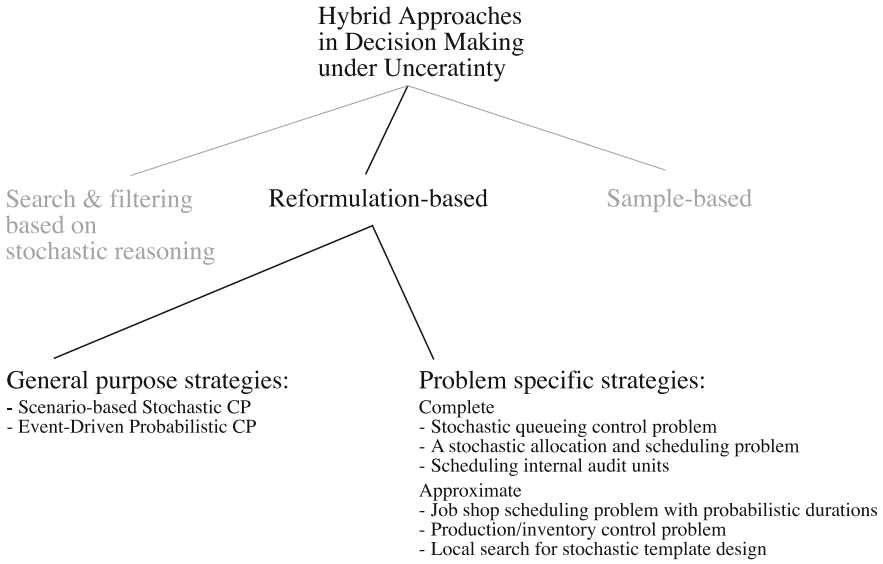


Fig. 8 A classification of hybrid approaches in CP-AI-OR for decision making under uncertainty: approaches based on a deterministic reformulation

some assumptions that affect the completeness and, thus, the quality of the solution found in the deterministic model. We will provide examples of applications in which a special purpose deterministic equivalent model is built, which is equivalent to the original model and also examples in which the deterministic model can only approximate the original stochastic model.

7.1 General Purpose Strategies

We survey two different general purpose strategies based on reformulation for modeling and solving classes of problems of decision making under uncertainty. These are Scenario-based Stochastic CP and Event-Driven Probabilistic Constraint Programming.

7.1.1 Scenario-Based Stochastic Constraint Programming

The first general purpose framework based on reformulation that we present is Scenario-based Stochastic Constraint Programming, which was proposed by Tarim et al. in [71]. The novelty in this work is the fact that the authors adopt a semantics for stochastic constraint programs based on scenario trees. By using this semantics, the authors can compile stochastic constraint programs into conventional (non-stochastic) constraint programs and they can therefore use existing constraint solvers to effectively solve this class of problems.

In a scenario based approach – frequently used in SP [11] – a scenario tree is generated which incorporates all possible realizations of discrete random variables into the model explicitly. A path from the root to an extremity of the event tree represents a scenario. With each scenario, a given probability is associated. Within each scenario, we have a conventional (non-stochastic) constraint program to solve. All we need to do is to replace the random variables by the values taken in the scenario, and ensure that the values found for the decision variables are consistent across scenarios, as certain decision variables are shared across scenarios. Constraints are defined (as in traditional constraint satisfaction) by relations of allowed tuples of values, and can be implemented with specialized and efficient algorithms for consistency checking. Furthermore, the scenario-based view of stochastic constraint programs also allows later-stage random variables to take values which are conditioned by the earlier-stage random variables. This is a direct consequence of employing the scenario representation, in which random variables are replaced with their scenario dependent values.

Scenario-based SCP has been outlined in Sect. 4.4. Tarim et al. [71] not only defined a general way to compile stochastic constraint programs into conventional constraint programs, but they also proposed a language, Stochastic OPL, which is based on the OPL constraint modeling language [28]. Using this language, the authors modeled optimization problems under uncertainty from a variety of fields, such as portfolio selection, agricultural planning, and production/inventory management (Sect. 3.6). We will not discuss the language in detail, but in the Appendix, we show how to model the single and multi-stage SKP problems of Sect. 2 by using the Stochastic OPL.

Among the benefits of the scenario based approach in [71] is the fact that it allows multiple chance-constraints and a range of different objectives to be modeled. The authors point out that each of these changes would require substantial modifications in the backtracking and forward checking algorithms proposed in [78]. The scenario based view allows each of these extensions to be modeled easily using stochastic OPL, compiled down into standard OPL, and solved by means of existing solvers. It should be noted that the approach is general and the compilation need not necessarily be performed using OPL, but it can be implemented using any available CP language and/or software package. The main drawback of this approach is the fact that the scenario tree required to model a given problem grows exponentially in size when random variable domains are large, thus leading to large models that are difficult to solve.

In addition to this general purpose modeling/solving framework the authors also proposed some techniques to improve the efficiency of the solution process. In order to do so, they proposed scenario reduction techniques, such as Monte Carlo Sampling or Latin Hypercube Sampling [65], to reduce the number of scenarios considered in the model. Their experimental results show the effectiveness of this approach, which in practice is able to find high quality solutions using a small number of scenarios. Finally, inspired by robust optimization techniques used in OR [35], the authors also proposed some techniques to generate robust solutions, that is, solutions that adopt similar (or the same) decisions under different scenarios.

7.1.2 Event-Driven Probabilistic Constraint Programming

We now briefly discuss a second general purpose framework based on reformulation: Event-Driven Probabilistic Constraint Programming [67]. This framework was introduced to address different problems than those for which SCP is a suitable modeling tool. Event-Driven Probabilistic Constraint Programming, as the name suggest, is connected to Probabilistic CSPs and, mainly, to Dependent-chance Programming [37, 38].

Sometimes, a complex probabilistic decision system undertakes multiple tasks, called *events* here, and the decision-maker wishes to maximize chance functions which are defined as the probabilities of satisfying these events. This is especially useful in situations where a particular measure of the "reliability" or "robustness" of a given plan has to be maximized. The Event-Driven Probabilistic Constraint Programming modeling framework allows users to designate certain probabilistic constraints, involving both decision and random variables, as *events* whose chance of satisfaction must be maximized, subject to hard constraints which should be always satisfied, and also logical dependencies among constraints. Event-Driven Probabilistic Constraint Programming builds on Dependent-chance Programming and provides more expressiveness to the user, in order to capture a more realistic and accurate measure of plan reliability [59]. It also provides an exact solution method, employing scenario-based reformulation, in contrast to the approximate genetic algorithm in [38].

7.2 Problem Specific Strategies

We now discuss some problem specific strategies based on deterministic equivalent reformulations.

7.2.1 Stochastic Queueing Control Problem

In [74, 75], the authors propose a set of deterministic equivalent CP models for solving the stochastic queueing control problem discussed in Sect. 3.1. [75] not only provides the first application of CP to solve a stochastic queueing control problem but it also provides a complete approach for a problem for which only a heuristic algorithm [8] existed. Three deterministic equivalent constraint programming models and a shaving procedure are proposed. The complete models provide satisfactory performances when compared with the heuristic procedure, which nevertheless remains superior in terms of solution quality over time. A hybrid method is therefore proposed, which combines the heuristic in [8] with the best constraint programming method. Such a hybrid approach performs better than either of these approaches separately.

The interesting aspect of this work is that, as in [60], all the stochastic information is encoded as constraints and expected values, and there is no need of random variables or scenarios. The three models proposed explore different sets of variables and different configurations for the constraint set, for instance using duality. Nevertheless, all the three models use predefined constraints available in standard CP solvers.

7.2.2 A Stochastic Allocation and Scheduling Problem

The problem, discussed in [39], is the scheduling problem described in Sect. 3.2 applied to multiprocessor systems on chip: given a conditional task graph characterizing a target application and a target architecture, with alternative memory and computation resources, the authors compute an allocation and schedule that minimize the expected value of communication costs, since – as they point out – communication resources are one of the major bottlenecks in modern multiprocessor systems on chips. The approach they propose is complete and efficient. As in the previous cases, it is based on a deterministic equivalent reformulation of the original stochastic integer linear programming model. More specifically, the authors employ logic-based Benders' decomposition. The stochastic allocation problem is solved through an Integer Programming solver, while the scheduling problem with conditional activities is handled with CP. The two solvers interact through nogoods. Once more, one of the main contributions is the derivation of an analytical deterministic expression employed in order to compute the expected value of communication costs in the objective function. This expression makes it possible for the authors to transform the original stochastic allocation problem into a deterministic equivalent one that can be solved using any available Integer Programming solver.

7.2.3 Scheduling Internal Audit Units

In [60], the authors analyze the problem of scheduling internal audit units discussed in Sect. 3.8. A stochastic programming formulation is proposed with Mixed Integer Linear Programming and CP certainty-equivalent models. Both the models transform analytically the chance-constraints in the model into deterministic equivalent ones. In experiments, neither approach dominates the other. However, the CP approach is orders of magnitude faster for large audit times, and almost as fast as the MILP approach for small audit times.

Finally, we discuss works in which the deterministic model obtained through reformulation for a given stochastic program is not "equivalent"; rather, it is based on some simplifying assumption, which makes it possible to obtain a compact deterministic formulation able to provide a near-optimal solution and an approximate value for the cost of such a solution, or a bound for such a cost.

7.2.4 Job Shop Scheduling with Probabilistic Durations

In [3], an approximate deterministic reformulation is employed to compute valid bounds to perform cost-based filtering. In this work, the authors analyze the JSP problem discussed in Sect. 3.4, in which the objective is to find the minimum makespan. In contrast to the classic formulation presented in [23], in [3] the authors assume that the job durations are probabilistic. The objective is therefore accordingly modified to account for uncertainty. More specifically, the authors search for a proactive plan, consisting of a partial order among activities and of resource-activity allocations, which attains the lowest possible makespan with probability greater or equal to a given threshold. For this problem, the authors propose a deterministic formulation, which depends on a given nonnegative parameter q. A correct choice of such a parameter guarantees that the minimum makespan for the deterministic model is a lower bound for the minimum makespan that can be attained with a certain threshold probability in the original model. This deterministic model can be efficiently solved with classic constraint programming techniques and can provide tight bounds at each node of the search tree that are employed to perform cost-based filtering. A number of heuristic techniques are proposed for correctly choosing a "good" value for the parameter q.

7.2.5 Production/Inventory Control Problem

Consider the production/inventory problem discussed in Sect. 3.6. The deterministic reformulation proposed in Tarim et al. [73] relies on some mild assumptions – discussed in [69] – concerning order-quantities. Under these assumptions, it was possible for the authors to obtain analytical deterministic expressions for enforcing the required service level in each period of the planning horizon, and to compute the expected total cost associated with a given set of policy parameters. By using these expressions, it was possible for the authors to formulate a deterministic model by employing standard constraints available in any CP solver. In [58], the authors compare the solutions obtained through a complete formulation with those obtained with the model in [73]. This comparison shows that the assumptions do not significantly compromise optimality, whereas they allow the construction of a model that can significantly outperform the complete one and solve real-world instances comprising long planning horizons and high demand values.

7.2.6 Local Search for Stochastic Template Design

In [52], the stochastic template design problem discussed in Sect. 3.7 is reformulated as a deterministic equivalent constrained optimization problem, using all possible scenarios and a novel modeling technique to eliminate non-linear constraints. The result is a standard integer linear program that proved to be hard to solve by

branch-and-bound. However, a local search algorithm design for linear integer programs performed very well, and was more scalable than the Bender's decomposition algorithm in [72].

8 Approaches Based on Sampling

In this section, we will discuss sample-based approximation strategies for solving problems of decision making under uncertainty. Due to the complexity of these problems in general, several works in the literature have been devoted to analyzing the effectiveness of heuristic approaches based on sampling. In Fig. 9, it is possible to observe how three main trends have been identified in the CP and AI literature, which apply sampling in a hybrid setting for solving problems of decision making under uncertainty: the Sample Average Approximation approach (SAA), Forward Sampling, and Sample Aggregation.

- In OR, and particularly in SP, the state-of-the-art technique that applies sampling in combinatorial optimization is the Sample Average Approximation approach [34]. In this approach, a given number of samples is drawn from the random variable distributions, and the combinatorial problem of interest is repeatedly solved by considering different samples as input in each run. The real expected

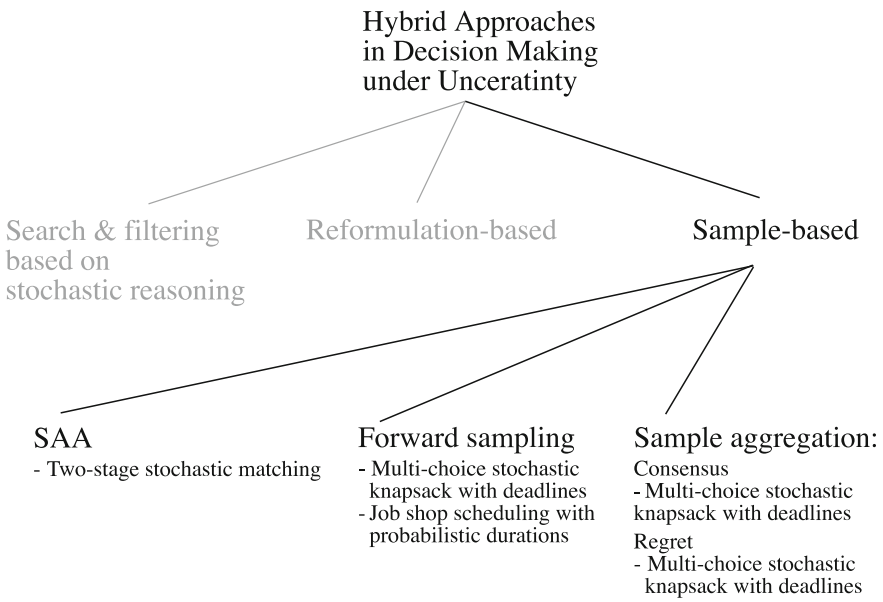


Fig. 9 A classification of hybrid approaches in CP-AI-OR for decision making under uncertainty: approaches based on sampling

cost/profit of a solution produced for a given sample is then computed by simulating a sufficient number of samples. Among all the solutions computed, the one that provides the minimum expected cost (or the maximum expected profit) is retained. Two criteria are given by the authors: one for deciding when a given sample size is no more likely to produce better solutions, and another for deciding if the increasing sample size may lead to better solutions.

- Forward sampling, as the name suggests, is a sort of forward checking that employs samples in order to make inference about which values are not consistent in decision variable domains or about the expected cost/profit of associated with a given (partial) assignment for decision variables, which is assessed against the generated samples by computing, for instance, the expected profit/cost of such an assignment with respect to these samples.
- Sample aggregation is a strategy in which a number of samples are generated, for each of these samples a deterministic problem is solved, then the results obtained for all these samples are aggregated and analyzed according to some rule. The "best" among these decisions is implemented in practice. For instance, a possible rule may always choose the decision that is optimal for the highest number of samples.

In the CP and AI literature, sampling is often applied in concert with a so called "online" optimization strategy. Online refers to the fact that decisions and observations are interleaved in the problem, and each time an observation occurs an optimization step takes place to compute the next decision, by taking into account the PDF of future random variables and the observed values for the past ones. It is easy to notice that a multi-stage stochastic program subsumes an online strategy if the decision maker has a complete knowledge of the PDF of the random variables in the problem. In this case, we may compute the entire solution tree at the beginning, and use it in order to find the best following decision each time a random variable is observed. Nevertheless, several reasons justify the use of an online strategy (also called a "rolling horizon" approach in the OR literature and especially in Inventory Control). The most compelling reason for using an online approach is that it does not require the decision maker to have a complete knowledge of the PDFs of the random variables. Consider, for instance, the SKP introduced in the previous sections. If the problem is formulated as a multi-stage stochastic program and we have a full knowledge about the possible weights that can be observed for all the objects, the policy tree will prescribe exactly what to do in each possible future course of action. Nevertheless, if at some stage one of the objects takes a weight that is not part of the PDF we considered for such an object, the policy tree will not be able to prescribe an appropriate action. In contrast, an online approach would simply take into consideration this weight in the following optimization step and it would, however, provide a valid decision to be implemented next.

Stochastic problems solved using online strategies, and to which either forward sampling or sample aggregation strategies are applied, appear in a number of works within the CP and AI literatures. In what follow, we shall classify some of these works on the basis of which sampling technique is applied.

8.1 Sample Average Approximation

In this section, we provide a pointer to a work that proposes to apply SAA to a modified version of a classic matching problem: the two-stage stochastic matching problem.

8.1.1 Two-stage Stochastic Matching Problem

In [33], Katriel et al. consider the two-stage stochastic matching problem discussed in Sect. 3.5. The authors prove lower bounds and analyze efficient strategies. We do not provide here a general survey for this work, as the reader may refer to the cited article for more details. Instead, we focus on one of the authors' contributions in which they firstly observe that, in this problem, with independently activated vertices the number of scenarios is extremely large. However, in such a situation, there is often a black box sampling procedure that provides, in polynomial time, an unbiased sample of scenarios; then they observe that one can use the SAA method to simulate the explicit scenarios case and, under some mild assumptions, obtain a tight approximation guarantee. The main observation is that the value of the solution defined by taking a polynomial number of samples of scenarios tightly approximates the value of the solution defined by taking all possible scenarios.

8.2 Forward Sampling

In this section, we survey two relevant works in which forward sampling is applied: the multi-choice stochastic knapsack with deadlines and the JSP with probabilistic durations.

8.2.1 Multi-Choice Stochastic Knapsack with Deadlines

In [5], the authors analyze different techniques for performing online stochastic optimization. A benchmark problem is proposed in order to assess all these different techniques. The benchmark is stemmed from the authors' industrial experience and it consist of a Multi-Choice Stochastic Knapsack with Deadlines. This problem corresponds, in practice, to the stochastic reservation problem discussed in Sect. 3.3 and it is used to test four different online strategies exploiting combinations of the stochastic and combinatorial aspects of the problem. These strategies are forward sampling, average values, most likely scenario analysis, and yield management techniques.

Initially, the authors propose two naive order handling policies: a first-come/first-serve policy and a best-fit policy. Furthermore, in order to assess the quality

of a given policy, the authors also discuss "far seeing" strategies, which assume advanced knowledge of the realized demand and can therefore solve the associated deterministic multi-choice knapsack problem.[4]

One of the strategies used in this work to estimate the quality of a given policy – for instance first-come/first-serve or best-fit – employs forward sampling in order to generate samples from the current date to the end of the planning horizon. The evaluation of a sample can be done, for instance, by simulating the behavior of a best-fit strategy for the specific sample. The policy evaluation then will be a measure (for instance the average) over the evaluations of many generated samples.

8.2.2 Job Shop Scheduling with Probabilistic Durations

Forward sampling is also employed in [3]. We recall that in this work the authors analyze the JSP problem discussed in Sect. 3.4, in which the authors assume that the job durations are probabilistic. A number of algorithms are proposed for solving this problem through sampling. First, a branch-and-bound procedure is introduced, which exploits at each node of the search tree a Monte Carlo simulation approach to compute – with respect to the partial assignment associated with such a node – a valid lower bound for the minimum possible makespan that may be attained with a probability greater or equal to the given threshold. Since sampling is employed for computing the bound, confidence interval analysis is employed to estimate if the attainment probability associated with the given makespan is a sufficiently reliable estimate. Second, the authors propose a number of heuristic techniques that aim to limit the amount of time spent on Monte Carlo simulation during the search by using the deterministic makespan as an oracle for selecting and simulating only the most promising plans in order to save CPU time and to dedicate more time to the exploration of the search space rather than on simulating non-promising plans. Finally, dedicated tabu search strategies are proposed in order to introduce a valid alternative to the constructive search techniques above, which are mainly based on tree-search.

8.3 Sample Aggregation

In this section, we discuss works in which two alternative sample aggregation strategies are employed: the "Consensus" strategy and the "Regret" strategy. The problem to which these strategies are applied is, once more, the multi-choice stochastic knapsack with deadlines.

[4] We recall that in SP this corresponds to using a wait-and-see policy and performing a posterior analysis.

8.3.1 Multi-Choice Stochastic Knapsack with Deadlines

In [27], the authors consider the same Online Multi-Choice Knapsack with Deadlines problem considered in [5]. In order to solve this problem, the authors employ the following online algorithm. The algorithm receives a sequence of online requests and starts with an empty allocation. At each decision point, the algorithm considers the current allocation and the current request, and chooses a bin in which to allocate the request, which is then included in the current assignment. Eventually, the algorithm returns the final allocation and the respective value. In order to decide in which bin to allocate a given request, the algorithm employs a function "chooseAllocation" which is based on two black boxes: a function "getSample" that returns a sample of the arrival distribution; and a function "optSol" that, given the current assignment and a request, returns an optimal allocation of the request by taking into account the past decisions. The authors then consider four possible options for implementing "chooseAllocation":

- The best-fit strategy discussed in [5].
- A strategy called "Expectation" – in practice performing a forward sampling — that generates future requests by sampling and that evaluates each possible allocation for a given request (i.e., in which bin to fit such a request) against the samples.
- A strategy called "Consensus", which was introduced in [47], and whose key idea is to solve each sample only once. More specifically, instead of evaluating each possible bin at a given time point with respect to each sample, "consensus" executes the optimization algorithm only once per sample. The bin to which the request is eventually allocated by this optimization step is then credited with the respective profit, while the other bins receive no credit. The algorithm eventually returns the bin with which the highest profit is associated.
- A strategy called "Regret" [6,7] based on a sub-optimality approximation, which is a fast estimation of the loss caused by sub-optimal allocations. The key steps in the process of choosing a bin resemble the "consensus" algorithm. But in "regret," instead of assigning some credit only to the bin selected by the optimal solution, the suboptimality approximation is used to compute, for each possible request allocation, an approximation of the best solution that makes such a choice. Therefore, every available bin is given an evaluation for every sample at a given time, at the cost of a single optimization.

Consensus and regret are two examples of what we previously defined as "sample aggregation" strategies.

9 Related Works

In this section, we will first briefly discuss Stochastic Dynamic Programming, a related and well established technique in OR that deals with decision making under uncertainty. We will also clarify why this technique has not been covered in the

former sections. Second, we will cast our work within a broader picture, and contrast our survey with existing similar works that address the topics of uncertainty and change.

9.1 Stochastic Dynamic Programming

An alternative and effective technique for modeling problems of decision making under uncertainty is Dynamic Programming. In [4], Bellman explicitly states that Dynamic Programming was initially conceived for modeling multi-stage decision processes. He also argues that these processes arise in practice in a multitude of diverse field and in many real life problems, for instance in stock control, scheduling of patients through a medical clinic, servicing of aircraft at an airfield, etc. Dynamic Programming has been applied to a multitude of deterministic multi-stage decision problems, but in [4], Bellman also discussed its application to stochastic multi-stage decision processes. As in the deterministic case, in the stochastic case, the modeling also relies mainly on the development of adequate functional equations capturing the dynamics of the system, and the expected cost (or profit) function associated with the possible decisions and affected by the random variables in the problem. The multi-stage decision process, in Dynamic Programming, is typically defined recursively, starting from a bounding condition that describes a degenerate state of the system that can be easily characterized. Depending on the specific nature of the process being analyzed (Markovian, Semi-Markovian, etc. – see [25], Chapter "Constraint Programming and Local Search Hybrids"), it is possible to exploit its structure to devise efficient solution methods or closed form solutions for the optimal control policy, which corresponds to the policy tree that constitutes a solution of a given Stochastic Program.

In this work, we mainly focused on the connections between and integration of SP, CP, and AI. So far Dynamic Programming has not played a role as significant as SP in the development of hybrids approaches for decision making under uncertainty. For this reason, Stochastic Dynamic Programming, and its extension to infinite horizon case Markov Decision Processes are not thoroughly covered here. For more details on Stochastic Dynamic Programming the reader may refer to the seminal work of Bellman [4], and to the works of Bertsekas [9], Warren [49], Sutton and Barto [66], and Gosavi [25].

9.2 Related Modeling Frameworks

Recently, the topic of decision making under *uncertain* and *dynamic* environment has been discussed in two literature surveys [14, 76]. Nevertheless, these two works discuss a variety of different problems that can hardly be classified within a unique group. For instance, consider a problem whose structure changes dynamically over time. As an example, we may refer to the Dynamic Constraint Network discussed in [18], in which, from time to time, new facts that become known about the model

induce a change in the constraint network. We find that such a problem has almost nothing in common with a problem where some parameters are random – thus may assume a certain value with a given probability – and a decision has to be taken proactively, before the realized values for these parameters are known. As an example for this second class, we may consider the proactive JSP problem discussed in [3], in which an optimal plan – that achieves a minimum makespan with a certain probability – has to be determined before the actual job durations are known. Also consider, as in [22], a CSP in which we allow some of the constraints to be violated by a solution, and in which we search for a solution that tries to satisfy the original constraint problem as much as possible; or, alternatively, consider a CSP, as in [26], in which some of the values in the decision variable domains may suddenly become unavailable after a solution has been computed and for which we are looking for robust solutions that can be "repaired" with little effort. These two latter examples, again, significantly differ from the previous ones and among each others.

A clear and comprehensive classification of all these different problems and frameworks is still missing. For this reason, in this section, we propose a classification in three distinct classes, and we try to position in each of these classes some of the frameworks proposed in the literature.

In our classification (Fig. 10), there are three criteria based on which a particular framework is classified: Degree of Change, Degree of Satisfiability, and Degree of Uncertainty.

- With respect to the *Degree of Change*, "static" refers to a classic, static CSP, while "dynamic" refers to the fact that the model is assumed to change dynamically, since constraints are added/removed. The solution has to be flexible enough to be adapted to these changes without too many modifications and with limited computational effort. Existing frameworks that, with respect to the Degree

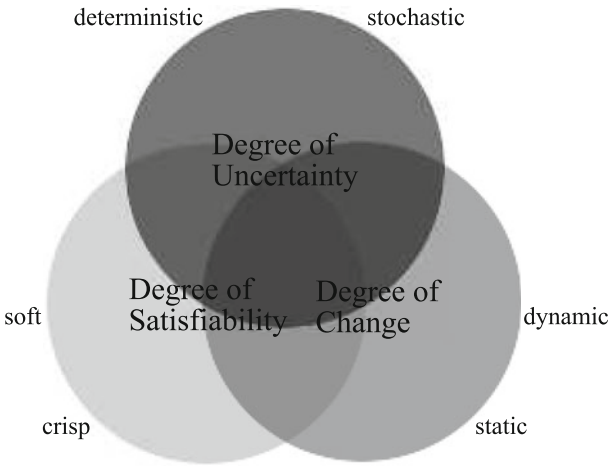


Fig. 10 A classification for existing frameworks based on problem structure

of Change, are classified as "dynamic" are Dynamic Constraint Satisfaction (Dechter [18]); Conditional CSP (Minton et al. [48]); and Super-solutions in CP (Hebrard et al. [26]).

- With respect to the *Degree of Satisfiability* "crisp" refers to a classic CSP in which all the constraints have to be satisfied by a given solution, while "soft" refers to the fact that some of the constraints in the model may be violated by a solution. The aim is to find a solution that typically violates the minimum number of constraints or that, in general, minimizes some violation measure. Existing frameworks that, with respect to the Degree of Satisfiability, are classified as "soft" are Partial Constraint Satisfaction (Freuder [20]); Constraint solving over semi-rings (Bistarelli et al. [12]); and Valued Constraint Satisfaction (Schiex et al. [63]).
- With respect to the *Degree of Uncertainty*, "deterministic" refers to classic CSPs, while "stochastic" refers to the existence of uncontrollable (random) variables in the model for which a probability distribution is given. Stochastic problems present an alternation of decisions and observations. Constraints are assigned a satisfaction threshold that must be met by any given solution.

Some of the frameworks presented in the literature do, in fact, cover more than one of the classes presented, and for this reason, the circles are intersecting each others. Clearly, this classification does not cover several other frameworks that in the years have been proposed to deal with other problem classes.

We have introduced pointers to relevant frameworks that can be either classified under Degree of Change ("dynamic") or Degree of Satisfiability ("soft"). Problems that are classified as "stochastic" with respect to their Degree of Uncertainty have been widely surveyed in the former part of this work. We argue that such a classification better positions the existing works with respect to aspects that are, in fact, orthogonal among each others.

10 Conclusions

In this survey, we focused on hybrid CP-AI-OR methods for decision making under uncertainty. First, we explicitly defined what "uncertainty" is and how it is possible to model by using SP, a well established existing modeling framework in OR. We surveyed additional existing frameworks – one from AI and one from CP – for modeling problems of decision making under uncertainty and we also identified the relevant connections among these frameworks. Second, we introduced a list of problems from the literature in which uncertainty plays a role and we categorized existing hybrid techniques that have been proposed for tackling these problems into three classes. In the first class, we identified general and special purpose approaches that perform "stochastic reasoning". In the second class, we listed approaches, once more general and special purpose, that use reformulation. In the third class, we categorized approximate techniques based on a variety of strategies employing sampling. Finally, we pointed out connections with other related works.

Appendix

In [71] Stochastic OPL, a language for modeling stochastic constraint programs, is proposed. We will now show how the single and multi-stage SKP problems introduced in Sect. 2 can be easily modeled using such a language.

In Fig. 11, the Stochastic OPL model for the single stage SKP is presented. As in the model presented in Fig. 1, the objective function maximizes the revenue brought by the objects in the knapsack minus the expected penalty for exceeding capacity. Chance-constraint `prob(sum(i in Items) W[i]*x[i] <= c) >=` θ ensures that the capacity is not exceeded with a probability higher than θ.

We now refer to the numerical Example 1 for SKP. In Fig. 12, the Stochastic OPL data file corresponding the numerical instance in Example 1 is presented. We recall that the strategy proposed in [71] employs a scenario-based formulation. In fact, it is easy to see that, given the random variables in the example and the values in their domains, there are a total of 32 scenarios that should be considered. Each row for variable `W` in Fig. 12 has, in fact, 32 entries (i.e. `[<10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8,8>]`). There are in total 5 rows, each having 32 entries, so a column – containing all the entries at the same position in each row – therefore, fully encodes one of the possible 32 scenarios. The probability of each of the 32 scenarios is provided using the array `myrand`. By using the compilation strategy proposed in [71], any model and data file written using Stochastic OPL can be easily compiled into a classic (deterministic) constraint program and solved by using classic solvers. The optimal solution for Example 1 – computed using

```
int k = ...;
int p = ...;
int c = ...;
float θ = ...;
range Items 1..k;
range onestage 1..1;
stoch myrand[onestage]=...;
float W[Items,onestage]^myrand = ...;
float r[Items] = ...;
dvar float+ z;
dvar int x[Items] in 0..1;

maximize sum(i in Items) x[i]*r[i] - expected(p*z)
subject to{
   z >= sum(i in Items) W[i]*x[i] - c;
   prob(sum(i in Items) W[i]*x[i] <= c) >= θ;
};
```

Fig. 11 Stochastic OPL formulation for the single-stage SKP

```
k = 5;
p = 2;
c = 30;
θ = 0.6;
W = [
            [<10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,10,8,8,8,
              8,8,8,8,8,8,8,8,8,8,8,8,8>],
            [<9,9,9,9,9,9,9,9,12,12,12,12,12,12,12,12,9,9,9,9,9,9,
              9,9,12,12,12,12,12,12,12,12>],
            [<8,8,8,8,13,13,13,13,8,8,8,8,13,13,
              13,13,8,8,8,8,13,13,13,13,8,8,8,8,13,13,13,13>],
            [<4,4,6,6,4,4,6,6,4,4,6,6,4,4,6,6,4,4,
              6,6,4,4,6,6,4,4,6,6,6,4,4,6,6>],
            [<12,15,12,15,12,15,12,15,12,15,12,15,12,15,12,15,
              12,15,12,15,12,15,12,15,12,15,12,15,12,15,12,15>]
];
myrand = [
            <0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),
            0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),
            0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),
            0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),
            0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),
            0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),
            0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),
            0.0 (0.03125),0.0 (0.03125),0.0 (0.03125),0.0 (0.03125)>
];
r = [16,16,16,5,25];
```

Fig. 12 Stochastic OPL Data File for the single-stage SKP

the compiled OPL code obtained from the Stochastic OPL model and data file presented – selects items $\{1, 4, 5\}$ and achieves an expected profit of 45.75, as shown in Fig. 2.

The SKP can be also formulated as a multi-stage stochastic constraint program as shown in Fig. 3. In Fig. 13, the Stochastic OPL model for the multi-stage SKP is presented. The model is similar to the one presented in Fig. 11. Nevertheless, now, the weight of each object is observed at a different decision stage. Therefore, we have an array of k random variables (`stoch W[Items]`) in contrast to the previous model that only had one random variable (`myrand`) to model the probability distribution of the possible scenarios. In Fig. 14, the data file corresponding to the numerical instance in Example 1 is presented. The optimal solution for Example 1, when the problem is formulated as a multi-stage Stochastic COP, can be computed using the compiled OPL code obtained from the Stochastic OPL model in Fig. 13 and from the data file presented in Fig. 14. This solution takes the form of a policy tree – graphically rendered in Fig. 4 – and achieves an expected profit of 47.75.

```
int k = ...;
int p = ...;
int c = ...;
float θ = ...;
range Items 1..k;
stoch W[Items]=...;
float r[Items] = ...;
dvar float+ z;
dvar int x[Items] in 0..1;

maximize expected(sum(i in Items) x[i]*r[i]) - p*expected(z)
subject to{
   z >= sum(i in Items) W[i]*x[i] - c;
   prob(sum(i in Items) W[i]*x[i] <= c) >= θ;
};
```

Fig. 13 Stochastic OPL formulation for the multi-stage SKP

Fig. 14 Stochastic OPL Data File for the multi-stage SKP

```
k = 5;
p = 2;
c = 30;
θ = 0.6;
W = [
          <10(0.5),8(0.5)>,
          <9(0.5),12(0.5)>,
          <8(0.5),13(0.5)>,
          <4(0.5),6(0.5)>,
          <12(0.5),15(0.5)>
];
r = [16,16,16,5,25];
```

Acknowledgements S. Armagan Tarim and Brahim Hnich are supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under Grant No. SOBAG-108K027. S. Armagan Tarim is supported also by Hacettepe University BAB.

R. Rossi has received funding from the European Community's Seventh Framework Programme (FP7) under grant agreement no 244994 (project VEG-i-TRADE). S.A. Tarim and Brahim Hnich are supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under SOBAG-1001. S.A. Tarim is supported by Hacettepe University-BAB.

References

1. Apt K (2003) Principles of constraint programming. Cambridge University Press, Cambridge
2. Balafoutis T, Stergiou K (2006) Algorithms for stochastic csps. In: Benhamou F (ed) Principles and practice of constraint programming, CP 2006, Proceedings. Lecture notes in computer science, vol 4204. Springer, Heidelberg, pp 44–58

3. Beck JC, Wilson N (2007) Proactive algorithms for job shop scheduling with probabilistic durations. J Artif Intell Res 28:183–232
4. Bellman RE (1957) Dynamic Programming. Princeton University Press, Princeton
5. Benoist T, Bourreau E, Caseau Y, Rottembourg B (2001) Towards stochastic constraint programming: a study of online multi-choice knapsack with deadlines. In: Walsh T (ed) Principles and practice of constraint programming, CP 2001, Proceedings. Lecture notes in computer science, vol 2239. Springer, Heidelberg, pp 61–76
6. Bent R, Van Hentenryck P (2004) Regrets only! online stochastic optimization under time constraints. In: Proceedings of the nineteenth national conference on artificial intelligence, sixteenth conference on innovative applications of artificial intelligence, San Jose, California, 25–29 July 2004, pp 501–506
7. Bent R, Katriel I, Van Hentenryck P (2005) Sub-optimality approximations. In: van Beek P (ed) Principles and practice of constraint programming- CP 2005. 11th International Conference, Sitges, Spain, 1–5 October 2005. Lecture notes in computer science, vol 3709. Springer, Heidelberg, pp 122–136
8. Berman O, Wang J, Sapna KP (2005) Optimal management of cross-trained workers in services with negligible switching costs. Eur J Oper Res 167(2):349–369
9. Bertsekas DP (1995) Dynamic programming and optimal control. Athena Scientific, Belmont
10. Bianchi L, Dorigo M, Gambardella L, Gutjahr W (2009) A survey on metaheuristics for stochastic combinatorial optimization. Nat Comput 8(2):239–287
11. Birge JR, Louveaux F (1997) Introduction to stochastic programming. Springer Verlag, New York
12. Bistarelli S, Montanari U, Rossi F (1995) Constraint solving over semirings. In: Proceedings of the fourteenth international joint conference on artificial intelligence, IJCAI '95, pp 624–630
13. Bordeaux L, Samulowitz H (2007) On the stochastic constraint satisfaction framework. In: SAC '07: Proceedings of the 2007 ACM symposium on applied computing, New York, pp 316–320
14. Brown KN, Miguel I (2006) Uncertainty and change. In: Rossi F, van Beek P, Walsh T (eds) Handbook of constraint programming, chapter 21. Elsevier, Amsterdam
15. Charnes A, Cooper WW (1963) Deterministic equivalents for optimizing and satisficing under chance constraints. Oper Res 11(1):18–39
16. Davis M, Logemann G, Loveland D (1962) A machine program for theorem-proving. Comm ACM 5(7):394–397
17. Davis M, Putnam H (1960) A computing procedure for quantification theory. J ACM 7(3):201–215
18. Dechter R, Dechter A (1988) Belief maintenance in dynamic constraint networks. In: Proceedings of the 7th national conference on artificial intelligence, AAAI '88, pp 37–42
19. Fargier H, Lang J, Martin-Clouaire R, Schiex T (1995) A constraint satisfaction framework for decision under uncertainty. In: UAI '95: Proceedings of the eleventh annual conference on uncertainty in artificial intelligence, 18–20 August 1995, Montreal, Quebec, Canada, pp 167–174
20. Freuder EC (1989) Partial constraint satisfaction. In: Proceedings of the eleventh international joint conference on artificial intelligence, IJCAI '89. Morgan Kaufmann, San Francisco, pp 278–283
21. Freuder EC, Hubbe PD (1995) Extracting constraint satisfaction subproblems. In: Proceedings of the fourteenth international joint conference on artificial intelligence, IJCAI '95, Montral, Qubec, Canada, 20–25 August 1995. Morgan Kaufmann, San Francisco, pp 548–557
22. Freuder EC, Wallace RJ (1992) Partial constraint satisfaction. Artif Intell 58(1–3):21–70
23. Garey MR, Johnson DS (1979) Computer and intractability. a guide to the theory of NP-completeness. Bell Laboratories, Murray Hill, New Jersey
24. Gomez FJ, Schmidhuber J, Miikkulainen R (2006) Efficient non-linear control through neuroevolution. In: Fürnkranz J, Scheffer T, Spiliopoulou M (eds) Machine learning: ECML 2006. 17th European conference on machine learning, Berlin, Germany, 18–22 September 2006 Proceedings. Lecture notes in computer science, vol 4212. Springer, Heidelberg, pp 654–662

25. Gosavi A (2003) Simulation-based optimization: parametric optimization techniques and reinforcement learning. Kluwer, Norwell
26. Hebrard E, Hnich B, Walsh T (2004) Super solutions in constraint programming. In: Régin J-C, Rueher M (eds) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems. First international conference, CPAIOR 2004, Nice, France, 20–22 April 2004, Proceedings. Lecture notes in computer science, vol 3011. Springer, Heidelberg, pp 157–172
27. Van Hentenryck P, Bent R, Vergados Y (2006) Online stochastic reservation systems. In: Beck JC, Smith BM (eds) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems. Third international conference, CPAIOR 2006, Cork, Ireland, 31 May–2 June 2006, Proceedings. Lecture notes in computer science, vol 3990. Springer, Heidelberg, pp 212–227
28. Van Hentenryck P, Michel L, Perron L, Régin J-C Constraint programming in opl. In: Nadathur G (ed) PPDP'99: Proceedings of the international conference on principles and practice of declarative programming. Lecture notes in computer science, vol 1702. 29 September–1 October 1999, pp 98–116
29. Hewahi NM (2005) Engineering industry controllers using neuroevolution. AI EDAM 19(1):49–57
30. Hnich B, Rossi R, Tarim SA, Prestwich SD (2009) Synthesizing filtering algorithms for global chance-constraints. In: Principles and practice of constraint programming, proceedings, CP 2009, Proceedings. Lecture notes in computer science, vol 5732. Springer, Heidelberg, pp 439–453
31. Jeffreys H (1961) Theory of probability. Clarendon Press, Oxford, UK
32. Kall P, Wallace SW (1994) Stochastic programming. Wiley, Chichester
33. Katriel I, Kenyon-Mathieu C, Upfal E (2007) Commitment under uncertainty: two-stage stochastic matching problems. In: Arge L, Cachin C, Jurdzinski T, Tarlecki A (eds) Automata, languages and programming. 34th international colloquium, ICALP 2007, Wroclaw, Poland, 9–13 July 2007, Proceedings. Lecture notes in computer science, vol 4596. Springer, Heidelberg, pp 171–182
34. Kleywegt AJ, Shapiro A, T Homem-De-Mello (2001) The sample average approximation method for stochastic discrete optimization. SIAM J Optim 12(2):479–502
35. Littman ML, Goldsmith J, Mundhenk M (1998) The computational complexity of probabilistic planning. J Artif Intell Res 9:1–36
36. Littman ML, Majercik SM, Pitassi T (2001) Stochastic boolean satisfiability. J Automat Reas 27(3):251–296
37. Liu B (1997) Dependent-chance programming: a class of stochastic optimization. Comput Math Appl 34:89–104
38. Liu B, Iwamura K (1997) Modelling stochastic decision systems using dependent-chance programming. Eur J Oper Res 101:193–203
39. Lombardi M, Milano M (2006) Stochastic allocation and scheduling for conditional task graphs in mpsocs. In: Benhamou F (ed) CP 2006: principles and practice of constraint programming. 12th international conference, CP 2006, Nantes, France, 25–29 September 2006, Proceedings. Lecture notes in computer science, vol 4204. Springer, Heidelberg, pp 299–313
40. Lombardi M, Milano M (2007) Scheduling conditional task graphs. In: Bessiere C (ed) CP 2007: principles and practice of constraint programming. 13th international conference, CP 2007, Providence, RI, USA, 23–27 September 2007, Proceedings. Lecture notes in computer science, vol 4741. Springer, Heidelberg, pp 468–482
41. Majercik SM (2000) Planning under uncertainty via stochastic satisfiability. PhD thesis, Durham, NC, USA. Supervisor-Littman, Michael L
42. Majercik SM (2007) Appssat: approximate probabilistic planning using stochastic satisfiability. Int J Approx Reason 45(2):402–419
43. Majercik SM (2009) Stochastic boolean satisfiability. In: Frontiers in artificial intelligence and applications, vol 185, chapter 27. IOS Press, Amsterdam, pp 887–925

44. Majercik SM, Boots B (2005) Dc-ssat: a divide-and-conquer approach to solving stochastic satisfiability problems efficiently. In: Veloso MM, Kambhampati S (eds) Proceedings of the twentieth national conference on artificial intelligence and the seventeenth innovative applications of artificial intelligence conference, 9–13 July 2005, Pittsburgh, Pennsylvania, USA. AAAI Press/The MIT Press, Cambridge, MA, pp 416–422
45. Majercik SM, Littman ML (1998) Maxplan: a new approach to probabilistic planning. In: Proceedings of the fourth international conference on artificial intelligence planning systems, Pittsburgh, Pennsylvania, USA. AAAI Press, pp 86–93
46. Majercik SM, Littman ML (2003) Contingent planning under uncertainty via stochastic satisfiability. Artif Intell 147(1–2):119–162
47. Michel L, Van Hentenryck P (2004) Iterative relaxations for iterative flattening in cumulative scheduling. In: Zilberstein S, Koehler J, Koenig S (eds) ICAPS 2004: proceedings of the fourteenth international conference on automated planning and scheduling, 3–7 June 2004, Whistler, British Columbia, Canada. AAAI Press, CA, USA, pp 200–208
48. Minton S, Johnston MD, Philips AB, Laird P (1992) Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. Artif Intell 58(1–3):161–205
49. Powell WB (2007) Approximate dynamic programming: solving the curses of dimensionality (Wiley Series in Probability and Statistics). Wiley-Interscience, New York
50. Prestwich SD, Tarim A, Rossi R, Hnich B (2008) A cultural algorithm for pomdps from stochastic inventory control. In: Blesa MJ, Blum C, Cotta C, Fernández AJ, Gallardo JE, Roli A, Sampels M (eds) Hybrid metaheuristics. 5th International Workshop, HM 2008, Málaga, Spain, 8–9 October 2008. Proceedings. Lecture notes in computer science, vol 5296. Springer, pp 16–28
51. Prestwich SD, Tarim A, Rossi R, Hnich B (2008) A steady-state genetic algorithm with resampling for noisy inventory control. In: Rudolph G, Jansen T, Lucas SM, Poloni C, Beume N (eds) PPSN X: parallel problem solving from nature. 10th international conference, Dortmund, Germany, September 13–17, 2008, Proceedings. Lecture notes in computer science, vol 5199. Springer, pp 559–568
52. Prestwich SD, Tarim SA, Hnich B (2006) Template design under demand uncertainty by integer linear local search. Int J Prod Res 44(22):4915–4928
53. Prestwich SD, Tarim SA, Rossi R, Hnich B (2009) Evolving parameterised policies for stochastic constraint programming. In: Principles and practice of constraint programming, CP 2009, Proceedings. Lecture notes in computer science, vol 5732. Springer, pp 684–691
54. Prestwich SD, Tarim SA, Rossi R, Hnich B (2009) Neuroevolutionary inventory control in multi-echelon systems. In: 1st international conference on algorithmic decision theory, Lecture notes in computer science, vol 5783. Springer, pp 402–413
55. Proll L, Smith B (1998) Integer linear programming and constraint programming approaches to a template design problem. INFORMS J Comput 10(3):265–275
56. Rossi R, Tarim SA, Hnich B, Prestwich SD (2007) Replenishment planning for stochastic inventory systems with shortage cost. In: Van Hentenryck P, Wolsey LA (eds) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems. 4th International Conference, CPAIOR 2007, Brussels, Belgium, 23–26 May 2007, Proceedings. Lecture notes in computer science, vol 4510. Springer Verlag, pp 229–243
57. Rossi R, Tarim SA, Hnich B, Prestwich SD (2008) Cost-based domain filtering for stochastic constraint programming. In: Stuckey PJ (ed) Principles and practice of constraint programming. 14th international conference, CP 2008, Sydney, Australia, 14–18 September 2008. Proceedings Lecture notes in computer science, vol 5202. Springer, pp 235–250
58. Rossi R, Tarim SA, Hnich B, Prestwich SD (2008) A global chance-constraint for stochastic inventory systems under service level constraints. Constraints 13(4):490–517
59. Rossi R, Tarim SA, Hnich B, Prestwich SD, Guran C (2009) A note on liu-iwamura's dependent-chance programming. Eur J Oper Res 198(3):983–986
60. Rossi R, Tarim SA, Hnich B, Prestwich SD, Karacaer S (2008) Scheduling internal audit activities: a stochastic combinatorial optimization problem. J Comb Optim
61. Rummery GA, Niranjan M (1994) On-line q-learning using connectionist systems. Technical report, CUED/F-INFENG/TR 166, Cambridge University

62. Sahinidis NV (2004) Optimization under uncertainty: State-of-the-art and opportunities. Comput Chem Eng 28:971–983
63. Schiex T, Fargier H, Verfaillie G (1995) Valued constraint satisfaction problems: hard and easy problems. In: Proceedings of the fourteenth international joint conference on artificial intelligence, IJCAI '95. Morgan Kaufmann, San Francisco, pp 631–639
64. Stanley KO, Miikkulainen R (2002) Evolving neural network through augmenting topologies. Evol Comput 10(2):99–127
65. Stein ML (1987) Large sample properties of simulation using latin hypercube sampling. Technometrics 29:143–151
66. Sutton RS, Barto AG (1998) Reinforcement learning: an introduction. The MIT Press, Cambridge
67. Tarim SA, Hnich B, Prestwich SD, Rossi R (2008) Finding reliable solution: event-driven probabilistic constraint programming. Ann Oper Res 171(1):77–99
68. Tarim SA, Hnich B, Rossi R, Prestwich SD (2009) Cost-based filtering techniques for stochastic inventory control under service level constraints. Constraints 14(2):137–176
69. Tarim SA, Kingsman BG (2004) The stochastic dynamic production/inventory lot-sizing problem with service-level constraints. Int J Prod Econ 88:105–119
70. Tarim SA, Kingsman BG (2006) Modelling and computing (R^n,S^n) policies for inventory systems with non-stationary stochastic demand. Eur J Oper Res 174:581–599
71. Tarim SA, Manandhar S, Walsh T (2006) Stochastic constraint programming: a scenario-based approach. Constraints 11(1):53–80
72. Tarim SA, Miguel I (2005) A hybrid benders' decomposition method for solving stochastic constraint programs with linear recourse. In: Hnich B, Carlsson M, Fages F, Rossi F (eds) Recent advances in constraints. Joint ERCIM/CoLogNET international workshop on constraint solving and constraint logic programming, CSCLP 2005, Uppsala, Sweden, June 20–22, 2005, revised selected and invited papers, Lecture notes in computer science, vol 3978. Springer, pp 133–148
73. Tarim SA, Smith B (2008) Constraint programming for computing non-stationary (R,S) inventory policies. Eur J Oper Res 189:1004–1021
74. Terekhov D, Beck JC (2008) A constraint programming approach for solving a queueing control problem. J Artif Intell Res 32:123–167
75. Terekhov D, Beck JC (2007) Solving a stochastic queueing control problem with constraint programming. In: Van Hentenryck P, Wolsey LA (eds) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems. 4th international conference, CPAIOR 2007, Brussels, Belgium, May 23–26, 2007, Proceedings. Lecture notes in computer science, vol 4510. Springer, Heidelberg, pp 303–317
76. Verfaillie G, Jussien N (2005) Constraint solving in uncertain and dynamic environments: a survey. Constraints 10(3):253–281
77. Walsh T (2000) Sat v csp. In: Dechter R (ed) Principles and practice of constraint programming, CP 2000, Proceedings. Lecture notes in computer science, vol 1894. Springer, Heidelberg, pp 441–456
78. Walsh T (2002) Stochastic constraint programming. In: van Harmelen F (ed) European conference on artificial intelligence, ECAI'2002, Proceedings. IOS Press, Amsterdam, pp 111–115
79. Zhuang Y, Majercik SM Walkssat: an approach to solving large stochastic satisfiability problems with limited time. Technical report

Constraint Programming and Local Search Hybrids

Paul Shaw

Abstract Constraint programming and local search are two different optimization paradigms which, over the last two decades or so, have been successfully combined to form hybrid optimization techniques. This chapter describes and compares a number of these works, with the goal of giving a clear picture of research in this domain. We close with some open topics for the future.

1 Introduction

This chapter describes ways in which constraint programming (CP) and local search (LS) can be usefully combined. Researchers have been looking at ways to combine these two approaches for around 20 years now. As we shall see, these combinations can take on a variety of guises: diverse examples include using LS in propagation and pruning rules, using CP to make moves in an LS process, and performing local moves on the decision path in a search tree. Focacci et al. describe a number of these in [28].

LS and CP offer two different and contrasting ways of solving combinatorial optimization problems. LS works on a complete assignment of values to variables and navigates through the search space by making "moves": each move is selected from one available in the "neighborhood" and modifies part of the complete assignment. Roughly speaking, the neighborhood is the set of possible moves available from an assignment (see [1] for elaboration) – we will not need a more formal definition here.

CP can be seen as both a modeling and a solving technology. In terms of modeling, CP provides high-level structural constraints, such as the element constraint and all-different, very often resulting in more compact and readable models than alternative approaches, such as integer programming. In terms of solving, CP

P. Shaw (✉)
IBM, 1681 route des Dolines, 06560 Valbonne, France
e-mail: paul.shaw@fr.ibm.com

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_8,
© Springer Science+Business Media, LLC 2011

builds up a partial solution constructively by heuristically making assignments to variables (which can later be undone through backtracking). Before each heuristic assignment, inference rules and algorithms are used to reduce the set of possible values for variables not yet in the partial solution (constraint propagation).

We examine different ways that CP and LS methods have been used together to solve combinatorial optimization problems. This chapter does not, however, explore what might be considered the more "classic" methods of combining complete and LS techniques which either: (a) Run an LS method to provide a good upper bound on the objective function, then switch to a complete method for establishing proof, or, (b) run an LS method on each solution found by a tree-based method, in order to produce better solutions faster and provide tighter bounds for the complete method, or, (c) more complex combinations of the above where the two methods are perhaps run in parallel with communication of solutions and bounds being carried out throughout the search process. These kinds of methods, although interesting in their own right, have no character specific to CP and are not mentioned further here.

2 Local Search on CP Models

In this chapter, when we refer to combinations of LS and CP, we are, for the most part, referring to a *combination of searches*: a fusion of an inherently LS method with the classic constructive incremental process of CP. In contrast to this general theme, this first section describes combinations using only the *modeling* aspects of CP in combination with LS methods for finding and improving solutions to these models.

2.1 Min-Conflicts and Related Algorithms

Min-conflicts is an appealing technique described in [64]. The idea is both simple and effective, and based on the authors' observations of the Guarded Discrete Stochastic (GDS) network [2], a system based on neural networks.

In min-conflicts, and indeed in many of the methods discussed in the chapter, constraints may be violated during the search process, and that violation is often measured and treated as an objective function in the classical sense.[1] The essential idea of min-conflicts is to always take a local decision which minimizes the total number of conflicts. One may consider the number of conflicts for a constraint as (an approximation of) the number of variables that must change value in order to

[1] These techniques (min-conflicts included) are often geared toward the solution of decision problems, but optimization problems can be solved in the usual manner, through a series of such decision problems with a tightening upper bound on the objective function.

satisfy it. For binary CSPs, the number of constraints violated is typically used. Two versions of min-conflicts are offered up in [64]: one based on LS (min-conflicts hill climbing), the other based on a complete tree-search technique. We concentrate only on the LS version here, but the other is mentioned in Sect. 5.

In min-conflicts, an initial solution to the problem is generated by assigning values to variables in turn, each instantiation choosing a value for the variable under investigation which minimizes the number of conflicts with variables that have already been given values. When this first "greedy" solution has been generated, min-conflicts enters a local improvement phase which tries to reduce the number of conflicts to zero by the following process:

while conflicts exist
    choose a variable x in conflict
    choose a value for x which minimizes conflicts on x
end while

How variables and values are chosen in case of ties (*what* conflicting variable, *which* value which minimizes conflicts), can depend upon implementation, but [64] suggests using random tie-breaking for the value.

To the best of our knowledge, the authors of the min-conflicts heuristic were the first to clearly communicate on a major reason that LS can work so much better than constructive search when good solutions are quickly sought. In [64], when speaking of the GDS network, the authors write:

> Our second hypothesis was that the network's search process uses information about the current assignment that is not available to a constructive backtracking program.

The point here is that the LS algorithm always has *global* information available to it on the quality of the current state of the algorithm: all variables contribute to its evaluation and the evaluation of its next move. By contrast, a constructive method has comparatively little information on which to base its next decision. Methods such as *promise* [34], *constrainedness* [37], and *impacts* [87] have been put forward for constructive backtracking algorithms, but their evaluations are essentially statistical in nature and hold none of the solidity of the conflicts measure.

Min-conflicts worked successfully, comparing so well to the GDS network that min-conflicts replaced it in the long-term scheduling component of the Hubble Space Telescope. However, it was realized by the authors and others that min-conflicts could fall foul of local minima. A simple example: consider the system, $A, B, C \in 0, 1$, $A + B + C = 1$, $A \neq C$, $B \neq C$. The only solution to this system is $ABC = 001$ (goal state). Further assume that the initial solution is 100, produced by assigning A,B, and C in order, minimizing conflicts on the way. This initial solution violates only one constraint: $B \neq C$, and so has a cost of 1. From this initial position, to reach the goal, the assignment must pass through one of the three neighboring assignments of the goal state: namely 101, 011, or 000 (recall that min-conflicts may only change one variable assignment at a time). Table 1 shows

Table 1 Cost contributions of assignments of min-conflicts at a local minimum

Assignment		Contribution			
Label	ABC	$A + B + C = 1$	$A \neq C$	$B \neq C$	Total
Start	100	0	0	1	1
Goal	001	0	0	0	0
Goal Nei. 1	000	1	1	1	3
Goal Nei. 2	011	1	0	1	2
Goal Nei. 3	101	1	1	0	2

the cost of the five important assignments (start state, goal state, and the three neighbors of the goal state). Notice that it is impossible for min-conflicts to reach the goal state from the start state as all states neighboring the goal are of higher cost than the starting state.

The local minimum problem of min-conflicts was addressed by the breakout method [66] and GENET [24].[2] The breakout method is largely inspired by min-conflicts, whereas GENET takes much of its design from GDS (which was the system on which min-conflicts itself was based). Both methods escape local minima by adjusting the way in which constraint violations are counted: each constraint is assigned a weight (initially one), which may be modified by a learning process.

In both methods, a greedy min-conflicts type of approach is pursued until a local minimum is reached. In the breakout method, a local minimum is one where the number of violated constraints cannot be strictly decreased. In GENET, sideways (cost neutral) moves are allowed and the greedy phase terminates when the network has had the same state for two successive iterations. When in the local minimum state, both methods increase the weights of *all* currently violated constraints by one, and the greedy process is restarted from the current point. In experiments, both methods are shown to strongly dominate the min-conflicts hill climbing method.

Yugami et al. [115] describe another method for breaking out of local minima, named EFLOP. The idea of EFLOP is that it can be used as a diversification method that a hill climber can call when confronted with a local minimum. EFLOP will change the value of a conflicting variable, even if that increases the number of conflicts, and then will go on to try to repair the new conflicts created, without ever changing the value of a variable more than once. In this way, the repair effect propagates through the network until no newly created conflicts can be repaired. The resulting assignment is then given back to the hill climber as a point from which to continue. The method is similar in spirit to that of an ejection chain in LS (see for example [113]) where changing a solution element has a cascading effect on others. The authors report significant gains when the EFLOP module is plugged into the min-conflicts hill climber or GSAT [98].

[2] Later, GENET was generalized to Guided Local Search [110], a meta-heuristic based on penalization of the cost function.

2.2 SAT-Based Methods

Another well-used technique for solving CSPs using LS methods is to translate the CSP problem to an SAT problem, and then use one of the well-known SAT solving techniques (such as WalkSAT [97]).

There are various ways of translating CSPs to SAT. In [112], Walsh describes two methods: the direct encoding and the logarithmic encoding. In the direct encoding, one propositional variable x_{ij} is introduced for each unary assignment $X_i = j$ in the CSP. Clauses of the form $\vee_{j \in \text{Domain}(X_i)} x_{ij}$ are introduced to assure that each CSP variable is assigned a value. Finally, each constraint is encoded by introducing one clause for each tuple of incompatible values. For example, if $X_1 = 1$ is incompatible with $X_2 = 3$, then this translates to the clause $\neg x_{11} \vee \neg x_{23}$. Note that clauses assuring that each CSP variable must take on *exactly one value*, rather than at least one value are not necessary. If, for example, a solution is found with both x_{11} and x_{12} true, then either the value 1 or 2 can be chosen for the variable X_1: both will give legal solutions.

In the logarithmic encoding, $b_i = \lceil \log_2(|\text{Domain}(X_i)|) \rceil$ variables are introduced to encode the binary representation of the value of X_i. This representation is more compact (in terms of variables) than the direct encoding, but the constraints of the CSP translate to longer clauses (proportional to the sums of the logs of the domain sizes of the variables in the constraint). Furthermore, for any i, if b_i is not a power of two, then additional clauses are required to rule out the combinations of values which do not correspond to a domain value of X_i in the CSP. Walsh showed that in both encodings, the unit propagation rule performs less filtering than arc consistency on the original problem.

Gent [38] describes a SAT encoding of CSPs called the *support encoding* which he showed as performing equivalent filtering to arc consistency on the original problem. The support encoding uses one propositional variable per CSP domain value as for the direct encoding but describes the constraints differently. The supports of each domain value are described by clauses. If, for example, on a binary constraint between X_1 and X_2, value 1 of X_1 is only compatible with values 3 and 5 of X_2, this would translate as a clause: $x_{11} \Rightarrow x_{23} \vee x_{25}$. In this formulation, clauses are also added to ensure that each CSP variable takes exactly one value. Of particular interest is that this formulation seems quite compatible with the operation of LS methods: a variant of WalkSAT performed on average over an order of magnitude faster on the support encoding than on the direct encoding.

More recently, more complete translations of CP models have been proposed: Huang [46] describes a universal framework for translating CP models to SAT models based on the logarithmic encoding, translating much of the MiniZinc language [102]. The Sugar solver [105, 106] also operates by translating CSP models to SAT, but in this case by using an *order encoding*, where each SAT variable $y_{x,a}$ represents the truth value of the relation $x \leq a$ in the original CSP formulation.

2.3 Localizer and Comet

Localizer [62, 63] was the first attempt at creating a language for the specification of LS algorithms. The modeling language is reminiscent of languages for CP, but with differences. First of all, the notion of an expression (say $x + y$) is central. Furthermore, when one writes $z = x + y$ in Localizer, this means that the value of z is *computed* from the values of x and y. (z is not a variable, has no domain, and its value cannot be decided by the system by a search procedure.) Localizer refers to the $\langle lhs \rangle = \langle \text{expr} \rangle$ construct as an *invariant*, and ensures that at any time during the computation, in our example of $z = x + y$, the value of z is equal to the value of x plus the value of y. This is performed by incremental calculations to maintain the value of z whenever x or y changes value. Invariants can also be created over more complex structures, such as sets. This means that Localizer can maintain, for example, the set of best variables to flip in a GSAT-type implementation, as shown in Fig. 1. The set *Candidates* maintains the set of variables which are among the best to flip. This set is updated incrementally whenever one of the *gain* expressions changes. These *gain* expressions are themselves computed from the current SAT assignment and a statement of the clauses of the problem (not shown). Localizer then makes a move by changing the value of one of these variables to its negation.

Localizer maintains invariants through a differencing approach, which we briefly describe. Consider the expressions and invariants in a Localizer program to form a directed acyclic graph. Each node represents an expression, with a directed arc connecting expression a to expression b when b directly mentions a.

Consider that arcs in the graph have the same rank as their source nodes. Nodes with no incoming arcs are labeled with rank 0. Other nodes are ranked one more than their highest ranked incoming arc. Consider the example $z = \sum_{1 \le i \le n} x_i + \sum_{2 \le i \le n} x_{i-1} x_i$. A graph corresponding to this invariant is shown in Fig. 2. In this graph, all "x" nodes have rank 0, all multiplication nodes have rank 1, and the node corresponding to z has rank 2.

Localizer proceeds by labeling each node according to its topological rank in its "planning" phase. Then, whenever the state of the system changes (here, expressed by the x_i expressions), the "execution" phase incrementally maintains the value of z by minimally recomputing parts of z following the topological order. Essentially, all changes are processed at level k before moving to level $k + 1$. This means that no part of an invariant is considered more than once, and node values are never changed

```
maxGain: int = max (i in 1..n) gain[i];
Candidates: {int} = {
  i : int | select i from 1..n where gain[i] = maxGain and gain[i] ≥ 0
};
...
move a[i] := !a[i]
where i from Candidates
```

Fig. 1 A Localizer code using a GSAT technique (taken from [63])

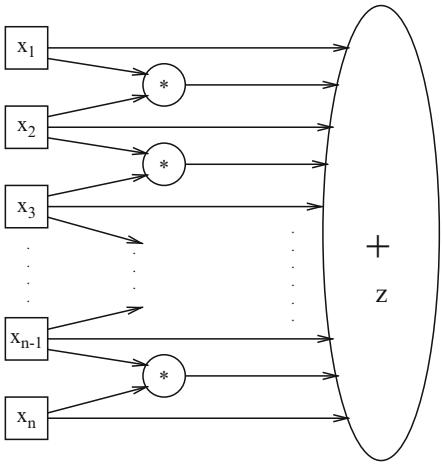


Fig. 2 Graph representation of a localizer invariant

more than once. The recalculation of the invariant is done in an incremental way. At level $k+1$, the current and previous values (before the update) of nodes at level k are available in order to recalculate the nodes at level $k+1$. For example, if the value of x_2 changes, then x_1x_2 and x_2x_3 are recalculated (in constant time), then z is recalculated in constant time by applying the *difference of the new and old values* of x_2, x_1x_2 and x_2x_3 to z.

Localizer also includes an element constraint which complicates the calculation of invariants as just described: $element(e, x_1, \ldots, x_n)$ evaluates to the eth element of the x array. However, often it is very natural to write invariants such as $pos_1 = 1 + element(prev_1, pos_1, \ldots, pos_n)$. This example is taken from a TSP-based model and maintains the position (pos) of each town according to the position of the previous town. Unfortunately, this invariant is deemed illegal by the topological rule because the same expression (pos_1) is mentioned on the left and right hand sides of the equality and so the topological ordering constraint rule is violated. In Localizer, this is treated by so-called dynamic invariants. This means that some of the planning phase is delayed until the index of an element constraint is known by the execution phase. At that point, the element expression $element(e, x_1, \ldots, x_n)$ depends only on the eth x expression. So long as this is not the same as the left-hand side of the invariant (which would provoke an error), a planning phase is invoked to produce a topological order for the graph after (in a topological sense) this point. Planning and execution phases are interleaved as necessary to evaluate all invariants.

The Comet system [107,108] grew out of work on Localizer, but has some important differences. Localizer is mostly declarative in nature with some imperative parts for generating initial solutions. By contrast, Comet is a full-function imperative programming language with features oriented toward modeling and search. However, as for other constraint modeling systems embedded in an imperative language (for example, IBM ILOG CP Optimizer in C++, Java, or .NET), the way in which the constraint model is created is declarative in style. Unlike Localizer, Comet has some built-in machinery to evaluate constraint violations in a constraint

```
function void minConflictSearch(ConstraintSystem S) {
  inc{int}[] var = S.getVariables();
  range Size = var.getRange();
  while (S.violations() > 0)
    selectMax(i in Size)(S.getViolations(var[i]))
      selectMin(v in var[i].getDomain())
        (S.getAssignDelta(var[i], v))
          var[i] := v;
}
```

Fig. 3 A comet implementation of min-conflicts (from [109])

system, which allows the specification of the constraint model in more traditional form (through constraints rather than invariants).

Comet also has language features to iterate over, evaluate, and select different decisions, allowing a compact definition of search methods. Additionally, the impact of a potential move on the objective function can be evaluated using methods such as `getAssignDelta(var, value)` (change the value of a single variable) and `getSwapDelta(var1, var2)` (swap the values of two variables). Finally, mechanisms such as *closures* and *checkpoints* [107], allow complex control methods such as events and neighborhood evaluation to be specified more easily.

Figure 3 shows a small code (taken from [109]) which implements a min-conflicts heuristic using Comet. `S.getViolations` is used to select the variable which is involved in the most violated constraints and `S.getAssignDelta` is used to choose a value for it which reduces the violation measure by the largest amount (even if this amount is negative). This process continues until the total number of violations reaches zero, meaning a solution is found that satisfies all the constraints. Of course, as for min-conflicts, this state might never be reached.

2.4 Some Other Methods

Without aiming to be exhaustive, but rather restricting ourselves to the most CP-oriented methods (see [45] for additional references), we also very briefly mention some other techniques where LS has been applied to solve CP-based models.

In [111], Walser reports results on an LS technique on linear pseudo-boolean problems. In particular, excellent results are reported on the progressive party problem [11], which was considered a very difficult problem at the time.

In [21], Codognet and Diaz use a tabu search method on CSP models. Their method changes the value of a single variable at each move and uses a variable penalty which is computed from a combination of constraint penalties. Experiments are carried out on simple examples such as n-queens, magic square, number partitioning, and all-interval series. Results show that on some problem classes, the method outperforms other approaches.

Solnon [101] describes a method for solving CSP using an ant colony optimization approach. In Solnon's method, each variable assignment is represented by a node in a graph, and an arc is present between all nodes which correspond to different variables. After each ant completes its journey, its trajectory is improved by a classic LS algorithm. Additionally, a local search is used as a fast way of initializing the pheromone trails.

Galinier and Hao [32] present a method for constraint solving using LS. In particular, they specify a set of supported constraints (including global constraints) along with precise penalty functions (following [67]). They apply their method to SAT, graph-coloring, max-clique, frequency assignment, and the progressive party problem. The method is demonstrated to be very competitive.

In [86], Årgen et al. describe methods for performing LS over set variables with incremental updates of data structures. The authors apply their methods to the progressive party problem. In [85], the same authors also describe a method of synthesizing incremental update algorithms for LS from a description based on second order logic.

Finally, [36] shows how LS can be effective on quantified boolean formulas, which is a generalization of the standard NP models treated by CP.

3 Using CP to Evaluate Neighborhoods

One main area where CP and LS meet is when an essentially LS method uses CP to make and evaluate moves in the neighborhood of the current solution. Typically, such methods address optimization problems and maintain a complete assignment which satisfies all problem constraints, in contrast with many of the algorithms already presented which try to reduce constraint violations. These methods then move from feasible solution to feasible solution, in an attempt to improve the objective value.

These methods maintain the current solution not in the constrained variables themselves, but in a so-called passive representation which is a simple array s holding, for each variable, its value in the current solution. By contrast, the active representation is the constraint model itself with variables, domains, and constraints acting between them. When a new solution to be accepted is produced in the active representation, it is copied to s for safekeeping.

This section presents various methods for using CP to evaluate and make moves in an LS process. All the techniques may be implemented by traditional CP engines supporting a sufficiently flexible user-definable depth-first search.

3.1 Constraint Checking

In [5], De Backer et al. propose to integrate CP into an LS method in perhaps the simplest possible way. The CP system is only used to check the validity of solutions and determine the values of auxiliary constrained variables (including the cost variable), not to search for solutions (via backtracking, for example). The search is performed by an LS procedure. When the procedure needs to check the validity of a potential solution, it is handed to the CP system. The authors studied their integration in the context of vehicle routing problems.

When the CP system performs propagation and validity checks, it instantiates the set of decision variables with the proposed solution. The constraints then propagate and constrained variables (for example, any auxiliary variables whose values are taken from propagation from the decision variables) have their domains reduced. If any variable has no legal values, the proposed solution is illegal.

Improvement heuristics generally only modify a very small part of a solution. Therefore, testing the complete solution in the above manner can be inefficient. The authors try to avoid this inefficiency in two ways: by reducing the amount of work carried out by the CP system to perform each check, and bypassing the CP checks altogether. In the context of vehicle routing, it is often the case that routes are independent (have no constraints acting between them). For these problems, only the routes involved in the modification proposed by the move need to be instantiated in the active representation. Normally, this means just one or two routes will be instantiated.

The second method of increasing efficiency is to perform custom tests on the more important "core" constraints without using the general propagation mechanism [92]. Only if these fast checks succeed is the proposed solution handed to the CP system to be fully evaluated.

3.2 Exploring the Neighborhood Using Tree Search

In [73,74], Pesant and Gendreau propose a framework for LS in which a new "neighborhood" constraint model is coupled with a standard model of the problem, and represents the neighborhood of the current solution. The idea is that the set of solutions to this new combined model is the set of legal neighbors of the current solution. This setup is depicted in Fig. 4.

The left part of the figure shows the variables and constraints of the model of the problem (the *principal* model). On the right, there is another set of variables and constraints, which together represent a model of the neighborhood of the current solution. These variables are coupled to the principal model via *interface constraints* which ensure that a choice of neighbor in the "neighborhood model" enforces the appropriate values of decision variables in the principal model with respect to the current solution.

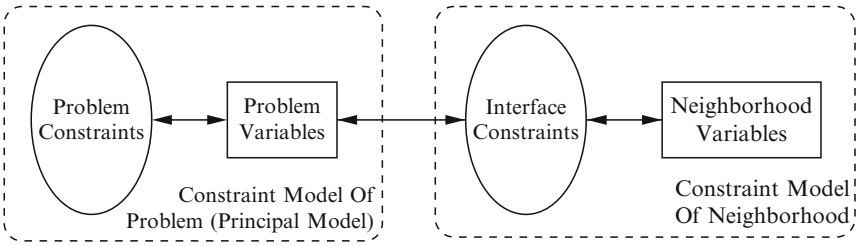


Fig. 4 Interaction of the neighborhood model with interface constraints

As a simple example, suppose that the principal model has n 0–1 variables $x[1] \ldots x[n]$ and that we wish to perform an LS over a neighborhood which flips each variable (changes its value from 0 to 1 or vice versa). We suppose a passive representation of the current solution (as in the previous section) which is a simple integer array s of n elements. In this case, the neighborhood model would consist of a single variable f with n domain values (say $1 \ldots n$). $f = a$ would mean that the ath 0–1 variable in the principal model would have a value different from that in the current solution, while the other constrained variables would take their values as in the current solution. The interface constraints are as follows:

$$(f = i) \Leftrightarrow (x[i] \neq s[i]) \quad \forall\, i \in \{1 \ldots n\}$$

To explore the neighborhood of the current solution, a backtracking search procedure then finds all solutions to the combined model, normally through exploring all combinations of values of the neighborhood variables: here, this means instantiating just the variable f (so a search tree of depth 1). This exploration is very natural in a CP context–one may also select the best (lowest cost) neighbor to move through a standard branch-and-bound search over the model.

It should be noted that the neighborhood model is configured according to the current solution vector s. That is, constants in the neighborhood model are taken from the values of the current solution. This means that, in general, the neighborhood model is recreated anew at each move, the previous one being discarded.

3.2.1 An Example

We give a more realistic example of a complex move which may be implemented using the constrained neighborhood framework. Consider an allocation problem, like a generalized assignment problem, where n objects are allocated to m containers. This problem might have capacity constraints on the containers, a complex cost function, and other side constraints, but that is not our concern here–we are only interested in how a solution is represented and the decision variables. Our passive representation of a solution will store the container s_i into which object i is placed. Our principal constraint model will contain n decision variables $x[1] \ldots x[n]$ each with domain

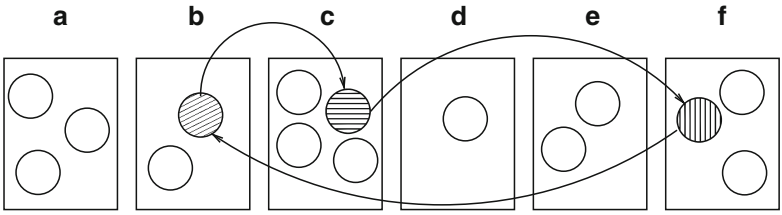


Fig. 5 A 3-rotation move

$\{1 \ldots m\}$. The neighborhood we consider is a rearrangement of the positions of any three objects without changing the numbers of objects in each container. We call this a 3-rotation. The move is depicted in Fig. 5. In the figure, after the move, the diagonally striped object will have moved to container C, the horizontally striped object to container F, and the vertically striped object to container B.

We will create a neighborhood model with three variables p, q, r, representing the objects involved in the move. Each of these variables has domain $\{1 \ldots n\}$ identifying the objects to be involved in the move. We will assume that in the move, object p will move to q's container, q to r's container, and r to p's container.

The interface constraints for this neighborhood can be written as:

$$p \neq q, p < r, q < r \tag{1}$$

$$x[p] = s[q], x[q] = s[r], x[r] = s[p] \tag{2}$$

$$p \neq i \wedge q \neq i \wedge r \neq i \Rightarrow x[i] = s[i] \;\; \forall i \in \{1 \ldots n\} \tag{3}$$

Line (1) constrains which objects can be involved in the move. Some symmetries need to be broken if moves are not to be duplicated. For example, $p = 1$, $q = 2, r = 3$ identifies the same move as $p = 2, q = 3, r = 1$. These can be broken by the constraints $p < r, q < r$. Note also that three *different* objects should be identified and so $p \neq q$. Line (2) places the objects involved in the move in their new positions. For this, the element expression is used on the solution vector s. Line (3) states that objects not involved in the move stay where they are currently.

Observe that whenever two objects appear in the same container, the move actually taken (so long as the third object is elsewhere) is a swap of two objects. If desired, we could limit ourselves to true 3-rotations by asserting that all objects must be in different containers. This could be done by replacing line (1) above by:

$$s[p] \neq s[q], s[p] < s[r], s[q] < s[r]$$

3.2.2 Flexibility and Efficiency

The main advantage of this framework is that propagation takes place between the principal and neighborhood model, and specifically, domain filtering may be carried out on the neighborhood variables, reducing the number of neighbors explored. This filtering is carried out by the interface constraints, based on the current domains of

the decision variables of the principal model. The final result is that neighbors which are illegal or do not meet acceptance criteria (like being better than the current solution in terms of cost) may be directly filtered out of the neighborhood variables. This filtering tends to be most effective when the search tree for the neighborhood variables is deep (neighborhoods are large) and the neighborhood has a large proportion of illegal neighbors. Experiments with the 3-opt operator on TSPs with time windows in [73] indicate that few neighbors are actually acceptable (typically a few percent), increasing the efficiency of the CP technique. GENIUS-CP [75] is very general technique for inserting customers into a route, and implemented in the CP framework. The authors report that one does pay a price for using CP in terms of efficiency (up to an order of magnitude over a custom approach), but their technique extends naturally to TSPs with multiple time windows and pickup-and-delivery problems, which the custom heuristic [35] was incapable of doing. In [91], the flexibility of the method is demonstrated via the application of several different constraint-based neighborhood operators in a style based on variable neighborhood search (VNS) [65].

Although the above framework is extremely flexible, natural, and exploits the power of CP's propagation in order to reduce neighborhood sizes, this benefit is not obtained when either the neighborhood is largely feasible, or when opportunities for propagation are limited (for instance when the depth of the search tree on the neighborhood variables is low). For example, in the "flip" example already given, exploring the neighborhood through a single depth search tree which instantiates the f variable could result in a total of n^2 variable fixings.

To attempt to address this, Shaw et al. [100] describe a method which reduces the number of variable fixings to $O(n \log n)$ for the "flip" example. The method works on an explicit representation of the neighborhood as a series of neighbors, each neighbor represented by an object called a "delta" which stores the variables to change together with their new values.[3] One attraction of this method is that it allows the creation of very general, unstructured, or randomly ordered neighborhoods, specifiable via general codes, and not constrained to what can be described by a constraint model. For this reason, this method is used in an open framework in the Solver module of IBM ILOG CP (which provides operators to randomize, concatenate, interleave and sample general neighborhood structures).

Figure 6 shows an actual C++ code sample for solving a vehicle routing problem using the IBM ILOG CP product. This code uses a greedy approach (via the use of `IloImprove`[4]) which accepts the first neighbor which decreases the overall cost. (By adding one additional line of code, it is also possible to accept the neighbor which decreases cost by the greatest amount.) The neighborhood is a combination of all possible two-opt [59] moves, all moves which relocate every node before each other node, and all moves which interchange the positions of two nodes.

[3] Not all neighbors have to be available before exploration can start, but can be generated on the fly to keep memory consumption low.

[4] Meta-heuristics such as simulated annealing or tabu search can be used by changing `IloImprove` to `IloSimulatedAnnealing` or `IloTabuSearch`.

```
// env = environment, a management object
// curSol = current solution
// solver = the CP solver instance
IloNHood nh = IloTwoOpt(env) + IloRelocate(env) + IloExchange(env);
IloNHood rnh = IloRandomize(env, nh);
IloGoal move = IloSingleMove(env, curSol, rnh, IloImprove(env));
while (solver.solve(move))
  cout << "Objective value = " << solver.getObjValue() << endl;
```

Fig. 6 IBM ILOG CP example of local search for vehicle routing

The "+"operator is used to concatenate these three different basic neighborhoods into one. This simple but powerful method can be used to implement a VNS [65]. Here, instead of using a VNS method (which could be achieved by using `nh` instead of `rnh` in the call to `IloSingleMove`), the order of moves in the neighborhood is randomized each time a move is accepted (by using `IloRandomize`), so that a *random* improving move will be taken each time. A loop calling `solve` on the CP solver actually performs the greedy moves. When no improving move exists in the neighborhood, the call to `solve` returns a false value, and the loop exits with the locally optimal solution in `curSol`.

The method described in [100], and used in IBM ILOG CP, gains efficiency by using a divide and conquer approach: a binary search tree is created where on the left branch the first half of the neighborhood is explored, and on the right branch, the second half is explored. This is continued recursively until only one neighbor remains at the leaf node. Each node in the search tree can thus be associated with a part of the complete neighborhood. If this part of the neighborhood does not change a variable x (that is, x does not appear in any "delta" for this part of the neighborhood), then the value of x in the current solution s can be assigned to x. This last rule, which allows different neighbors to share variable fixings, reduces the number of fixings by a factor of $O(n/\log n)$ when neighbors change a constant number of variables.

Pesant and Gendreau also propose an adaption of their method which allows similar logarithmic efficiency gains to be achieved. The idea is to map the divide and conquer approach of Shaw et al. to a variable instantiation strategy. Instead of instantiating the neighborhood variables by fixing them, a deeper search tree can be created by dividing the domain of a neighborhood variable in two at each branch. This will eventually instantiate the neighborhood variables but will crucially allow propagation to the principal model's decision variables higher in the (now deeper) search tree.

3.3 Large Neighborhood Search

Large Neighborhood Search (LNS) is a technique first coined in [99], but whose origins date back to the shuffling technique described in [3]. Similar work uses different nomenclature: Mimausa [60], forget-and-extend [17] and

ruin-and-recreate [94]. In [40], Sect. 10.7 also briefly mentions such an approach in general terms, there called *referent-domain* optimization, and [104] proposes a technique with similar motivation.

LNS has been applied to a variety of problems, with most of these having a vehicle routing [10,76,99], scheduling [3,41], or a network/graph aspect [15,20]. It is also used in commercial products, such as IBM ILOG CP Optimizer [55].

The simplest way to think about LNS is that of iteratively relaxing a part (normally called the *fragment*) of the current solution and then re-optimizing that part. Figure 7 shows how this method works for the traveling salesman problem (TSP). Figure 7a shows a solution to the TSP, in Fig. 7b,c a set of arcs is chosen and removed from a region of the space, and then in Fig. 7d this partial solution is completed to produce a new, improved solution of lesser total distance.

An outline of how LNS is normally implemented is shown in Fig. 8 (adapted from [72]). At a high level, the technique is extremely simple. It is normally a hill climber (however see [76]) which is executed until some time limit is exhausted, although other criteria can be used, such as a number of iterations, or a number of iterations without improving the current solution.

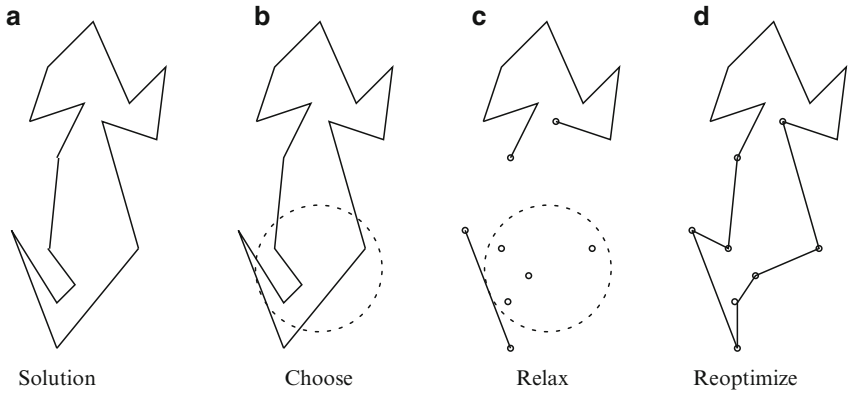


Fig. 7 Operation of LNS on the traveling salesman problem

```
Create initial feasible solution
while optimal not found and time remaining do
    Choose part of problem (the fragment)
    Freeze the part of the solution not in the fragment
        (that is, fix it to its value in the current solution)
    Search the remaining sub-problem for an improving solution
    if improvement found then
        Adopt the improved solution as the current one
    end if
end while
```

Fig. 8 Typical implementation of large neighborhood search

The general description of LNS makes reference to two components: a method to choose the fragment of the problem to relax, and a method to re-optimize the chosen fragment: we discuss both, beginning with the latter.

3.3.1 The Re-Optimization Method

LNS is specified in very general terms, and this leaves the implementation of the re-optimization method open. Techniques which have already been used with LNS are greedy methods [76], LS [103], decomposition-oriented techniques [68], mixed-integer programming methods [79] (for linear models), or CP [99]. In fact, virtually any technique which can be used to solve a complete problem can be used to re-optimize a fragment in LNS. However, for CP techniques, the re-optimization is generally based on a tree-search, which could be depth-first search, limited discrepancy search (LDS) [44], restarts [52], or indeed any other techniques one might imagine. Using CP as the re-optimization method allows the addition of a constraint on the objective which excludes non-improving solutions. This can accelerate the search through increased propagation.

Using CP for the re-optimization process also increases the robustness of the search as side-constraints can be efficiently handled in the re-optimization, meaning that LNS is much better able to handle richer models with more complex constraints than standard neighborhood search. This is demonstrated in [53] where side constraints are progressively added to a problem, and in [9], which successfully extends an LNS solution for vehicle routing [8] to one for pickup-and-delivery, which is considerably more constrained. Likewise, Røpke and Pisinger [76] and Laborie and Godard [55] introduce LNS methods which perform well over a wide range of vehicle routing and scheduling problem classes.

When re-optimizing the fragment, we have found that a tree search which works better than others for solving the *whole* problem will continue to work better than others when embedded in LNS as a re-optimization procedure.[5] We know of no strong contradictions of this rule of thumb–if any exist, their analysis might yield interesting directions of the design of LNS re-optimization methods. That said, an almost ubiquitous modification is made to any global solution method when it is embedded in LNS, and that it to limit its run time in some way ([13] is an exception). Common methods are to limit the number of backtracks in a depth-first search [16, 71], or to limit the number of discrepancies [6] in a depth-first search [99]. Perron [70] compares both of these methods, amortized search [58], as well as hybrids.

The general idea is to find a balance between avoiding exponential time exploration of unpromising fragments, and attempting to improve the solution as much as possible at each re-optimization. Another choice in the re-optimization is whether

[5] This is important, as here we can draw on a wealth of knowledge on general-purpose and problem-dependent branching heuristics.

to stop as soon as an improving solution is found (as done in [71]), or to continue up until the limit to see if better improvements can be found in the same re-optimization (as done in [13, 99]).

3.3.2 Selecting the Fragment

One natural way of selecting the fragment of the solution to be relaxed is to do this in a completely random fashion. This technique can sometimes be effective. For example, [69] uses a completely random technique (but then moved to a more structured technique in [70]), [23] shows good results with random selection (although biased selection is shown to perform better), and [71] uses a random selection as one of two methods for fragment selection. Although a random choice has its merits (it may avoid pathological behavior, and is thus useful in portfolio-based techniques–see Sect. 3.3.3), using *only* random choice is seldom the best way to produce good solutions quickly. This was recognized from early on, for example, the shuffle of [3] relaxes the order of all operations on a set of machines–each machine has either a fully fixed or fully free order (while still respecting problem constraints). Later, other more general shuffles were performed in [16], but here again the fragments chosen have a certain cohesion, for example, operations falling in a certain time window, or operations on the critical path.

We should ask ourselves the question, *what makes a good fragment*? To be useful, a fragment must contain at least one good solution. (In the typical implementation of LNS, this solution should be better than the current solution, but some implementations allow cost-neutral moves [71] or degradation of the current solution [76].) For some problems, we can determine some basic conditions that need to be satisfied if cost is to have a hope of being reduced. For example, in a job-shop scheduling problem with a makespan objective, we know that the critical path must be changed to reduce the makespan. This dictates that some operations on the critical path should be present in the fragment, if the fragment is to be useful [13].

Other less strict rules can be used to select the fragment which we might consider would generate fragments with improving solutions. For example, Fig. 7 shows a fragment selection criterion based on distance from a randomly chosen node n in the TSP tour: for example, one might relax the k closest nodes to the central node n. The intuition here is that when we relax the fragment, the relaxed nodes will have the possibility for position interchange in the current tour. Now, assuming the current tour is not too bad (if it is bad, then almost any reasonable fragment selection will allow improvements to be found), then nodes which are far apart are also far apart in the tour. This implies that interchange of these nodes would introduce long arcs, and hence a large distance penalty. By contrast, when the fragment is localized, interchange of nodes can be carried out without introducing long arcs.

Given the fact that a fragment can be selected that contains a better solution–or has a reasonable chance of containing one–the fact that the solution is available is not enough. (We could place the whole problem in the fragment, which would guarantee it contained the global optimum.) In reality, the re-optimization method

(Assume solution elements are $x_1 \ldots x_n$)
Initial fragment $F = \{i\}$ where i is randomly drawn from $\{1 \ldots n\}$ (F's universe)
while $|F| < f$ **do**
  Let e = random element of F
  Let z be an non-increasing ordering of all $e' \in F^C$ according to $R(s, e, e')$
  (F^C is the complement of F, the indexes of the elements *not* in the fragment)
  $k = 1 + \lfloor |F^C| y^d \rfloor$ where y is uniformly and randomly chosen in $[0 \ldots 1)$
  Add the kth element of z to F
end while
return F

Fig. 9 Shaw's method for choosing a fragment

must have a reasonable chance of finding the solution using the limited resources at its disposal. There is thus an interplay between the fragment selection and the re-optimization method, but the general idea is to keep the fragment as small as possible, while maintaining a reasonable chance of finding an improving solution. In this way, the re-optimization process has the best chance of finding the good solution or solutions. Similarly, smaller fragments allow the re-optimization process to be limited more, which increases the speed of examination of different fragments. A common mechanism, first used in [99] but since replicated elsewhere, is to begin with a small fragment, and gradually increase its size when improving solutions become difficult to find.

Although the use of some problem structure is useful for choosing the fragment, the first generic framework for doing this was introduced in [99]. We outline the approach in Fig. 9. It assumes the existence of a real valued function $R(s, x, y)$ which takes the current solution s, and two solution elements. The solution elements could be variables, but could also correspond to more structured objects such as a nodes in a TSP or operations in a scheduling problem. R delivers a measure of how related x and y are in solution s. The interpretation of this is that if $R(s, x, y)$ is high, then x and y should tend to be included in the fragment together, or not at all, and if $R(s, x, y)$ is low, then putting both x and y in the fragment is less desirable. In [99], vehicle routing problems were treated, where an objective which was to reduce the number of vehicles used, and then total distance as a secondary objective. For two customers x and y, $R(s, x, y) = 1/(C(x, y) + V_s(x, y))$ where $C(x, y)$ is a normalized distance (in the range $0 \ldots 1$) between the two customers x and y and $V_s(x, y)$ evaluates to 0 if the two customers are served by the same vehicle in the current solution s and to 1 otherwise.[6] The idea of favoring customers served by the same vehicle for inclusion in the fragment comes from the observation that the only way of reducing the number of vehicles (primary objective) is to free all customers on at least one vehicle. Unless some specific provision is provided for that, the distance measure alone is unlikely to achieve it.

[6] An error appeared in the original paper which indicated the opposite result for $V_s(x, y)$.

The selection method also has two parameters $f \geq 1$ (fragment size) and $d \geq 1$ (determinism). The former indicates the desired size of the fragment, and the latter a measure of how deterministically the selection will follow R: $d = 1$ means that R will be ignored and random choice will be used, while $d = \infty$ means that R will be followed for each choice.[7] Shaw [99] reports that the tuning of the determinism parameter was reasonably simple, and for medium sized VRPs (around 100 customers) values around 10–20 were reasonable, with performance suffering badly only for values of d outside $3 \ldots 30$. Pisinger and Røpke [78] also discuss fragment selection methods based on the distance between customers in a vehicle routing context. They use Kruskal's algorithm [54] to generate clusters of customers which are close together.

3.3.3 Learning in Large Neighborhood Search

A few ideas have been put forward to attempt to ease the design and tuning of LNS methods (see discussion in [14]). The first is the aforementioned one of starting with a small fragment size and increasing it when finding improving solutions is becoming too hard. This is normally controlled by a number of LNS iterations without an improvement in the current solution.

In an attempt to alleviate the work involved in designing fragment selection, Perron and Shaw [72] propose a method called Propagation Guided LNS which tries to determine what solution elements should be placed in the fragment together. The method works by using the propagation mechanism itself. It can be viewed as a variation on Fig. 9 where each time a solution element is added to the fragment, it is assigned as in the current solution s, and propagation on the sub-problem outside the fragment is observed. The strength of the propagation of x to y (where x is newly added to the fragment and y is outside the fragment) defines $R(s, x, y)$. The method continues like this until the desired fragment size is attained, at which point it records the fragment for use in the subsequent re-optimization process, but discards the partial assignment corresponding to it in the constraint solver. Perron and Shaw show that the method can outperform hard-designed fragment selection on the car sequencing problem.

A common learning mechanism applied in the context of LNS is that of algorithm portfolios [43]. LNS is particularly amenable to such a technique as it is an iterative combination of two separate techniques: fragment choice and re-optimization. Thus, a portfolio of algorithms can be proposed for each technique. In [70], Perron uses a portfolio of algorithms for the re-optimization technique for a network design problem. He ascribes a weight to each algorithm and uses reinforcement learning to reward (increase the weight of) algorithms which produce improving solutions. The weights drop after a certain number of trials without improvement. An algorithm

[7] Setting $d = \infty$ does not eradicate randomness from the algorithm, only for one part of it.

is selected proportionally according to its weight in relation to others. However, Perron's main aim in [70] is to investigate restart strategies, and does not perform any detailed analysis of his portfolio scheme.

Røpke and Pisinger [77] present a method which uses portfolios for both the fragment selection method and the re-optimization method for pickup-and-delivery problems. They use three fragment selection methods, five re-optimization methods and optionally apply noise to the objective function to help diversify the (largely greedy) re-optimization method. A feedback learning approach controls a biased choice of fragment selection, re-optimization method, and the application of noise to the objective function. Their fragment selection methods are based on that of [99], a completely random method, and a method which tries to place high-cost solution elements into the fragment (reminiscent of squeaky wheel optimization [47]). Re-optimization methods are founded on a regret-based technique. The authors method demonstrates greatly increased robustness over a standard LNS approach (allowing them to develop a very general heuristic applicable to many different types of routing problem [76]), and produces state-of-the-art results on pickup-and-delivery problems with time windows.

Laborie and Godard [55] also present a portfolio method and apply it to a large number of different single-mode scheduling problems. Their framework also proposes using a feedback learning approach on fragment selection and re-optimization methods, but their study is limited to the selection of fragment. Three fragment selection methods are used: random selection, selection of operations in a time window, and selection of operations occurring in connected components or strongly connected components of the precedence graph of the problem. Results show that the resulting algorithm is extremely robust over a large number of scheduling problem classes and, despite its generality, can improve on the best ad-hoc methods created for several of these classes.

In [18], Caseau et al. present a general method for learning combinations of heuristics for vehicle routing problems. They use LNS with fragment selection as in Fig. 9 as one of the building blocks of their system. Their LNS method is parametrized with a learned fragment size f, a learned determinism parameter d, and a learned re-optimization method, which can be constructed from other basic building blocks. Results show that the learning process can better hand-crafted algorithms.

4 Local Search for Pruning and Propagation

LS has been used as a means of demonstrating that the current partial solution to a problem has no legal extension (resulting in pruning) or that certain value assignments cannot be made in any legal extension (resulting in domain filtering, or propagation as it is also known). Although this area has not been widely explored, this section describes some pieces of work which have successfully used LS to reduce search tree size through pruning and/or propagation.

In [96], Harvey and Sellmann describe how to increase pruning and propagation on the Social Golfer Problem. The social golfer problem can be stated as follows: *32 golfers want to play in 8 groups of 4 each week for 9 weeks, such that no pair of players play together more than once. Is this possible?* The problem can be generalized to one of determining a w week schedule of g groups of golfers, with each group of size s. Harvey and Sellmann's method depends on finding witnesses to the insolubility of the current sub-problem. These witnesses are clique structures contained in a residual graph representation of the sub-problem.

Consider that a golfing schedule is being built week by week, in chronological order, and that a graph is maintained in which each golfer is represented by a node and a pair of golfers that have already played together in previous weeks is represented by an arc connecting their respective nodes. If a clique of size k is present in this graph, then it means that the golfers involved in the clique must play in different groups from now on. If, when scheduling the current week by building groups for that week, we get to a situation where none of the golfers in the clique have been scheduled, but less than k groups remain available this week, then the current partial assignment cannot be completed as each of the k golfers in the clique must be put into a different group. Harvey and Sellmann use an extension of this pruning rule to perform propagation and describe an orthogonal rule which reasons on golfers rather than on groups in a week. The authors use a randomized LS approach with intensification and diversification steps to find the cliques. Results show that significant gains can be made both in terms of number of backtracks and run time.

Focacci and Shaw [30] present a method for pruning the search tree based on the application of a dominance rule, the detection of the rule's applicability at any node being carried out by an LS process. The method could be considered an implementation of Symmetry Breaking by Dominance Detection [26, 29]. Focacci and Shaw argue that aggressive pruning of a search tree by dominance rules can be detrimental to the success of a search algorithm. For example, consider a Euclidean TSP where a 2-opt [59] dominance rule is applied. That is, any branch of the search tree which constructs a partial tour where two arcs cross (and is therefore not 2-optimal) is pruned, as we know that any tour which is not 2-optimal cannot be globally optimal. This aggressive pruning rule may result in no solution being found for some considerable time, which could be a problem for online optimization. Furthermore, since solutions may be found considerably later with the dominance rule active, even time to produce a proof may suffer since good upper bounds (which can also significantly help pruning) are not provided quickly. The solution proposed in [30] is quite general (a similar technique is used in [31]) and can be applied to decision as well as optimization problems, but the following description is based on the TSP.

The authors propose to apply a dominance rule which prunes the current branch only if there is no hope of extending it to a solution which can better the best solution found so far. This is done in the context of the TSP and TSP with time windows. Here, imagine that we are to construct a simple path between nodes A and Z, passing through nodes $\{B, C, \ldots, Y\}$. Imagine two partial TSP tours $t_1 = A \rightarrow B \rightarrow C$ and $t_2 = A \rightarrow D \rightarrow B \rightarrow E \rightarrow C \rightarrow F \rightarrow G$. Assume that all extensions of t_1 have already been fully explored and that t_2 is the current partial tour. We form t

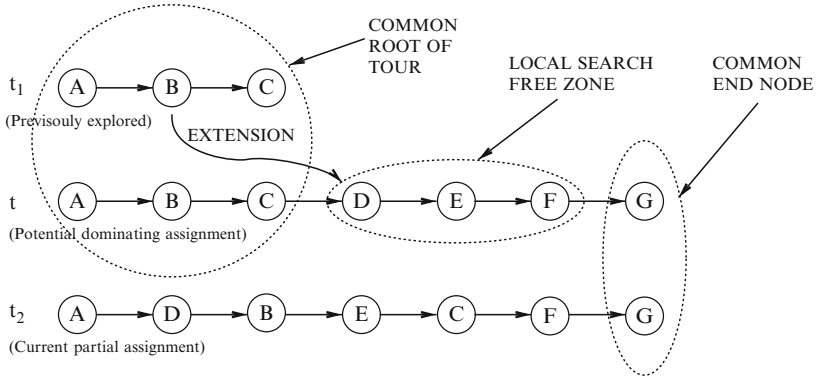


Fig. 10 Operation of Focacci and Shaw's dominance detection

from t_1 by appending to it all nodes in t_2 not already in t_1, ensuring that t ends at the same node as t_2, node G in our case.[8] Assume that after performing this operation $t = A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \rightarrow G$. Observe that with this construction, t always visits at least all the nodes in t_2 and terminates at the same node. Figure 10 shows the relationships between the three partial tours. Now, assume that the length of a (partial) tour is given by the function L and that in our case $L(t) \leq L(t_2)$. Given that t visits at least all of the nodes of t_2 and any extension of t will start at the same node as t_2, we can safely say that the best extension of t will be better than the best extension of t_2. However, we have already explored the best extension of t, because we have explored the best extension of t_1 (a smaller partial tour). Thus, we can prune the current search tree branch t_2. If $L(t) > L(t_2)$, then we still have options. In particular, we formed t from t_1 by an arbitrary extension order. Any extension which maintains G at the end of the partial tour is legal to form t. If any one of these extensions verifies $L(t) \leq L(t_2)$, then search can be pruned.

Focacci and Shaw propose to search for an improving extension t via an LS process which, in the case of our example, would perturb the order of the nodes D, E, F, while keeping G at the end as needed, as well as the root of the partial tour (ABC) coming from t_1. This is done using a relocate operator which can move any node in the free part of the partial tour to another position in the free part (see Fig. 10). The extension to the TSP with time windows simply adds another condition to check. Assume that the total time of a (partial) tour is given by the function T. Then, search can be pruned if $L(t) \leq L(t_2) \wedge T(t) \leq T(t_2)$.

Results reported indicate that the proposed approach can often work much better than a pure CP approach without dominance rules or indeed than standard application of dominance rules which prune every branch which corresponds to a partial tour that can be improved using the relocation operator.

[8] If t_1 already contained G, then the dominance detection would not be possible using t_1.

Galinier et al. [33] describe a method to facilitate propagation of the *some-different* constraint [90]. The some-different constraint is a generalization of the all-different constraint [88] where only a subset of all pairs, freely specifiable to the constraint, must be different. This global constraint is capable of encoding any graph coloring problem and hence achieving arc consistency for it is NP-hard.

The authors approach the problem by using a tabu search algorithm to find *supports* (solutions to the constraint). The aim is to quickly find a support for each domain value of each variable. For those variable-value assignments for which a support cannot be easily found, the suspicion is that they have no support, and this can be found out by the application of a complete technique, such as DSATUR [12]. The LS algorithm is reminiscent of min-conflicts but is based on tabu search [40] and works as follows. Initially, each variable is assigned an arbitrary value from its domain. Then, the search attempts to minimize the number of conflicts ($\neq$ constraints violated) using a tabu criterion which forbids a variable taking on its previous value for a certain number of iterations. When a solution to the constraint (a support) is found, then it is added to a "support store".

After the first support is found, the algorithm changes its behavior, as it is no longer based solely on reducing the number of conflicts, but a weighted combination of conflicts and on covering (in the support sense) variable-value assignments not yet represented in the support store. In this way, the tabu search algorithm tries to provide a support for as many variable-value assignments as quickly as possible. The tabu search algorithm is stopped after a certain number of iterations after which it is considered that continued search is not as productive as the use of other methods. After the tabu search has found as many supports as possible, a complete method (DSATUR) is launched for each variable-value assignment for which a support has not yet been found. This is done for $x = a$ by assigning a to x and then using DSATUR to try to find a solution to the remaining sub-problem. If this sub-problem proves soluble, then a support for $x = a$ has been found, otherwise a is removed from the domain of x.

Combined with a number of reduction procedures, Galinier et al. report that their algorithm reduces average run time on real workforce management data by a factor of 34 over a standard CP approach.

5 Other Integrations

Various other integrations of LS in CP have been proposed where the interaction of the CP and LS techniques is more intimate. Often, the methods are based on the combination of a constructive technique (including propagation) and a repair technique, but could simply be the use of a non-systematic search idea inside a complete search method. Much work could potentially fall into this fairly weak classification, and so while this section is necessarily not exhaustive, it does attempt to present a reasonable overview of the current situation.

In [114], Yokoo describes a *weak commitment* strategy for solving constraint satisfaction problems. The algorithm is complete and the main idea is based on the complete min-conflicts method [64]. The complete min-conflicts method is essentially a depth-first search using a min-conflicts heuristic for variable and value choice. First, a tentative assignment to each variable is made which greedily attempts to minimize total conflicts as described in Sect. 2.1. Then, a depth-first search chooses first a variable in conflict, assigning it a value which does not conflict with previous assignments and conflicts minimally with future tentative assignments. If all values conflict with previous assignments, the algorithm backtracks.

Yokoo observes that in depth-first search mistakes are very costly to undo. Yokoo's weak commitment strategy also uses tentative assignments, but instead of backtracking one level when a dead end is reached, it entirely abandons the current partial assignment. Search then continues by building a new partial assignment, using, where possible, the previous assignment as new tentative assignment values and adding the previous abandoned assignment to a no-good store to avoid re-exploration of the same space. One can think of the weak commitment method as a type of improved *iterative sampling* [56], where the variable/value choice heuristic is based on min-conflicts, and a no-good is added at each dead end.

Yokoo observed good results on graph coloring and 3-SAT problems, better than either the complete min-conflicts method or the breakout method.

Richards and Richards [89] describe Learn-SAT, whose principles draw heavily on the weak commitment method. Their algorithm uses the same process of abandoning the search process when a dead end is encountered, and of no-good addition. However, Learn-SAT also including a "learning-by-merging" component which performs resolution on the clauses which lead to a variable's domain becoming empty, generating a new clause for each dead end encountered. Experiments demonstrate that the method can outperform *relsat* [48], the best complete SAT solver of the time.

Ginsberg and McAllester [39] describe how gradient methods from GSAT can be used in a complete technique. The method used, known as partial order dynamic backtracking (PDM), works by generating no-goods in *directed form* ($A = a \wedge B = b \wedge C \neq c \Rightarrow D \neq d$ instead of $A \neq a \vee B \neq b \vee C = c \vee D \neq d$) when a violation is detected. The value of right hand side (conclusion) variable is then modified by the algorithm in order to remove the violation. PDM allows some freedom in the choice of the conclusion variable of any new no-good generated. This freedom allows PDM to start from a tentative total assignment, and then use a GSAT-inspired conflict-based rule to repair (that is, to add as conclusions in a no-good set) the variables which are most in conflict. In this way, a complete search is created, based on information normally only available to an LS technique.

It should be noted, however, that the choice of next variable to repair is not as free in PDM as in GSAT. PDM introduces so-called *safety conditions* to guarantee integrity of the algorithm, which specify partial orders on the variable choice. Ginsberg and McAllester demonstrate that their methods compare very well with WalkSAT and TABLEAU, dominating TABLEAU on unsatisfiable SAT instances.

Jussien and Lhomme [49] introduce the decision-repair algorithm which combines a constructive search and propagation with an LS over the decisions made

in the constructive search path. This incomplete method works by constructing a solution using heuristics and constraint propagation in the "classic" way until the last decision D made creates an inconsistency. Instead of undoing D in the normal depth-first search manner, decision-repair tries to find a *conflict*, which is a strict subset of the decisions already taken (not including D) which are together inconsistent with D. At least one of these decisions needs to be negated if D is to be kept. The authors introduce a heuristic to decide which decision(s) to negate based on a tabu search criterion. The most recent k conflicts found during the search are maintained, and the algorithm first chooses to negate a decision which appears most often in this conflict set. If more decisions need to be negated to ensure D can be accepted, then these follow the same rule. Experiments on open shop scheduling problems show the technique to be competitive with leading approaches.

Prestwich [80] introduces CLS (constrained local search), later to be renamed IDB (incomplete dynamic backtracking) [83]. Like [49], the method is incomplete, and can be considered as performing an LS on the search decisions already taken in a tree search. The difference between decision-repair and IDB is based on how the repair of past search decisions is carried out. In [49], conflicts are used both as a predicate and as a heuristic to decide on the decisions(s) whereas IDB either uses random selection, or an ad-hoc heuristic with random tie-breaking. Once the repair variables are chosen, in [49], each repaired decision is negated, whereas IDB simply discards the chosen decisions. Prestwich has applied IDB quite widely with excellent results: see for example [80–82]. In particular, in [81], Prestwich argues that IDB delivers the scalability of LS compared to classic backtracking *because* of the flexible choice of the variable to change.

In [95], Sellmann and Ansótegui describe Disco-Novo-GoGo, a restarting technique which uses backtracking search "probes" to perform an LS over a complete assignment. The essential idea is to encode a value choice heuristic as a complete (normally illegal) assignment H. A backtrack-limited depth-first search then instantiates the problem variables in a random order, fixing each variable, where possible, to its preferred value from H: this is done on the left branch of the search tree, whereas other values are considered equivalent and are explored in random order on other branches. If the backtracking search finds a solution or proves there is none, the entire search process can be stopped. If the backtracking limit has been attained (which is usual), then the heuristic assignment is updated as follows. For any variable which is fixed to a different from its value in H, its current value is copied to H. For any unassigned variable that has had its value from H filtered from its domain, a random value from its current domain is transferred to H. The remainder of H remains unchanged. This basic mechanism is embedded in a double loop which varies the fail limit and a number of trials. The authors report significant speedups over standard restart techniques.

Zhang and Zhang [116] report on a method which combines an LS procedure with backtracking and propagation. The idea is to successively generate, in a randomized way, partial solutions to the problem which do not violate the constraints between variables in the partial solution. Then, for each one of these partial solutions, a combined LS and backtracking phase is entered where first a tree search

technique with propagation tries to extend the partial solution to a complete one. This tree search is normally limited in the number of branches it can carry out. If the tree search does not succeed, a local move is carried out on the partial solution to try to improve its quality. The authors use the criterion of the number of constraints satisfied by the partial solution. This extension attempt using propagation and backtrack followed by a local movement on the partial solution is repeated until a solution is found or an iteration limit is exceeded. In the latter case, the method begins again with a completely new partial solution. The authors succeed in closing some open quasi-group problems using their method.

Schaerf [93] describes an incomplete method. The technique extends a partial solution until a dead end is reached. (A static variable ordering is used and Schaerf defines a dead end as being when the next variable to be assigned has no legal values consistent with previous assignments.) At this point, an LS method is invoked which attempts to change the values of the variables in the current partial assignment to allow further constructive search to take place. The method continues in this way, interleaving construction and repair, until a solution is found.

When repair takes place, the objective function takes into account (a) conflicts existing between variables in the partial solution, (b) the value of the bound on the objective function, and (c) a "lookahead" which is a count of the number of domain values remaining in the variables not yet given a value. The idea is to reduce (a) and (b), but increase (c). The LS procedure is largely greedy but is randomized and may make sideways (non-objective changing) moves. In addition, the different constraint types in the problem have independent and dynamically changing weights according to how often they are violated.

The algorithm is evaluated on timetabling and tournament scheduling problems. Schaerf compares LS (for example min-conflicts) to a combination of the LS method embedded (and suitably modified) in the constructive procedure. Results show that the combined method significantly outperforms the LS method on these problems.

At around the same time, David [25] also introduced an incomplete method very similar to that of Schaerf. A partial solution is extended until a dead-end is reached, at which point action is taken to try to repair the current partial assignment. David concentrates on examination timetabling problems, and the repair methods are dedicated to the problem. Four different repair techniques are tried before the method marks an examination as "not scheduled," and continuing with the remaining examinations. Thus, the method may produce partial schedules (which are presumably executed by temporarily procuring extra staff), but produces results in a deterministic time frame (under a few seconds).

Caseau and Laburthe [19] present an incremental local optimization approach to solving vehicle routing problems where a traditional constructive approach to building a solution is followed, but the partial solution is improved each time the solution is extended. The authors begin with an empty schedule and use a classical insertion approach to solution construction. At each insertion, the most "constrained" customer is inserted. Before insertion, a limited search examines the potential impact of insertion in different routes and the best insertion point is chosen. After insertion, a more thorough LS process improves the solution. Results show that the method

scales much better than performing all insertions and then applying LS. The authors find very good solutions to problems with around 500 customers in around 10 s. Finally, to further improve their solutions, the authors also examine overlaying a limited amount of backtracking over their methods. For example, the decision to open a new route or not when inserting a customer could be the subject of a backtrack point.

Kamarainen and El Sakkout [51] describe a method called local probing, a type of probe-backtrack search [50]. In probe-backtrack search, at each node in a tree search, a *probe* is executed which tries to find a solution. This probe is some heuristic or algorithm which treats only a subset of the constraints of the problem (the "easy" constraints). If the probe produces a solution which also happens to satisfy the remaining "hard" constraints of the problem, search can stop. Otherwise, the solution provided by the probe is analyzed to find a violation of the "hard" constraints, and a variable involved in the conflict is chosen for branching. This method as described is not related to LS, but LS may be used in the probe. In [51], the authors concentrate on scheduling problems. A probe is implemented which respects only precedence constraints and the objective, while relaxing (ignoring) capacity constraints of resources. Resource capacities which are violated by the probe mean that too many tasks execute at the same time. For a time when resources are oversubscribed, a branch is created which orders a pair of tasks (via the addition of a new precedence constraint in the "easy" constraints), driving the search toward satisfaction of the capacity constraints.

Lever [57] studies a network routing problem and proposed a combination of tree search and LS for solving it. At a high level, the search is one based upon tree search with propagation which decides for each potential demand whether it will be routed or not. After each positive decision in the search tree, the demand is routed with those already present, which may result in capacity violations in the network. The infeasibility is repaired using an LS process. If the LS process does not succeed in legally placing the new demand in conjunction with those already placed within a certain limit, then the branch is pruned in the search tree, and the order is not placed. Thus, the method is not complete, but can use constraint propagation to determine that certain demands must be routed, for example. Lever demonstrates that his method compares well to probe-backtrack search [50] on this problem.

Crawford [22] uses an LS technique to identify hard-to-satisfy clauses in an SAT formulation and from that, variables which should be prioritized for early assignment inside a complete technique. Although the idea seems promising, results indicate that only a 10% reduction in the number of backtracks is obtained.

Mazure et al. [61] mostly follow the same idea as [22], the essential difference being that Mazure et al. run the LS process before each instantiation of a variable in the complete process. After each LS, the variable which was the most "unsatisfiable" (by appearing most in unsatisfied clauses) is chosen for branching. (The technique can also find solutions when the LS technique delivers an assignment which satisfies all constraints.) The authors compare their method to standard techniques (such as choosing a variable in the shortest clause) and demonstrate large speedups on some problems.

Benoist and Bourreau [7] describe "branch-and-move," a technique which uses LS to guide a traditional depth-first search process. The idea of branch and move is to perform LS on a tentative assignment to the unassigned variables, trying to reduce constraint "unhappiness." The unhappiness is defined for each constraint as distance measure of the tentative assignment from the closest support. The algorithm uses this information to generate branching decisions which concentrate on taking corrective action to please the most unhappy constraint. The authors apply their algorithm to a TV-break scheduling problem, and find that their methods outperform MIP, LS, and standard CP techniques.

6 Conclusion

We have described, through reference to work carried out over the last two decades, how CP and LS methods can be combined. While we could not be exhaustive (in particular, there is much research in the SAT community which could be influential, for instance, using LS to prove non-satisfiability of SAT instances [4, 27, 84]), we have tried to show how the domain has progressed in many different ways. That said, we believe that there is still much to be done in this promising area. We list some possibilities:

1. LNS has been an extremely effective technique, but to our knowledge, there is very little work using LNS as a move operator inside another meta-heuristic technique, such as tabu search ([76] uses simulated annealing). Likewise, we also know of no work trying to apply LNS to decision problems rather than optimization problems. Finally, it would be interesting to see if intensification techniques such as streamlining [42] could be used to better explore larger fragments.
2. LS for increasing filtering power [30, 96] is also a technique which deserves more attention. The idea of finding a witness which shows that extensions to a partial solution are fruitless is a powerful technique. Until now, these techniques have been applied at a *problem* level. Of interest would be to find out if such techniques can be applied to certain global constraints, or groups of constraints, allowing their inferences to be re-used in different problems.
3. As it maintains a complete assignment and an exact evaluation of this assignment, LS can be a good technique for finding out where the difficult part of a problem may lie. We believe that more work (in the spirit of [61]) deserves to be carried out to analyze the information derived from LS in order to perform better branching.

As the problems we face become larger and more challenging and application areas for optimization and automated reasoning widen, solving these problems will result in hybridization through necessity: no single domain has all the best tricks. So, we should look forward to more hybridization in the future, in which CP and LS will certainly play a big part.

References

1. Aarts E, Lenstra JK (eds) (1997) Local search in combinatorial optimization. Princeton University Press, Princeton
2. Adorf HM, Johnston MD (1990) A discrete stochastic neural network algorithm for constraint satisfaction problems. In: Proceedings of the international joint conference on neural networks
3. Applegate D, Cook W (1991) A computational study of the job-shop scheduling problem. ORSA J Comput 3(2):149–156
4. Audemard G, Simon L (2007) GUNSAT: a greedy local search for unsatisfiability. In: Proceedings of IJCAI-07, pp 2256–2261
5. De Backer B, Furnon V, Shaw P, Kilby P, Prosser P (2000) Solving vehicle routing problems using constraint programming and metaheuristics. J Heuristics 6(4):501–523
6. Beck JC, Perron L (2000) Discrepancy-bounded depth first search. In: Proceedings of CP-AI-OR 2000
7. Benoist T, Bourreau E (2003) Improving global constraints support by local search. In: The CP 2003 workshop on cooperative solvers in constraint programming
8. Bent R, Van Hentenryck P (2004) A two-stage hybrid local search for the vehicle routing problem with time windows. Transporatation Sci 38(4):515–530
9. Bent R, Van Hentenryck P (2006) A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows. Comput Oper Res 33(4):875–893
10. Bent R, Van Hentenryck P (2007) Randomized adaptive spatial decoupling for large-scale vehicle routing with time windows. In: Proceedings of AAAI-07, pp 173–178
11. Brailsford SC, Hubbard PM, Smith B, Williams HP (1996) The progressive party problem: a difficult problem of combinatorial optimisation. Comput Oper Res 23:845–856
12. Brelaz D (1979) New methods to color the vertices of a graph. Commun ACM 22(4):251–256
13. Carchrae T, Beck JC (2005) Cost-based large neighborhood search. In: Workshop on the combination of metaheuristic and local search with constraint programming techniques
14. Carchrae T, Beck JC (2008) Principles for the design of large neighborhood search. J Math Model Algorithm 8(3):245–270
15. Caseau Y (2006) Combining constraint propagation and meta-heuristics for searching a maximum weight hamiltonian chain. Oper Res 40:77–95
16. Caseau Y, Laburthe F (1995) Disjuntive scheduling with task intervals. Technical Report 95–25, Laboratoire d'Informatique de l'École Normale Superieure, Departement de Mathématiques et d'Informatique
17. Caseau Y, Laburthe F, Le Pape C, Rottembourg B (2001) Combining local and global search in a constraint programming environment. Knowl Eng Rev 16:41–68
18. Caseau Y, Silverstein G, Laburthe F (2001) Learning hybrid algorithms for vehicle routing problems. Theor Pract Logic Program 1(6):779–806
19. Caseau Y, Laburthe F (1999) Heuristics for large constrained vehicle routing problems. J Heuristics 5:281–303
20. Chabrier A, Danna E, Le Pape C, Perron L (2004) Solving a network design problem. Ann Oper Res 130:217–239
21. Codognet P, Diaz D (2001) Yet another local search method for constraint solving. In: Proceedings of the international symposium on stochastic algorithms: foundations and applications, pp 73–90
22. Crawford JM (1993) Solving satisfiability problems using a combination of systematic and local search. In: Second DIMACS challenge: cliques, coloring, and satisfiability
23. Danna E, Perron L (2003) Structured vs. unstructured large neighborhood search: a case study on job-shop scheduling problems with earliness and tardiness costs. In: Proceedings of CP 2003, pp 817–821
24. Davenport A, Tsang E, Wang C, Zhu K (1994) GENET: a connectionist architecture for solving constraint satisfaction problems by iterative improvement. In: Proceedings of AAAI-94

25. David P (1997) A constraint-based approach for examination timetabling using local repair techniques. In: Selected papers from the second international conference on practice and theory of automated timetabling, pp 169–186
26. Fahle T, Schamberger S, Sellmann M (2001) Symmetry breaking. In: Proceedings of CP 2001, pp 93–107
27. Fang H, Ruml W (2004) Complete local search for propositional satisfiability. In: Proceedings of AAAI-04, pp 161–166
28. Focacci F, Laburthe F, Lodi A (2001) Local search and constraint programming. In: Proceedings of the metaheuristics international conference (MIC '01), pp 451–454
29. Focacci F, Milano M (2001) Global cut framework for removing symmetries. In: Proceedings of CP 2001, pp 77–92
30. Focacci F, Shaw P (2002) Pruning sub-optimal search branches using local search. In: Proceedings of CP-AI-OR 2002, pp 181–189
31. Fukunaga AS, Korf RE (2005) Bin-completion algorithms for multicontainer packing and covering problems. In: Proceedings of IJCAI-05, pp 117–124
32. Galinier P, Hao J-K (2004) A general approach for constraint solving by local search. J Math Model Algorithm 3(1):78–88
33. Galinier P, Hertz A, Paroz S, Pesant G (2008) Using local search to speed up filtering algorithms for some NP-hard constraints. In: Proceedings of CP-AI-OR 2008, pp 298–302
34. Geelen PA (1992) Dual viewpoint heuristics for binary constraint satisfaction problems. In: Proceedings of ECAI-92, pp 31–35
35. Gendreau M, Hertz A, Laporte G, Stan M (1998) A generalized insertion heuristic for the traveling salesman problem with time windows. Oper Res 46(3):330–335
36. Gent I, Hoos H, Rowley A, Smyth K (2003) Using stochastic local search to solve quantified boolean formulae. In: Proceedings of CP 2003, pp 348–362
37. Gent IP, MacIntyre E, Prosser P, Walsh T (1996) The constrainedness of search. In: Proceedings of AAAI-96, pp 246–252
38. Gent IP (2002) Arc consistency in SAT. In: Proceedings of ECAI – 02, pp 121–125
39. Ginsberg M, McAllester DA (1994) GSAT and dynamic backtracking. In: International conference on the principles of knowledge representation (KR94), pp 226–237
40. Glover F, Laguna M (1997) Tabu Search. Kluwer, Boston
41. Godard D, Laborie P, Nuijten W (2005) Randomized large neighborhood search for cumulative scheduling. In: Proceedings of ICAPS, pp 81–89
42. Gomes C, Sellmann M (2004) Streamlined constraint reasoning. In: Proceedings of CP 2004, pp 274–289
43. Gomes C, Selman B (1997) Algorithm portfolio design: theory vs. practice. In: Proceedings of UAI-97, pp 190–197
44. Harvey W, Ginsberg M (1995) Limited discrepancy search. In: Proceedings of IJCAI-95, pp 607–615
45. Hoos H, Tsang E (2006) Chapter 5. Local search methods. In: Handbook of constraint programming. Elsevier, Amsterdam
46. Huang J (2008) Universal booleanization of constraint models. In: Proceedings of CP 2008, pp 144–158
47. Joslin DE, Clements DP (1999) "Squeaky wheel" optimization. J Artif Intell Res 10:353–373
48. Bayardo RJ Jr, Schrag RC (1997) Using CSP look-back techniques to solve real-world SAT instances. In: Proceedings of AAAI-97, pp 203–208
49. Jussien N, Lhomme O (2002) Local search with constraint propagation and conflict-based heuristics. Artif Intell 139(1):21–45
50. Kamarainen O, El Sakkout H (2002) Local probing applied to network routing. In: Proceedings of CP-AI-OR 2004, pp 173–189
51. Kamarainen O, El Sakkout H (2002) Local probing applied to scheduling. In: Proceedings of CP 2002, pp 155–171
52. Kautz H, Horvitz E, Ruan Y, Gomes C, Selman B (2002) Dynamic restart policies. In: Proceedings of AAAI-02, pp 674–682

53. Kilby P, Prosser P, Shaw P (2000) A comparison of traditional and constraint-based heuristic methods on vehicle routing problems with side constraints. Constraints 5(4):389–414
54. Kruskal JB (1956) On the shortest spanning subtree of a graph and the traveling salesman problem. Proc Am Math Soc 7(1):48–50
55. Laborie P, Godard D (2007) Self-adapting large neighborhood search: application to single-mode scheduling problems. In: Proceedings of MISTA-07, pp 276–284
56. Langley P (1992) Systematic and nonsystematic search strategies. In: Proceedings of the first international conference on artificial intelligence planning systems, pp 145–152
57. Lever J (2005) A local search/constraint propagation hybrid for a network routing problem. Int J Artif Intell Tool 14(1):43–60
58. Lhomme O (2002) Amortized random backtracking. In: Proceedings of CP-AI-OR 2002, pp 21–32
59. Lin S (1965) Computer solutions of the traveling salesman problem. Bell Syst Tech J 44:2245–2269
60. Mautor T, Michelon P (1997) Mimausa : a new hybrid method combining exact solution and local search. In: Second international conference on meta-heuristics
61. Mazure B, Saïs L, Grégoire É (1998) Boosting complete techniques thanks to local search methods. Ann Math Artif Intell 22:319–331
62. Michel L, Van Hentenryck P (1999) Localizer: a modeling language for local search. INFORMS J Comput 11(1):1–14
63. Michel L, Van Hentenryck P (2000) Localizer. Constraints 5:43–84
64. Minton S, Johnston MD, Philips AB, Laird P (1992) Minimizing conflicts: a heuristic repair method for constraint-satisfaction and scheduling problems. Artif Intell 58(1):161–205
65. Mladenovic N, Hansen P (1997) Variable neighborhood search. Comput Oper Res 24(11): 1097–1100
66. Morris P (1993) The breakout method for escaping from local minima. In: Proceedings of AAAI-93, pp 40–45
67. Nareyek A (2001) Using global constraints for local search. In: Constraint programming and large scale discrete optimization. DIMACS, Vol 57, pp 9–28
68. Palpant M, Artigues C, Michelon P (2004) LSSPER: Solving the resource-constrained project scheduling problem with large neighborhood search. Ann Oper Res 131:237–257
69. Perron L (2002) Parallel and random solving of a network design problem. In: AAAI workshop on probabilistic approaches in search, pp 35–39
70. Perron L (2003) Fast restart policies and large neighborhood search. In: Proceedings of CP-AI-OR 2003
71. Perron L, Shaw P (2004) Combing forces to solve the car sequencing problem. In: Proceedings of CP-AI-OR 2004, pp 225–239
72. Perron L, Shaw P, Furnon V (2004) Propagation guided large neighborhood search. In: Proceedings of CP 2004, pp 468–481
73. Pesant G, Gendreau M (1996) A view of local search in constraint programming. In: In Proceedings of CP '96, pp 353–366
74. Pesant G, Gendreau M (1999) A constraint programming framework for local search methods. J Heuristics 5(3):255–279
75. Pesant G, Gendreau M, Rousseau J-M (1997) GENIUS-CP: A generic single-vehicle routing algorithm. In: Proceedings of CP '97, pp 420–433
76. Pisinger D, Røpke S (2007) A general heuristic for vehicle routing problems. Comput Oper Res 34(8):2403–2435
77. Røpke S, Pisinger D (2006) An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. Transportation Sci 40:455–472
78. Røpke S, Pisinger D (2006) A unified heuristic for a large class of vehicle routing problems with backhauls. Eur J Oper Res 171(3):750–775
79. Prescott-Gagnon E, Desaulniers G, Rousseau L-M (1999) A branch-and-price-based large neighborhood search algorithm for the vehicle routing problem with time windows. Networks 54(4):190–204

80. Prestwich S (2000) A hybrid search architecture applied to hard random 3-SAT and low-autocorrelation binary sequences. In: Proceedings of CP 2000, pp 337–352
81. Prestwich S (2001) Local search and backtracking vs non-systematic backtracking. In: AAAI 2001 Fall symposium on using uncertainty within computation, pp 109–115
82. Prestwich S (2001) Trading completeness for scalability: hybrid search for cliques and rulers. In: Proceedings of CP-AI-OR 2001, pp 159–174
83. Prestwich S (2002) Combining the scalability of local search with the pruning techniques of systematic search. Ann Oper Res 115:51–72
84. Prestwich S, Lynce I (2006) Local search for unsatisfiability. In: Proceedings of SAT 2006, pp 283–296
85. Årgen M, Flener P, Pearson J (2005) Incremental algorithms for local search from existential second-order logic. In: Proceedings of CP 2005, pp 47–61
86. Årgen M, Flener P, Pearson J (2005) Set variables and local search. In: Proceedings of CP-AI-OR 2005, pp 19–33
87. Refalo P (2004) Impact-based search strategies for constraint programming. In: Proceedings of CP 2004, pp 557–571
88. Régin JC (1994) An filtering algorithm for constraints of difference in CSPs. In: Proceedsings of AAAI-94, pp 362–367
89. Richards ET, Richards B (1998) Non-systematic search and learning: an empirical study. In: Proceedings of CP-98, pp 370–384
90. Richter Y, Freund A, Naveh Y (2006) Generalizing AllDifferent: the SomeDifferent constraint. In: Proceedings of CP 2006, pp 468–483
91. Rousseau L-M, Gendreau M, Pesant G (2002) Using constraint-based operators to solve the vehicle routing problem with time windows. J Heuristics 8(1):43–58
92. Savelsbergh MWP (1985) Local search in routing problems with time windows. Ann Oper Res 4:285–305
93. Schaerf A (1997) Combining local search and look-ahead for scheduling and constraint satisfaction problems. In: Proceedings of IJCAI-97, pp 1254–1259
94. Schrimpf G, Schneider J, Stamm-Wilbrandt H, Dueck G (2000) Record breaking optimization results using the ruin and recreate principle. J Comput Phys 159:139–171
95. Sellmann M, Ansótegui C (2006) Disco-Novo-GoGo: integrating local search and complete search with restarts. In: Proceedings of AAAI-06, pp 1051–1056
96. Sellmann M, Harvey W (2002) Heuristic constraint propagation – using local search for incomplete pruning and domain filtering of redundant constraints for the social golfer problem. In: Proceedings of CP-AI-OR 2002, pp 191–204
97. Selman B, Kautz HA, Cohen B (1994) Noise strategies for improving local search. In: Proceeedings of AAAI-94, pp 337–343
98. Selman B, Levesque H, Mitchell D (1992) A new method for solving hard satisfiability problems. In: Proceedings of AAAI-92, pp 440–446
99. Shaw P (1998) Using constraint programming and local search methods to solve vehicle routing problems. In: Proceedings of CP '98, pp 417–431
100. Shaw P, De Backer B, Furnon V (2002) Improved local search for CP toolkits. Ann Oper Res 115:31–50
101. Solnon C (2002) Ants can solve constraint satisfaction problems. IEEE transactions on evolutionary computation 6(4):347–357
102. Stuckey PJ, De La Banda MG, Maher M, Marriott K, Slaney J, Somogyi Z, Wallace M, Walsh T (2005) The G12 project: mapping solver independent models to efficient solutions. In: Proceedings of the 21st international conference on logic programming. Lecture notes in computer science, vol 3668 . Springer, Heidelberg, pp 9–13
103. Taillard E (2003) Heuristic methods for large centroid clustering problems. J Heuristics 9(1):51–73
104. Taillard ED, Voss S (2001) POPMUSIC: Partial optimization metaheuristic under special intensification conditions. In: Hansen P, Ribeiro C (eds) Essays and surveys in metaheuristics. Kluwer, pp 613–629

105. Tamura N, Banbara M (2008) Sugar: A CSP to SAT translator based on order encoding. In: Second international CSP solver competition, pp 65–69
106. Tamura N, Taga A, Kitagawa S, Banbara M (2006) Compiling finite linear CSP into SAT. In: Proceedings of CP 2006, pp 590–603
107. Van Hentenryck P, Michel L (2003) Control abstractions for local search. In: Proceedings of CP 2003, pp 65–80
108. Van Hentenryck P, Michel L (2005) Constraint-based local search. MIT Press, Cambridge
109. Van Hentenryck P, Michel L, Liu L (2004) Constraint-based combinators for local search. In: Proceedings of CP 2006, pp 47–61
110. Voudouris C, Tsang E (1999) Guided local search and its application to the travelling salesman problem. Eur J Oper Res 113:469–499
111. Walser JP (1997) Solving linear pseudo-boolean constraint problems with local search. In: Proceedings of AAAI-97, pp 269–274
112. Walsh T (2000) SAT v CSP. In: Proceedings of CP 2000, pp 441–456
113. Yagiura M, Ibaraki T, Glover F (2004) An ejection chain approach for the generalized assignment problem. INFORMS J Comput 16(2):133–151
114. Yokoo M (1994) Weak-commitment search for solving constraint satisfaction problems. In: Proceedings of AAAI-94, pp 313–318
115. Yugami N, Ohta Y, Hara H (1994) Improving repair-based constraint satisfaction methods by value propagation. In: Proceedings of AAAI-94, pp 344–349
116. Zhang J, Zhang H (1996) Combining local search and backtracking technqiues for constraint satisfaction. In: Proceedings of AAAI-96, pp 369–374

Hybrid Metaheuristics

Christian Blum, Jakob Puchinger, Günther Raidl, and Andrea Roli

Abstract One of the most interesting recent trends for what concerns research on metaheuristics is their hybridization with other techniques for optimization. In fact, the focus of research on metaheuristics has notably shifted from an algorithm-oriented point of view to a problem-oriented point of view. In other words, in contrast to promoting a certain metaheuristic, as, for example, in the eighties and the first half of the nineties, nowadays researchers focus much more on solving, as best as possible, the problem at hand. This has inevitably led to research that aims at combining different algorithmic components for the design of algorithms that are more powerful than the ones resulting from the implementation of pure metaheuristic strategies. Interestingly, the trend of hybridization is not restricted to the combination of algorithmic components originating from different metaheuristics, but has also been extended to the combination of exact algorithms and metaheuristics. In this chapter, we provide an overview of the most important lines of hybridization. In addition to representative examples, we present a literature review for each of the considered hybridization types.

1 Introduction

The term *metaheuristics* was coined in the Artificial Intelligence and Operations Research communities [23, 50]. It refers to general techniques for optimization that are not specific to a particular problem. Genetic and evolutionary algorithms, tabu search, simulated annealing, iterated local search, ant colony optimization, etc., are typical representatives falling under this generic term. Each of them has an individual historical background, follows certain paradigms and philosophies, and puts one or more particular strategic concepts in the foreground.

C. Blum (✉)
ALBCOM Research Group, Universitat Politècnica de Catalunya, Barcelona, Spain
e-mail: cblum@lsi.upc.edu

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_9,
© Springer Science+Business Media, LLC 2011

Over the last years, a large number of algorithms were reported that do not purely follow the concepts of one single traditional metaheuristic. On the contrary, they combine various algorithmic ideas, often originating from other branches of Artificial Intelligence, Operations Research, and Computer Science in general. These approaches are commonly referred to as *hybrid metaheuristics*. The lack of a precise definition of this term is sometimes subject to criticism. In our opinion, however, the relatively open nature of this term is rather helpful, as strict borderlines between related fields of research are often a hindrance for creative thinking and the exploration of new research directions.

The motivation behind the hybridization of different algorithmic concepts is usually to obtain better performing systems that exploit and unite advantages of the individual pure strategies, that is, hybrids are believed to benefit from *synergy*. In fact, choosing an adequate combination of multiple algorithmic concepts is often the key for achieving top performance in solving many hard optimization problems. However, developing a highly effective hybrid approach is, in general, a difficult task and sometimes even considered an art. Nevertheless, there are several hybridization types that have proven successful on many occasions, and they can provide some guidance.

The growing popularity of this line of research is documented by rather recent conferences and workshops such as *CPAIOR* [81, 106], *Hybrid Metaheuristics* [6, 13], and *Matheuristics* [73]. Moreover, the first book specifically devoted to hybrid metaheuristics has recently been published in 2008 [19]. In this chapter, we provide an overview of hybrid metaheuristics by illustrating prominent and paradigmatic examples, which range from the integration of metaheuristic techniques among themselves to the hybridization of metaheuristics with constraint and mathematical programming. The interested reader can find other reviews on hybrid metaheuristics in [19, 32, 39, 72, 92, 94].

2 Examples and Literature Overview

This section is devoted to the presentation of examples and short literature overviews concerning five important categories of hybrid metaheuristics. More specifically we focus on the hybridization of metaheuristics with (meta-)heuristics, constraint programming, tree search methods, problem relaxations, and dynamic programming. Each of the five categories mentioned above is treated in its own subsection. In each subsection, first, two representative examples are outlined, and then, a short literature overview is provided.

2.1 Hybridization of Metaheuristics with (Meta-)Heuristics

The hybridization of metaheuristics with (meta-)heuristics is quite popular, especially for what concerns the use of local search methods inside population-based methods. Indeed, most of the successful applications of evolutionary

computation and ant colony optimization make use of local search procedures for refining the generated solutions. This is because the major strength of population-based methods is to be found in their exploration capability. At the start of the search, they generally try to capture a global picture of the search space, and typically, rather simple and less problem-dependent operations are then iteratively applied to derive diverse new solutions successively focusing the search on more promising regions of the search space. Conversely, the strength of local search methods is their rather fast intensification capability, that is, the capability of quickly finding better solutions in the vicinity of given starting solutions. In summary, population-based methods are good in identifying promising areas of the search space in which local search methods can then quickly determine the best solutions. Therefore, this type of hybridization is often very successful. In the field of evolutionary algorithms, these hybrids even carry their own name, *memetic algorithms* [66].

In contrast to rather standard lines of hybridization, in this section, we present two examples that recently have received considerable attention. First, we present so-called *multilevel techniques*, and as a second example, we shortly outline the main idea of *hyper-heuristics*.

2.1.1 Multilevel Techniques

Multilevel techniques [110, 111] are heuristic frameworks with the potential of making the search process of a metaheuristic more effective and efficient. They have their origin in multigrid methods (see [24] for an overview) and have been discovered for combinatorial optimization by Walshaw et al. with the application to mesh partitioning [112], the traveling salesman problem [109], and graph coloring [110].

The basic idea of a multilevel technique is simple. Starting from the original problem instance under consideration, smaller and smaller instances are obtained by successive coarsening until some stopping condition is satisfied. This process generates a hierarchy of problem instances in which the problem instance assigned to a certain level is always smaller than (or of equal size as) the problem instance assigned to the next higher level. Then, a solution to the smallest problem instance is computed and successively transformed into a solution to the instance of the next level until a solution for the original problem instance is obtained. At each level, the determined solution may be subject to a refinement process. This is where metaheuristics come into play. They may be used for the generation of the solution to the lowest level, as well as for the refinement step at the other levels. This idea is illustrated in Fig. 1. In some multilevel approaches, a solution obtained for the original problem in this way is further used to guide a successive re-coarsening, which is again followed by the refinement process, and both of these phases are iterated to obtain even better solutions.

In general, applying metaheuristics in the context of a multilevel framework is only possible if an efficient and sensible way of coarsening a problem instance, and expanding solutions to higher levels, can be found. This is the case, for example,

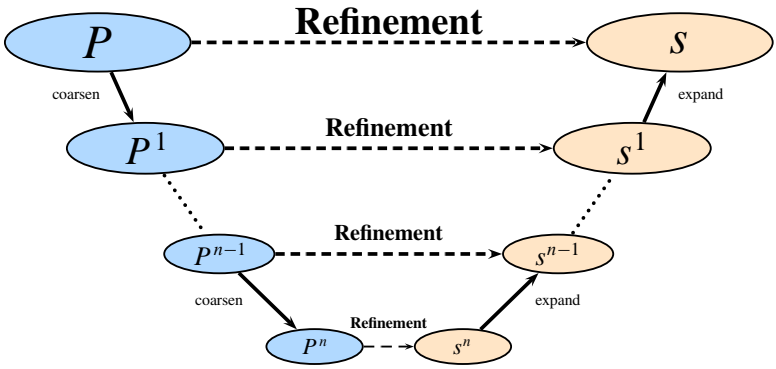


Fig. 1 General idea of a multilevel technique. The original problem instance is P. By means of an iterative process this problem instance is simplified until a stopping criterion causes to stop this process at some level n with instance P^n. Then, a metaheuristic may be used to generate a solution s^n for instance P^n, which is subsequently expanded into a solution $s^{n-1'}$ to instance P^{n-1} of the next higher level. A metaheuristic may then be used to refine solution $s^{n-1'}$, resulting in a possibly improved solution s^{n-1}. This process is repeated until a solution for the original problem instance is obtained

for many graph-based problems, where the coarsening of a problem instance is done by means of edge contractions. A recent survey over existing applications is given in [111].

Finally, so-called *variable fixing* strategies are to some extent related to multilevel techniques. Hereby, variables of the original problem are fixed to certain values (according to some, usually heuristic, criterion) and so optimization is performed over the resulting restricted search space. Examples of effective variable fixing strategies are the *core concepts* for knapsack problems [85,90].

2.1.2 Hyper-Heuristics

One of the main arguments in favor of metaheuristics has always been their generality. In principle, any metaheuristic can be applied to any combinatorial optimization problem. However, in recent years the main focus of many metaheuristic works has been on performance rather than generality. In other words, in many cases researchers tried to obtain the best possible performance for a specific problem, which led to a higher degree of customization. More customization, in turn, led to a decreasing amount of generality.

With this in mind, a recent branch of research on metaheuristics explicitly aims at the development of general methods that can potentially be applied to many related problems without much adaption. The aim is to raise the level of generality at which optimization systems operate. Algorithms adhering to this goal are nowadays known as *hyper-heuristics* [25]. They work on a higher level than classical metaheuristics, in the sense that they do not directly operate on the search space of the problem under consideration. Instead, they operate on a search space

consisting of lower-level heuristics–or even metaheuristics–for the tackled problem. Hyper-heuristics are broadly concerned with selecting the right (meta-)heuristic at any situation.

Timetabling and scheduling problems have shown to be a natural field of application for hyper-heuristics (see, for example, [26, 34]). In fact, one of the first applications of a hyper-heuristic is the one to the open shop scheduling (OSS) problem [41], even though in the term hyper-heuristics was still to come. The OSS problem can be formalized as follows. Given is a finite set of operations $O = \{o_1, ..., o_n\}$ which is partitioned into disjoint subsets $M = \{M_1, \ldots, M_{|M|}\}$. The operations in $M_i \in M$ have to be processed on the same machine. For the sake of simplicity, each set $M_i \in M$ of operations is identified with the machine they have to be processed on, and therefore M_i is referred to as a machine. Set O is additionally partitioned into disjoint subsets $J = \{J_1, \ldots, J_{|J|}\}$, where the set of operations $J_j \in J$ is called a job. Moreover, each operation $o \in O$ has a fixed processing time $p(o)$. Each machine can process at most one operation at a time. Operations must be processed without preemption (that is, once the processing of an operation has started, it must be completed without interruption). Operations belonging to the same job must be processed sequentially. A solution is obtained by assigning to each operation a time at which its processing is started. Together with the processing times, these starting times imply stopping times for all operations. The goal is to minimize the maximal stopping time, which is known as makespan optimization.

Fang et al. [41] proposed the use of an evolutionary hyper-heuristic that works on individuals with the following representation. Each individual is a string $s = (a_0, h_0, \ldots, a_n, h_n)$ where all $a_i \in \{1, \ldots, |J|\}$ refer to jobs and all $h_i \in H$ refer to heuristics identifiers, where H is a set of different heuristics. Given a string s, a schedule builder generates a solution as follows. For $i = 1, \ldots, n$, it schedules the first unscheduled operation of the a_i-th job with unscheduled operations according to heuristic h_i. The schedule builder keeps a circular list of uncompleted jobs to determine the a_i-th job with a modulo operation. The resulting hyper-heuristic can accordingly be seen as an evolutionary algorithm that works with an indirect representation.

2.1.3 Literature Overview

As mentioned already in the introduction to this section, the most popular way of hybridizing different (meta-)heuristic components concerns the use of local search methods within population-based techniques. In the field of evolutionary computation (EC), this type of hybridization has even developed into its own branch of research known as memetic algorithms. Interestingly, in recent years, a few examples have appeared for the use of components from population-based methods within metaheuristics based on local search. One of these examples concerns *population-based iterated local search* [103] where iterated local search is extended from working on a single solution to working on a population which is managed in the style of evolution strategies. Concerning additional examples, the paper by Lozano and García-Martínez [68] advocates the use of an evolutionary algorithm as a

perturbation technique within iterated local search, while Resende et al. [96] devise several versions of a hybrid algorithm based on greedy randomized adaptive search procedures (GRASP) and path relinking methodologies for the max–min diversity problem. The hybrids include, for example, evolutionary path relinking where the pool of elite solutions is evolved in order to be both diverse and of high quality.

Another example is the so-called *proximate optimality principle* (POP), which was first mentioned by Glover and Laguna in the context of tabu search [49]. It refers to the general intuition that good solutions are likely to have a similar structure and can therefore be found close to each other in the search space. Fleurent and Glover transfered this principle in [44] from complete to partial solutions in the context of GRASP. They suggested that mistakes introduced during the construction process may be undone by applying local search during (and not only at the end of) the GRASP construction phase. They proposed a practical implementation of POP in GRASP by applying local search at a few stages of the construction phase only. Another application of this concept can be found in [11] for the job shop scheduling problem.

Recent hybridizations based on a technique called *problem kernelization* are related to multilevel strategies and variable fixing. Problem kernelization can be seen as a more systematic approach based on tools from the field of parameterized complexity. The idea is to reduce a given problem instance in polynomial time to a so-called problem kernel such that an optimal solution to the problem kernel can be transformed in polynomial time to an optimal solution to the original problem instance. In [47], Gilmour and Dras propose several different ways of using the information given by the kernel of a problem instance, making the ant colony system more efficient for solving the minimum vertex cover problem. The most intuitive version applies the ant colony system directly to the problem kernel and subsequently transforms the best solution obtained for the kernel into a solution to the original problem instance.

An important branch of hybridization is the enhancement of metaheuristics with additional techniques for improving run-time, results, or both. Montemanni and Smith [78] propose an algorithm to solve the frequency assignment problem that is based on tabu search. Hereby, tabu search is enhanced by *heuristic manipulation*, a mechanism based on the idea that adding constraints to a problem results in a search space reduction, which, in turn, may facilitate the solution of the problem. Another example is the paper by Chaves et al. [28] that proposes a solution method for the capacitated centered clustering problem based on so-called *clustering search*. Hereby, a metaheuristic is used to create a set of initial solutions. These solutions are then clustered. Finally, local search is used to find possibly better solutions within the cluster regions.

2.2 Hybridization of Metaheuristics With Constraint Propagation Techniques

The rationale for integrating metaheuristics and constraint programming (CP) is that their respective strengths have a high degree of complementarity, and therefore, it is

natural to try to combine them in order to exploit possible synergies. Informally, we may ascribe the success of such combinations to the fact that metaheuristic algorithms are very effective in finding good-quality solutions to optimization problems while requiring a limited amount of computational resources. CP, instead, is particularly effective in finding feasible solutions to constrained problems. Moreover, the respective strengths are also a weakness of the counterpart, as it is known that metaheuristics are, in general, not particularly effective in tackling constraint satisfaction problems, while CP alone usually does not achieve an extraordinary high performance in solving (loosely constrained) optimization problems.

The integration of (meta-)heuristics and CP dates back to the late 1990s with the works by Pesant and Gendreau [82, 83] and their follow-ups [36, 100]. It is to be also noted that CPAIOR conferences have contributed considerably to research in this specific area.

In this section, we outline two among the most recent and paradigmatic examples of combination of metaheuristics and constraint propagation techniques. In Sect. 2.3, examples concerning tree-search will be illustrated. These two examples are chosen on purpose as they have a quite different nature and they show two of the main perspectives from which the integration of metaheuristics and CP can be conceived. The interested reader is also referred to Chap. 8 of this book which provides an in-depth discussion of the integration of local search and CP methods.

2.2.1 Ant Colony Optimization Hybridized with Constraint Programming

The combination of ant colony optimization (ACO) and CP is a representative example of the hybridization of two, considerably different, constructive techniques. ACO is a population-based metaheuristic inspired by the ant foraging behavior and further formalized as a model-based search metaheuristic [14,38]. In ACO, solutions are built in parallel by a probabilistic constructive procedure. The parameters of the probabilistic model upon which the procedure is based are dynamically adjusted by using a learning mechanism, very similar in spirit to reinforcement learning. In ACO terminology, it is common to denote the construction of one solution by the action of an *ant* which iteratively adds a solution component to the current partial solution. The choice of the component is probabilistic and biased toward components with a higher value of *pheromone*, which accounts for the attractiveness of a solution component. Once the solution is built, its components are rewarded by adding a quantity of pheromone positively correlated with solution quality. This operation increases the probability that a component belonging to a high-quality solution is chosen in the successive iteration. A more formal, yet slightly simplified, description of the general algorithm consists in viewing the ant's solution construction as the definition of one path along the search tree from the root note to one leaf. In one iteration of the algorithm, m solutions are independently built. The decisions taken at the choicepoints are based on a probabilistic model, whose parameters are updated as a function of the quality of the m solutions constructed. Thus, paths leading to good-quality solutions are *reinforced* from iteration to iteration and decisions which led

to good-quality solutions are more likely to be taken in the future. The overall result of this process is that good-quality solutions emerge by sampling the tree on the basis of a dynamic probabilistic model. For the sake of completeness, we also remark that the probabilistic model is usually combined with heuristic information, and the resulting search process is a mixture of randomized greedy search and learning.

The integration of ACO and CP holds the hope of effectively combining the learning capabilities of ACO with CP's power of constraint handling. The description we provide in the following is a succinct illustration of the main ideas behind the techniques proposed by Meyer in [76], in which this combination is described along with experimental results for a machine scheduling problem with sequence-dependent setup times.

The combination of ACO and CP can be seen from two dual standpoints. It is worthwhile to briefly outline both of them, even though the resulting method is virtually the same. The first perspective consists in focusing on solution construction in ACO and viewing CP as a tool employed by the ants while constructing a solution. Indeed, the usual approach for constraint handling in ACO–and metaheuristics in general–is to relax (a subset of) the problem constraints and penalize complete solutions violating such constraints. This procedure might not be very effective, especially in case of tightly constrained optimization problems. Hence, if ants use CP for finding a feasible solution, search is concentrated on finding a good-quality solution among the feasible ones and a large amount of computational effort can be saved. It is crucial at this point to observe that, in this algorithmic scheme, the probabilistic/greedy decision mechanism of ACO comes into play in the context of variable and value selection. In other words, while CP provides filtering, ACO is in charge of performing labeling. Reverting this point of view, we can interpret ACO as a heuristic for guiding search at each node inside a CP framework. Thus, variable and value selection is not only based on heuristics, but also on information derived from search history in the form of a probabilistic model, dynamically updated by means of a learning mechanism. Hence, the second perspective for interpreting ACO and CP hybrids is that ACO learns an effective labeling inside a CP solution process. As an example, let us consider Algorithm 1, which is a slight variation of the original one from [76], in which we show a general scheme of a solver based on CP and ACO. The algorithm is a variant of a classical CP search. The main differences are the procedure propagate_and_label that makes use of the probabilistic construction mechanism of ACO for variable and value ordering and the procedure update probabilistic model that updates the pheromone values.

Besides the already mentioned work by Meyer (and references therein), for completeness, we also mention another work that combines ACO with CP for tackling CSPs. In this work, the objective function that ACO tries to maximize is the number of assigned variables[1] and the algorithm has been successfully applied to the car sequencing problem [64].

[1] A very similar approach has been proposed by Prestwich [88].

Algorithm 1 CP-with-ACO

Input: Variables $x_1, \ldots, x_n$, domains $D_1, \ldots, D_n$, constraints $c_1, \ldots, c_n$
Output: A feasible assignment, optimal w.r.t. a given objective function or **nil** if no feasible solution exists

```
Setup initial domains for variables x_1, ..., x_n
Post initial constraints
while search not completed do
  for each ant do
      propagate_and_label(x_1, ..., x_n)
  end for
  update probabilistic model
  if new best solution found then
      Post upper bound constraint
  end if
end while
```

2.2.2 Solution-Guided Multi-Point Constructive Search

Solution-guided multi-point constructive search (SGMPCS) was introduced by Beck [9]. It defines a framework in which it is possible to combine any CP technique with search characteristics typical for metaheuristics. It is commonly acknowledged that effective metaheuristics optimize the balance between intensification–extensive search in the proximity of good-quality solutions and exploitation of search information–and diversification–exploration of unvisited search space areas [23]. This very property is exploited inside a CP search by SGMPCS, whose pseudo-code is shown in Algorithm 2, as it appears in [9]. The algorithm starts by building a set of elite solutions E which will be used to guide solution construction; this set of solutions will also be updated during search, and it can be initially derived by collecting the best solutions returned by several runs of the best performing metaheuristic algorithms for the problem at hand. In the following, we will assume that the cost function has to be minimized. In this description, we focus only on the part of the algorithm from lines 11 to 18, as the first part is a slight variation of local search without solution guidance. Here, one elite solution r is randomly chosen among the set E and used as a reference for the value heuristic of the CP search process (line 14). Before starting the CP search, an upper bound on the objective function is set (line 4 and line 12). Two policies can then be adopted, depending on the propagation strength: (a) *global bound*, that is, the upper bound is set to $c^* - 1$, where c^* is the (integer) cost of the best solution found so far, and (b) *local bound*, that is, the bound is either set to the cost of the worst elite solution minus one, or the cost of the starting solution minus one.

The CP search is performed in line 14. Let x be the decision variable chosen by the variable heuristic to be assigned, and let v be the value assigned to x in r; if v is in $dom(x)$ then the assignment $\langle x = v \rangle$ is made, otherwise another value in $dom(x)$ is chosen by any other value heuristic. In this way, the elite solution serves as a reference and search is first focused on the *neighborhood* of r, thus

Algorithm 2 SGMPCS-basic

```
1: initialize elite solution set E
2: while termination criteria unmet do
3:   if rand[0, 1) < p then
4:     set upper bound on cost function
5:     set fail limit, l
6:     s ← search(∅, l)
7:     if s ≠ ∅ ∧ s is better than worst(E) then
8:       replace worst(E) with s
9:     end if
10:  else
11:    r ← randomly chosen element of E
12:    set upper bound on cost function
13:    set fail limit, l
14:    s ← search(r, l)
15:    if s ≠ ∅ ∧ s is better than r then
16:      replace r with s
17:    end if
18:  end if
19: end while
20: return best(e)
```

performing a kind of intensification.[2] Search then continues until the search limit is reached, and the set of elite solutions is eventually updated. The search limit l can be chosen in such a way that it is progressively increased and the algorithm ultimately performs an exhaustive search, thus preserving the completeness of the CP search.

The combination of the solution-guided value heuristic and CP search, with a bound on the objective function, provides a very effective and robust solver, as indeed proved by recent results on the job shop scheduling problem [113], where an iterated tabu search is used to initialize SGMPCS.

2.2.3 Literature Overview

The publications on the integration of metaheuristics and CP are abundant in the literature and range from theory of algorithms to applications. A survey on possible ways of integrating metaheuristics and CP is provided by Focacci et al. in [45]. With a bit of oversimplification, we can consider three main approaches for the integration of metaheuristics (especially local search) and CP:

1. Metaheuristics are applied before CP, providing a valuable input, or vice versa.
2. Metaheuristics, mainly local search, use CP to efficiently explore the neighborhood of the current solution.

[2] A similar strategy has also been proposed in [77].

3. CP applies a metaheuristic in order to improve a solution (i.e., a leaf of the tree) or a partial solution (i.e., an inner node). Metaheuristic concepts can also be used to obtain incomplete but efficient tree exploration strategies.

The first approach can be seen as an instance of cooperative search, and it represents a rather loose integration. The second approach combines the advantages of a fast search space exploration by means of a metaheuristic with the efficient neighborhood exploration performed by a systematic method. A prominent example of such a kind of integration are large neighborhood search and related approaches [27,99] and CP-based local branching [65]. More examples can be found in [37, 82, 83]. The third approach preserves the search space exploration based on a systematic search (such as tree search), but sacrifices the exhaustive nature of the search [48,55,56,77]. The hybridization is usually achieved by integrating concepts and working mechanisms developed for metaheuristics (e.g., probabilistic choices, aspiration criteria, heuristic construction) into tree search methods. For example, instead of a chronological backtracking, a backjumping based on search history or information retrieved from local search samples can be performed. Other examples of this approach can be found in [63, 87, 98].

2.3 Hybridizing Metaheuristics with Tree Search

Optimization techniques can be characterized by their way of exploring the search space. Some algorithms consider the search space of an optimization problem in form of a tree, the so-called *search tree*, which is generally defined by an underlying solution construction mechanism. Each path from the root node of the search tree to one of the leafs corresponds to a step-by-step construction of a candidate solution. Inner nodes of the tree are partial solutions to the given problem. The process of moving from an inner node to one of its child nodes is called a solution construction step, or an extension of a partial solution.

The above mentioned class of tree search algorithms comprises incomplete and complete techniques. Examples of incomplete methods are constructive heuristics, such as greedy methods, and metaheuristics such as ACO [38] and GRASP [42]. Construction-based metaheuristics are iterative algorithms that employ repeated probabilistic solution constructions at each iteration. While ACO algorithms include a learning component, GRASP algorithms generally do not. An example of a complete method is branch & bound and its variants. An intersting heuristic version of a breadth-first branch & bound is *beam search* [80]. While branch & bound (implicitly) considers all nodes of a certain level of the search tree, beam search restricts the search to a certain number of nodes based on the bounding information.

In the following, we outline two different examples of hybridizing metaheuristics with tree search techniques. The first one uses branch & bound concepts within the solution construction process of a metaheuristic, while the second one exploits the fact that sub-problems of the original problem instance can often be efficiently solved by MIP solvers.

2.3.1 Beam-ACO

Both incomplete and complete tree-search algorithms have advantages as well as disadvantages. While ACO and GRASP generally find good solutions in a reasonable amount of computation time, they do not incorporate mechanism for avoiding the waste of computation time due to visiting the same solution more than once. Complete techniques on the other side guarantee to find an optimal solution. However, a user might not be prepared to accept overly large running times. Therefore, a relatively recent line of research promotes the incorporation of features originating from deterministic branch & bound derivatives such as beam search into construction-based metaheuristics. Examples are probabilistic beam search (PBS) [20], incomplete and non-deterministic tree search (ANTS) procedures [69–71], and Beam-ACO algorithms [15,17]. In the following, we outline the idea of Beam-ACO, which is the most prominent example.

As explained already in Sect. 2.2, standard ACO algorithms work basically as follows. At each iteration, first, a number of n_a artificial ants probabilistically construct solutions independently from each other. Hereby, the probabilities for the different extensions of the current partial solution are generated by means of greedy information and so-called pheromone values. Second, some of the solutions constructed in the current iteration, or in earlier iterations, are used for updating the pheromone values. This causes a change in the probability distribution over the search space that is computed on the basis of the greedy information and the pheromone values. The aim of the pheromone update is to focus the search over time on areas of the search space where high quality solutions can be found.

The central idea behind beam search is to allow the extension of partial solutions in several possible ways. At each step, the algorithm chooses at most $\lfloor \mu k_{bw} \rfloor$ feasible extensions of the partial solutions stored in a set B, called the *beam*. Hereby, k_{bw} is the so-called *beam width* that limits the size of B, and $\mu \geq 1$ is a parameter of the algorithm. The choice of feasible extensions is done deterministically by means of a greedy function that assigns a weight to each feasible extension. At the end of each step, the algorithm creates a new beam B by selecting up to k_{bw} partial solutions from the set of chosen feasible extensions. For this purpose, beam search algorithms calculate–in the case of minimization–a lower bound value for each chosen extension. Only the maximally k_{bw} best extensions–with respect to the lower bound–are chosen to constitute the new set B. Finally, the best found complete solution is returned.

The main idea of Beam-ACO is the *non-independent* probabilistic construction of n_a solutions per iteration, in the way of beam search. In other words, Beam-ACO performs at each iteration a beam search where $k_{bw} = n_a$. Hereby, the choice of feasible extensions is done probabilistically in the ACO-way. This algorithm has the advantage of using complementary types of information about the problem at hand: greedy information as well as bounding information. The benefits of using both types of information can be easily explained by means of an example: Let us consider the search tree shown in Fig. 2. The unique optimal solution is depicted in gray. For simplicity let us assume that the greedy information does not differentiate

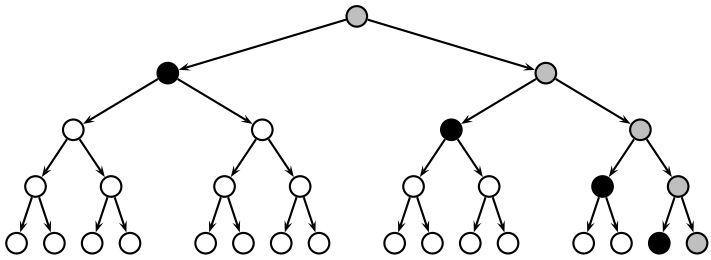


Fig. 2 Example of a search tree. The path of *light gray nodes* corresponds to the construction of the unique optimal solution. Moreover, available bounding information indicates that the *black nodes* cannot belong to the optimal solution

between the two children of a father node (that is, the two extensions of any father node have the same greedy value). Moreover, let us assume that available bounding information indicates that the nodes colored in black cannot belong to the optimal solution. Now, let us assume that we employ a probabilistic solution construction mechanism in which the probabilities for different extensions are proportional to their (relative) greedy value. This means that all extensions have a probability of 0.5 to be selected. An algorithm that does not consider the bounding information has, for each solution construction, a probability of 0.0625 to generate the optimal solution. On the other side, a (probabilistic) beam search algorithm with $k_{\text{bw}} \geq 2$ will solve this problem in only one run. A deeper analysis of the benefits of the use of complementary problem information is given in [22].

2.3.2 Large Neighborhood Search Based on MIP Solvers

Similarly, as CP is sometimes used for searching large neighborhoods (see Sect. 2.2), other tree search methods are also utilized for this purpose. Especially linear programming based branch & bound techniques, including branch-and-cut, are often a promising option when the problem at hand can be expressed by a mixed integer programming (MIP) model. The availability of highly effective general purpose MIP solvers, which are typically based on sophisticated branch-and-cut frameworks but nevertheless can be relatively easily applied, makes this approach particularly interesting in practice.

In many cases, directly solving the whole MIP is practically impossible due to excessive time or memory requirements. By fixing an appropriate portion of the variables and/or adding further constraints to the MIP in conjunction with a current incumbent solution, the whole search space is restricted to a certain large neighborhood of the incumbent. The intention is that this sub-problem can then be solved efficiently by the MIP-solver, possibly yielding a new incumbent. By iterating this approach, a large neighborhood search is performed. Of course, this concept can also be embedded into other, more sophisticated metaheuristic frameworks.

In the literature, numerous successful examples exist for such approaches. Among the more generally applicable ones is *local branching* [43], which works on MIPs with binary variables $(x_1, \ldots, x_n) \in \{0, 1\}^n$ and defines a neighborhood for a current solution $\overline{x} = (\overline{x}_1, \ldots, \overline{x}_n)$ in correspondence to the classical k-opt neighborhood by applying the so-called *local branching constraint*:

$$\Delta(x, \overline{x}) := \sum_{j \in \overline{S}} (1 - x_j) \quad + \sum_{j \in \{1,\ldots,n\} \setminus \overline{S}} x_j \leq k \ , \tag{1}$$

where $\overline{S}$ corresponds to the index set of the 0–1 variables that are set to one in the incumbent solution, i.e., $\overline{S} = \{j = 1, \ldots, n \mid \overline{x}_j = 1\}$. $\Delta(x, \overline{x})$ resembles the Hamming distance between x and $\overline{x}$ for integer values. The choice of parameter k is, in general, critical, because too small values lead to strongly restricted neighborhoods that often do not contain better solutions, while too large values result in excessive running times for solving the sub-problem. Therefore, the local branching framework includes a strategy for dynamically adapting k.

A successful problem-specific example for large neighborhood search by means of solving sub-MIPs via branch-and-cut has been described by Prandtstetter and Raidl for the Car Sequencing Problem (CarSP) [86]. This problem comes from the automobile manufacturing industry and has been used by the French Operations Research Society for the ROADEF Challenge 2005, which was a publicly open competition. The goal is to find a cost-effective arrangement of commissioned cars along a production line, i.e., a permutation. Although the individual cars are similar, each automobile requires particular components to be installed by different working bays along the assembly line. In addition to the different configurations of cars, each vehicle has to be painted with a specific color. The objective is to minimize the number of color changes while at the same time smoothing the workload in the assembly bays. To achieve the latter, restrictions of the form *"No more than l_c cars are allowed to require component c in any subsequence of m_c consecutive cars"* are imposed, and their violation is penalized by additional costs in the objective function.

In their work, Prandtstetter and Raidl describe a hybrid generalized variable neighborhood search (VNS) with embedded variable neighborhood descent (VND) [53]. Eight different types of neighborhood structures are exploited within the VND. Straight-forward are the neighborhoods induced by swapping two cars or moving one car to some other position, respectively. They can efficiently be examined by direct enumeration and incremental evaluation. Applying only these neighborhood structures, however, leads in general to solutions of moderate quality only. Therefore, the more powerful κ-exchange neighborhood structure is also included: A set of κ cars is selected either uniformly at random or by some greedy strategy that prefers cars involved in conflicts or color changes inducing high costs. These cars are then released from their current positions and reassigned in an optimal way, while all other cars remain fixed. A direct enumeration of this neighborhood would lead to excessive running times already for rather small κ, as the neighborhood size grows exponentially with κ. Instead, Prandtstetter and Raidl

introduced a special integer linear programming (ILP) formulation for the CarSP and utilize the MIP-solver ILOG CPLEX[3] for searching these neighborhoods. To avoid too long running times for larger κ, CPLEX is aborted when a certain time limit is exceeded and the so far best solution (if available) is returned.

Using this MIP-solver based large neighborhood search within the VNS substantially improves the overall solution quality. In the documented tests, κ is increased up to 65, i.e., 65 cars are released and tried to be optimally reassigned. Although this VNS did not take part at the competition as it was developed a little later and a fair comparison to the winning algorithms is hard due to the different hardware and imposed time limits, results clearly document that this VNS is highly competitive.

2.3.3 Literature Overview

The hybridization of metaheuristics with tree search techniques is probably one of the most popular hybridization approaches. Therefore, it is impossible to mention all works that have appeared in this field. In the following, we focus on a representative selection of papers, different to the ones mentioned already above, that is, different to Beam-ACO and large neighborhood search.

One of the first works on a combination of branch & bound with an evolutionary algorithm is the one by Nagar et al. [79] for a two-machine flow-shop scheduling problem. The proposed algorithm falls into the category of *multi-stage approaches*. Candidate solutions are represented as permutations of jobs. In the first stage, branch & bound is executed down to a predetermined level k. Hereby, suitable bounds are calculated and stored at each node of the branch & bound tree. The second stage consists in the execution of the evolutionary algorithm. Each generated partial solution is mapped onto the corresponding tree node. If the associated bounds indicate that no path below this node can lead to an optimal solution, the partial solution is subject to the application of a guided mutation operator.

Exact tree search methods have been used quite a few times in *solution merging*, which is based on the idea of deriving new and hopefully better solutions from the attributes originating from two or more input solutions. Solution merging is used in crossover operators of evolutionary algorithms, but also, for example, in *path relinking* [51]. Applegate et al. [3,4] were among the first to apply tree search methods in the context of merging. They present an application to the traveling salesman problem (TSP) where a set of different solutions is derived by a series of runs of the chained Lin-Kernighan iterated local search algorithm. The sets of edges of all these solutions are merged and the TSP is finally solved to optimality on the resulting (reduced) graph. Solutions are achieved that are typically superior to the best ones obtained by the iterated local search approach alone. Cotta and Troya [33] discuss merging in the context of a more general framework for combining branch & bound with evolutionary algorithms. They show the usefulness of applying branch & bound for identifying optimal offspring for different problems.

[3] `http://www.ilog.com`

The examples mentioned above are characterized by a subordinate use of an exact method within the metaheuristic. However, the literature also offers examples where metaheuristics are used for guiding the search process of an exact technique, or a heuristic derivate. For example, Rothberg [97] suggests a tight integration of an evolutionary algorithm in a branch-and-cut based MIP solver. The evolutionary algorithm is applied at regular intervals as a branch & bound tree node heuristic. Another example concerns the works presented in [21, 46]. In the proposed algorithms, the control flows of beam search and a memetic algorithm are intertwined: phases of beam search and the memetic algorithm alternate. Beam search purges its queue of open partial solutions by excluding those ones whose upper bounds are worse than the value of the best solution found by the memetic algorithm. On the other side, beam search guides the search of the memetic algorithm by injecting information about promising regions of the search space into the population.

Another example where metaheuristics may be used as a subordinate technique is *diving*, which is a mechanism for focusing the search process of branch & bound in an initial phase to neighborhoods of promising incumbents in order to quickly identify high-quality solutions. For example, Danna et al. [35] describe guided dives, a diving variant consisting in temporarily switching from breadth-first search to depth-first search. The branch to be processed next is chosen to be the one in which the branching variable is allowed to take the value it has in an incumbent solution. This results in a bias of the search process toward the neighborhood of this solution. Guided dives are repeatedly applied at regular intervals during the whole optimization process.

Finally, we want to mention some examples for the hybridization of metaheuristics with backtracking. In [57], the authors describe applications of various hybrid metaheuristics to problems ranging from car sequencing and graph coloring to scheduling. For example, a tabu search algorithm for the job shop scheduling problem is presented, combining local search with complete enumeration as well as limited backtracking search. *Nested partitioning* proposed by Shi and Ólafsson [101] is another example were breadth-first search combined with backtracking is used to explore the search space under the guidance of a metaheuristic. However, instead of variable-value assignments, in nested partitioning, the nodes of the search tree correspond explicitly to subspaces of solutions which are evaluated by a metaheuristic. In [1], for example, ACO is applied for this evaluation process, whereas in [102] local search is used for the same purpose.

2.4 Hybridization of Metaheuristics With Problem Relaxation

Guiding metaheuristics by *problem relaxation* has become another extremely popular hybridization approach in recent years. A so-called relaxed problem is obtained by simplifying or omitting constraints from the original problem formulation. The hope is, first, that the relaxed problem can be efficiently solved, and second, that the structure of an optimal solution to the relaxed problem together with its objective

function value can be used in some way for solving the original problem. For example, the optimal solution value of a relaxed problem can be seen as a bound for the optimal solution value of the original problem. Therefore, it can be used in a branch & bound algorithm for discarding parts of the search tree. An important type of relaxation in combinatorial optimization concerns dropping the integrality constraints of the involved variables from an MIP formulation. The resulting linear programming (LP) relaxation can then be solved to optimality by efficient methods such as the well-known *simplex algorithm*.

In the following, we present two examples of hybrid metaheuristics based on problem relaxations. The first one concerns a two-phase approach, in which problem relaxation is the crucial component of the first phase, and tabu search is used in the second phase. The second example concerns relaxation guided VNS, where the order in which neighborhoods are considered depends on their improvement-potential, which in turn is determined by problem relaxation.

2.4.1 Searching in the Vicinity of Non-Integral Solutions

The algorithm by Vasquez and Hao [107, 108] for the multi-dimensional 0–1 knapsack problem (MKP) is a prime example for a hybrid metaheuristic guided by problem relaxation. The main idea consists in solving a number of relaxed problems obtained by dropping the integrality constraints to optimality. This is done in a first phase. Afterward, in a second phase, tabu search is used to search around the optimal solutions to the relaxed problems. Hereby, care is taken that tabu search always stays within a sphere around the non-integral solutions. In the following, we outline this approach in more detail.

The MKP can be stated as follows. Given are n objects, where each object i has a profit c_i. Moreover, given are m resources, where each resource j has a capacity b_j, and each object i has a requirement a_{ij} of each resource j. Introducing a binary decision variable $x_i \in \{0, 1\}$ for each object i, the goal is the maximization of the total profit. This can be expressed as the following integer program:

$$\textbf{max} \sum_{i=1}^{n} c_i \cdot x_i \tag{2}$$

subject to

$$a_{ij} \cdot x_i \leq b_j \qquad j = 1, \ldots, m \tag{3}$$

$$x_i \in \{0, 1\} \qquad i = 1, \ldots, n \tag{4}$$

Note that the LP relaxation, which is obtained by replacing (4) by inequalities $0 \leq x_i \leq 1, \forall i = 1, \ldots, n$, may not be very helpful as it is known that the structure of an optimal solution to it may have not much in common with the structure of

an optimal solution to the original MKP. Therefore, the main idea by Vasquez and Hao was to add a constraint that fixes the number of items to be packed to a certain value k, i.e.,

$$\sum_{i=1}^{n} x_i = k, \tag{5}$$

where $k \in I\!N$, in addition to dropping the integrality constraints. In this way, the optimal solutions to the relaxed problems are generally closer in structure to good (or optimal) solutions to the original MKP. Relaxed problems are then solved to optimality for all k between a lower bound $k_{\min}$ and an upper bound $k_{\max}$, where $0 \leq k_{\min} \leq k_{\max} \leq n$. These bounds are computed by solving a linear program for each of them. We refer to [107] for more details.

In the second phase of the algorithm, tabu search is used to search around the non-integral solutions of the optimally solved relaxed problems. For each k, tabu search is restricted to solutions where exactly k objects are selected. Moreover, the distance of considered solutions to the computed optimal solution to the corresponding relaxed problem is restricted. In [108], the presented method is further enhanced by various variable fixing strategies.

2.4.2 Relaxation Guided Variable Neighborhood Search

VNS, including diverse variants such as VND, is a relatively simple but successful metaheuristic concept [53]. It relies on the principle of searching different neighborhood structures in a systematic way. When a sequence of neighborhood structures is used in which one neighborhood structure is entirely contained in the next one and a best improvement step function is performed, it is clear that these neighborhood structures will be applied in increasing order with respect to the size. In other cases, especially when neighborhood structures of different types but rather comparable size are to be used, the best order is usually unknown, and it also remains unclear how it can be determined. Even worse, in different phases of the search, different neighborhood orderings might be beneficial.

Puchinger and Raidl [89] therefore introduced *relaxation guided variable neighborhood search* (RGVNS). It follows the general VNS scheme but incorporates relaxation guided VND (RGVND), where the neighborhoods are ordered according to so-called *improvement-potentials*. These estimates are determined by computing bounds on the objective function values of the optimal solutions within each neighborhood. Such bounds are obtained by solving relaxations of the original problems associated with the neighborhoods.

It is therefore a prerequisite for developing an RGVNS approach that relaxed versions of the considered neighborhoods are available. Solving such relaxations should be much faster than searching the original neighborhoods. Another important aspect is that the bounds need to be reasonable estimates of the objective values of the best solutions in the neighborhood. Cases where the relaxations are misleading should not occur too frequently.

In [89], the multi-dimensional knapsack problem is used as an example to evaluate the RGVNS concept. Two neighborhood structures that are both parameterized by an integer $k \geq 1$ are defined and formulated as ILPs. In the first neighborhood structure, k items are removed from the knapsack, and any combination of items that were outside the knapsack before the removal are allowed to be added as long as the solution remains feasible. In the second neighborhood structure, k items not yet packed are forced to be added to the knapsack and any previously packed item is allowed to be removed to ensure the feasibility of the solution.

The RGVNS uses these neighborhoods with the item number k ranging from 1 to 10. The order in which the neighborhoods are considered is always dynamically determined by solving the LP-relaxations and sorting the neighborhoods according to decreasing solution values.

2.4.3 Literature Overview

In the following, we provide a representative selection of different approaches where LP relaxations are exploited in combination with metaheuristic approaches. A more general overview on combinations of metaheuristics with LP and ILP techniques is given in [72].

The probably most obvious way to utilize an optimal solution to the LP relaxation of a problem at hand is to directly derive a heuristic integer solution which is feasible for the original problem. Depending on the problem, this can be achieved by simple rounding or more sophisticated repairing strategies. For example, Raidl and Feltl [93] present a hybrid genetic algorithm (GA) for the generalized assignment problem. The LP relaxation of the problem is solved and its solution is exploited by a randomized rounding procedure to create an initial population of promising integral solutions. As these solutions are often infeasible, randomized repair and improvement operators are applied, yielding an even more meaningful initial population for the genetic algorithm.

Optimal solutions to LP relaxations are often exploited for guiding local improvement or for repairing infeasible candidate solutions. In [91], the multi-dimensional knapsack problem is considered again. The items are sorted according to increasing LP-values of their corresponding variables. A greedy repair procedure removes the items in this order from the knapsack until all constraints are fulfilled. In a greedy improvement procedure, items are considered in reverse order and included in the knapsack as long as no constraint is violated.

Dual variable values that come as a by-product of solving LP relaxations can also be exploited. Chu and Beasley [29] make use of them in their GA for the MKP by calculating pseudo-utility ratios for the variables and using them in similar ways as described above for the primal solution values. These pseudo-utility ratios tend to give better indications of the likeliness of the corresponding items to be included in an optimal solution.

A successful example for using further relaxation techniques is the hybrid Lagrangian GA for the prize collecting Steiner tree problem by Haouari and Siala [54]. It is based on a Lagrangian decomposition of a minimum spanning

tree such as ILP formulation of the problem. The volume algorithm is used for solving the Lagrangian dual [5]. After its termination, the GA is started, exploiting results obtained from the volume algorithm: The original graph is reduced by discarding edges, meaningful initial solutions are generated, and the objective function is modified by considering reduced costs.

For the knapsack constrained maximum spanning tree problem, a similar combination of Lagrangian decomposition and a genetic algorithm is described in Pirkwieser et al. [84]. A combination of a Lagrangian relaxation approach and a VND metaheuristic, which is also based on similar principles, has recently been developed for a real-world fiber optic network design problem by Leitner and Raidl [67].

Tamura et al. [104] tackle a job-shop scheduling problem and start from its ILP formulation. For each variable, they take the range of possible values and partition it into a set of subranges, which are then indexed. The encoded solutions of a GA are defined so that each position represents a variable, and its value corresponds to the index of one of the subranges. The fitness of such a chromosome is calculated using Lagrangian relaxation in order to obtain a bound on the optimal solution subject to constraints on the variable values which must fall within the represented ranges. When the GA terminates, an exhaustive search of the region identified as the most promising is carried out in a second stage.

Reimann [95] introduces an ACO algorithm for the symmetric TSP where an optimal solution to the minimum spanning tree (MST) relaxation is used for biasing the search of the artificial ants toward edges that form part of the minimum spanning tree. The proposed algorithm is based on computational experience indicating that an optimal solution to the symmetric TSP has about 70–80% of the edges in common with an optimal MST solution.

2.5 Hybridization of Metaheuristics With Dynamic Programming

Dynamic programming (DP) is another example of an optimization method from Operations Research and control theory that can be successfully integrated with metaheuristics, both in the case of constructive and local search techniques. DP provides a method for defining an optimal strategy that leads from an initial state to the final goal and it has been successfully applied to many optimization and control problems [10].

In this section, we illustrate two representative examples of hybrid solvers obtained by integrating DP with metaheuristics such as iterated local search, EC and ACO.

2.5.1 Iterated Dynasearch

Iterated dynasearch is a hybrid metaheuristic that uses DP as a neighborhood exploration strategy inside iterated local search [59]. The rationale behind this integration

is that, in simple local search strategies such as iterative improvement (a.k.a. descent search or hill climbing), the larger the neighborhood, the better the quality of the local optimum returned (on average). Suitable neighborhoods are often of exponential size, making it impractical to perform an explicit exhaustive lexicographic enumeration. Therefore, more computationally efficient neighborhood exploration techniques are required. In some cases, DP can make it possible to completely explore an exponential size neighborhood in polynomial time and space. In this paragraph, we illustrate the principles of *iterated dynasearch* with respect to its application to the single-machine total weighted tardiness scheduling problem (SMTWTSP) [31]. Further contributions to this work can be found in the recent literature [2, 52].

The SMTWTSP consists in finding the processing order of n jobs on one machine such that the total tardiness is minimized. More formally, for each of the n jobs, a processing time p_j, a positive weight w_j, and a due date d_j are given. Jobs are available at time zero and must be processed one at a time on the machine without interruption. Once a job ordering is provided, for each job, a completion time C_j can be computed, along with its tardiness $T_j = \max\{\mathrm{C_j} - \mathrm{d_j}, 0\}$. Hence, the function to be minimized is $\sum_{j=1}^{n} w_j T_j$.

For the moment, let us simply focus on the design of a suitable neighborhood structure for a best improvement local search. A natural neighborhood structure can be defined in terms of job permutations. Any permutation of n objects can be obtained by the repeated application of *swaps*. In a swap, two objects are exchanged and the resulting neighborhood is called $2-$exchange. In general, the $k-$exchange neighborhood, defined by sequences of swaps involving k objects, has an $O(n^k)$ size. Therefore, for efficiency concerns, usually only the cases of $k \in \{2, 3\}$ are considered.

The *dynasearch swap* neighborhood of a job sequence $\sigma = (\sigma(1), \ldots, \sigma(n))$ is composed of all the permutations of σ that can be generated by a series of *independent* swaps. Two swap moves $\{i, j\}$ and $\{k, l\}$ are independent if $\max\{i, j\} < \min\{k, l\}$ or $\min\{i, j\} > \max\{k, l\}$. This neighborhood has size $2^{n-1} - 1$; however, the independence of moves makes it possible to define a recursive enumeration algorithm based on DP such that the resulting exploration is polynomial in time and space.

Let σ_k be the partial job sequence ordering with minimum total weighted tardiness among the possible allowed orderings of the sequence $(\sigma(1), \ldots, \sigma(k))$ and let $F(\sigma_k)$ be the total weighted tardiness of σ_k. This partial sequence can be obtained from a partial optimal sequence σ_i, $0 \leq i < k$, by adding job σ_k. Two cases must be considered:

1. $i = k - 1$: job σ_k is simply appended to σ_i.
2. $i < k - 1$: job σ_k is first appended to σ_i and then immediately swapped with job $\sigma(i+1)$, hence the final sequence becomes $(\sigma(1), \ldots, \sigma(i), \sigma(k), \ldots, \sigma(i+1))$.

In both cases, the total tardiness $F(\sigma_k)$ can be easily evaluated by choosing the minimum of tardiness values computed as sum of independent contributions.

Algorithm 3 Iterated dynasearch

```
1: s ← GenerateInitialSolution()
2: ŝ ← BestImprovement(s ; dynasearch swap neighborhood)
3: while termination conditions not met do
4:    s′ ← Perturbation(ŝ; sequence of random swaps)
5:    ŝ′ ← BestImprovement(s′ ; dynasearch swap neighborhood)
6:    ŝ ← ApplyAcceptanceCriterion(ŝ′, ŝ, history)
7: end while
```

The best sequence σ_n can be computed recursively in a DP algorithm that runs in $O(n^3)$ and requires $O(n)$ space.[4]

A best-improvement local search based on the dynasearch neighborhood has, on average, a better performance than a best-improvement local search using the 2−exchange or the 3−exchange neighborhoods. In other words, the average total tardiness of the local optimum returned in the case of the dynasearch neighborhood is lower. Furthermore, this local search can be taken as the inner local search component for an iterated local search (ILS) algorithm [103], as illustrated in Algorithm 3. The algorithm iteratively perturbs the current solution s to provide an initial solution for a best improvement local search[5]. The local optimum found by the local search replaces the current solution s depending on the given acceptance criterion. In a nutshell, ILS performs a local search in the sampled space of local optima, rather than in the global search space.

2.5.2 Dynamic Programming for Solving Subproblems

In [18], Blum and Blesa present the use of a DP algorithm in two different metaheuristics for the k-cardinality tree (KCT) problem. The general idea of their approaches is not limited to the KCT problem and can, potentially, be used for other subset problems. Basically, the idea is to let the metaheuristic generate objects that are bigger than solutions, containing in general an exponential number of solutions to the problem under consideration. DP is then used to efficiently find for each object the best solution that it contains. In the following, we explain how this idea was implemented in the context of the KCT problem.

Technically, the KCT problem can be described as follows. Let $G(V, E)$ be an undirected graph in which each edge $e \in E$ has a weight $w_e \geq 0$, and each node $v \in V$ has a weight $w_v \geq 0$. Furthermore, we denote by $\mathcal{T}_k$ the set of all k-cardinality trees in G, that is, the set of all trees in G with exactly k edges. The problem consists of finding a k-cardinality tree $T_k \in \mathcal{T}_k$ that minimizes

[4] For brevity, we omit the details and point the interested reader to [31].

[5] In general, any local search algorithm can be used.

$$f(T_k) = \left(\sum_{e \in E_{T_k}} w_e \right) + \left(\sum_{v \in V_{T_k}} w_v \right), \tag{6}$$

where E_{T_k} denotes the set of edges and V_{T_k} the set of nodes of T_k. When G is itself a tree, the KCT problem can be solved efficiently by DP [16]. This fact was used by Blum and Blesa within their ACO approach and their EC approach presented in [18].

The main idea of the ACO approach is as follows. At each iteration, a number of n_a artificial ants step-by-step construct trees in G. However, instead of aborting the tree construction once k edges have been added, the tree construction is continued until the tree has $k < l \leq |V| - 1$ edges. Then, DP is applied to each of the constructed l-cardinality trees in order to find the best k-cardinality trees they contain. The resulting k-cardinality trees are then used for updating the pheromone values.

Concerning the EC approach presented in [18], DP is used within the crossover operator, which needs two different k-cardinality trees as input. Then, the two trees are merged, resulting in a tree that contains more than k edges. Finally, DP is applied to this bigger tree in order to obtain the best k-cardinality tree it contains. This resulting tree is the output of the crossover operator.

Both algorithm versions obtain better solutions faster than their *standard* counterparts. This holds especially for large input graphs.

2.5.3 Literature Overview

Besides the approaches outlined above, a few other hybrids involving DP have appeared in the literature. In the following, we shortly present a representative selection. For example, Hu and Raidl [60] use DP within an evolutionary algorithm as a mechanism for generating the best solution that can be obtained from an incomplete solution. They consider the generalized TSP in which a clustered graph is given and a shortest tour visiting exactly one node from each cluster is requested. Their algorithm is based on VNS. Among other ways of representing a candidate solution, a permutation of clusters is given, representing the order in which the clusters are to be visited. A DP procedure is then used to derive a corresponding optimal selection of particular nodes from each cluster.

A somewhat related approach is presented in [40] where a dynamic facility layout problem with unequal sizes of departments is considered. Department sizes may even change from one period to the next. The proposed algorithm combines an evolutionary technique and DP in the following way. A number of T evolutionary algorithms run in parallel, one for each of T periods. In each case, a solution is a layout for the respective period. However, a solution to the original problem is a sequence of T periods. Therefore, the evaluation of a layout of a single period must take into account the best combination of layouts that can be generated given the current populations. This is done by DP.

In [61], DP is used purely as a decoder for tackling the rectangle packing problem with general spatial costs, which consists in packing given rectangles without overlap in the plane so that the maximum cost of the rectangles is minimized.

The following examples deal with hybridizations based on problem decompositions. In [114], the authors propose a hybrid method that combines adaptive memory, sparse DP, and reduction techniques to reduce and explore the search space. The first step consists in the generation of a bi-partition of the variables. This bi-partition results in a small core problem with at most 15 variables. This small problem is solved using the forward phase of DP. The space defined by the remaining variables is explored using tabu search. Hereby, each partial solution is completed with the information stored during the forward phase of DP. The authors state that their approach can be seen as a global intensification mechanism, since at each iteration, the move evaluations involve solving a reduced problem implicitly.

The application of DP to subproblems is also proposed in [105], in which the authors introduce and tackle a multidrug cancer chemotherapy model to simulate the possible response of the tumor cells under drug administration. The objective is to minimize the tumor size under a set of constraints. A so-called adaptive elitist GA is combined with a local search technique called *iterative dynamic programming*. This local search technique works by subdividing the problem into subproblems and optimizing the subproblems separately by DP.

Finally, we would like to point out an interesting heuristic version of DP known as *bounded dynamic programming* in which at each level the number of states is heuristically reduced. In this way, the authors of [8] were able to find most optimal solutions to benchmark instances of the simple assembly line balancing problem in a reduced amount of computation time.

3 Discussion and Conclusions

The process of designing and implementing effective hybrid metaheuristics can be rather complicated and involves knowledge about a broad spectrum of algorithmic techniques, programming and data structures, as well as algorithm engineering and statistics. In fact, it is hardly possible to provide guidelines for the successful development of hybrid metaheuristics. For the development of well-performing algorithms, the authors can only recommend (1) the careful search of the literature for the most successful optimization approaches for the problem at hand or for similar problems, and (2) the study of different ways of combining the most promising features of the identified approaches. This chapter may serve as a starting point for this purpose.

Indeed, for the extraction of useful guidelines for the development of hybrid metaheuristics, it is probably necessary to improve the research methodology that is nowadays commonly used in the metaheuristics field. The used research

methodology is characterized by an ad hoc approach that consists in mixing different algorithmic components without many serious attempts to identify the contribution of different components to the algorithms' performance. In our opinion, the research community should move toward a sound scientific methodology consisting of theoretical models for describing properties of hybrid metaheuristics and using an experimental methodology as done in natural sciences. In fact, among the key points of the engineering process of a hybrid metaheuristic are *scientific testing* and the *statistical assessment* of the results.

The goal of scientific testing [58] is to abstract from actual implementations and study, empirically and through predictive models, the effect of algorithmic components. This investigation approach can be particularly useful in the case of conjectures on algorithm behavior that, while being widespread in the community, have not yet been subject to validation. Scientific testing of algorithms requires scientists to formulate one or more hypotheses and test them both via experiments and theoretical models, trying to clearly isolate the algorithmic components under investigation (i.e., comparisons must be made *ceteris paribus* –all other things being equal). This methodology assures not only the generation of sound results, but it also makes it possible to generalize them.

Testing (metaheuristic) algorithms is also an empirical process with several sources of stochasticity–concerning both the algorithm and the problem instances. Hence, a proper statistical methodology has to be adopted, both to support conjectures and inferences and to validate results. This part is often overlooked, while it is of fundamental importance. In particular, the application of a sound statistical analysis should be one of the mandatory requirements for publications. Guidelines on a sound statistical approach to Artificial Intelligence can be found in the book by Cohen [30].

Researchers interested in this topic can find useful contributions in the literature about Artificial Intelligence and Operations Research addressing the issues of experimental methodology. Besides the already cited paper by Hooker [58], we mention the very well known paper by Johnson [62] that can be seen as an introduction to empirical testing from a theoretician's point of view. Furthermore, discussions on the overall experimental methodology or just one of its issues, such as parameter tuning or the statistical assessment of results, can be found in [7, 12, 74, 75].

We are convinced that research on hybrid metaheuristics is still in its early stages. In the years to come, most publications on metaheuristic applications will be concerned with hybrids. We hope that this chapter contributes to give some more structure and guidance to this very interesting line of research.

Acknowledgements This work was supported by grant TIN2007-66523 (FORMALISM) of the Spanish Government. In addition, Christian Blum acknowledges support from the *Ramón y Cajal* program of the Spanish Ministry of Science and Innovation.

References

1. Al-Shihabi S (2004) Ants for sampling in the nested partition algorithm. In: Blum C, Roli A, Sampels M (eds) Proceedings of HM 2004 – first international workshop on hybrid metaheuristics, pp 11–18
2. Angel E, Bampis E (2005) A multi-start dynasearch algorithm for the time dependent single-machine total weighted tardiness scheduling problem. Eur J Oper Res 162(1):281–289
3. Applegate DL, Bixby RE, Chvátal V, Cook WJ (1998) On the solution of the traveling salesman problem. Documenta Mathematica, Extra Volume ICM III:645–656
4. Applegate DL, Bixby RE, Chvátal V, Cook WJ (2007) The traveling salesman problem: a computational study. Princeton Series in Applied mathematics. Princeton University Press, Princeton
5. Barahona F, Anbil R (2000) The volume algorithm: producing primal solutions with a subgradient method. Math Program A 87(3):385–399
6. Bartz-Beielstein T, Blesa Aguilera MJ, Blum C, Naujoks B, Roli A, Rudolph G, Sampels M (eds) (2007) In: Proceedings of HM 2007 – fourth international workshop on hybrid metaheuristics. Lecture notes in computer science, vol 4771. Springer, Berlin
7. Bartz-Beielstein T, Chiarandini M, Paquete L, Preuss M (eds) (2009) Empirical methods for the analysis of optimization algorithms. Springer, Heidelberg
8. Bautista J, Pereira J (2009) A dynamic programming based heuristic for the assembly line balancing problem. Eur J Oper Res 194(3):787–794
9. Beck JC (2007) Solution-guided multi-point constructive search for job shop scheduling. J Artif Intell Res 29:49–77
10. Bertsekas DP (2007) Dynamic programming and optimal control, 3rd edition. Athena Scientific, Nashua
11. Binato S, Hery WJ, Loewenstern D, Resende MGC (2001) A GRASP for job shop scheduling. In Ribeiro CC, Hansen P (eds) Essays and surveys on metaheuristics. Kluwer, Boston, pp 59–79
12. Birattari M (2009) Tuning metaheuristics: a machine learning perspective, vol 197. Studies in computational intelligence. Springer, Berlin
13. Blesa Aguilera MJ, Blum C, Cotta C, Fernández AJ, Gallardo JE, Roli A, Sampels M (eds) (2008) In: Proceedings of HM 2008 – fifth international workshop on hybrid metaheuristics. Lecture notes in computer science, vol 5296. Springer, Berlin
14. Blum C (2005) Ant colony optimization: introduction and recent trends. Phys Life Rev 2(4):353–373
15. Blum C (2005) Beam-ACO–hybridizing ant colony optimization with beam search: an application to open shop scheduling. Comput Oper Res 32:1565–1591
16. Blum C (2007) Revisiting dynamic programming for finding optimal subtrees in trees. Eur J Oper Res 177(1):102–115
17. Blum C (2008) Beam-ACO for simple assembly line balancing. INFORMS J Comput 20(4):618–627
18. Blum C, Blesa MJ (2009) Solving the KCT problem: large-scale neighborhood search and solution merging. In: Alba E, Blum C, Isasi P, León C, Gómez JA (eds) Optimization Techniques for Solving Complex Problems. Wiley, Hoboken, pp 407–421
19. Blum C, Blesa Aguilera MJ, Roli A, Sampels M (eds) (2008) Hybrid metaheuristics – an emerging approach to optimization. Studies in computational intelligence, vol 114. Springer, Berlin
20. Blum C, Cotta C, Fernández AJ, Gallardo JE (2007) A probabilistic beam search algorithm for the shortest common supersequence problem. In: Cotta C, van Hemert JI (eds) Proceedings of EvoCOP 2007 – seventh european conference on evolutionary computation in combinatorial optimisation. Lecture notes in computer science, vol 4446. Springer, Berlin, pp 36–47
21. Blum C, Cotta C, Fernández AJ, Gallardo JE, Mastrolilli M (2008) Hybridization of metaheuristics with branch and bound derivatives. In: Blum et al. (eds) Hybrid metaheuristics – an emerging approach to optimization. Studies in computational intelligence, vol 114. Springer, Berlin, pp 85–116

22. Blum C, Mastrolilli M (2007) Using branch&bound concepts in construction-based metaheuristics: exploiting the dual problem knowledge. In: Bartz-Beielstein et al. (eds) Proceedings of HM 2007 – fourth international workshop on hybrid metaheuristics. Lecture notes in computer science, vol 4771. Springer, Berlin, pp 123–139
23. Blum C, Roli A (2003) Metaheuristics in combinatorial optimization: overview and conceptual comparison. ACM Comput Surv 35(3):268–308
24. Brandt A (1988) Multilevel computations: review and recent developments. In: McCormick SF (ed) Multigrid methods: theory, applications, and supercomputing, proceedings of the 3rd copper mountain conference on multigrid methods. Lecture notes in pure and applied mathematics, vol 110. Marcel Dekker, New York, pp 35–62
25. Burke EK, Kendall G, Newall J, Hart E, Ross P, Schulenburg S (2003) Hyper-heuristics: an emerging direction in modern search technology. In: Glover F, Kochenberger G (eds) Handbook of metaheuristics. International series in operations research & management science, vol 57. Kluwer, Dordrecht, pp 457–474
26. Burke EK, McCollum B, Meisels A, Petrovic S, Qu R (2007) A graph-based hyper-heuristic for educational timetabling problems. Eur J Oper Res 176(1):177–192
27. Caseau Y, Laburthe F (1999) Effective forget-and-extend heuristics for scheduling problems. In: Proceedings of CP-AI-OR'02 – fourth international workshop on integration of AI and OR techniques in constraint programming for combinatorial optimization problems
28. Chaves AA, Correa FA, Lorena LAN (2008) Clustering search heuristic for the capacitated p-median problem. In: Corchado E, Corchado JM, Abraham A (eds) Innovations in hybrid intelligent systems. Advances in soft computing, vol 44. Springer, Berlin, pp 136–143
29. Chu PC, Beasley JE (1998) A genetic algorithm for the multidimensional knapsack problem. J Heuristics 4:63–86
30. Cohen PR (1995) Empirical methods for artificial intelligence. The MIT Press, Cambridge
31. Congram RK, Potts CN, van de Velde SL (2002) An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. INFORMS J Comput 14(1):52–67
32. Cotta C (1998) A study of hybridisation techniques and their application to the design of evolutionary algorithms. AI Comm 11(3–4):223–224
33. Cotta C, Troya JM (2003) Embedding branch and bound within evolutionary algorithms. Appl Intell 18:137–153
34. Cowling PI, Chakhlevitch K (2007) Using a large set of low level heuristics in a hyperheuristic approach to personnel scheduling. In: Dahal KP, Tan KC, Cowling PI (eds) Evolutionary scheduling. Springer, Berlin, pp 543–576
35. Danna E, Rothberg E, Le Pape C (2005) Exploring relaxation induced neighborhoods to improve MIP solutions. Math Program Ser A 102:71–90
36. De Backer B, Furnon V, Shaw P (2000) Solving vehicle routing problems using constraint programming and metaheuristics. J Heuristics 6:501–523
37. Dell'Amico M, Lodi A (2002) On the integration of metaheuristic strategies in constraint programming. In: Rego C, Alidaee B (eds) Metaheuristic optimization via memory and evolution. Operations research/computer science interfaces series, vol 30. Springer, Berlin, pp 357–371
38. Dorigo M, Stützle T (2004) Ant colony optimization. MIT Press, Cambridge
39. Dumitrescu I, Stuetzle T (2003) Combinations of local search and exact algorithms. In: Günther R, Raidl et al. (eds) Applications of evolutionary computation. Lecture notes in computer science, vol 2611. Springer, Berlin, pp 211–223
40. Dunker T, Radons G, Westkämper E (2005) Combining evolutionary computation and dynamic programming for solving a dynamic facility layout problem. Eur J Oper Res 165(1):55–69
41. Fang H-L, Ross PM, Corne C (1994) A promising hybrid GA/Hybrid approach for open-shop scheduling problems. In: Cohn A (ed) Proceedings of ECAI 1994 – eleventh european conference on artificial intelligence. Wiley, Hoboken, pp 590–594
42. Feo TA, Resende MGC (1995) Greedy randomized adaptive search procedures. J Global Optim 6:109–133

43. Fischetti M, Lodi A (2003) Local branching. Math Program Ser B 98:23–47
44. Fleurent C, Glover F (1999) Improved constructive multistart strategies for the quadratic assignment problem using adaptive memory. INFORMS J Comput 11:198–204
45. Focacci F, Laburthe F, Lodi A (2003) Local search and constraint programming. In: Glover F, Kochenberger G (eds) Handbook of metaheuristics. International series in operations research & management science, vol 57. Kluwer, Dordrecht, pp 369–403
46. Gallardo JE, Cotta C, Fernández AJ (2007) On the hybridization of memetic algorithms with branch-and-bound techniques. IEEE Trans Syst Man Cybern B 37(1):77–83
47. Gilmour S, Dras M (2006) Kernelization as heuristic structure for the vertex cover problem. In: Dorigo M et al. (eds) Proceedings of ANTS 2006 – 5th international workshop on ant colony optimization and swarm intelligence. Lecture notes in computer science, vol 4150. Springer, pages 452–459
48. Ginsberg ML (1993) Dynamic backtracking. J Artif Intell Res 1:25–46
49. Glover F (1968) Surrogate constraints. Oper Res 16(4):741–749
50. Glover F, Kochenberger G (eds) (2003) Handbook of metaheuristics. International series in operations research & management science, vol 57. Kluwer, Dordrecht
51. Glover F, Laguna M, Martí R (2000) Fundamentals of scatter search and path relinking. Control Cybern 39(3):653–684
52. Grosso A, Della Croce F, Tadei R (2004) An enhanced dynasearch neighborhood for the single-machine total weighted tardiness scheduling problem. Oper Res Lett 32(1):68–72
53. Hansen P, Mladenović N (1999) An introduction to variable neighborhood search. In: Voss S, Martello S, Osman IH, Roucairol C (eds) Metaheuristics, advances and trends in local search paradigms for optimization. Kluwer, Boston, pp 433–458
54. Haouari M, Siala JC (2006) A hybrid Lagrangian genetic algorithm for the prize collecting Steiner tree problem. Comput Oper Res 33(5):1274–1288
55. Harvey WD (1995) Nonsystematic backtracking search. PhD thesis, CIRL, University of Oregon, Eugene, Orgeon
56. Harvey WD, Ginsberg ML (1995) Limited discrepancy search. In: Mellish CS (ed) Proceedings of IJCAI 1995 – 14th international joint conference on artificial intelligence, vol 1. Morgan Kaufmann, San Mateo, pp 607–615
57. Van Hentenryck P, Michel L (2005) Constraint-based local search. MIT Press, Cambridge
58. Hooker JN (1995) Testing heuristics: we have it all wrong. J Heuristics 1(1):33–42
59. Hoos H, Stützle T (2005) Stochastic local search – foundations and applications. Morgan Kaufmann , San Francisco
60. Hu B, Raidl GR (2008) Effective neighborhood structures for the generalized traveling salesman problem. In: van Hemert JI, Cotta C (eds) Evolutionary computation in combinatorial optimisation – EvoCOP 2008. Lecture notes in computer science, vol 4972. Springer, pp 36–47
61. Imahori S, Yagiura M, Ibaraki T (2005) Improved local search algorithms for the rectangle packing problem with general spatial costs. Eur J Oper Res 167(1):48–67
62. Johnson DS (2002) A theoretician's guide to the experimental analysis of algorithms. In: Johnson DS, Goldwasser MH, McGeoch CC (eds) Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS implementation challenges. American Mathematical Society, Providence, pp 215–250
63. Jussien N, Lhomme O (2002) Local search with constraint propagation and conflict-based heuristics. Artif Intell 139:21–45
64. Khichane M, Albert P, Solnon C (2008) Integration of ACO in a constraint programming language. In Ant colony optimization and swarm intelligence, 6th international workshop, ANTS 2008. Lecture notes in computer science, vol 5217. Springer, Berlin
65. Kiziltan Z, Lodi A, Milano M, Parisini F (2007) CP-based local branching. In: Principles and practice of constraint programming – CP 2007. Lecture notes in computer science, vol 4741. Springer, Berlin
66. Krasnogor N, Smith J (2005) A tutorial for competent memetic algorithms: model, taxonomy, and design issues. IEEE Trans Evol Comput 9(5):474–488

67. Leitner M, Raidl GR (2008) Lagrangian decomposition, metaheuristics, and hybrid approaches for the design of the last mile in fiber optic networks. In: Blesa Aguilera et al. (eds) Proceedings of HM 2008 – fifth international workshop on hybrid metaheuristics. Lecture notes in computer science, vol 5296. Springer, Berlin, pp 158–174
68. Lozano M, García-Martínez C (2010) Hybrid metaheuristics with evolutionary algorithms specializing in intensification and diversification: overview and progress report. Comput Oper Res 37(3):481–497
69. Maniezzo V (1999) Exact and approximate nondeterministic tree-search procedures for the quadratic assignment problem. INFORMS J Comput 11(4):358–369
70. Maniezzo V, Carbonaro A (2000) An ANTS heuristic for the frequency assignment problem. Future Generat Comput Syst 16:927–935
71. Maniezzo V, Milandri M (2002) An ant-based framework for very strongly constrained problems. In: Dorigo M, Di Caro G, Sampels M (eds) Proceedings of ANTS 2002 – 3rd international workshop on ant algorithms Lecture notes in computer science, vol 2463. Springer, Berlin, pages 222–227
72. Maniezzo V, Stützle T, Voss S (eds) (2009) Matheuristics – hybridizing metaheuristics and mathematical programming. Annals of information systems, vol 10. Springer, Heidelberg
73. Matheuristics 2008. http://astarte.csr.unibo.it/matheuristics2008/. Accessed April 2009
74. McGeoch CC (1996) Toward an experimental method for algorithm simulation. INFORMS J Comput 8(1):1–15
75. McGeoch CC (2001) Experimental analysis of algorithms. Not Am Math Soc 48(3):304–311
76. Meyer B (2008) Hybrids of constructive meta-heuristics and constraint programming: a case study with ACO, chapter 6. Blum et al. (eds) Hybrid metaheuristics – an emerging approach to optimization. Studies in computational intelligence, vol 114. Springer, Berlin
77. Milano M, Roli A (2002) On the relation between complete and incomplete search: an informal discussion. In: Proceedings of CP-AI-OR'02 – fourth international workshop on integration of AI and OR techniques in constraint programming for combinatorial optimization problems pp 237–250
78. Montemanni R, Smith DH (2010) Heuristic manipulation, tabu search and frequency assignment. Comput Oper Res 37(3):543–551
79. Nagar A, Heragu SS, Haddock J (1995) A meta-heuristic algorithm for a bi-criteria scheduling problem. Ann Oper Res 63:397–414
80. Ow PS, Morton TE (1988) Filtered beam search in scheduling. Int J Prod Res 26:297–307
81. Perron L, Trick MA (eds)(2008) Proceedings of CPAIOR 2008 – 5th international conference on the integration of ai and or techniques in constraint programming for combinatorial optimization problems. Lecture notes in computer science, vol 5015. Springer, Berlin
82. Pesant G, Gendreau M (1996) A view of local search in constraint programming. In: Principles and practice of constraint programming - CP'96. Lecture notes in computer science, vol 1118. Springer, Heidelberg, pp 353–366
83. Pesant G, Gendreau M (1999) A constraint programming framework for local search methods. J Heuristics 5:255–279
84. Pirkwieser S, Raidl GR, Puchinger J (2007) Combining Lagrangian decomposition with an evolutionary algorithm for the knapsack constrained maximum spanning tree problem. In: Cotta C, van Hemert JI (eds) Evolutionary computation in combinatorial optimization – EvoCOP 2007. Lecture notes in computer science, vol 4446. Springer, Berlin, pp 176–187
85. Pisinger D (1999) Core problems in knapsack algorithms. Oper Res 47:570–575
86. Prandtstetter M, Raidl GR (2008) An integer linear programming approach and a hybrid variable neighborhood search for the car sequencing problem. Eur J Oper Res 191(3):1004–1022
87. Prestwich S (2002) Combining the scalability of local search with the pruning techniques of systematic search. Ann Oper Res 115:51–72
88. Prestwich S (2008) The relation between complete and incomplete search, chapter 3. Blum et al. (eds) Hybrid metaheuristics – an emerging approach to optimization. Studies in computational intelligence, vol 114. Springer, Berlin
89. Puchinger J, Raidl GR (2008) Bringing order into the neighborhoods: relaxation guided variable neighborhood search. J Heuristics 14(5):457–472

90. Puchinger J, Raidl GR, Pferschy U (2006) The core concept for the multidimensional knapsack problem. In: Gottlieb J, Raidl GR (eds) Evolutionary computation in combinatorial optimization – EvoCOP 2006. Lecture notes in computer science, vol 3906. Springer, Berlin, pp 195–208
91. Raidl GR (1998) An improved genetic algorithm for the multiconstrained 0–1 knapsack problem. In: Fogel DB et al. (eds) Proceedings of the 1998 IEEE international conference on evolutionary computation. IEEE Press, pp 207–211
92. Raidl GR (2006) A unified view on hybrid metaheuristics. In: Almeida F, Blesa Aguilera MJ, Blum C, Moreno Vega JM, Pérez Pérez M, Roli A, Sampels M (eds) Proceedings of HM 2006 – third international workshop on hybrid metaheuristics. Lecture notes in computer science, vol 4030. Springer, Berlin, pp 1–12
93. Raidl GR, Feltl H (2004) An improved hybrid genetic algorithm for the generalized assignment problem. In: Haddadd HM et al (eds) Proceedings of the 2003 ACM symposium on applied computing. ACM Press, pp 990–995
94. Raidl GR, Puchinger J, Blum C (2010) Metaheuristic hybrids. In: Gendreau M, Potvin JY (eds) Handbook of metaheuristics. Springer, 2nd edition
95. Reimann M (2007) Guiding ACO by problem relaxation: a case study on the symmetric TSP. In: Bartz-Beielstein et al. (eds) In: Proceedings of HM 2007 – fourth international workshop on hybrid metaheuristics. Lecture notes in computer science,vol 4771. Springer, Berlin, pp 45–55
96. Resende MGC, Martí R, Gallego M, Duarte A (2010) GRASP and path relinking for the max–min diversity problem. Comput Oper Res 37(3):498–508
97. Rothberg E (2007) An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS J Comput 19(4):534–541
98. Schaerf A (1997) Combining local search and look-ahead for scheduling and constraint satisfaction problems. In: Proceedings of IJCAI 1997 – 15th international joint conference on artificial intelligence. Morgan Kaufmann, San Mateo, CA, pp 1254–1259
99. Shaw P (1998) Using constraint programming and local search methods to solve vehicle routing problems. In: Maher M, Puget J-F (eds) Principle and practice of constraint programming – CP98. Lecture notes in computer science, vol 1520. Springer, Berlin, pp 417–431
100. Shaw P, De Backer B, Furnon V (2002) Improved local search for CP toolkits. Ann Oper Res 115:31–50
101. Shi L, Ólafsson S (2000) Nested partitions method for global optimization. Oper Res 48(3):390–407
102. Shi L, Ólafsson S, Chen Q (2001) An optimization framework for product design. Manage Sci 47(12):1681–1692
103. Stützle T (2006) Iterated local search for the quadratic assignment problem. Eur J Oper Res 174(3):1519–1539
104. Tamura H, Hirahara A, Hatono I, Umano M (1994) An approximate solution method for combinatorial optimisation. Trans Soc Instrum Control Eng 130:329–336
105. Tse S-M, Liang Y, Leung K-S, Lee K-H, Mok TS-K (2007) A memetic algorithm for multiple-drug cancer chemotherapy schedule optimization. IEEE Trans Syst Man Cybern B 37(1):84–91
106. Van Hentenryck P, Wolsey LA (eds) (2007) Proceedings of CPAIOR 2007 – 4th international conference on the integration of AI and OR techniques in constraint programming for combinatorial optimization problems. Lecture notes in computer science, vol 4510. Springer, Berlin
107. Vasquez M, Hao J-K (2001) A hybrid approach for the 0–1 multidimensional knapsack problem. In: Nebel B (ed) Proceedings of the 17th international joint conference on artificial intelligence, IJCAI 2001, Seattle, Washington. Morgan Kaufman, pp 328–333
108. Vasquez M, Vimont Y (2005) Improved results on the 0–1 multidimensional knapsack problem. Eur J Oper Res 165(1):70–81
109. Walshaw C (2002) A multilevel approach to the travelling salesman problem. Oper Res 50(5):862–877
110. Walshaw C (2004) Multilevel refinement for combinatorial optimization problems. Ann Oper Res 131:325–372

111. Walshaw C (2008) Multilevel refinement for combinatorial optimisation: boosting metaheuristic performance. In: Blum et al. (eds) Hybrid metaheuristics – an emerging approach to optimization. Studies in computational intelligence, vol 114. Springer, Berlin, pp 85–116, pages 261–289
112. Walshaw C, Cross M (2000) Mesh partitioning: a multilevel balancing and refinement algorithm. SIAM J Sci Comput 22(1):63–80
113. Watson J-P, Beck JC (2008) A hybrid constraint programming/local search approach to the job-shop scheduling problem. In: Integration of AI and OR techniques in constraint programming for combinatorial optimization problems. Lecture notes in computer science, vol 5015. Springer, Berlin, pp 263–277
114. Wilbaut C, Hanafi S, Fréville A, Balev S (2009) Tabu search: global intensification using dynamic programming. Control Cyber 35(3):579–598

Learning in Search

Philippe Refalo

Abstract This chapter focuses on the recent improvements in solution search that are based on learning. We will describe some learning methods applied in areas such as mixed-integer programming, constraint programming, and those used for satisfaction problems. Instead of being exhaustive, we will concentrate on some of the most exciting advances. In particular, we will focus on pseudo-cost strategies used in general-purpose mixed-integer programming solvers, on the strategy learning used for automatic search in constraint programming, and on no-good generation in SAT solvers. Several examples are given to illustrate the effectiveness of learning in these areas. Some practical results are also given using the integration of different learning techniques.

1 Introduction

Search has always been a fundamental aspect of problem solvers. Recently, in areas such as mixed-integer programming (MIP), constraint programming (CP), or satisfiability (SAT), search techniques have experienced a revival, for various reasons.

MIP solvers have put emphasis on the linear relaxation since the beginning through the development and strengthening of cutting planes [27, 28]. Although generic strategies were developed since the 1970s for solving integer programs (see Sect. 2.1 of this chapter), tightening linear relaxations was not sufficient. During the last years, the integration of local search techniques in integer programming has reinforced the efficiency of MIP solvers in finding good solutions [13, 14].

In clause SAT solvers, the recent tremendous progress is due to better propagation techniques [26] but mostly to new schemes for learning clauses during search that in turn improve search strategies [16, 25].

In CP, the global constraint euphoria of the nineties has given way to an emphasis on search techniques. This cultural evolution took some time to occur. The reason

P. Refalo (✉)
IBM, Les Taissounieres, 1681, route des Dolines, 06560 Sophia Antipolis, France
e-mail: philippe.refalo@fr.ibm.com

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_10,
© Springer Science+Business Media, LLC 2011

is that one of the biggest source of pride of the CP community about their solvers was the possibility to program complex search for solutions. This has turned to the situation where, for many problems, there was no hope to find any solution without programming search.

The need to give a larger audience access to CP solvers has lead to the development of *autonomous searches* for CP. CP solvers have moved from toolkits where users needed to program their own strategies to almost back-box solvers including efficient search strategies mixing backtrack search, restart, and local search.

The overall trend is to have an efficient search that requires as little as possible from the user. Because the user knows his model but usually not the way it is going to be solved. As a consequence, the solution search needs to extract knowledge from the model and also to learn via self-observation. Learning is thus obviously closely related to search autonomy. Learning plays an important role not only to learn the right solver parameters but to acquire knowledge about the structure of the problem. This knowledge acquisition that we will call *structural learning* permits one to infer high-level information on problem solutions in order to improve problem solving. In practice, this is exploited by inferring new constraints and by being able to distinguish important variables and values in order to make appropriate decisions.

In this chapter, we will try to mention several approaches to learning in search but instead of being exhaustive, we will concentrate on some of the most exciting advances.

We will first focus on recent advances in strategy learning. Historically, integer programming solvers were the first to face the problematic of designing general-purpose solvers able to solve efficiently very different problems in terms of size and structure. We will review the existing strategies' improvements by learning, and we will present recent CP strategies that have been inspired by integer programming strategies and the variations that have been developed subsequently.

After a section on search restarts that plays a fundamental role in recent CP and SAT solvers, we will overview clause learning in SAT solvers. Intelligent clause learning is a key technique in the efficient solution of satisfiability problems. Although clause learning has been used for a long time in CP, the exploitation of clauses is different in SAT. We will present some recent works that integrate a SAT-like clause management in CP. Finally, we will illustrate the combination of clause generation, restart, and strategy learning.

2 Strategy Learning

The search strategies we consider under analysis are those used for a complete solution of a problem, whether it is an integer program, a CP, or an SAT problem (a set clauses to satisfy). By *complete*, we mean that the search is able to find a solution to the problem, or to prove that there is none, or to prove optimality in the case of an optimization problem.

A common approach for searching solutions is *backtrack search*. At each step (or node of the search tree), backtrack search chooses a nonassigned variable x of the problem and a set of values E or a value (in that case $E = \{v\}$) and states the choice point

$$x \in E \text{ or } x \notin E$$

The constraint $x \in E$ is added to the current problem (the node problem) and a constraint processing is triggered. That processing can be constraint propagation which infers domain reductions, linear relaxation solving which computes a relaxed optimal solution, and a lower bound to the problem or unit clause propagation in SAT solvers that fixes Boolean variables to *true* or *false*. This constraint processing checks that no constraint is violated. When there are violated constraints, the search procedure backtracks to the last choice point by undoing constraint additions and tries the first unexplored branch encountered. When every variable is instantiated, and no constraint is violated, a solution is found.

The performance of backtrack search (or the number of nodes it traverses) varies dramatically depending on the strategy used to choose a variable and values. Moreover, the strategy may need to be different depending on the nature of the problem. In an optimization problem, finding a solution is often easy and the emphasis is on finding good quality solutions. This may involve different strategies than for a problem where we just want a solution. Similarly, a strategy to find good solutions may be different from the ones that prove optimality.

However, there are a few general principles for reducing the search effort that can be widely applied and that are worth mentioning here.

Since all variables have to be instantiated, first choosing the variable that maximally constrains the rest of the search space reduces the search effort in general. This principle called *first-fail* in the CP community is popular and widely applied [19]. It is often implemented by choosing the variable having the smallest domain first or the one that participates in the largest number of constraints or a combination of both. In integer programming, the chosen variable is the one that constrains the objective function the most. In SAT solvers, this strategy uses variations around choosing the variable appearing in the largest number of clauses.

As to the choice of a value, a solution can be reached more quickly if one chooses values that maximize the number of possibilities for future assignments. As we will see in the next section, for finding good quality solutions, integer programming solvers apply the principle by choosing the values that constrain the least the objective function. But choosing the right value is often difficult and is less likely to obey general rules. A knowledge of the structure of the problem (or of the structure of the solutions) is often helpful.

Finally, a general principle is to make good choices at the top of the search tree. Choices made at the top of the search tree have a huge impact on its size and a bad choice can have disastrous effects. This is especially true for depth-first search backtrack algorithms where the whole sub-tree below a choice needs to be explored before reconsidering that choice. This leads to a high variability in terms of computation time [17] that can be broken by search restart (see Sect. 3).

It is interesting to see that the integer programming community has applied these principles since the early seventies. In integer programming, the emphasis is on finding optimal solutions. As described in the next section, an estimation of the objective function improvement is associated to a variable. According to these principles, the variable chosen first is the one involving the largest improvement; then the branch chosen first is the one that involves the least improvement.

2.1 Strategy Learning in Integer Programming

In integer programming solvers, designing a search strategy is considered a complex task. It is true that the underlying concepts of integer programming that need to be understood (i.e., relaxed optimal solution, dual values, reduced costs) are not very intuitive. As a consequence, a class of techniques for efficient general-purpose strategies in integer programming have emerged. The emphasis of integer programming being optimization, these techniques are based on estimating the importance of a variable with respect to the variation of the objective function value. This criterion is called a *pseudo-cost* and is learned by observing the variations of the objective function value through the search space.

Pseudo-costs are introduced briefly in [3] and fully described in [15]. They are widely used in modern integer programming solvers such as CPLEX [1].

In integer programming problems, constraints are linear, and the domain of a variable is an interval of real or integer values. In addition to that, there is a linear cost function to be minimized (the maximization case is ignored, as it is similar). Integer programming solvers maintain a relaxed optimal solution at each node. This solution is given by a linear solver (such as the simplex method) applied to the node problem where integrality conditions on integer variables have been removed. This relaxed solution on a variable x is noted x^*. The value of the objective function in this solution is z^*. It is a lower bound on the optimal value of the problem. The value z^* increases as variables are instantiated during search. It is the amount of that increase that is used to select variables.

Consider a variable x whose noninteger value is $x^* = \lfloor x^* \rfloor + f$ where $0 < f < 1$. One can force x to an integer value by creating the choice point

$$x \leq \lfloor x^* \rfloor \text{ or } x \geq \lceil x^* \rceil$$

Let z^* be the objective function value before the choice point. Let Δ_{down} be the increase of the objective value when adding the constraint $x \leq \lfloor x^* \rfloor$ and Δ_{up} be the increase of the objective value when adding the constraint $x \geq \lceil x^* \rceil$.

The values Δ_{down} and Δ_{up} can be computed for each variable having a noninteger value by solving two linear programs. From the first principle above, the variable to be chosen is the one having a maximum impact on the objective. In practice, it is the one that maximizes the weighted sum

$$v(x) = \alpha \min(\Delta_{\text{down}}, \Delta_{\text{up}}) + \beta \max(\Delta_{\text{down}}, \Delta_{\text{up}})$$

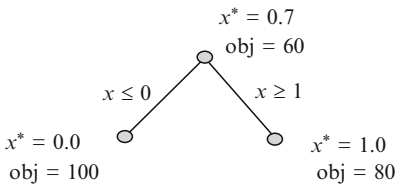


Fig. 1 Example of a search node in integer programming

Usually, more importance is given to the maximum of Δ_{down} and Δ_{up}. For $\alpha = 1$, choosing β greater than 3 gives good results (see [23]).

From the second principle above, the first branch to explore is the one that creates the smallest improvement in the objective function value with the hope of getting solutions with a low objective value earlier.

The pseudo-costs of a variable x measure the increase of the objective function value per unit of change of x when adding the constraint $x \leq \lfloor x^* \rfloor$ or $x \geq \lceil x^* \rceil$. The *down pseudo cost* $PC_{\text{down}}(x) = \Delta_{\text{down}}/f$ corresponds to the decrease of x^* and the *up pseudo-cost* $PC_{\text{up}}(x) = \Delta_{\text{up}}/(1-f)$ corresponds to the increase of x^*.

Example 1. Consider the excerpt of a search tree in Fig. 1. The search branches on variable x by stating first $x \leq 0$ and then $x \geq 1$. Each branching involves a variation of the objective function value (obj). For the branch $x \geq 1$ the objective values moves up from 60 to 80 while the value x^* of the variable moves from 0.7 to 1. Thus, the variations are $\Delta_{\text{down}} = 40$ and $\Delta_{\text{up}} = 20$ and the pseudo-costs are $PC_{\text{down}}(x) = 40/0.7 = 57.1$ and $PC_{\text{up}}(x) = 20/0.3 = 66.6$.

A fundamental observation about pseudo-costs is that experiments reveal that the pseudo-costs of a variable tend to be close from one node to another (see [15] and [23]). As a consequence, we can avoid solving two linear programs (one for Δ_{down} and one for Δ_{up}) for each noninteger variable at each node, which is computationally very expensive. We can estimate that the up and down pseudo-costs of a variable x at a node are likely to be the average of the ones observed so far when choices are made on x. This averaged pseudo-cost (PC^e) is used to compute an estimation of $v(x)$ from an estimation of the objective function variation:

$$\Delta^e_{\text{down}} = f * PC^e_{\text{down}}(x) \qquad \text{if } x \leq \lfloor x^* \rfloor \text{ is added}$$
$$\Delta^e_{\text{up}} = (1-f) * PC^e_{\text{up}}(x) \quad \text{if } x \geq \lceil x^* \rceil \text{ is added}$$

At the beginning of search, pseudo-costs are unknown. Having pseudo-costs at this time is extremely important since choices made high in the tree are crucial. Computing explicit pseudo-costs by bounding up and down variables that have a noninteger value can degrade performance significantly [3]. A trade-off consists of performing a limited number of dual simplex iterations [23]. The approximated pseudo-cost computed in this way is usually replaced by the first one observed.

2.2 Impacts-Based Strategies in Constraint Programming

The most popular strategies in CP are based on first selecting variables having the minimum domain size [19]. Ties can be broken with the dynamic degree of variables [7]. A variation consists in using the ratio between the size of the domain and the degree of the variable [5] which is often considered as the best strategy for solving binary constraint satisfaction problems(CSP). Another one is to look at the neighboring structure of the variable [4,33]. All these heuristics are dynamic but do not adapt themselves to the problem by learning information. In this section, as well as in the next one, we described heuristics that can be seen as a generalization of minimum domain size and degree-based choices to incorporate some learning.

Impact-ordering heuristics were first described in [30] for finding solutions to constraint programs. The idea is to provide efficient heuristics for solving constraint programs inspired by pseudo-costs strategies. Impacts are based on learning. It exploits parts of the search tree that are apparently not useful – because they do not lead to a solution – in order to learn information about the importance of variables and values. Following this idea, some variations have been developed in [9] that are presented in the next section.

As with pseudo-costs, the basic idea of impacts is rather intuitive. In CP, when a value is assigned to a variable, constraint propagation reduces the domains of other variables. We consider that the number of all possible combinations of values for the variables (the Cartesian product) is an estimation of the search size. An impact measures the importance of an assignment $x = a$ for the search space reduction. Impacts are obtained from the domain reduction involved by assignments made during search.

Therefore, an estimation of the size of the search tree is the product of every variable domain size:

$$P = \left|D_{x_1}\right| \times \ldots \times \left|D_{x_n}\right|$$

If we look at this product before (P_{before}) and after (P_{after}) an assignment $x_i = a$, we have an estimation of the importance of this assignment for reducing the search space. This reduction rate is called the *impact* of the assignment

$$I(x_i = a) = 1 - \frac{P_{\text{after}}}{P_{\text{before}}}$$

The higher the impact, the greater the search space reduction. From this definition, an assignment that fails has an impact of 1.

The impact of assignments can be computed for every value of all noninstantiated variables, but this can create a huge overhead. From the experiments we have made, impacts, like pseudo-costs, do not vary much from node to node. The impact value distribution of a given assignment almost always presents a sharp and unique peak centered on the average value. For instance, the distribution of impact values thorough the search tree of an assignment in a multidimensional knapsack problem is shown in Fig. 2. An important consequence is that the impact of an assignment at

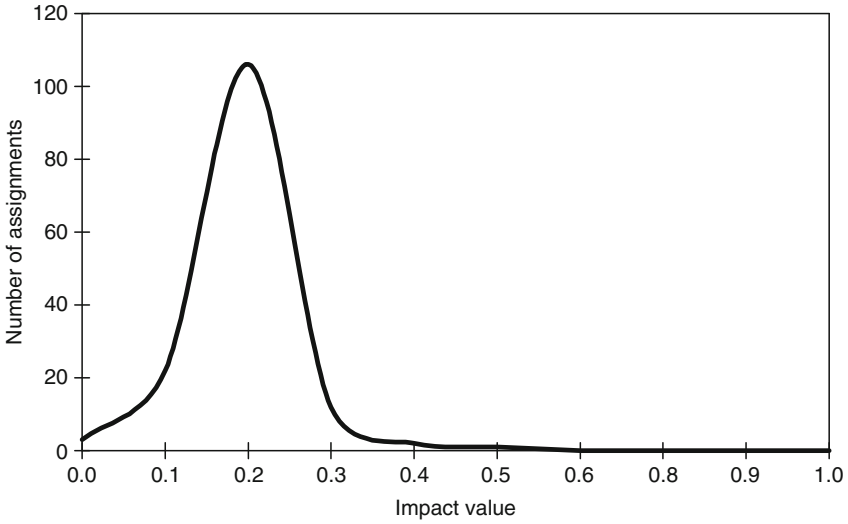


Fig. 2 Impact distribution of an assignment $x = 1$ in a multi-dimensional knapsack problem

a given node can be the average of the observed impacts of this assignment up to this point. If K is the index set of impacts observed so far for assignment $x_i = a$, $\bar{I}$ is the averaged impact:

$$\bar{I}(x_i = a) = \frac{\sum\limits_{k \in K} I^k(x_i = a)}{|K|}$$

An important consequence is that impacts do not need to be computed explicitly at each node, but are available almost for free.

The impact of a variable x_i can be the average of impacts $\bar{I}(x_i = a)$ for $a \in D_{x_i}$. A more accurate measure would use only the values remaining in its domain. Thus, if the current domain of x_i at a node is D'_{x_i}, we have

$$\tilde{I}(x_i) = \frac{\sum\limits_{a \in D'_{x_i}} \bar{I}(x_i = a)}{\left|D'_{x_i}\right|}$$

This approach is not accurate enough. The goal is to choose a variable having the largest impact when assigning to it one of the values remaining in its domain. Since it is assumed that each value will be tried (the hypothesis is that the problem is infeasible as mentioned above), we need to consider the search reduction if *every* value of the domain is tried. Let P be the product of the domain sizes at a node and

consider a variable x_i. The estimation of the size of the search space when trying $x_i = a$ with $a \in D'_{x_i}$ is

$$P \times (1 - \bar{I}(x_i = a))$$

This is an estimation of the size of the search tree for $x_i = a_j$. If we were to try every value, an estimation of the search tree would be the *sum* of the estimation for each value remaining in the domain:

$$\sum_{a \in D'_{x_i}} P \times (1 - \bar{I}(x_i = a))$$

The value P is a constant at a node, and it is not relevant to compare the impact of several variables at the same node. Finally, the impact of a variable that depends on its current domain is defined as

$$\mathscr{I}(x_i) = 1 - \sum_{a \in D'_{x_i}} 1 - \bar{I}(x_i = a)$$

Experiments we have made comparing the use of the average impact $\tilde{I}(x)$ and the use of the sum of impacts on values $\mathscr{I}(x)$ show that using $\mathscr{I}(x)$ is much more efficient over all the problems we have tested.

Example 2. A simple illustration of impact averaging is made on an instance of a bin-packing problem. Let us consider instance u120_00 of Falkenauers bin-packing problems, from OR-Library[1] (120 objects needs to be packed and the bins capacity is 150). Here is a straightforward CP model for this problem.

$x_o \in \{0, \ldots, 120\}$ for $o = 1$ to 120 // object position variable
$l_b \in \{0, \ldots, 150\}$ for $b = 1$ to 120 // load of a bin
minimize $(l_1 > 0) + (l_2 > 0) + \ldots + (l_{120} > 0)$
s.t. $w_1 \times (x_1 = b) + w_2 \times (x_2 = b) + \cdots + (x_{120} = b) = lb$ for $b = 1$ to 120

We ran several random searches on this problem (by choosing variables and values randomly) for a few thousand nodes to give an equal chance for all variables and value to be instantiated in different parts of the tree.

Figure 3 shows the impact value obtained for the weight of each position variable. Weights are sorted increasingly. It is interesting to see the clear correlation between the object size and the impact of the corresponding variable. The increasing impacts variable ordering matches the increasing weight ordering of the objects. Starting with variables having the largest impact amounts to starting with heaviest objects first. Thus, impact measure rediscovers an efficient strategy for placing objects in bin-packing problems that starts with the heaviest or largest objects first.

[1] instances are described at http://people.brunel.ac.uk/mastjjb/jeb/orlib/binpackinfo.html

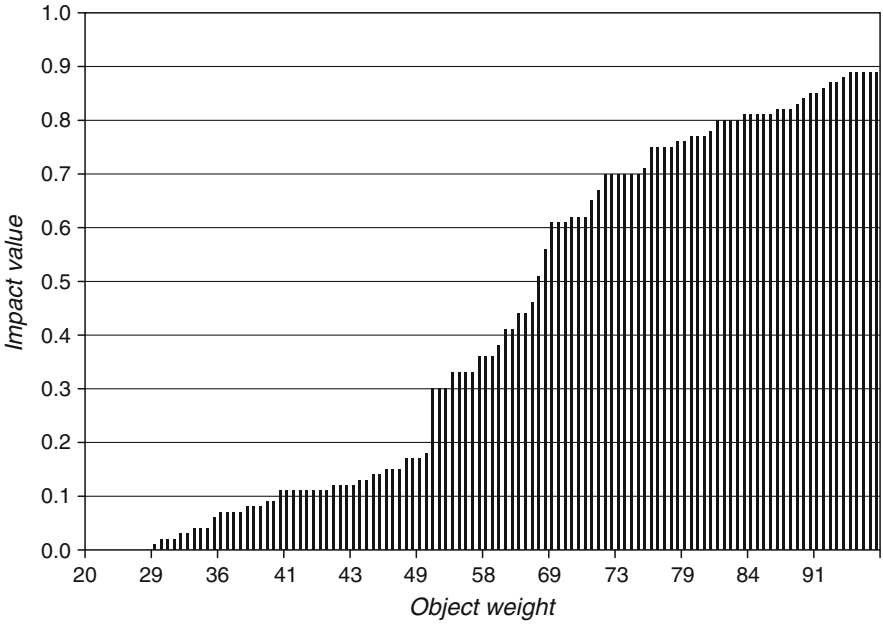


Fig. 3 Size of objects and impact of the corresponding position variable in a bin-packing problem

As was said previously, some effort must be made before starting search to compute impacts. This helps differentiate variables at the root node and to make better choices at the beginning of the search when this is crucial. One approach is to try every value for every variable. However, on problems where domains are large, this can be more costly than solving the problem itself. In [30], an efficient method is described that recursively splits the domain of a variable until a given depth. This has the advantage of being independent from the size of the variable domain and still to provide information precise enough to avoid starting search in the wrong way.

Figure 4 shows the effect of impacts when solving decision multi-dimensional knapsack problems. These problems are modeled by a set of linear constraints over binary variables. They are usually hard for CP solvers. Five problems come from the smaller set of the operations research library[2] where the cost function is constrained to take its optimal value (problems `mknap1-*`). Four other problems are isolated sub-problems of a real-life configuration problem (problems `mc*`). These problems have up to 11 constraints and 50 variables. The time limit is 1,500 s and we used depth-first search together with constraint propagation.

Classical CP search strategies do not make sense here because the domain size of uninstantiated variables is always two and the degree tends to be the same for all variables. We instead run a random strategy that serves as a reference. We then

[2] Problems are available at http://mscmga.ms.ic.ac.uk/jeb/orlib/mdmkpinfo.html

	Random		Impact w/o init.		Impact + init.	
Problems	Time	Ch.pts.	Time	Ch.pts.	Time	Ch.pts.
mknap1-0	0.02	2	0.02	2	0.03	2
mknap1-2	0.02	10	0.05	15	0.03	26
mknap1-3	0.03	408	0.06	304	0.05	186
mknap1-4	0.55	11485	0.24	3230	0.05	434
mknap1-5	48.91	1031516	1.7	29418	0.22	4247
mknap1-6	>1500		>1500		50.46	902319
mc3923	14.67	491445	197.65	5862508	0.38	11768
mc3800	2.28	75270	248.81	7348618	0.06	1769
mc3888	53.40	1784812	33.93	1007735	1.36	44682
mc3914	26.91	899114	305.56	9084462	0.44	14390

Fig. 4 Impact strategies on multiknapsack decision problems

compare the use of impact strategies with and without initializing impacts. The strategy is randomized in the sense that ties are broken randomly. Impacts are initialized here by instantiating every variable to 0 and then to 1 and computing the corresponding impact. One can observe that initialization of impacts dramatically improves performance by allowing good choice at the beginning of the search. Making wrong choices high in the tree can have disastrous effects with depth-first search.

2.3 Conflict-Directed Strategies in Constraint Programming

A classical strategy in CP makes use of the dynamic degree of a variable. The degree of a variable is the number of variables it is linked to via constraints. The dynamic degree is simply the number of *uninstantiated* variables linked to the variable. This measure is widely used for solving CSP and constraint programs and often for breaking ties when selecting variables with the minimum domain size strategy.

In [6], a new degree measure was introduced. It can be seen as a generalization of the dynamic degree. It is based on the learning of constraints that fail the most. Each time a dead-end is reached (due to a domain wipe-out), the method increases a counter fails(c) of each constraint c responsible for the wipe-out.

Using these counters, they define a new estimation (called weighted-degree) for a variable that counts the number of failures of constraints connected to the variable:

$$E_{\text{wd}}(x) = \sum_{c \in S, x \in \text{vars}(c)} \text{fails}(c)$$

Here, vars(c) is the set of variables of the constraint c. The estimation used in [6] is indeed more dynamic as it sums up the fails(c) counters only for constraints having at least a free variable. When solving binary CSPs, one can observe that the dynamic degree of a variable amounts to the estimator $E_{\text{wd}}(x)$ where fails(c) = 1.

The strategy is then quite intuitive. If one wants to apply the first-fail principle and choose the most constrained variable at a node, one can start with variables having the largest $E_{\text{wd}}(x)$ value. In [6], this estimator is combined with the domain size and the strategy chooses the variable having the smallest ratio

$$domain\_size(x) \ / \ E_{\text{wd}}(x)$$

They compared the weighted-degree against the classical degree measure by replacing it in various standard strategies such as the one minimizing the ratio domain size over the degree. Their most interesting results were obtained on frequency allocation problems where they could achieve orders of magnitude improvements over standard strategies. Here also, learning a low-level characteristic (the number of failures) of a solver behavior on a given problem can lead to significant improvements. The intuition behind the estimator is easy to understand. However, a formal expression of it in terms of model formulation and models data is difficult because the estimator operates at the frontier between the constraint formulation and the solver.

Starting from the work of [30], Cambazard and Jussien proposed in [9] another measure for the impact of an assignment based on explanations provided by the CP solver. An explanation is the reason for the removal of a value a in the domain of the variable x, it is a set E containing the set of instantiations D made by the search procedure and the set of constraints C of the model involved in the inference such that

$$D \cup C \models \{x \neq a\}$$

In practice, the solver keeps a trace of the inference chain each time an instantiation occurs to compute sets D and C for every value removed. As a consequence for every explanation, performing constraint propagation on the set $D \cup C$ leads to the right-hand side (the removal of the value a from the domain of x). At any stage of the search, all explanations are stored in the set E. The set of explanations of a removal $x \neq a$ computed through the whole search is noted $E(x \neq a)$. This is where the learning occurs: exploiting the set of explanations can lead to a different – and more accurate in some cases – view on the problems in terms of variable and value selection strategy.

In [9], statistics on explanations are used to compute new estimators for values. The impact of an assignment $y = b$ is the number of times it appears in all explanations or in explanations for removal of values that are still in the domains (the latter is theoretically more accurate). The size of explanations is also taken into account in order to give more importance to occurrences in shorter explanations. Their evaluation of an assignment is thus

$$C(y = b) = \sum_{e \in E \text{ such that } (y=b) \in e} 1/|e|$$

The estimator for a variable is computed by summing up the value evaluations over the current domain of the variable, exactly as for regular impacts (see Formula 2.2). They also propose taking into account the age $A(d, e)$ of an

instantiation d in an explanation. In the formula above this amounts to sum up $\frac{1}{|e| \times A(y=b,e)}$ instead of $1/|e|$ only.

Following the work of [6] on conflict-based degree, they also introduce an evaluation of constraints where the evaluation sums up the $1|e|$ value for each explanation e where the constraint appears. The new weighted degree of a variable x is obtained by summing up the evaluation $C(c)$ for each constraint c where x appears.

Experiments were conducted in [9]. The explanation-based impacts cannot compete with regular impact on multidimensional knapsack problems (some of those presented in Fig. 4). However, they could achieve better results on the structured random problems they generated. They also show that they are able to produce results comparable to Boussemart et al. [6] on frequency allocation problems by using a strategy that minimizes the ratio domain size over the explanation based degree described above.

3 Search Restarts for Exploiting Learning

Restarting search, that is stopping the search process and restarting it from scratch from time to time depending of a given policy, is an effective technique when using randomized strategies [17]. The basic idea is to give equal chances to all parts of the search space to be explored at the beginning of the search. As an example, if the assignment $x = 1$ is the first made in backtrack search, the other branch $x \neq 1$ will be explored only when the whole sub-tree under $x = 1$ is fully traversed. This sub-tree can be huge and restarting search from time to time gives the choice $x \neq 1$ some probability to be explored earlier. Search restarts have proved to be practically very effective and it is nowadays a common component of Boolean satisfaction (SAT) solvers and CP solvers (see [2, 18, 26] and [30]).

When learning the importance of variables and value, restarting search plays the fundamental role of permitting the solver to use the information learned during the previous run to start a new, and hopefully smaller, search tree by making more appropriate choices at the beginning when this is crucial. Taking the example of impact measures described earlier, as the search progresses, we get more and more accurate impacts. We learn in a way what the important assignments and variables are. Consequently, we may discover that variables assigned high in the search tree are not as important as they looked at the beginning. Therefore, it can be beneficial to restart search from time to time in order to use the most recent impact information with the hope that the new search tree will be smaller.

An important aspect is to decide about restart policy. Restart policies can be grouped in three classes:

1. A fixed restart criteria: once a limit is reached (number of branching failures, time, or any other measure) the search is restarted.

2. A geometrically increasing limit: this makes the restart procedure complete, because we will surely reach a limit sufficiently large to be able to find a solution or to prove that none exists.
3. An optimal approach was proposed by Luby et al. in [24]. It is based on the following sequence of run lengths: 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, ... This policy is optimal in the sense that the expected running time is only a logarithmic factor slower than the true optimal policy when using randomized algorithms such as Las Vegas algorithms.

Search restarts can lead to huge improvements in computation time. It has been used extensively in SAT and CP solvers. An example of improvement obtained by restarts is given in Sect. 7.

4 Clause Learning

Clause learning aim is to improve the performance of backtrack search algorithms by generating explanations for failure (backtrack) points, and then adding the explanations as new constraints to the original problem (initiated in [34] and further developed for instance in [11]) . These explanations are mostly clauses, named *conflict-clauses* or *no-goods* [32]. The goal of learning clauses from a conflict is to improve the solver's efficiency in finding a solution or proving that the problem is infeasible by avoiding to recreate the same failure conditions in another part of the search. Judicious clause learning is one of the main reasons for the tremendous improvements in SAT problem solving over the last decade.

4.1 Clause Learning for Satisfiability Problems

Solving SAT problems in a complete way is based on variable instantiation and backtracking when a failure is encountered [10]. At each node of the search tree, similarly to CP some inferences are performed called *unit propagation*. Assume we have a clause $x \vee \neg y \vee z$ and that the search has set x to *false* and y to *true*. Unit propagation consists in deducing that z must be set to *true* to satisfy the clause and propagates this fixing by seeking another clause where this new assignment permits to fix a variable. This process is performed until a fix-point is reached (no more fixing can be deduced).

As the search progresses, the reasons for all these assignments are recorded in an oriented graph called the *implication graph* [25]. Such a graph is represented in Fig. 5. On the left part are the assignments made by the search procedure (with a grey background) and on the right part are the fixings made by propagation. For instance in this figure, a has been instantiated to *true* by the search procedure and this has fixed p to *true* (due to a clause $\neg a \vee p$ in the model). Then, b was fixed to *false* and together with a fixed to *true* propagation has fixed q to *false* (due to a

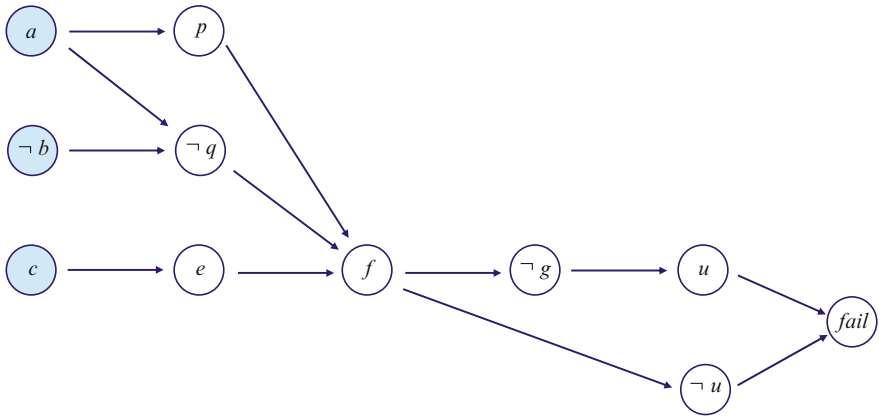


Fig. 5 Implication graph leading to a failure

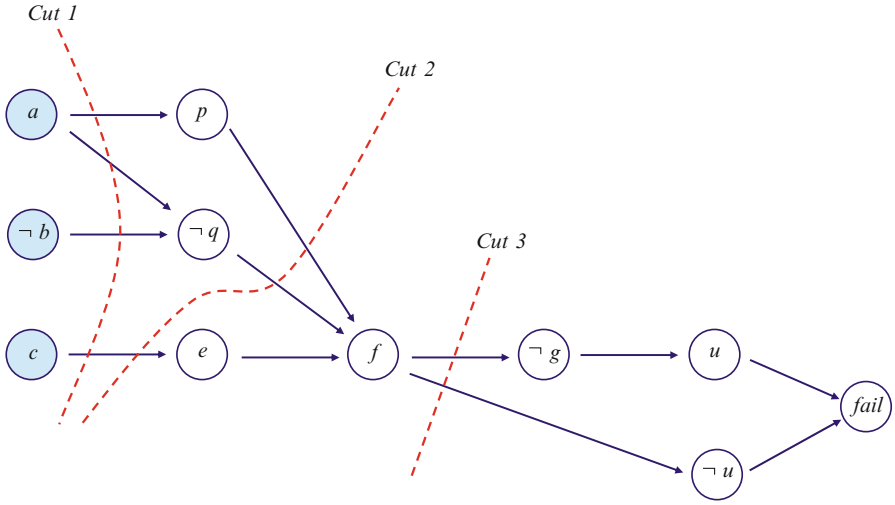


Fig. 6 Edge-cuts of the implication graph

clause $\neg a \vee b \vee \neg q$ in the model). Observe that the graph in Fig. 5 shows a failure that is obtained when fixing c to true (we were able to fix u to *true* and also to *false*). This graph is thus called a *conflict graph* because it represents an explanation for the failure. There might be other failures involved at this level but for efficiency reasons, SAT solvers usually analyze only the first one encountered.

Once this graph is generated, how to generate additional clauses? This is quite simple: any cut that separates the vertices of the instantiation level and the *fail* vertex is called an *edge-cut*, and it corresponds to a clause that does not exclude any solution of the problem. The clause contains the negation of the literal that is on the left side of every cut edges. For instance in Fig. 6.

- The edge-cut Cut 1 separates the instantiations from the rest of the graph and leads to the clause $\neg a \vee b \vee \neg c$ that forbids the same configuration of instantiations. This is unlikely to happen if we backtrack on the last choice ($c = true$) and such cuts are generally avoided.
- The edge-cut Cut 2 leads to the clause $\neg p \vee q \vee \neg c$ which is more likely to be encountered in another part of the search tree because it does not depend directly on the instatiation made by search; we could have other reasons than $a = true$ and $b = false$ to obtain $p = true$ and $q = false$.
- Finally, the edge-cut Cut3 leads to the clause $\neg f$ and such a unary clause permits to fix immediately f to false because fixing f to true always leads to a failure.

Now, the main decision that a search procedure needs to take is that: which conflict clause to add? A naive way of benefiting from conflict-clauses is to add a clause for every edge-cut. This approach can generate a huge number of clauses. However, it was used by earliest satisfiability solvers. Many improvements were made since the satisfiability solver GRASP [25] that was the first to select a clause to accelerate the search. Recent solvers use heuristics for choosing a few (in practice, often just one) interesting conflict clauses to add to the original problem. These clauses are then managed as a database where clauses are added but also removed when their activity level or the date of the latest activity is too low.

The extension of a clause model with additional clauses maintains model homogeneity and permits one to refine the search strategies that are mostly based on statistics about literal occurrences in clauses. This is somewhat related to cutting plane generation in integer programming solvers [35] where linear constraint additions refine the linear relaxations and improve the search strategy.

While clause learning is now an old technique in CP, clauses were handled differently. The huge performance improvement that it brings for solving SAT problems has motivated some recent research about handling clauses in a similar way as they are in SAT solvers.

4.2 Clause Learning in Constraint Programming

Learning from failures is not new in the constraint satisfaction problem field. *Nogood recording* [12] as well as conflict-directed *backjumping* [11, 29] has been used for a while for solving CSP. A *no-good* is a set of assignments that cannot be extended to a solution. When a failure is encountered during a depth-first search, the set of assignments that led to the failure is obviously a no-good. By combining different no-goods responsible for the failure of every choice in a choice point, one can obtain a no-good whose deepest assignment in the search tree is higher than the decisions that created the failures. Backtracking higher to this deepest assignment is called *backjumping*.

Clauses generated by SAT solvers and no-goods in constraint satisfaction are exploited differently but they are closely related. Both are based on the resolution principle. As a no-good

$$\neg(x_1 = v_1 \wedge x_2 = v_2 \wedge \ldots \wedge x_n = v_n)$$

is equivalent to the clause

$$(x_1 \neq v_1 \vee x_2 \neq v_2 \vee \ldots \vee x_n \neq v_n)$$

It is tempting to manage these clauses, as in SAT solvers, not only for backjumping but also for generating new constraints from failures that will be propagated to further reduce the variables domains. In [21], no-goods are generated from the reasons (or explanation) of the removal of a value in a domain. These no-goods are combined to create new ones that augment the solved model. This obviously increases domain reduction.

The fundamental difference with SAT solvers lies in the uniformity of the constraint model: in SAT solvers, clause generation augments the basic model and has a direct impact on further clause generation, on logical inferences and on the search strategy which is mostly based on statistics about literal occurrences. Clause generation in SAT solvers also helps recover hidden model structures. In CP, the benefit can be less significant. CP are often structured, and the strong inferences made by global constraints make clause generation no longer an essential feature but rather a way among others to make domain reductions. As for global constraints, the impact on performance can be tremendous.

A restricted but effective use of no-good is presented in [22] where search restarts are used. After each run, and before the next restart, a set of clauses forbidding the already explored search space is added to the model. This avoids restarts scanning the same parts of the search space several times. This restricted use has the advantage of being efficiently implemented and does not generate a huge amount of clauses since restart cutoffs are often large and the number of clauses depends on the number of variables and on the number of restarts. This idea has been exported to CP in [31] and in CSP in [8]. Clauses generated after each run are handled in a global constraint exploiting the well-known watched literals propagation mechanism [26].

Practical results on structured problems demonstrate the effectiveness of the method. It benefits from the advantages of restart with a negligible overhead compared to depth-first search. In [8], an order of magnitude improvement is reported on solving hard radio-link frequency decision problems. In [31], the combination of impacts, restarts, and clause generation involves significant improvements. It is at the basis of the automatic search procedure available in the CP Optimizer product [20].

We give here some details about the combination and experimental results as it illustrates the combination of several types of learning during search. The idea is to use impact based strategies and search restarts to exploit the impacts learned. In addition, conflict clauses are added to avoid re-exploring the same parts of the search and to restart more often, without increasing the cutoff.

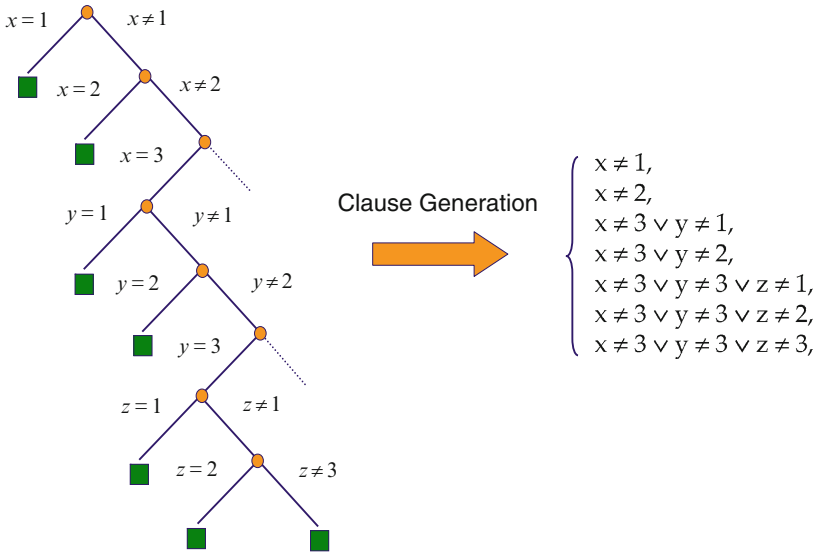


Fig. 7 Clause generation from search restarts

Figure 7 shows an example explaining the way these clauses are generated. On the left part of the figure, a search tree exploration fails because a fail limit has been reached. From this partial tree, clauses are generated to avoid re-exploring the same nodes. For instance, to avoid re-exploring the sub tree below $y = 1$, the clause $x \neq 3 \vee y \neq 1$ is added. These clauses (which are no-goods in this case since they are the negation of an assignment) are grouped together in a global constraint and propagated using the watched-literal technique. Clauses with a low activity are removed from time to time to avoid memory explosion.

This approach has been tested on decision multi-knapsack problems from OR-Library. These are optimization multi-knapsack problems whose objective function is replaced by an equation forcing the function to be equal to the optimal value of the problem. In Fig. 8, the comparison of four methods is presented:

1. Impact estimators described in Sect. 2.2 with a depth-first search procedure
2. Impacts with increasing restarts (the cutoff is a fail limit initialized at 100 and it increases geometrically)
3. Impacts with increasing restarts and the no-good generation described above
4. Impact with constant restarts (cutoff = 1,000) and no-good generation

Before search, impacts are initialized by branching once on each value (0 and 1) for every variable. The Figure gives the computation time in seconds. The last line show the number of instances unsolved. The first result is that restart helps a lot. Conflict addition helps also but when it allows search to restart every 1,000 fails, we can solve more instances and reduce significantly the computation time for instances that could be solved otherwise.

Instances	Impact + depth-first search	Impacts + incr. restarts	Impact + incr. restarts + conflicts	Impact + constant restarts + conflicts
SENTO1	0.17	5.00	0.39	2.30
SENTO2	1.38	1.94	0.22	0.61
WEING1	3.97	22.98	1.78	19.28
WEING8	20.11	8.92	4.17	0.42
WEISH01	156.78	1000.00	635.02	5.39
WEISH015	2.22	3.06	0.41	2.14
WEISH016	0.23	1.94	0.22	0.34
WEISH017	24.55	0.53	0.59	1.98
WEISH018	6.30	0.09	6.66	5.86
WEISH019	432.67	66.13	1.83	12.38
WEISH020	36.72	28.88	0.73	1.67
WEISH021	59.42	27.70	54.59	4.72
WEISH022	50.69	1.06	0.16	2.94
WEISH023	360.92	1000.00	184.09	17.81
WEISH024	885.64	191.31	13.77	45.36
WEISH025	1000.00	1000.00	166.47	11.31
WEISH026	1000.00	0.31	336.20	4.16
WEISH027	1000.00	1000.00	477.28	5.83
WEISH028	1000.00	1000.00	1000.00	6.97
WEISH029	1000.00	1.05	1000.00	8.56
WEISH030	1000.00	4.19	812.52	3.78
# unsolved	**7**	**6**	**2**	**0**

Fig. 8 Effect of clause generation on decision multi-dimensional knapsack problems

5 Conclusion

This chapter has emphasized some of the learning techniques that are responsible for recent improvement in SAT and CP solvers.

Strategies based on learning (like impact-based strategies as well as conflict-based strategies) have proved their efficiency on practical problems. The underlying variables and values selection methods remains quite intuitive and they can even compete with special-purpose strategies. Interestingly, learning-based strategies are still exploring the first-fail principle. However, whatever the strategy, it must not be trusted in any case and all the time. Some randomization needs to be introduced.

This not only introduces diversity in a strategy that is conservative but also can improve it by applying learning to search areas that were not considered promising a priori.

Structural learning carried out by clause generation is an essential reason for the improvements of SAT solvers during the last decade. In CP, the problem of obtaining some structural information for solving a model has always been deferred

to the user. Thus, the large number of global constraints developed for CP solvers. Despite this, clause learning can still be an effective method for solving CP models.

Many of the ideas presented in this chapter have been there for quite some time (pseudo-costs in integer programming or no-goods for CP, for instance). However, they are now used again, improved and adapted in the perspective of designing an autonomous search. There is a strong need for autonomous search from people that solve problems. In industry, the time devoted to optimization projects has decreased. The size of optimization teams in companies where optimization is not the primary objective has decreased as well. Prototypes that solve complex problems must often be developed in a few days and they need to produce good solutions from the beginning. There is often no time for developing a dedicated method even if it exists in the academic literature. Flexibility and rapid adaptation to model changes are the priority. Thus, the need for rapid modeling tools but also for solvers that find good solutions without needing an expert user. Here is the challenge for the learning schemes of search methods and this will certainly be an active research area in the next decade.

References

1. ILOG CPLEX 11.0. (2009) User Manual. ILOG, S.A., Gentilly, France
2. Baptista L, Marques-Silva J (2000) Using randomization and learning to solve hard real-world instances of satisfiability. In: CP '02: Proceedings of the 6th international conference on principles and practice of constraint programming, London, 2000. Springer, London, pp 489–494
3. Benichou M, Gauthier JM, Girodet P, Hentges G, Ribiere G, Vincent O (1971) Experiments in mixed-integer linear programming. Math Program (1):76–94
4. Bessiere C, Chmeiss A, Sais L (2001) Neighborhood-based variable ordering heuristics for the constraint satisfaction problem. In: Proceedings of CP 2001, pp 565–569
5. Bessière C, Régin J-C (1996) MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems. In: CP96: second international conference on principles and practice of constraint programming, Cambridge, pp 61–75
6. Boussemart F, Hemery F, Lecoutre C, Sais L (2004) Boosting systematic search by weighting constraints. In: 16th European conference on artificial intelligence (ECAI'04)e, Valencia, Spain, August 2004, pp 149–164
7. Brélaz D (1979) New methods to color the vertices of a graph. Comm ACM (22):251–256
8. Tabary S, Lecoutre C, Sas L, Vidal V (2007) Nogood recording from restarts. In: Proceedings of the 20th international joint conference on artificial intelligence (IJCAI'07), Hyderabad, India, 2007, pp 131–136
9. Cambazard H, Jussien N (2006) Identifying and exploiting problem structures using explanation-based constraint programming. Constraints 11(4):295–313
10. Davis M, Logemann G, Loveland D (1962) A machine program for theorem proving. Comm ACM (5):394–397
11. Dechter R (1986) Learning while searching in constraint-satisfaction-problems. In: 5th AAAI, Philadelphia, pp 178–185
12. Dechter R (1990) Enhancement schemes for constraint processing: backjumping, learning and cutset decomposition. Artif Intell 41(3):273–312
13. Rothberg E, Danna E, Le Pape C (2005) Exploring relaxation induced neighborhoods to improve MIP solutions. Math program 102(1):71–90
14. Fischetti M, Lodi A (2003) Local branching. Math program 98:23–47

15. Gauthier J-M, Ribiere G (1977) Experiments in mixed-integer linear programming using pseudo-costs. Math program (12):26–47
16. Goldberg E, Novikov Y (2007) Berkmin: A fast and robust sat-solver. Discrete Appl Math 155(12):1549–1561
17. Gomes C (2003) Complete randomized backtrack search (survey). In: Milano M (ed) Constraint and integer programming: toward a unified methodology. Kluwer, Dordrecht, pp 233–283
18. Gomes CP, Selman B, Kautz H (1998) Boosting combinatorial search through randomization. In: 15th AAAI, Madison, July 1998, pp 431–437
19. Haralick R, Elliot G (1980) Increasing tree search efficiency for constraint satisfaction problems. Artif Intell (14):263–313
20. IBM ILOG CP OPtimizer 1.0 (2007) User Manual. IBM, Gentilly, France
21. Katsirelos G, Bacchus F (2003) Unrestricted no-good recording in CSP search. In: Proceedings of CP'03, pp 873–877
22. Lynce I, Baptista L, Marques-Silva J (2001) Complete search restart strategies for satisfiability. In: IJCAI workshop on stochastic search algorithms, pp 1–5
23. Linderoth J, Savelsberg M (1999) A computational study of search strategies for mixed integer programming. INFORMS J Computing 11(2):173–187
24. Luby M, Sinclair A, Zuckerman D (1993) Optimal speedup of Las Vegas algorithms. Inform Process Lett pp 173–188
25. Marques-Silva J, Sakallah K (1996) Grasp—a new search algorithm for satisfiability. In: ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on computer-aided design, Washington, DC, USA, 1996. IEEE Computer Society, pp 220–227
26. Moskewicz MW, Madigan CF, Malik S (2001) Chaff: engineering an efficient SAT solver. In: Design automation conference, pp 530–535
27. Nemhauser G, Wolsey L (1988) Integer and combinatorial optimization. Wiley, New York
28. Padberg M, Rinaldi G (1991) A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems. SIAM Rev 33(1):60–100
29. Prosser P (1993) Hybrid algorithms for the constraint satisfaction problem. Comput Intell 9:268–299
30. Refalo P (2004) Impact-based search strategies for constraint programming. In: Proceedings of 6th international conference CP 2004, Toronto, October 2004. Springer, pp 369–383
31. Refalo P (2005) No-good learning and search restart for impact based strategies. In: Proceedings of INFORMS annual meeting, San Francisco, October 2005
32. Schiex T, Verfaillie G (1993) Nogood recording for static and dynamic CSP. In: Proceeding of the 5th IEEE international conference on tools with artificial intelligence (ICTAI'93), Boston, MA, November 1993, pp 48–55
33. Smith B (1999) The Brelaz heuristic and optimal static ordering. In: Proceedings of CP'99, (Alexandria, VA), pp 405–418
34. Stallman R, Sussman G (1977) Forward reasoning ad dependency-directed backtracking in a system for computer aided circuit analysis. Artif Intell (9):135–196
35. Wolsey LA (1998) Integer programming. Wiley, New York

What Is Autonomous Search?

Youssef Hamadi, Eric Monfroy, and Frédéric Saubion

Abstract Autonomous search is a particular case of adaptive systems that improve their solving performance by modifying and adjusting themselves to the problem at hand, either by self-adaptation or by supervised adaptation. We propose a general definition and a taxonomy of search processes with respect to their computation characteristics. For this purpose, we decompose solvers into components and their configurations. Some computation rules between computation stages are used to formalize the solver modifications and adaptations. Using these rules, we then sketch out and classify some well known solvers and try to answer the question: "What is Autonomous Search?"

1 Introduction

The selection and the correct setting of the most suitable algorithm for solving a given problem has already been investigated many years ago [87]. The proposed abstract model suggested to extract features in order to characterize the problem, to search for a suitable algorithm in the space of available algorithms and then to evaluate its performances with respect to a set of measures. These considerations are still valid and this general problem can indeed be considered at least from two complementary points of view:

- Selecting solving techniques or algorithms from a set of possible available techniques
- Tuning an algorithm with respect to a given instance of a problem

To address these issues, the proposed approaches include tools from different computer science areas, especially from machine learning. Moreover, they have

Y. Hamadi (✉)
Microsoft Research, 7 JJ Thomson Avenue, Cambridge, CB3 0FB, United Kingdom
and
LIX Ecole Polytechnique, F-91128 Palaiseau, France
e-mail: youssefh@microsoft.com

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_11,
© Springer Science+Business Media, LLC 2011

been developed to answer the algorithm selection problem in various fields as described in the recent survey of K. Smith-Miles [90]. In this chapter, we focus on the restriction of this general question to constraint satisfaction and optimization problems. Therefore, constraint solvers and optimization techniques constitute the core of our study.

1.1 Related Approaches

The problem of finding the best configuration in a search space of heuristic algorithms is also related to the recent notion of hyper-heuristics [10, 11, 21]. Hyper-heuristics are methods that aim at automating the process of selecting, combining, generating, or adapting several simpler heuristics (or components of such heuristics) to efficiently solve computational search problems. Hyper-heuristics are also defined as "heuristics to choose heuristics" [23] or "heuristics to generate heuristics" [8]. This idea was pioneered in the early 60's with the combination of scheduling rules [20, 36]. Hyper-heuristics that manage a set of given available basic search heuristics by means of search strategies or other parameters have been widely used for solving combinatorial problems (see Burke et al. [11] for a recent survey).

From a practical point of view, Burke et al. [12] proposed a comprehensive classification of hyper-heuristics considering two dimensions: the nature of the heuristics and the source of the feedback for learning. They thus distinguish between heuristics that select heuristics from a pre-existing set of search heuristics and heuristics that generate new heuristics from basic components. Concerning the feedback, they identify three categories: online learning, offline learning, and no learning. The distinction between online and offline processes was previously proposed in order to classify parameter setting in evolutionary algorithms [29], distinguishing parameter tuning (offline) from parameter control (online).

- As classical offline mechanisms, we may mention *portfolio* algorithms [50, 100], where previously acquired knowledge is used in order to select the suitable solving method with regard to a given problem instance. M. Gagliolo et al. [44] use reinforcement learning based techniques for algorithm selection.
- Online control of heuristics has been widely addressed, for instance, in adaptive strategies in evolutionary computation [67, 93], in adaptive neighborhood selection for local search [19, 59, 85], or in constraint programming solvers [27].

When considering parameter setting, the space of possible algorithms is the set of possible configurations of a given algorithmic scheme induced by the possible values of its parameters that control its computational behavior. Parameter tuning of evolutionary algorithms has been investigated for many years (we refer the reader to the book [68] for a recent survey). Adaptive control strategies were also proposed for other solving approaches such as local search [58,84]. Offline mechanisms

are also available for tuning parameters, such as the work of Hutter et al. [51], which proposes to use a local search algorithm in order to automatically find a good (i.e., efficient) configuration of an algorithm in the parameters space. Including this work, a more complete view of the configuration of search algorithms is presented in the PhD thesis of F. Hutter [61]. Revac [80, 81] is a method that uses information theory to identify the most important parameters and calibrate them efficiently. We may also mention that racing techniques [17, 101–103] can be used to choose suitable parameter settings when faced to multiple choices.

Another important research community that focuses on very related problems has been established under the name *Reactive Search* by R. Battiti et al. [5, 7]. After focusing on local search with the seminal works on reactive tabu [18] or adaptive simulated annealing [62], this community is now growing through the dedicated conference LION [4].

It clearly appears that these approaches share common principles and purposes and have been developed in parallel in different but connected communities. Their foundations rely on the fact that, since the solving techniques and search heuristics are more and more sophisticated and the problems structures more and more intricate, the choice and the correct setting of a solving algorithm is becoming an intractable task for most users. Therefore, there is a rising need for an alternative problem solving framework. According to the above brief historical review, we have remarked that these approaches have indeed their own specificities that are induced by their seminal supporting works. In this chapter, we propose to integrate the main motivations and goal into the more general concept of Autonomous Search (AS) [54, 55].

1.2 Autonomous Search

An Autonomous Search system should provide the ability to advantageously modify its internal components when exposed to changing external forces and opportunities. It corresponds to a particular case of adaptive systems with the objective of improving its problem solving performance by adapting its search strategy to the problem at hand. Internal components correspond to the various algorithms involved in the search process – heuristics, inference mechanisms, etc. External forces correspond to the evolving information collected during this search process – search landscape analysis (quality, diversity, entropy, etc.), external knowledge (prediction models, rules, etc.), etc . This information can be either directly extracted from the problem or indirectly computed through the perceived efficiency of algorithm's components. Examples of collected information include the size of the search space (exact or estimated), the number of sub-problems, etc. Computed information includes the discriminating degree of heuristics, the pruning capacity of inference techniques, etc. Information can also refer to the computational environment which can often vary, e.g., number of CPU cores.

Many pieces of work are distributed over various research areas and we see benefits in providing an overview of this new trend in problem solving. Nevertheless, Autonomous Search is particularly relevant to the constraint programming community, where many work has been conducted to improve the efficiency of constraint solvers. These improvements often rely on new heuristics, parameters, or hybridization of solving techniques and therefore, solvers are becoming more and more difficult to design and manage. In this chapter, we thus propose to focus on recent advances in Autonomous Search for constraint solving. To this end, we propose a description of the architecture of generic constraint solvers. Our purpose is not to present an abstract computation rule system but rather to give a taxonomy of the various components and configurations of a solver. We then want to propose a classification of basic search processes. For example, we try to differentiate *offline* tasks from *online* processes, *tuning* (adjustment of parameters and heuristics before solving) from *control*, etc. Note that previous taxonomies and studies have already been proposed for specific families of solving techniques or for specific solving purposes [12, 25, 29, 31, 44]. Therefore, these works may use different terms and concepts. Based on these works, our purpose is thus to provide a comprehensive classification methodology that would be able to handle these previously identified principles and that would be precise enough to cover wider families of solving methods, especially with regard to their components and solving processes. Here, we apply our classification methodology to various representative solvers. We can then define Autonomous Search as search processes that integrate *control* in their solving process, either by self adaptation (in which the rule application strategy is modified during solving by the solver itself) or by supervised adaptation (in which an "external" mechanism modifies the solving process). We finally propose some challenges for performance evaluation and continuous search.

1.3 Chapter Organization

This chapter is organized as follows. In Sect. 2, we describe the general architecture of solvers we consider and the specificities of autonomous solvers. We also formalize the basic solving mechanisms based on a set of rules. In Sect. 3, we illustrate different solver architectures by providing examples from the literature and we characterize these solvers using our previous description framework. In Sect. 4, we point out some future research directions and highlighting the main challenges that must be tackled for designing more autonomous systems.

2 Solvers Architecture

In this section, we present the general basic concepts related to the notion of solver in the context of general constraint problems solving, which provides a general introduction on problem solving. By general problems, we mean optimization or

constraint satisfaction problems, whose variables may take their values over various domains (integers, real numbers, Boolean, etc.). In fact, solving such problems is the main interest of different but complementary communities in computer science: operation research, global optimization, mathematical programming, constraint programming, artificial intelligence, etc. Among the different underlying paradigms that are associated to these research areas, we may try to identify common principles, which are shared by the resulting solving algorithms and techniques that can be used for the ultimate solving purpose.

2.1 General Architecture

As it has finally been suggested by the notion of metaheuristics [40], solvers could be viewed as a general skeleton whose components are selected according to the problem or the class of problems to be solved. Indeed, from our point of view, we want to look carefully at the components of the solver that define its structural properties and at its parameters or external features that define its behavior. On one hand, one has to choose the components of the solver and on the other hand one should configure how these internal components are used during the solving process. We identify the core of the solver which is composed by one or several solving algorithms. Note that here we distinguish between the solver and the solving algorithm, which is a part of the solver but corresponds to the real operational solving process. A basic solving algorithm corresponds to the management of solving techniques, abstracted by the notion of operators, making use of a solving strategy that schedules the use of these operators. A solving algorithm is designed of course according to the internal model, that defines the search space, and uses a function to evaluate the elements of the search spaces. All these components can be subjected to various parameters that define their behavior. A given parametrization defines thus what we call a configuration of the solver. At this level, a control layer can be introduced, especially in an autonomous solver, to manage the previous components and modify the configuration of the solver during the solving process. The general description of a solver architecture is illustrated by Fig. 1.

2.1.1 Problem Modeling/Encoding

The encoding of the problem is considered apart from the solver itself. In fact, most of the time, a solver is designed for a specific encoding framework that induces a specific internal representation that corresponds to the model. While the classic Constraint Satisfaction Problem (CSP) modeling framework [95] is commonly used as a description tool for all solving methods, the internal encoding of the problem and its possible configurations involve different representations (e.g., complete vs. partial

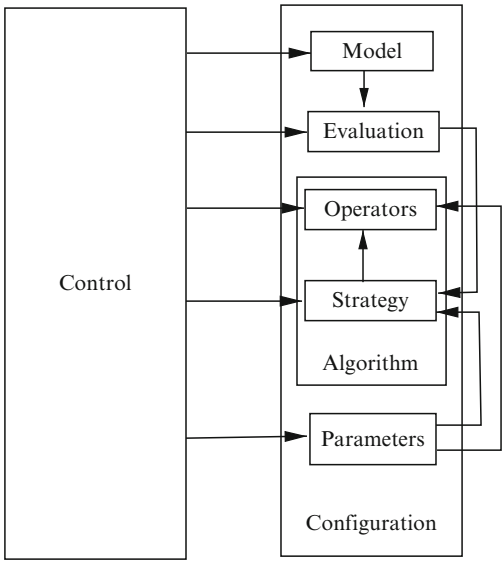


Fig. 1 The general architecture of a solver

assignments, etc.). One should note that different modeling and encoding paradigms can be used. In constraint programming [3, 26, 48, 75], one could encode constraints as tuples of allowed values or using a more declarative first order language with relations and functions. Moreover, other paradigms can be used to encode CSPs, such as SAT [14], and various transformation schemes have been investigated [15, 57, 96]. On the metaheuristics side, the encoding of the possible configurations of the problem has a direct impact on the search space and on the search landscape. For instance, one may include directly some of the constraints of the problem in the encoding as this is the case when using permutations for the Traveling Salesman Problem (TSP [1]), which corresponds to the constraint: *Each city is visited once and only once*. In genetic algorithms [24, 33, 72] or local search [2, 53], encoding may have a significant impact on the performance of the algorithm. The encoding of continuous optimization problems (i.e., over real numbers) also requires providing suitable data structures, for instance, floating point representation for genetic algorithms [63] or continuous and interval arithmetic in constraint programming [9]. The internal representation of the model can be considered as a component of the solver. This representation has of course a direct computational impact on the evaluation function and also on the solving techniques that are implemented through operators.

2.1.2 The Evaluation Function

The evaluation function is related to the nature of the problem. From a general point of view, a function is needed to evaluate possible configurations of the problem with

regard to its constraints and variables values. An evaluation function may evaluate the number of conflicts or check the satisfiability of a given constraint set, or use particular consistency notions (global or local). Such a function can also be used to prune the search space when dealing with optimization problems. Again, this notion is more classically used in the context of metaheuristics than in classic complete constraint programming solvers. But it seems rather intuitive to have such a function to assess the current search state in order to be able to check if the solver has reached a solution or not. Moreover, this evaluation function clearly appears when dealing with constraint optimization problems and using branch and bound algorithms.

2.1.3 The Solving Algorithm

Our purpose is to distinguish between the basic structure of the algorithm and its configurable components. For instance, in a classic complete constraint solver, the skeleton of the algorithm is the basic backtracking process whose heuristics and propagation rules can be configured. In an evolutionary algorithm, the core of the solver is constituted by the population management. A solver may include the following components that we have to take into account:

- *A set of operators*: operators are used in the solving process to compute search states. These operators may basically achieve variable instantiation, constraint propagation, local moves, recombination or mutation operators, selection, etc. Most of the time, they are parameterized and use an evaluation function to compute their results (e.g., number of violated constraints or evaluation of the neighborhood in local search algorithms). Note that these operators may be used to achieve a complete search (i.e., able to find a solution or prove unsatisfiability of the problem) or to perform an incomplete search (i.e., find a solution if possible or a suboptimal solution).
  - Concerning tree search based methods, the notion of operator for performing solving steps during the search process rather corresponds to basic solving techniques. For instance, if we consider a classic backtracking based solver in constraint programming, we need an enumeration operator that is used to assign values to variables and reduction operators that enforce consistencies in order to reduce the domains of the variables. The search process then corresponds to the progressive construction of a search tree whose nodes are subjected to the application of the previously described operators. When considering numerical variables over intervals, we may add splitting operators. Of course, these operators may include heuristics concerning the choice of the variables to be enumerated, the choice of the values but also other parameters to adjust their behavior. Indeed, constraint propagation can be formalized by means of rules [3, 34], which support operators-based description and provide a theoretical framework to assess properties of the solver such as termination.
  - On the metaheuristics side, in evolutionary computing [24, 33, 43], we usually consider variation operators (mutation operators and recombination

operators) and selection operators. Considering an evolutionary algorithm, it is possible to establish some convergence properties such as the famous schemata theorem [56]. There exist some general purpose operators as, for instance, the uniform crossover [91] or the Gaussian mutation [64]. To get better performances, these operators are often designed with respect to the specificities of the problem to be solved. In local search[2], local moves are based on neighborhoods functions.

All these operators are most of the time subjected to parameters that may modify their behavior but, more importantly that also control their application along the search process.

- *A solving strategy*: the solving strategy schedules how operators are used. Back to previous example, in a complete tree-based search process, the strategy will consist in alternating enumeration and constraint propagation. The strategy can be subjected to parameters that will indicate which operators to choose in the general scheduling of the basic solving process. Concerning metaheuristics, it is clear that the strategy corresponds for instance to the management of the temperature schedule in simulated annealing or the tabu list in tabu search.

2.1.4 Configuration of the Solver: The Parameters

The solver usually includes parameters that are used to modify the behavior of its components. A configuration of the solver is then an instance of the parameters together with its components. Parameters are variables that can be used in the general search process to decide how the other components are used. These parameters may correspond to various data that will be involved in the choice of the operator to be applied at a given search state. For instance, we may consider the probability of application of the operators (e.g., genetic operators in evolutionary algorithms, the noise in random walk for local search algorithms [89]) or to some tuning of the heuristics themselves (e.g., tabu list length in Tabu Search [41]).

Setting of the parameters is an important issue for evolutionary algorithms [68]. Setting of the parameters for local search algorithms is also handled in [7]. In constraint programming, much work has been done to study basic choice heuristics (see [27] for instance), but also to evaluate the possible difficulties related to the classic use of basic heuristics such as heavy-tailed problems [45] (these studies particularly demonstrate the benefit of randomization when solving multiple instances of a given family of problem compared to the use of a single predefined heuristics) .

2.1.5 Control

Modern solvers also include external or internal mechanisms that allow the solver to change its configuration by selecting the suitable operators to apply, or tuning the parameters, or adding specific information to the model. These mechanisms often

include machine learning techniques and will be detailed later. Of course, control rules will often focus on the management of the parameters and/or of the operators of the solver.

2.1.6 Existing Classifications and Taxonomies

As mentioned in the introduction, we may identify at least three important domains where related work has already been conducted. These lines of work have led to the use of different terminologies and concepts that we try to recall here.

In evolutionary computing, parameter setting [68] constitutes a major issue and we may recall the taxonomy proposed by Eiben et al. [29] (see Fig. 2).

Methods are classified depending on whether they attempt to set parameters before the run (tuning) or during the run (control). The goal of parameter tuning is to obtain parameters values that could be useful over a wide range of problems. Such results require a large number of experimental evaluations and are generally based on empirical observations. Parameter control is divided into three branches according to the degree of autonomy of the strategies. Control is deterministic when parameters are changed according to a previously established schedule, adaptive when parameters are modified according to rules that take into account the state of the search, and self-adaptive when parameters are encoded into individuals in order to evolve conjointly with the other variables of the problem.

In [88], Eiben and Smit recall the difference between numeric and symbolic parameters. In [82], symbolic parameters are called components whose elements are operators. In this chapter, we choose to use the notions of parameters for numeric parameters. As defined above, the operators are configurable components of the solver that implement solving techniques.

In [5], reactive search is characterized by the integration of machine learning techniques into search heuristics. A classification of the source of information that is used by the algorithm is proposed to distinguish between problem dependent information, task dependent information, and local properties.

In their survey [12], Burke et al. propose a classification of hyper-heuristics that are defined as "search methods or learning mechanisms for selecting or generating heuristics to solve computational search problems." As mentioned above, this classification also distinguishes between two dimensions: the different sources of feedback information and the nature of the heuristics search space. This classification is summarized in Fig. 3.

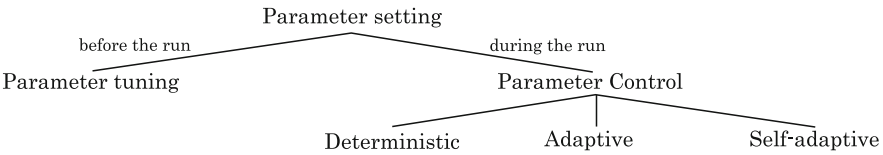


Fig. 2 Control taxonomy proposed by Eiben et al. [29]

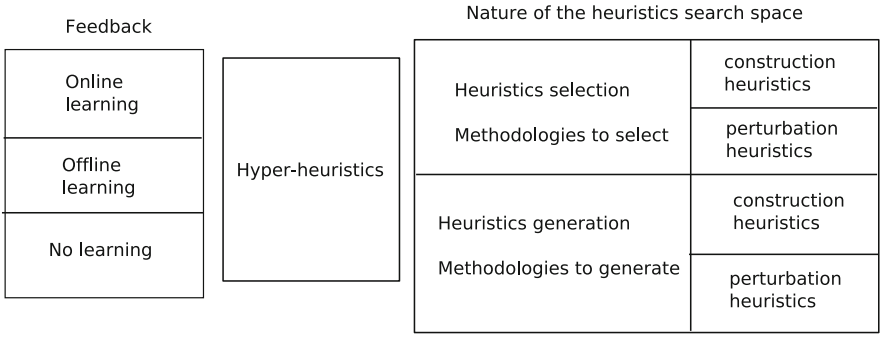


Fig. 3 Classification of hyper-heuristics proposed by Burke et al. [12]

The feedback, when used, corresponds here to the information that is learned during solving (online) or using a set of training instances (offline). The authors identify two families of low level heuristics : construction heuristics (used to incrementally build a solution) and perturbation heuristics (used to iteratively improve a starting solution). The hyper-heuristics level can use heuristics selection methodologies, that produce combinations of pre-existing low level heuristics, or heuristics generation methodologies, that generate new heuristics from basic blocks of low level heuristics.

Another interesting classification is proposed in [44], in which Gagliolo et al. are interested in the algorithm selection problem [87] and describe the different selection techniques according to the following points of views. The problem consists in assigning algorithms from a set of possible alternative solving methods to a set of problem instances in order to improve the performance. Different dimensions are identified with regard to this algorithm selection problem :

- The nature of the problems to be solved : decision vs. optimization problems
- The generality of the selection process : selection of an algorithm for a set of instances or selection of an algorithm for each instant
- The reactivity of the selection process : the selection can be static and made before ruining all the selected algorithms or can be dynamically adapted during execution
- The feedback used by the selection process : the selection can be made from scratch or using previously acquired knowledge
- The source of feedback: as in the previous classification, when learning is used in the selection process, one may consider offline (using separated training instances) or online (updating information during solving) learning techniques

As claimed in the introduction, autonomous search aims at providing a more uniform description and characterization of these different trends, which have close relationships.

2.2 Architecture of Autonomous Solvers

We may define autonomous solvers as solvers that contain control in their search process (i.e., the solvers described in Sect. 3.2). We want to study such autonomous systems with respect to their specific control methods.

A general control process includes a strategy that manages the modification of some of the solver's components and behavioral features after the application of some solving functions. The overall strategy to combine and use components and parameters can be based on learning that uses information from the current solving process or from previous solved instances (see remarks in Sect. 2.1.6). Therefore, modifications are often based on a subset of search states. Given a solver, we have to consider the interactions between the heuristics and the strategy which selects the heuristics at a meta-level (notion of hyper-heuristics).

On the one hand, one can consider the solver and its history and current environment (i.e., the previously computed search states and eventually other external information related to previous computations) as an experimental system, which is observed from an external point of view. Such a supervised approach then consists in correctly controlling the solver by adjusting its components according to criteria and decision rules (these rules may be automatically generated by means of statistics and machine learning tools or even by human experts). On the other hand, one may consider that the solver changes the environment at each step of the solving process and that this environment returns feedback information to the solver in order to manage its adaptation to this changing context (different types of feedback may be taken into account as mentioned in Sect. 2.1.6). In this case, we will use self adaptation. To illustrate these ideas, we propose a high level picture of an autonomous search system (see Fig. 4).

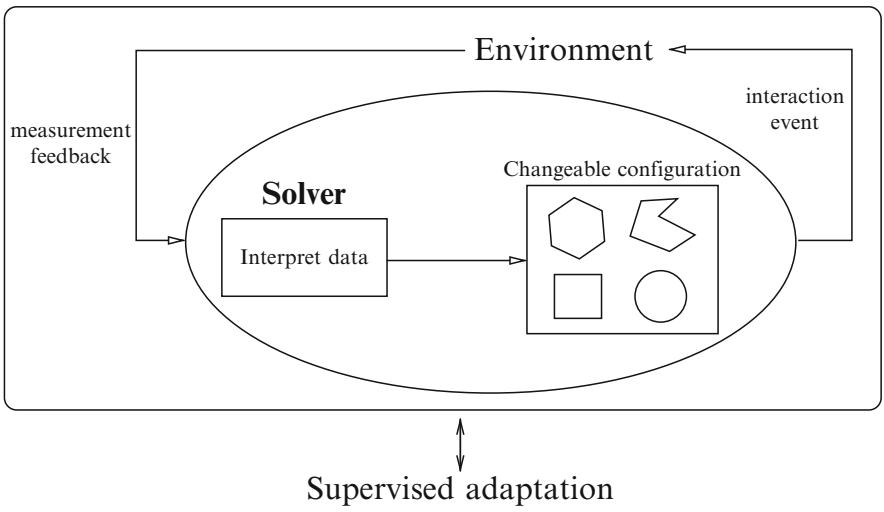


Fig. 4 The global architecture of an autonomous search system

2.2.1 Control by Self Adaptation

In self adaptation, the adaptive mechanism is coupled with the search components, directly changing them in response to their actions. Self-adaptive techniques are tightly integrated with the search process and should usually require little overhead. The algorithm is observing its own behavior in an online fashion, modifying its parameters accordingly. This information can be either directly collected on the problem or indirectly computed through the perceived efficiency of individual components. Because the adaptation is done online, there is an important trade-off between the time spent computing heuristic information and the gains that are to be expected from this information. Therefore, we can consider that the strategy depends on the set of computed states and changes during solving.

2.2.2 Control by Supervised Adaptation

Supervised adaptation works at a higher level. It is usually external and its mechanisms are not coupled with the search process. It can be seen as a monitor that observes the search and analyzes it. Then, it modifies the components of the solver (or requires the solver to modify its components) in order to adapt it. Supervised adaptation can use more information, e.g., learning-based knowledge. In some cases, we can imagine that typical supervised actions could be "compiled" into self-adaptive mechanisms.

2.3 Searching for a Solution vs. Solutions for Searching

It appears now that the problem of building a good Autonomous Search solver is more ambitious than finding a solution to a given instance of a problem. Indeed, inspired by the seminal consideration of John Rice [87] when he was abstracting the problem of finding the best algorithm for solving a given problem, we need to take into account at least three important spaces in which an autonomous search process takes place.

- *The search space*: the search space is induced by the encoding of the problem and corresponds to the set of all potential configurations of the problem that one has to consider in order to find a solution (or to find all solutions, or to find an optimal solution). This search space can also be partitioned, for optimization problems, into the set of feasible solutions and infeasible solutions with respect to the constraints of the problem.
- *The search landscape*: the search landscape is related to the evaluation function that assigns a quality value to the elements of the search space. If indeed this notion is rather of limited use in the area of complete solvers, this is a crucial notion when using heuristics or metaheuristics, search algorithms whose purpose is to explore and exploit this landscape in order to find solutions. Most of the

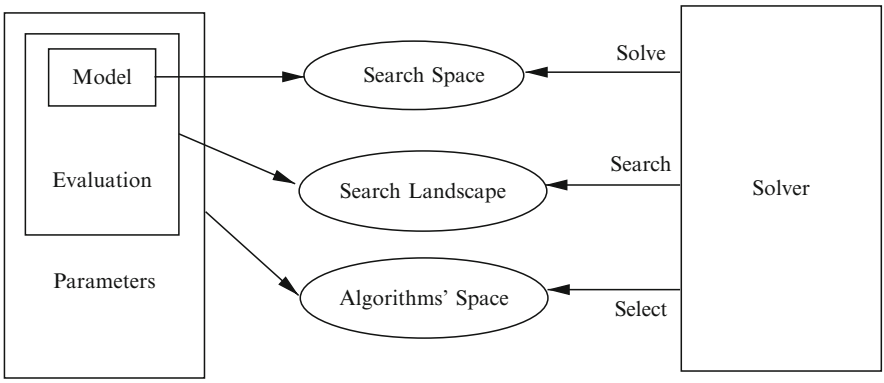


Fig. 5 The solver and its action with respect to different spaces

metaheuristics are designed accordingly to the management of this exploration–exploitation balance and the characteristics of the search landscapes, often use geographical metaphors: How to travel across plateaus? How to escape from a local optimum by climbing hills?

- *The algorithms' space*: according to the previous description of solvers architecture, we have highlighted that a solver consists of components that define its structural properties together with a set of behavioral features (parameters and control rules). As mentioned before, given a basic algorithmic skeleton, we may consider a set of possible solvers that correspond to the possible components choices and configurations. This algorithms' space can also be composed of different solvers when dealing with portfolio-based algorithm selection.

The relationships between these spaces are illustrated in Fig. 5. Indeed, the ultimate autonomous search purpose can be formulated as: finding a suitable algorithm that is able to efficiently explore and exploit the search landscape in order to suitably manage the search space and find solutions to the initial problem.

2.4 A Rule-Based Characterization of Solvers

As already mentioned, the solving techniques used for solving such problems may include very different features from complete tree based solvers to local search or evolutionary algorithms. In this presentation, we will attempt to abstract theses solving features in order to be able to address general solving algorithms, focusing on their autonomous aspects as described above. Indeed, such rule-based formalizations have already been proposed for modeling some constraint programming solving processes [3, 34] and also for hybrid solvers including local search [79]. Here, our purpose is not really to prove some properties of the solvers but rather to highlight their basic operational mechanisms in order to classify them with regard to their behavioral and structural characteristics.

When using a solver, one may distinguish two main tasks that correspond indeed to different but closely related levels of technical accuracy that can be achieved by more or less specialized users:

- *The component design*: this phase, consists in choosing the suitable components described in Sect. 2.1.3 that should be included in the solver with regard to the problem characteristics for instance. As mentioned above, these components constitute the architecture of the solver.
- *The configuration of the solver through parameters settings and control*: this second phase consists in defining through control features how the components can be used during the solving process.

Based on this consideration and on the general solver architecture depicted in Fig. 1, we propose a formal description in the next section.

2.4.1 Formal Description

We define here some basic notions in order to characterize the behavior of solvers with a computationally oriented taxonomy. This approach will allow us to characterize the solvers. We first recall some basic concepts related to constraint satisfaction and optimization problems.

Definition 1 (CSP). A CSP is a triple $(X, D, \mathscr{C})$, where $X = \{x_1, \ldots, x_n\}$ is a set of variables whose values are restricted to given domains $D = \{D_1, \ldots, D_n\}$. There exists a bijective mapping that assigns each variable x_i to its corresponding domain, that will be noted D_{x_i}. We consider a set of constraints $\mathscr{C}$ as a set of relations over the variables X.

Definition 2 (Search Space). The search space $\mathscr{S}$ is a subset of the possible configurations of the problem and can be the Cartesian product of domains $\Pi_{x \in X} D_x$. The choice of the internal representation (i.e., the model) defines the search space. An element s of the search space will be called a **candidate solution**.

Definition 3 (Solution). A feasible solution is an assignment of values to variables, which can be seen as an element of $\mathscr{S}$ (i.e., given an assignment $\theta : X \rightarrow \Pi_{i=1}^{n} D_i$, $\theta(x_i) \in D_{x_i}$), and which satisfies all the constraints of $\mathscr{C}$. In the context of optimization problems, we also consider an objective function $f : \mathscr{S} \rightarrow \mathbb{R}$. An optimal solution is a feasible solution maximizing or minimizing, as appropriate, the function f.

We have now to define, according to Sect. 2, the different elements that are included in the solver.

Definition 4 (Evaluation Functions). We denote by E the set of evaluation functions $e : \mathscr{S} \rightarrow \mathbb{R}$.

Definition 5 (Parameters). We denote by P the set of parameters and a parametrization π is a mapping that assigns a value to each parameter. We denote by Π the set of parameterizations.

Definition 6 (Solving operators). We denote by Ω a set of solving operators (operators for short) that are functions $o : 2^{\mathcal{S}} \rightarrow 2^{\mathcal{S}}$.

Definition 7 (Solving strategy). We denote by H the set of solving strategies that are functions $h : 2^{\mathcal{S}} \times \Pi \times E \rightarrow \Omega$.

For the, sake of simplicity, in the following, we refer to solving strategies as strategies. Solving strategies and solving operators are the key-points of the solving algorithm (see Fig. 1): a strategy manages some operators to compute the solutions. We obtain:

$$\textbf{Solving algorithm} = \textbf{solving strategy} + \textbf{solving operators}$$

The granularity to describe a solving algorithm can vary a lot. We now give some examples for various granularities.

Consider a complete solver based on domain reduction and enumeration (usual backtrack search). First, let us consider a fine granularity. Then, the operators could be functions to reduce one variable with a given constraint or enumeration function to split the domain of a variable. A strategy h for achieving the maintaining of arc-consistency (MAC) would be to schedule (with respect to the given parametrization) and apply reduction functions up to obtaining a candidate solution (i.e., the Cartesian product of the current domains in terms of usual CSP) which is general arc consistent (GAC). The fact that the candidate solution is GAC or not would be given by the evaluation function. Then, h would apply an enumeration, which it would select with respect to the parametrization, e.g., a first fail to select the variable with the smallest domain. Then again, h would apply a pruning phase by scheduling operators up to reaching solutions.

Now, consider a coarse granularity. We could consider operators to enforce GAC and the other ones to enumerate with respect to a given enumeration strategy. The role of h in this case would only be to sequence a reduction operator (leading directly to a GAC candidate solution) with an enumeration (selected with respect to the same parameters as before). The process (one reduction, one operation) would iterate and terminate when reaching the solutions. In this case, the evaluation function is just needed to verify that a solution is reached.

Similarly, we can consider the same two granularities for a local search solver. The fine granularity would consider operators that are computation of neighbors, moves, or restart; the strategy would schedule these operators with respect to the parameters: for best-improvement hill climbing search, it could select the neighborhood function that computes all the neighbors and return the best one; for first improvement search, an operator that computes a better neighbor. The strategy would thus iterate neighborhood computation and moves until reaching a solution that cannot be improved (with respect to the evaluation function or given a number of moves). It could then perform a restart, and so on.

Coarse grain operators would be a complete local search or a restart. The strategy would then iterate a local search followed by a restart (or by a perturbation of the current optimum as in Iterated Local Search [60]).

We can see here that, depending on the level of details we want for describing a solving algorithm, we can model operators and strategies with different granularities. Note that there is a close relationship between the strategy and the parameters, and changes in the strategy can also be managed by means of parameters related to the strategy.

We now formalize the solving processes as transitions using rules over computation states.

Definition 8 (Computation State). Given a CSP $(X, D, \mathscr{C})$, a search space $\mathscr{S}$, a set of operators Ω, a set of evaluation functions E, a set of parameters P, and a set of solving strategies H, a computation state is a tuple $< O, \mathscr{S}, e, \pi, h|S >$ where:

- $O \subseteq \Omega$, where O is the set of operators currently used in the solver,
- $S \subseteq \mathscr{S}$, is the current subset of candidate solutions,
- $e \in E$, is an evaluation function,
- $\pi \in \Pi$ is the current parametrization,
- $h \in H$ is the current solving strategy.

Comments:

- It is important to note that Ω, E, and H are sets that may not be yet computable. For example, H represents the set of all possible strategies, either already existing or that will be discovered by the solver (as defined in Definition 12). Similarly, all the operators of Ω are not known since they can be designed later by the solver. However, O is known and all its operators as well.
- S corresponds to the internal basic search structure: the search state. For instance, if we consider a genetic algorithm, the search state will be a population. In the case of a complete backtracking solver, it will consist in an incomplete assignment, maybe associated with some no good records.
- O is the current set of operators available in the solver at a given stage and that are extracted from a set Ω of potential operators that could be used in this solver. Indeed, some solvers may use new solving operators that are produced online or offline according to a general specification or according to design rules. Note that an operator allows the solver to perform a transition from one search state to another. This is therefore the key concept of the solving process and we want to keep it as general as possible to handle various solving paradigms (as mentioned above).
- The evaluation function e must evaluate the candidate solutions. This evaluation is used by the strategy in order to drive the basic solving task and by the control in order to drive the solver behavior.
- The solving strategy h will be used to select the suitable operator to apply on the current candidate solutions with respect to the current parametrization π and the evaluation function e.

Note that, for the sake of simplicity, we restrict here solvers to have only one evaluation function and one search space at a time. This is generally the case, but this framework could be easily generalized to capture more "exotic" situations.

We denote by CS the set of computation states. Note that a computation state corresponds in fact to a search state together with the current configuration of the solver.

Definition 9 (Computation Rules). A computation rule is a rule $\frac{\sigma'}{\sigma}$ where σ and σ' are computation states from CS.

2.4.2 Identification of Computation Rules

We identify here specific families of computation rules with respect to the way they modify the computation states.

- *Solving*: The fundamental solving task of a classic solver consists in computing a new state from the current one according to a solving strategy that chooses the suitable operator to apply with respect to the current candidate solution, the parametrization, and the evaluation function. This corresponds to the following rule:

  [Solv] Solving
  $$\frac{< O, \mathscr{S}, e, \pi, h|S >}{< O, \mathscr{S}, e, \pi, h|S' >}$$
  where $S' = o(S)$ and $o = h(S, \pi, e) \in O$.

- *Parametrization*: The modification of the solver's parameters changes its configuration and can be used either to tune the solver before running it or to adjust its behavior during the run. A parametrization rule can be abstracted as:

  [Par] Parametrization
  $$\frac{< O, \mathscr{S}, e, \pi, h|S >}{< O, \mathscr{S}, e, \pi', h|S >}$$

- *Evaluation function modification*: Since we address here autonomous systems that are able to modify not only their configuration trough their parameters but also their internal components, we have to consider more intricate rules. A first way to adapt the solver to changes is to modify its evaluation function, which directly induces changes on the search landscape. This is the case when changing weights or penalties in the evaluation function (there are many examples, for instance [66, 83]).

[EvalCh] Evaluation modification

$$\frac{< O, \mathscr{S}, e, \pi, h|S >}{< O, \mathscr{S}, e', \pi, h|S >}$$

- *Operators modification*: Another possibility to modify the internal configuration of the solver is to change its set of operators. Note that operators can be added or discarded from the set O.

  [OpCh] Operators modification

$$\frac{< O, \mathscr{S}, e, \pi, h|S >}{< O', \mathscr{S}, e, \pi, h|S >}$$

- *Strategy modification*: Similarly, solving strategies can be changed to manage differently the operators and achieve a different solving algorithm. As mentioned above, a backtracking algorithm can apply different strategies for enforcing local consistency at each node, or in hybrid solving one may switch from complete to approximate methods.

  [StratCh] Strategy modification

$$\frac{< O, \mathscr{S}, e, \pi, h|S >}{< O, \mathscr{S}, e, \pi, h'|S >}$$

- *Encoding modification*: We also have to take into account solvers that will be able to change their encoding during execution. As this is the case for the evaluation modification, such changes will affect the search landscape.

  [EncCh] Encoding modification

$$\frac{< O, \mathscr{S}, e, P, h|S >}{< O, \mathscr{S}', e, P, h|S >}$$

Note that applying one of these rules (except [Res]) will generally require applying other computation rules. For example, a change of encoding ([EncCh]) will certainly require a change of operators ([OpCh]), of evaluation function ([EvalCh]), of strategy ([StratCh]), and of parametrization ([Par]). However, a change of strategy does not always imply a change of operators. Consider the fine grain description of the complete solver given above (reduction functions and enumeration operators). Then, the same operators can be used for an MAC or a full look-ahead algorithm: only the strategy has to be changed.

2.4.3 Control of the Computation Rules and Solvers

The most important part of our characterization concerns the control of the algorithm to finally build the solver. The control is used to act on the configuration of

the solver through its parameters but also to modify the internal components of the solver (parameters, operators, strategies, etc.).

Definition 10 (Control). Let $\mathscr{S}_{CS}$ be the set of all the finite sequences of elements of CS. A control function $K : \mathscr{S}_{CS} \rightarrow R$ is a function that selects a computation rule from the set R according to a sequence of computation states.

A solver state can be defined by a set of computation rules, and a sequence of computation states that have previously been computed.

Definition 11 (Solver State). A solver state is a pair (R, Σ) where:

- R is a set of computation rules as defined above
- Σ is a sequence of computation states that are recorded along the solving process.

Starting from a solver state (R, Σ), with $\Sigma = (\sigma_0, \ldots, \sigma_n)$ the next state is obtained as (R, Σ') where $\exists r \in R$, such that $K(\Sigma) = r$ and $\Sigma' = (\sigma_0, \ldots, \sigma_n, \sigma_{n+1} = r(\sigma_n))$.

Note that in practice, a solver state does not contain the complete history. Thus, the sequence of computation states is either limited to a given length, or only the most relevant computation states are kept.

Definition 12 (Solver). A solver is a pair (K, R) composed of a control function K and a set of computation rules R that will define a sequence of solver states.

A way of describing a solver is to use regular expressions to schedule computation rules to describe its control. Let's come back to the rules defined in Sect. 2.4. We consider the set of rules $R = Par \cup Res \cup EvalCh \cup EncCh \cup OpCh \cup StratCh$ where *Par* represents some parametrization rules [*Par*], *EvalCh* some evaluation modification rules [*EvalCh*]. Given two subsets R_1 and R_2 of R, R_1^* means that zero or more rules of R_1 are sequentially applied and $R_1 R_2$ means the sequential application of one rule of the subset R_1 followed by the application of one rule of R_2. $R_1 | R_2$ corresponds to use of one rule from R_1 or one from R_2. These notations will be used in the following section to highlight the characteristics of the solvers by means of the sequences of rules that they apply in their solving processes. We now have:

$$\textbf{Solver} = \textbf{Control} + \textbf{Configured Solving Algorithms}$$

We recall that we stated before that Solving algorithm = Solving Strategy + Solving Operators. Coming back to Fig. 3 that shows a classification of hyper-heuristics, we can notice that we obtain similar distinction here: solvers correspond to the hyper-heuristics of Fig. 3, solving algorithms to heuristics search space, strategies to heuristics selection or generation, and operators to construction or perturbation heuristics. We can finally identify an autonomous solver:

Definition 13 (Autonomous Solver). Consider a solver given by a regular expression *ex* of computation rules from $R = Par \cup Solv \cup EvalCh \cup EncCh \cup$

OpCh ∪ *StratCh*. A solver is autonomous if *ex* contains at least a rule from *Par* ∪ *EvalCh* ∪ *EncCh* ∪ *OpCh* ∪ *StratCh* (i.e., *ex* is not only composed of rules from *Solv*).

In fact, an autonomous solver is a solver that modifies its configuration during solving, using a control rule. Of course, there are various degrees in this autonomy scale. We can now come back to the previous taxonomy of offline/tuning and online/control (e.g., for parameters). Consider a solver given by a regular expression *ex* of computation rules from $R = Par \cup Solv \cup EvalCh \cup EncCh \cup OpCh \cup StratCh$, and the word *w* given by flattening this expression *ex*. The offline/tuning of a solver consists of the rules that appear in *ex* before the first Solv rule of *ex*. The online/control is composed of all the rules that appear after the first rule Solv and that are not of the Solv family of rules.

In the next section, we illustrate how these rules are used in real solvers and how they can be used to characterize families of solvers within our autonomous search scope.

3 Case Studies

In this section, we will not attempt to present an exhaustive view of existing solvers but we will rather choose some representative solvers or algorithms in order to illustrate different solving approaches and how the previous computation rules can be used to characterize these approaches. As mentioned in the introduction, autonomous search has been indeed investigated for many years, across many different areas and under different names. Therefore, we could not imagine providing an exhaustive discussion of all approaches. In our opinion, since autonomous search is a rising trend in problem solving, we rather propose to identify how autonomous mechanisms have been used in the literature and try to point out some future challenges for designers of such solvers.

3.1 Tuning Before Solving

As in [29,68], we use the word tuning for the adjustment of the different components of the algorithm before trying to solve an instance (see end of Sect. 2.4.3).

3.1.1 Preprocessing Techniques

Even if preprocessing is not directly linked to the core of the solving mechanism but relies on external processes, we have to consider it as an important component in the design of modern solvers. Nowadays, efficient solvers (e.g., DPLL) use simplification preprocessing before trying to solve an instance (see for instance the SAT solver

SatElite [30]). Note that the model transformation can maintain equisatisfiability or a stronger equivalence property (the set of solutions is preserved).

3.1.2 Parameter Tuning on Preliminary Experiments

Such a tuning phase may consist in setting correct parameters in order to adjust the configuration of the solver. Here, these settings are performed according to a given set of preliminary experiments. Tuning before solving will correspond to the configuration of the solver and then its use for properly solving the problem. Therefore, the general profile of the solvers will be mainly described as :

$$[Config]Solv^*$$

where $[Config]$ is of the form $(Par|EvalCh|OpCh|EncCh)^*$.

Empirical Manual Tuning: we include in this family the classic tuning task involved when using single metaheuristics based solvers where experiments are required to tune the various parameters [82, 88]. Of course, there exist similar studies in constraint programming to choose the suitable variable and value choice heuristics, and this task is often not formalized. Most of the time, parameters are tuned independently since it appears difficult to control their mutual interaction without a sophisticated model. Here, the parametrization is not really part of the solver but rather a preliminary experimental process.

$$Solver : Solv^*$$

Deciding the Size of a Tabu List: experiments or other previous analysis can be used to extract general parameters or heuristics' settings. In the context of Tabu Search for SAT, [77] have used an extensive offline experimental step to determine the optimal length of a tabu list. They used simple regression to derive the length of the list according to the number of variables n. Remarkably, the length is independent of the size of the constraints, and their formula applies to any hard-random k-SAT instance. Therefore, the parametrization can be included as a the first step of the solving process.

$$Solver : Par\_Solv^*$$

Automatic Parameter Tuning by an External Algorithm: recently, [51] proposed an algorithm to search for the best parameters in the parameter space and therefore to automatically tune a solver. Now, if we consider that this automated process is included in the solver, we have then the following description.

$$Solver : (Solv^* Par)^* Solv^*$$

Note that $(Solv^* Par)^*$ corresponds to a series of runs and parameter tuning, which is achieved automatically.

3.1.3 Components Setting Before Solving

We consider here methods that consist in choosing the correct components of the solver by using experiments and/or external knowledge that has been acquired apart from the current solving task. This knowledge can be formulated as general rules, can use more or less sophisticated learning techniques, or may also use an external computation process.

A. Learning Solver's Components

External mechanisms can be used before tuning to discover or learn efficient components for the solver.

Discovering heuristics: in [37], genetic programming is used to discover new efficient variable selection heuristics for SAT solving with local search algorithms. Candidate variable selection heuristics are evaluated on a set of test instances. This automatic process can be inserted before solving (the variable selection heuristics can induce a change of parameters or operators depending on the description granularity). Note that here the first $Solv^*$ is not applied to the problem at hand.

$$Solver : (Solv^*(OpCh|Par))^* Solv^*$$

The choice heuristics can be parameters of the operators in our formalism, heuristics discovering can be considered as the selection of suitable operators and their parametrization.

Learning evaluation functions: in [16], a new method is proposed in order to learn evaluation functions in local search algorithms and improve search efficiency based on previous runs.

$$Solver : (Solv^* EvalCh)^* Solv^*$$

B. Empirical Prediction of Instances Hardness

The following techniques are based on a learning component (e.g., clustering tools), which can be used to detect automatically the suitable heuristics and strategies to apply.

Portfolio-based: in SATzilla [100], offline linear basis function regression and classifiers are used on top of instances-based features to obtain models of SAT solvers runtime. During the exploitation phase, instances features are used to select the best algorithm from a portfolio of tree and local search based SAT solvers. We may also cite the works of Gebruers et al. [38] and Guerri et al. [42] that use case based reasoning and learning techniques to choose the appropriate solving technique among

constraint programming and integer linear programming. In these solvers schemes, the first $Solv^*$ corresponds again to preliminary experiments.

$$Solver : Solv^*(OpCh|StratCh|Par|EvalCh)^*Solv^*$$

Parameter-based : in [49,50], the authors use an approach similar to SATzilla. They showed that it is possible to predict the runtime of two stochastic local searches (SLS). In this work, the selection of the best method to apply on a given instance is changed into the selection of the best parameters of a given SLS algorithm.

$$Solver : ParSolv^*$$

3.2 Control During Solving

The control of the solver's behavior during the run can be achieved by either modifying its components and/or its parameters. This corresponds, for instance, to an online adjustment of the parameters or heuristics. Such control can be achieved by means of supervised control schemes or by self adaptive rules. Of course, such approaches often rely on a learning process that tries to benefit from previously encountered problems along the search or even during the solving of other problems. Therefore, the profile of the solvers will generally be:

$$([Config]Solv^*)^*$$

where $[Config]$ is of the form $(Par|EvalCh|OpCh|EncCh)^*$. Note that the outter * loop represents indeed the control loop.

3.2.1 Controlling Encoding

Hansen [47] proposes an adaptive encoding in an evolutionary algorithm in order to solve continuous function optimization problems. The representations of the solutions are changed along the search to reach an optimal representation that could simplify the solving of the initial problem.

$$Solver : (EncChSolv^*)^*$$

3.2.2 Controlling Variable Orderings and Values Selection in Search Heuristics

We consider here approaches where the heuristics functions change during the search with respect to the current state and parameters.

Hybrid approaches to discover efficient variable ordering: To illustrate this kind of approach, we may mention the SAT solving technique of [78] where a Tabu Search is used at each node of a DPLL to find the next variable to branch on.

$$Solver : ((OpChStratCh)Solv^* ParSolv^*)^*$$

A conflict-driven heuristic: in [13], important variables are deemed to be the ones linked to constraints that have frequently participated in dead-ends. During the search, this information is collected and used to order variables. Eventually, the system has enough knowledge to branch on important variables and quickly solve the problem. The system "learns" weights from conflicts that are used in the computation of the variable selection heuristics, this corresponds to an update of the parameters each time a conflict is met.

$$Solver : (ParSolv^*)^*$$

Implicit feed-back loops in modern DPLL solvers: in modern SAT solvers such as the one presented in [32], many implicit feed-back loops are used. For instance, the collection of conflicts feeds the variable selection heuristic, and the quality of unit propagation is sometimes used to control the restart strategy. Similarly, the deletion of learned clauses which is necessary to preserve performances uses activity-based heuristics that can point to the clauses that were the least useful for the unit propagation engine. Therefore, it induce changes in the model itself and in the heuristics parameters.

$$Solver : ((EncCh|Par)Solv^*)^*$$

Adapting neighborhood during the search: variable neighborhood search [59, 71, 85] consists in managing simultaneously several neighborhood functions and or parameters (according to the description granularity) in order to benefit from various exploration/exploitation facilities.

$$Solver : ((OpCh|Par)Solv)^*$$

3.2.3 Evolving Heuristics

Hyper-heuristics: hyper-heuristics [10] is a general approach that consists in managing several meta heuristics search methods from a higher strategy point of view. Therefore, it is closely related to autonomous search and has already been applied for many problems (e.g., SAT solving [8]). Since they switch from one solving technique to another, hyper-heuristics could be characterized by:

$$Solver : ((OpCh|StratCh|Par|EvalCh)^* Solv^*)^*$$

Learning Combinations of Well-known Heuristics: in the ACE project [28], learning is used to define new domain-based weighted combinations of branching heuristics (for variable and value selection). ACE learns the weights to apply through a voting mechanism. Each low-level heuristic votes for a particular element of the problem (variable, value). Weights are updated according to the nature of the run (successful or not). The learning is applied to a given class of problems. The combination is learned on a set of representative instances and used during the exploitation step. A similar approach has been used in [39] in order to learn efficient reduction operators when solving numerical CSPs.

$$Solver : (ParSolv^*)^*$$

3.2.4 Controlling Evaluation Function

This aspect may concern local search algorithms that use, for instance, adaptive weighting of the constraints in their evaluation function [74, 94]. Constraint weighting schemes solve the problem of local minima by adding weights to the cost of violated constraints. These weights increase the cost of violating a constraint and so change the shape of the cost surface with respect to the evaluation function. Note that these techniques are also widely used in SAT solvers [14].

$$Solver : (EvalChSolv^*)^*$$

3.2.5 Parameters Control in Metaheuristics Algorithms

We consider here approaches that change the parameters during the search with respect to the current state and other parameters. Of course, these parameters have a direct influence on the heuristics functions, but these latter functions stay the same during the solving process.

Reactive Search: in [7] (formerly presented in [6]), Battiti et al. propose a survey of so-called reactive search techniques, highlighting the relationship between machine learning and optimization processes. In reactive search, feedback mechanisms are able to modify the search parameters according to the efficiency of the search process. For instance, the balance between intensification and diversification can be automated by exploiting the recent past of the search process through dedicated learning techniques.

$$Solver : (ParSolv^*)^*$$

Adaptive Genetic Algorithms: adaptability is well-known in evolutionary algorithms design. For instance, there are classical strategies to dynamically compute the usage probability of GA search operators [92, 97, 98]. Given a set of search operators, an adaptive method has the task of setting the usage probability of each operator. When an operator is used, a reward is returned. Since the environment is

nonstationary during evolution, an estimate of the expected reward for each operator is only reliable over a short period of time [99]. This is addressed by introducing a quality function, defined such that past rewards influence operator quality by an extent that decays exponentially with time. We may also mention other works that use more sophisticated evaluation functions, rewards computation, and operator probability adjustment in order to manage dynamically the application parameters of the EA [35, 70, 76].

$$Solver : (ParSolv^*)^*$$

3.3 Control During Solving in Parallel and Distributed Search

The solvers described in this section also belong to the previous family of solvers that include control within their proper solving process. But here, due to the parallel/distributed architecture of solver, the sequence of computation rules is more difficult to schedule. Thus, the profile could be described as $([Config]|Solv^*)^*$

Value-ordering in Portfolio-based Distributed Search: in [86], the authors present a portfolio-based distributed search. The system allows the parallel execution of several agent-based distributed search. Each search requires the cooperation of a set of agents which coordinate their local decisions through message passing. An agent is part of multiple distributed search, and maintains the context of each one. Each agent can aggregate its context to dynamically rank the values of its local variables. The authors define several efficient portfolio-based value-ordering heuristics. For instance, one agent can pick up the value which is used most frequently in competing search, or the one which is most supported in other searches, etc.

$$Solver : (Par|Solv^*)^*$$

Adaptive Load-balancing Policies in Parallel Tree-based Search: Disolver is an advanced Constraint Programming library which particularly targets parallel search [46]. This search engine is able to dynamically adapt its interprocesses knowledge-sharing activities (load balancing, bound sharing). In Disolver, the end-user can define constraint-based knowledge sharing policies by adding new constraints. This second modeling can be linked to the constraint-based formulation of the problem to control the knowledge sharing according to the evolution of some problem components. For instance, the current value of the objective function can be used to allow answers to incoming load-balancing requests when the quality of the current subtree is perceived as good, etc. Interestingly, since the control of the knowledge sharing policies is made through classical constraints, it is automatically performed by the constraint propagation engine. We can see this as a dynamic adjustment of knowledge sharing activities, and assimilate it to model (learned clauses) and parameters (selection heuristics) change.

$$Solver : ((EncCh|Par)|Solv^*)^*$$

Control-based Clause Sharing in Parallel SAT Solving: Conflict driven clause learning, one of the most important component of modern DPLL, is crucial to the performance of parallel SAT solvers. Indeed, this mechanism allows clause sharing between multiple processing units working on related (sub-)problems. However, without limitation, sharing clauses might lead to an exponential blow up in communication or to the sharing of irrelevant clauses. In [52], the authors propose new innovative policies to dynamically select shared clauses in parallel solvers. The first policy controls the overall number of exchanged clauses whereas the second one additionally exploits the relevance or quality of the clauses. This dynamic adaptation mechanism allows to reinforce/reduce the cooperation between different solvers which are working on the same SAT instance.

$$Solver : (Par|Solv^*)^*$$

These case studies led us to consider some open questions for the development of these future solvers.

4 Challenges and Opportunities

In the following, we discuss some of the main challenges that must be taken into account if one wants to define and implement a solver based on the previously described architecture.

4.1 Performance Evaluation

There exists an optimal search strategy for a particular problem. However, determining such strategy could require much more computational power than solving the problem at hand. One possible way to assess the performance of AS systems is to run them on artificial problems where the optimal strategy is well known and to see if their adaptive mechanisms are able to build a strategy close to the optimal. One typical example is the mix "n-queens + pigeon-hole" problem [13], where a naive strategy endlessly finds a solution to the first subproblem before failing on the second one. In contrast, more suitable heuristics are able to detect the difficulty of the second problem and focus on it early in the search process[1].

The efficiency of an AS system can also be measured as its ability to maintain the competitiveness of its search strategy in a changing environment. Here, the goal is more to assess the reaction-time of the system under changing settings rather

[1] Remark here that an optimal strategy would state the unsatisfiability of the problem from the analysis of the second subproblem.

than the ultimate quality of the produced strategies. This second level of efficiency can be measured through the dynamic addition of subproblems. With the previous example, the pigeon-hole subproblem could be introduced during the search and a good strategy would quickly focus on it. Optimal strategy and reaction time are important to assess the performance against a particular problem.

Combinatorial search is often used in well identified domains which generate similar instances. The performance of an AS system can therefore be assessed against a particular domain. For instance, supervised adaptation can benefit from these similarities to converge on an efficient strategy. Note that supervised adaptation faces the problem of instance characterization. A perfect characterization is able to detect the repetition of an instance while a partial one is just able to detect the repetition of instance characteristics [49, 50]. This second case is more realistic both spatially, and technically: we can hardly imagine production scenarios where an end-user would solve the same problem multiple times. However, it is clear for us that ideally, an AS solver should be better at solving the same instance for the second time.

A major challenge associated to AS is that classical tools for algorithm analysis typically provide weak support for understanding the performance of autonomous algorithms. This is because autonomous algorithms exhibit a complex behavior that is not often amenable to a worst-case/average case analysis. Instead, autonomous algorithms should be considered as full-fledged complex systems and studied as such.

4.2 Continuous Search

Continuous computation addresses the issue not of finding the best (boundedly optimal) use of time in solving a given problem, but the best use of idle computational resources between bouts of problem solving. This approach broadens the definition of a "problem" to include not just individual instances, but the class of challenges that a given computational system is expected to face its lifetime. Eventually, the end of the current search is just another event for the AS system. As an effect, the priority of its long lasting self-improving task is raised and the task becomes foreground. The latest resolution is here to enrich the knowledge of the system and is eventually exploited during this new task. We can envision a wide range of actions that can be overtaken by the search algorithm while it is idle, in order to improve its solving strategy:

- Analyzing the strategies that have succeeded and failed during the last runs.
- Performing costly machine learning techniques in order to improve a supervised tuning method.
- Using knowledge compilation techniques in order to compile new deduction rules, or new patterns that were detected in the recently solved problems and that can prove useful for future problems of the same application area.

- Exchange gained knowledge with similar AS systems, e.g., features-based prediction function.

In fact, such a continuous system would include a self-adaptive strategy during the solving process while it could switch to a supervised controller while waiting for another problem instance. This architecture would allow it to react dynamically to incoming event during solving and to exploit the knowledge acquired through its successive experiences.

The performance evaluation of an AS to be able to work in continuous search mode is also an important problem which is highly related to the arrival rate and to the quality of new problem instances. Here quality corresponds on how good the instances are for the AS to gain important knowledge on the whole problem class.

4.3 Automated Generation of Components

As previously cited, some studies have explored the possibility to automatically generate components of the algorithm, for instance in order to discover heuristics as in [37] or to manage a set of operators that are generated during the run as in [73]. Starting from these preliminary works, one could look one step forward and try to automatize the full algorithmic design process. For instance, the specification of the problem could be used to automatically generate suitable neighborhood for local search operators [69]. Indeed, one could detect that the problem corresponds to a permutation encoding and then define a swap-based neighborhood. If the domain of the variable is binary, then one could use a flip based neighborhood. Once the skeleton of the operators has been defined, it could be interesting to benefit from approaches that could allow the solver to build new operators, such a genetic programming [65]. Indeed, there already exist some tools to facilitate the design of local search based algorithms [53] or for evolutionary algorithms [22], but they appear more as library for developers than integrated autonomous solutions for users.

5 Conclusion

In this chapter, we have proposed a taxonomy of search processes with respect to their computation characteristics. To this end, we have presented the general basic concepts of a solver architecture: the basic components of a solver and its configurations. We have then identified autonomous solvers as solvers that can control their solving process, either by self adaptation (internal process) or by supervised adaptation (external process).

We have proposed a rule-based characterization of autonomous solvers: the idea is to formalize solvers adaptations and modifications with some computation rules that describe solver transformation. Using our formalism, we could then classify,

characterize, and identify in the scope of autonomous search some representative existing solvers by outlining their global mechanism.

Our description framework allows us to handle solving techniques:

- Of various and different types: either complete, incomplete, or hybrid,
- Based on different computation paradigms: sequential, distributed, or parallel
- Dedicated to different problem families: CSP, SAT, optimization.

This work was also an attempt to highlight the links and similarities between different communities that aim at building such autonomous solvers and that may benefit from more exchanges and more collaborative approaches (including constraint programming, SAT, machine learning, numerical optimization, clustering, etc.).

We have identified the notion of control in autonomous constraint solvers and two main techniques for achieving it: control by supervised adaptation and control by self-adaptation, depending on the level of interaction between the solver, its environment, and the control itself. These two control management approaches are indeed complementary. Moreover, they open new challenges for the design of more autonomous search systems that would run continuously, alternating (or combining or executing in parallel), solving and self-improving phases.

Of course, an important remaining issue is evaluating performances of such systems with respect to or compared to classical criteria, used in solver competitions for instance. We think that the performance evaluation of an autonomous search may actually focus on three points:

- Show that an autonomous search can (re)discover the best known or approximate a very good strategy for a specific problem
- Show the ability of an autonomous search to adapt itself to a changing environment
- Show that an autonomous search could adapt itself and converge to an efficient strategy for a class of problems

One might consider the development of new types of autonomous search systems, by considering missing ones, for instance, the rule-based formulation presented here could be used here to identify new patterns that no solver matches. An other future direction is also to focus on general meta-level control of the adaptation strategy: this could be seen as the solving of an optimization problem whose solutions would contain the optimal configuration of the autonomous search system. As described in Sect. 4.2, continuous search is also a challenge: how to continuously adapt the solver for the current task while preparing it and improving it as well for the future tasks. Last but not least, we think that the automated generation of solver components should further develop and evolve.

Acknowledgements The authors would like to thank warmly Michela Milano and Pascal Van Hentenryck for their invitation to contribute to this book and the reviewers of this chapter for their helpful comments.

References

1. Applegate D, Bixby R, Chvatal V, Cook W (2007) The traveling salesman problem: a computational study (Princeton Series in Applied Mathematics). Princeton University Press, Princeton
2. Aarts E, Lenstra JK (eds) (2003) Local search in combinatorial optimization. Princeton University Press, Princeton
3. Apt K (2003) Principles of constraint programming. Cambridge University Press, Cambridge
4. Battiti R, Brunato M (eds) (2008) Learning and intelligent optimization second international conference, LION 2007 II, selected papers. Lecture Notes in Computer Science, vol 5313. Springer, Berlin
5. Battiti R, Brunato M (2010) Handbook of Metaheuristics, 2nd edn. chapter Reactive search optimization: learning while optimizing. Springer
6. Battiti R, Brunato M, Mascia F (2007) Reactive search and intelligent optimization. Technical report, Dipartimento di Informatica e Telecomunicazioni, Univerita di Tranto, Italy
7. Battiti R, Brunato M, Mascia F (2008) Reactive search and intelligent optimization. Operations research/computer science interfaces, vol 45. Springer, Heidelberg
8. Bader-El-Den M, Poli R (2008) Generating sat local-search heuristics using a gp hyper-heuristic framework, artificial evolution. In: 8th International conference, Evolution Artificielle, EA 2007. Revised selected papers. Lecture notes in computer science, vol 4926. Springer, Berlin, pp 37–49
9. Benhamou F, Granvilliers L (2006) Continuous and interval constraints. In: Rossi F, van Beek P, Walsh T (eds) Handbook of constraint programming, chapter 16. Elsevier, Amsterdam
10. Burke EK, Kendall G, Newall J, Hart E, Ross P, Schulenburg S (2003) Handbook of meta-heuristics, chapter Hyper-heuristics: an emerging direction in modern search technology. Kluwer, Dordrecht, pp 457–474
11. Burke EK, Hyde M, Kendall G, Ochoa G, Ozcan E, Qu R (2009) A survey of hyper-heuristics. Technical Report No. NOTTCS-TR-SUB-0906241418-2747, School of Computer Science and Information Technology, University of Nottingham, Computer Science
12. Burke EK, Hyde M, Kendall G, Ochoa G, Ozcan E, Woodward J (2010) Handbook of meta-heuristics, chapter A classification of hyper-heuristics approaches, Springer
13. Boussemart F, Hemery F, Lecoutre C, Sais L (2004) Boosting systematic search by weighting constraints. In: López de Mántaras R, Saitta L (eds) Proceedings of the 16th European conference on artificial intelligence, ECAI'2004. IOS Press, Amsterdam, pp 146–150
14. Biere A, Heule M, van Maaren H, Walsh T (eds) (2009) Handbook of satisfiability. Frontiers in artificial intelligence and applications, vol 185. IOS Press, Amsterdam
15. Bordeaux L, Hamadi Y, Zhang L (2006) Propositional satisfiability and constraint programming: a comparative survey. ACM Comput Surv 9(2):135–196
16. Boyan J, Moore A, Kaelbling P (2000) Learning evaluation functions to improve optimization by local search. J Mach Learn Res 1:1–2000
17. Birattari M, Stützle T, Paquete L, Varrentrapp K (2002) A racing algorithm for configuring metaheuristics. In: GECCO '02: Proceedings of the genetic and evolutionary computation conference. Morgan Kaufmann, San Francisco, CA, pp 11–18
18. Battiti R, Tecchiolli G (1994) The reactive tabu search. INFORMS J Comput 6(2):126–140
19. Crispim J, Brandão J (2001) Reactive tabu search and variable neighbourhood descent applied to the vehicle routing problem with backhauls. In: Proceedings of the 4th metaheuristics international conference, Porto, MIC 2001, pp 631–636
20. Crowston W, Glover F, Thompson G, Trawick J (1963) Probabilistic and parametric learning combinations of local job shop scheduling rules. Technical report, ONR Research Memorandum No. 117, GSIA, Carnegie-Mellon University, Pittsburg, PA, USA
21. Cowling P, Kendall G, Soubeiga E (2002) Hyperheuristics: a tool for rapid prototyping in scheduling and optimisation. In: Applications of evolutionary computing, EvoWorkshops 2002: EvoCOP, EvoIASP, EvoSTIM/EvoPLAN. Lecture notes in computer science, vol 2279. Springer, London, pp 1–10

22. Cahon S, Melab N, Talbi E, Schoenauer M (2003) Paradiseo-based design of parallel and distributed evolutionary algorithms. In: Artificial evolution, 6th international conference, evolution artificielle, EA 2003. Lecture notes in computer science, vol 2936. Springer, Berlin, pp 216–228
23. Cowling P, Soubeiga E (2000) Neighborhood structures for personnel scheduling: a summit meeting scheduling problem (abstract). In: Burke EK, Erben W (eds) Proceedings of the 3rd international conference on the practice and theory of automated timetabling, Constance, Germany
24. De Jong K (2006) Evolutionary computation: a unified approach. The MIT Press, Cambridge, MA
25. De Jong K (2007) Parameter setting in EAs: a 30 year perspective. In: Lobo F, Lima C, Michalewicz Z (eds) Parameter setting in evolutionary algorithms. Studies in computational intelligence, vol 54. Springer, Berlin, pp 1–18
26. Dechter R (2003) Constraint processing. Morgan Kaufmann, San Francisco, CA
27. Epstein S, Freuder E, Wallace R, Morozov A, Samuels B (2002) The adaptive constraint engine. In: Principles and practice of constraint programming – CP 2002, 8th international conference. Lecture notes in computer science, vol 2470. Springer, London, pp 525–542
28. Epstein S, Freuder E, Wallace R (2005) Learning to support constraint programmers. Comput Intell 21(4):336–371
29. Eiben AE, Hinterding R, Michalewicz Z (1999) Parameter control in evolutionary algorithms. IEEE Trans Evol Comput 3(2):124–141
30. Eén N, Mishchenko A, Sörensson N (2007) Applying logic synthesis for speeding up sat. In: Theory and applications of satisfiability testing – SAT 2007. Lecture notes in computer science, vol 4501. Springer, Heidelberg, pp 272–286
31. Eiben AE, Michalewicz Z, Schoenauer M, Smith JE (2007) Parameter control in evolutionary algorithms. In: Lobo F, Lima C, Michalewicz Z (eds) Parameter setting in evolutionary algorithms. Studies in computational intelligence, vol 54. Springer, Berlin, pp 19–46
32. Eén N, Sörensson N (2003) An extensible sat-solver. In: Theory and applications of satisfiability testing, 6th international conference, SAT 2003. Lecture notes in computer science, vol 2919. Springer, Heidelberg, pp 502–518
33. Eiben A, Smith JE (2003) Introduction to evolutionary computing. Natural computing series. Springer, Heidelberg
34. Fruewirth T, Abdennadher S (2003) Essentials of constraint programming. Springer, Heidelberg
35. Fialho A, Da Costa L, Schoenauer M, Sebag M (2008) Extreme value based adaptive operator selection. In: Rudolph G et al (ed) Parallel problem solving from nature - PPSN X, 10th international conference. Lecture notes in computer science, vol 5199. Springer, Berlin, pp 175–184
36. Fisher H, Thompson L (1963) Industrial scheduling, chapter Probabilistic learning combinations of local job-shop scheduling rules. Prentice Hall, Englewood Cliffs, NJ
37. Fukunaga A (2008) Automated discovery of local search heuristics for satisfiability testing. Evol Comput 16(1):31–61
38. Gebruers C, Guerri A, Hnich B, Milano M (2004) Making choices using structure at the instance level within a case based reasoning framework. In: Integration of AI and OR techniques in constraint programming for combinatorial optimization problems, First international conference, CPAIOR. Lecture notes in computer science, vol 3011. Springer, Berlin, pp 380–386
39. Goualard F, Jermann C (2008) A reinforcement learning approach to interval constraint propagation. Constraints 13(1–2):206–226
40. Glover F, Kochenberger G (2003) Handbook of metaheuristics (International series in operations research & management science). Springer, Berlin
41. Glover F, Laguna M (1997) Tabu search. Kluwer Academic, Dordrecht
42. Guerri A, Milano M (2004) Learning techniques for automatic algorithm portfolio selection. In: Proceedings of the 16th European conference on artificial intelligence, ECAI'2004. IOS Press, Amsterdam, pp 475–479

43. Goldberg D (1989) Genetic algorithms in search, optimization, and machine learning. Addison-Wesley Professional, Boston
44. Gagliolo M, Schmidhuber J (2008) Algorithm selection as a bandit problem with unbounded losses. Technical report, Tech. report IDSIA - 07 - 08
45. Gomes C, Selman B, Crato N, Kautz H (2000) Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. J Autom Reason 24(1/2):67–100
46. Hamadi Y (2003) Disolver : a distributed constraint solver. Technical Report MSR-TR-2003-91, Microsoft Research
47. Hansen N (2008) Adaptative encoding: how to render search coordinate system invariant. In: Parallel problem solving from nature – PPSN X, 10th international conference. Lecture notes in computer science, vol 5199. Springer, Berlin, pp 204–214
48. Van Hentenryck P (1989) Constraint satisfaction in logic programming. The MIT Press, Cambridge, MA
49. Hutter F, Hamadi Y (2005) Parameter adjustment based on performance prediction: towards an instance-aware problem solver. Technical Report MSR-TR-2005-125, Microsoft Research, Cambridge, UK
50. Hutter F, Hamadi Y, Hoos H, Brown KL (2006) Performance prediction and automated tuning of randomized and parametric algorithms. In: 12th International conference on principles and practice of constraint programming (CP'06)
51. Hutter F, Hoos H, Stützle T (2007) Automatic algorithm configuration based on local search. In: Proceedings of the 22nd conference on artifical intelligence (AAAI '07), pp 1152–1157
52. Hamadi Y, Jabbour S, Sais L (2009) Control-based clause sharing in parallel SAT solving. In: IJCAI 2009, Proceedings of the 21st international joint conference on artificial intelligence, pp 499–504
53. Van Hentenryck P, Michel L (2005) Constraint-based local search. The MIT Press, Cambridge, MA, USA
54. Hamadi Y, Monfroy E, Saubion F (2008) Special issue on autonomous search. In: Constraint programming letters, vol 4
55. Hamadi Y, Monfroy E, Saubion F (2008) What is autonomous search? Technical Report MSR-TR-2008-80, Microsoft Research
56. Holland J (1975) Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor, MI
57. Hoos H (1999) Sat-encodings, search space structure, and local search performance. In: Proceedings of the 16th international joint conference on artificial intelligence, IJCAI 99. Morgan Kaufmann, San Francisco, CA, pp 296–303
58. Hoos H (2002) An adaptive noise mechanism for walksat. In: AAAI/IAAI, pp 655–660
59. Hu B, Raidl G (2006) Variable neighborhood descent with self-adaptive neighborhood-ordering. In: Proceedings of the 7th EU meeting on adaptive, self-adaptive and multilevel metaheuristics
60. Hoos H, Stuetzle T (2004) Stochastic local search: foundations & applications (The Morgan Kaufmann Series in Artificial Intelligence). Morgan Kaufmann, San Francisco, CA
61. Hutter F (2009) Automating the configuration of algorithms for solving hard computational problems. PhD thesis, Department of Computer Science, University of British Columbia
62. Ingber L (1989) Very fast simulated re-annealing. Math Comput Model 12(8):967–973
63. Janikow C, Michalewicz Z (1991) An experimental comparison of binary and floating point representations in genetic algorithms. In: 4th International conference on genetic algorithms, pp 3136
64. Kjellström G (1991) On the efficiency of gaussian adaptation. J Optim Theory Appl 71(3):589–597
65. Koza J (1992) Genetic programming: on the programming of computers by means of natural selection. The MIT Press, Cambridge, MA
66. Kazarlis S, Petridis V (1998) Varying fitness functions in genetic algorithms: studying the rate of increase of the dynamic penalty terms. In: Parallel problem solving from nature – PPSN V, 5th international conference. Lecture notes in computer science, vol 1498, pp 211–220

67. Kramer O (2008) Self-adaptive heuristics for evolutionary computation. Springer, Berlin
68. Lobo F, Lima C, Michalewicz Z (eds) (2007) Parameter setting in evolutionary algorithms. In: Studies in computational intelligence, vol 54. Springer, Berlin
69. Monette J-N, Deville Y, Van Hentenryck P (2009) Operations research and cyber-infrastructure, chapter Aeon: synthesizing scheduling algorithms from high-level models. Springer, New York, pp 43–59
70. Maturana J, Fialho A, Saubion F, Schoenauer M, Sebag M (2009) Compass and dynamic multi-armed bandits for adaptive operator selection. In: Proceedings of IEEE Congress on evolutionary computation CEC, IEEE Press, Piscataway, NJ, pp 365–372
71. Mladenovic N, Hansen P (1997) Variable neighborhood search. Comput Oper Res 24(11):1097–1100
72. Michalewicz Z (1992) Genetic algorithms + data structures = evolution program. Artificial intelligence. Springer, Berlin
73. Maturana J, Lardeux F, Saubion F (2010) Autonomous operator management for evolutionnary algorithms. J Heuristics, Springer, pp 1–29
74. Morris P (1993) The breakout method for escaping from local minima. In: Proceedings of the 11th national conference on artificial intelligence (AAAI93). AAAI Press, Menlo Park, CA, pp 40–45
75. Marriott K, Stuckey P (1998) Programming with constraints: an introduction. The MIT Press, Cambridge, MA
76. Maturana J, Saubion F (2008) A compass to guide genetic algorithms. In: Rudolph G et al (ed) Parallel problem solving from nature – PPSN X, 10th international conference. Lecture notes in computer science, vol 5199. Springer, Berlin, pp 256–265
77. Mazure B, Sais L, Grégoire E (1997) Tabu search for sat. In: AAAI/IAAI, pp 281–285
78. Mazure B, Sais L, Grégoire E (1998) Boosting complete techniques thanks to local search methods. Ann Math Artif Intell 22(3-4):319–331
79. Monfroy E, Saubion F, Lambert T (2004) On hybridization of local search and constraint propagation. In: Logic programming, 20th international conference, ICLP 2004. Lecture notes in computer science, vol 3132. Springer, Berlin, pp 299–313
80. Nannen V, Eiben AE (2006) A method for parameter calibration and relevance estimation in evolutionary algorithms. In: Genetic and evolutionary computation conference, GECCO 2006, proceedings. ACM, New York, NY, pp 183–190
81. Nannen V, Eiben AE (2007) Relevance estimation and value calibration of evolutionary algorithm parameters. In: IJCAI 2007, proceedings of the 20th international joint conference on artificial intelligence, pp 975–980
82. Nannen V, Smit S, Eiben A (2008) Costs and benefits of tuning parameters of evolutionary algorithms. In: Parallel problem solving from nature – PPSN X, 10th international conference. Lecture notes in computer science, vol 5199. Springer, Berlin, pp 528–538
83. Pullan WJ, Hoos HH (2006) Dynamic local search for the maximum clique problem. J Artif Intell Res (JAIR) 25:159–185
84. Patterson D, Kautz H (2001) Auto-walksat: a self-tuning implementation of walksat. Electron Notes Discrete Math 9:360–368
85. Puchinger J, Raidl G (2008) Bringing order into the neighborhoods: relaxation guided variable neighborhood search. J Heuristics 14(5):457–472
86. Ringwelski G, Hamadi Y (2005) Boosting distributed constraint satisfaction. In: Principles and practice of constraint programming – CP 2005, 11th international conference. Lecture notes in computer science, vol 3709. Springer, Berlin, pp 549–562
87. Rice J (1975) The algorithm selection problem. Technical Report CSD-TR 152, Computer science department, Purdue University
88. Smit S, Eiben A (2009) Comparing parameter tuning methods for evolutionary algorithms. In: Proceedings of the 2009 IEEE Congress on evolutionary computation (CEC 2009). IEEE Press, Piscataway, NJ, pp 399–406
89. Selman B, Kautz H, Cohen B (1994) Noise strategies for improving local search. In: AAAI, pp 337–343

90. Smith-Miles K (2008) Cross-disciplinary perspectives on meta-learning for algorithm selection. ACM Comput Surv 41(1):1–25
91. Sywerda G (1989) Uniform crossover in genetic algorithms. In: Proceedings of the third international conference on genetic algorithms. Morgan Kaufmann, San Francisco, CA, pp 2–9
92. Thierens D (2005) An adaptive pursuit strategy for allocating operator probabilities. In: Beyer H-G (ed) Proc. GECCO'05. ACM, New York, NY, pp 1539–1546
93. Thierens D (2007) Adaptive strategies for operator allocation. In: Lobo FG, Lima CF, Michalewicz Z (eds) Parameter setting in evolutionary algorithms. Springer, Heidelberg, pp 77–90
94. Thornton J (2000) Constraint weighting for constraint satisfaction. PhD thesis, School of Computing and Information Technology, Griffith University, Brisbane, Australia
95. Tsang E (1993) Foundations of constraint satisfaction, 1st edn. Academic, London
96. Walsh T (2000) SAT v CSP. In: Proceedings of CP 2000. Lecture notes in computer science, vol 1894. Springer, Berlin, pp 441–456
97. Wong Y-Y, Lee K-H, Leung K-S, Ho C-W (2003) A novel approach in parameter adaptation and diversity maintenance for GAs. Soft Comput 7(8):506–515
98. Whitacre J, Tuan Pham Q, Sarker R (2006) Credit assignment in adaptive evolutionary algorithms. In: Genetic and evolutionary computation conference, GECCO 2006. ACM, New York, NY, pp 1353–1360
99. Whitacre J, Pham T, Sarker R (2006) Use of statistical outlier detection method in adaptive evolutionary algorithms. In: Proceedings of the genetic and evolutionary computation conference (GECCO). ACM, New York, NY, pp 1345–1352
100. Xu L, Hutter F, Hoos HH, Leyton-Brown K (2008) Satzilla: portfolio-based algorithm selection for sat. J Artif Intell Res 32:565–606
101. Yuan B, Gallagher M (2004) Statistical racing techniques for improved empirical evaluation of evolutionary algorithms. In: Yao X et al (ed) Parallel problem solving from nature – PPSN VIII, 8th international conference. Lecture notes in computer science, vol 3242. Springer, Berlin, pp 172–181
102. Yu-Hui Yeh F, Gallagher M (2005) An empirical study of hoeffding racing for model selection in k-nearest neighbor classification. In: Gallagher M, Hogan J, Maire F (eds) IDEAL. Lecture notes in computer science, vol 3578. Springer, Berlin, pp 220–227
103. Yuan B, Gallagher M (2007) Combining meta-eas and racing for difficult EA parameter tuning tasks. In: Lobo F, Lima C, Michalewicz Z (eds) Parameter setting in evolutionary algorithms. Studies in computational intelligence, vol 54. Springer, Berlin, pp 121–142

Software Tools Supporting Integration

Tallys Yunes

Abstract This chapter provides a brief survey of existing software tools that enable, facilitate, and/or support the integration of different optimization techniques. We focus on tools that have achieved a reasonable level of maturity and whose published results have demonstrated their effectiveness. The description of each tool is not intended to be comprehensive. We include references and links to detailed accounts of each tool for the interested reader, and we recommend that the reader consult the developers and/or vendors for the latest information about upgrades and improvements. Our purpose is to summarize the main features of each tool, highlighting what it can (or cannot) do, given the current version at the time of writing. We conclude the chapter with suggestions for future research directions.

1 Introduction

This chapter provides a brief survey of existing software tools that enable, facilitate, and/or support the integration of different optimization techniques. By *optimization techniques*, we mean linear programming (LP), mixed-integer linear programming (MILP), nonlinear programming (NLP), constraint programming (CP), local search (LS), large neighborhood search (LNS), and their derivatives. Integration can occur in many ways and at different levels. It ranges from a simple information exchange between two solvers implementing two versions of the same mathematical model, to a full-fledged synchronization of multiple methods that interact closely during an iterative search procedure. Decomposition methods such as branch-and-price (B&P) [12] and Benders decomposition [14] also lend themselves to effective integrated

T. Yunes (✉)
Department of Management Science, School of Business Administration,
University of Miami, Coral Gables, FL 33124-8237, USA
e-mail: tallys@miami.edu

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_12,
© Springer Science+Business Media, LLC 2011

approaches and are, therefore, also covered here. Due to space limitations, we were not able to include a comprehensive list of all the available tools. We focus, however, on those tools that have achieved a reasonable level of maturity and whose published results have demonstrated their effectiveness.

The tools typically come in two main categories: *high level* and *low level*. High-level tools are those that offer a high-level modeling language to the user and do not necessarily require computer programming at the level of languages such as C, C++, Java , or Prolog. Low-level tools are those that require the user to write low-level code to be able to take advantage of their integration abilities. This is usually done in one of two ways: writing code that includes calls to a library of functions or object classes (the tool's API), or writing code in a computer programming language that has been enriched with new declarations and commands that provide access to the underlying algorithms and solvers (e.g., extensions to the Prolog language). We decided, however, not to separate the tools into the above two groups. Because of the rapidly evolving nature of optimization software and the ongoing introduction of new features and interfaces, such a classification would soon be obsolete. Moreover, some tools are actually a mixture of both high- and low-level features. Therefore, we have opted to list the tools in plain alphabetical order.

The description of each tool focuses on its defining characteristics and does not attempt to list all of its facilities. We include references and links to detailed accounts of each software for the interested reader, and we recommend that the reader consult the developers and/or vendors of each tool for the latest information about upgrades and improvements. Our purpose is to summarize the main features of each tool, highlighting what it can (or cannot) do, given the current version at the time of writing. The contents of the following sections have been taken from published material about each tool, such as: user manuals, web sites, books, and research papers. We have tried, to the extent possible, to stay true to the software descriptions provided by the respective authors and developers. Changes to those descriptions have been kept to a minimum, and were made with the sole intent of improving clarity and removing biases and subjective statements. Finally, the code excerpts presented in this chapter are for illustration purposes only and do not intend to represent the best way of modeling any particular problem.

This chapter is organized as follows. Section 2 presents the main features that are currently available to facilitate the integration of optimization techniques. Descriptions of each tool appear in Sects. 3–11. For each tool, excluding those in Sect. 11, we include a *data sheet*, which is a table summarizing some of its characteristics for easy reference, such as: supported features, solvers and platforms, details on availability, references, and additional notes. When available, developers' names are listed in alphabetical order. Some tools also include a "coming soon" paragraph with new features that are currently under development. Section 12 concludes the chapter with final remarks and directions for future research.

2 Existing Features That Facilitate Integration

In this section, we provide a classification scheme for the different kinds of features and functionalities that can appear in a software system supporting the integration of different optimization techniques. The purpose of this classification is to create an organized hierarchy of features that can be easily referenced in the subsequent sections.

1. *User interface*: how the user interacts with (i.e., furnishes a model to) the tool
   a. Graphical interface/IDE available (assumes the tool has a high-level modeling language)
   b. Tool can read an input file with model description written in a high-level language
   c. User writes code to use the tool's API
2. *Solver support*: kinds of solvers that are part of (or can interface with) the tool
   a. LP, MILP
   b. NLP
   c. CP
   d. SAT
3. *Relaxation support*: kinds of model relaxations that are supported by the tool
   a. Linear
   b. Nonlinear
   c. Lagrangian
   d. Domain store (from a CP solver)
4. *Inference mechanisms*: forms of inference supported by the tool
   a. Preprocessing (upfront elimination/tightening of variables and constraints)
   b. Cutting planes
   c. Domain filtering/reduction
   d. Logical resolution mechanisms (as in satisfiability problems)
5. *Search mechanisms*: forms of search supported by the tool
   a. Complete branching (e.g., tree search, branch-and-bound)
   b. Incomplete branching (e.g., LDS)
   c. Simple local search (no metaheuristic)
   d. Metaheuristics (e.g., tabu search, simulated annealing)
6. *Decomposition mechanisms*: forms of decomposition supported by the tool
   a. Dantzig–Wolfe
   b. Column generation/B&P (master problem and pricing problem can potentially be solved by different techniques)
   c. Benders

7. *Search control*: how much control the user has over the search mechanisms
   a. High-level (via modeling language), declarative (i.e., fixed forms of control)
   b. High-level, imperative (detailed and flexible control)
   c. Low-level (coding through an API)

We will refer back to this classification scheme in the data sheet of each tool. For instance, to indicate that a certain tool has an IDE, interfaces with a SAT solver, and supports Benders decomposition, we use the codes 1a, 2d, and 6c.

The classification scheme above refers to the support for integration only. Hence, if a given tool accepts LP files as input (a kind of high-level input), but requires low-level coding for access to its integration features, its user interface will be classified as 1c, rather than 1b.

3 BARON

BARON [57] was developed with the main purpose to find global solutions to nonlinear and mixed-integer nonlinear programs. It is based on the idea of *branch-and-reduce*, which can be seen as a modification of branch-and-bound in which a variety of logical inferences made during the search tree exploration help reduce variable domains and tighten the problem relaxation. Two of the main domain reduction techniques used by BARON are:

Optimality-based range reduction: Let $x_j - x_j^U \leq 0$ be a constraint that is active at the optimal solution of the relaxation of the current search node (i.e., variable x_j is at its upper bound), and let $\lambda_j > 0$ be the corresponding Lagrange multiplier. If we know an upper bound U and a lower bound L on the value of the optimal solution, a valid lower bound for x_j can be calculated as $x_j^U - (U - L)/\lambda_j$. (An analogous procedure can be used to calculate an improved upper bound for x_j.). If variable x_j is at neither of its bounds in the solution of the relaxation, we can *probe* its bounds (temporarily fix x_j at its bounds, and resolve the relaxation) to obtain possibly better bounds. This technique extends to arbitrary constraints of the type $g_i(x) \leq 0$.

Feasibility-based range reduction: This is a process that generates constraints that cut off infeasible parts of the search space. For instance, given a constraint $\sum_{j=1}^{n} a_{ij} x_j \leq b_i$, one of the constraints below is valid for a pair (i, h) with $a_{ih} \neq 0$:

$$x_h \leq \frac{1}{a_{ih}} \left(b_i - \sum_{j \neq h} \min \left\{ a_{ij} x_j^U, a_{ij} x_j^L \right\} \right), \quad a_{ih} > 0 \tag{1}$$

$$x_h \geq \frac{1}{a_{ih}} \left(b_i - \sum_{j \neq h} \min \left\{ a_{ij} x_j^U, a_{ij} x_j^L \right\} \right), \quad a_{ih} < 0 \tag{2}$$

The above inequalities can be interpreted as an approximation to the optimal bound tightening procedure that involves solving the $2n$ linear programs

$$\left\{ \min \ \pm x_k \text{ s.t.} \sum_{j=1}^{n} a_{ij} x_j \leq b_i, \ i = 1, \ldots, m \right\}, \ k = 1, \ldots, n \tag{3}$$

BARON makes extensive use of (1) and (2) throughout the search, along with a limited use of (3), mostly at the root node, for selected variables. Nonlinear analogues of (1) and (2) are also utilized throughout the search.

For a given problem, BARON's global optimization strategy integrates conventional branch-and-bound with a wide variety of range reduction tests. These tests are applied to every subproblem of the search tree in pre- and postprocessing steps to contract the search space and reduce the relaxation gap. Many of the reduction tests are based on duality and are applied when the relaxation is convex and solved by an algorithm that provides the dual, in addition to the primal, solution of the relaxed problem. Another crucial component of the software is the implementation of heuristic techniques for the approximate solution of optimization problems that yield improved bounds for the problem variables. Finally, the algorithm incorporates a number of compound branching schemes that accelerate convergence of standard branching strategies.

Typically, the relaxed problem is constructed using factorable programming techniques (see [42, 43]), so that the relaxations are exact at the variable bounds. Therefore, the tightness of the relaxation depends on the tightness of the variable bounds. BARON implements the construction of underestimators for many nonconvex terms such as: bilinearities ($x_i x_j$), linear fractional terms (x_i / x_j), univariate terms such as x^n with n odd, etc. Note that outer approximations do not need to be linear.

BARON comes in the form of a callable library. The software has a core component which can be used to solve any global optimization problem for which the user supplies problem-specific subroutines, primarily for lower and upper bounding. In this way, the core system is capable of solving very general problems. In addition to the general purpose core, the BARON library also provides ready-to-use specialized modules covering several classes of problems. These modules work in conjunction with the core component and require no coding on the part of the user. Some of the available problem classes are: separable concave quadratic programming, separable concave programming, problems with power economies of scale (Cobb–Douglas functions), fixed-charge programming, fractional programming, univariate polynomial programming, linear and general linear multiplicative programming, indefinite quadratic programming, mixed integer linear programming, and mixed integer semi-definite programming.

BARON includes a factorable NLP module, which is the most general of the supplied modules that do not require any coding from the user. The input to this module is through BARON's parser and it can solve fairly general nonlinear programs where the objective and constraints are factorable functions (i.e., recursive

Table 1 BARON data sheet

Feature	Description
Tool name	BARON: Branch-and-Reduce Optimization Navigator
User interface	1b, 1c
Solver support	2a, 2b
Relaxation support	3a, 3b
Inference mechanisms	4a, 4b, 4c
Search mechanisms	5a, 5c
Decomposition mechanisms	N/A
Search control	7a + some added flexibility through GAMS, 7c
Written in	C and Fortran
Supported platforms	Linux, Windows, AIX
Main developers	Nikolaos Sahinidis and Mohit Tawarmalani
Availability	Available under the AIMMS and GAMS modeling languages. See http://archimedes.cheme.cmu.edu/baron/baron.html.
References	[57]
Notes	Also available on the NEOS server.

compositions of sums and products of functions of single variables). Most functions of several variables used in nonlinear optimization are factorable and can be easily brought into separable form. The types of functions currently allowed in this module include $\exp(x)$, $\ln(x)$, x^α for $\alpha \in \mathbb{R}$, and β^x for $\beta \in \mathbb{R}$.

The theoretical and empirical work that is embodied in BARON earned Nikolaos Sahinidis and Mohit Tawarmalani the 2004 INFORMS Computing Society Prize and the 2006 Beale–Orchard–Hays Prize from the Mathematical Programming Society. BARON is customarily used as a benchmark code when it comes to solving global optimization problems.

Most of BARON's specialized modules require the use of an LP solver. An NLP solver is optional, but frequently beneficial. The current release (8.1) can work with many different LP, NLP, and SDP solvers such as OSL, CPLEX, CONOPT, MINOS, SDPA, and SNOPT.

BARON's data sheet appears in Table 1.

Coming soon to BARON: The ability to solve MILP relaxations for problems that involve a combinatorial component.

4 Comet

One of the main innovations of Comet [59] is constraint-based local search (CBLS); a computational paradigm based on the idea of specifying local search algorithms as two components: a high-level model describing the application in terms of constraints, constraint combinators, and objective functions, and a search procedure expressed in terms of the model at a high level of abstraction. CBLS makes it possible to build local search algorithms compositionally, to separate modeling from search, to promote reusability across many applications, and to exploit problem

structure to achieve high performance. The computational model underlying CBLS uses constraints and objectives to drive the search procedure toward feasible or high-quality solutions. Constraints incrementally maintain their violations, and objectives maintain their values. Moreover, constraints and objectives are differentiable and can be queried to evaluate the effect of moves on their violations and values.

Comet is an object-oriented language featuring both a constraint-based local search engine and a constraint-programming solver. Comet models for these two paradigms are essentially similar and are expressed in modeling languages featuring logical and cardinality constraints, global/combinatorial constraints, numerical (possibly nonlinear) constraints, and vertical extensions for specific application areas (e.g., scheduling abstractions). The search components are also expressed in a rich language for controlling graph- and tree-search explorations. The constraint-programming solver also views combinatorial optimization as the combination of model and search components. It provides the traditional expressiveness of constraint programming with a language for expressing search algorithms.

Comet also features high-level abstractions for parallel and distributed computing, based on parallel loops, interruptions, and work stealing. In addition, it provides a declarative language for specifying model-driven visualizations of various aspects of an optimization algorithm. Finally, being an open language, Comet allows programmers to add their own constraints and objectives, as well as their own control abstractions.

To illustrate some of Comet's modeling constructs, let us consider the well-known n-queens problem. We are given an $n \times n$ chessboard. The objective is to place n queens on the chessboard so that no two queens can attack each other (i.e., no two queens lie on the same row, column or diagonal). Figure 1 shows a

```
01. import cotls;
02. Solver<LS> m();
03. int n = 16;
04. range Size = 1..n;
05. UniformDistribution distr(Size);
06. var{int} queen[Size](m,Size) := distr.get();
07.
08. ConstraintSystem<LS> S(m);
09. S.post(alldifferent(queen));
10. S.post(alldifferent(all(i in Size) queen[i] + i));
11. S.post(alldifferent(all(i in Size) queen[i] - i));
12. m.close();
13.
14. int iter = 0;
15. while (S.violations() > 0 && iter < 50*n) {
16.    selectMax(q in Size)(S.violations(queen[q]))
17.       selectMin(v in Size)(S.getAssignDelta(queen[q],v))
18.          queen[q] := v;
19.    iter++;
20. }
```

Fig. 1 Comet model for the n-queens problem

Comet model for the n-queens problem using local search. The algorithm is based on the min-conflict heuristic [45] but uses a greedy heuristic to select the variables. Because no two queens can be placed on the same column, we can assume that the i-th queen is placed on the i-th column, and variable `queen[i]` represents the row assigned to the i-th queen. We start by including the local solver library and the local solver (lines 01 and 02). Line 06 declares the incremental variables (`queen`) and performs an initial random assignment of queens to rows. The three all different constraints in lines 09–11 check for violations of the row and diagonal constraints. Then, at each iteration, we choose the queen violating the most constraints (line 16) and move it to a position (row) that minimizes its violations (line 17). These steps are repeated until a feasible solution is found (the constraint system `S` has no violations) or the number of iterations becomes greater than $50n$ (line 15). The expression `S.getAssignDelta(queen[q],v)` queries the constraint system to find out the impact of assigning value `v` to queen `q`. This query is performed in constant time because of the invariants maintained in each of the constraints. The assignment `queen[q]:= v` in line 18 induces a propagation step that recomputes the violations of all constraints.

More recently, Comet has added support for CP, LP, and MIP solvers, as well as the ability to do column (variable) generation during search. To access the above solvers, the user can write statements like `Solver<CP> m();`, `Solver<LP> m();` or `Solver<MIP> m();` instead of `Solver<LS>`. Variable creation and constraint posting are modified accordingly. To create finite domain CP variables for the n-queens problem, we can write `var<CP>int queen[Size](m,Size).`

The linear solvers supported by the current Comet release (1.2) are LP_SOLVE and the COIN-OR LP solver CLP.

Comet's data sheet appears in Table 2.

Coming soon to Comet: A uniform GUI/Visualization tool and CP set variables.

Table 2 Comet data sheet

Feature	Description
Tool name	Comet
User interface	1b, 1c
Solver support	2a, 2c
Relaxation support	3a, 3d
Inference mechanisms	4a, 4c
Search mechanisms	5a, 5b, 5c, 5d
Decomposition mechanisms	6b
Search control	7a, 7b, 7c
Written in	C++ and Assembly
Supported platforms	Linux, Mac OS X, Windows XP
Developers	Laurent Michel and Pascal Van Hentenryck
Availability	Free for noncommercial use at http://www.dynadec.com. See also http://www.comet-online.org.
References	[59]
Notes	Also includes an ODBC module to grant access to databases.

5 ECL^{i}PS^e

ECL^{i}PS^e [7, 61] is a Prolog-based system whose aim is to serve as a platform for integrating various Logic Programming extensions, in particular, Constraint Logic Programming (CLP). The kernel of ECL^{i}PS^e is an efficient implementation of standard (Edinburgh-like) Prolog as described in basic Prolog texts [20]. It is built around an incremental compiler which compiles the ECL^{i}PS^e source code into WAM-like code [62], and an emulator of this abstract code. The ECL^{i}PS^e logic programming system was originally an integration of ECRC's SEPIA [44], Megalog [16], and parts of the CHIP [24] system. It was then further developed into a Constraint Logic Programming system with a focus on hybrid problem solving and solver integration. ECL^{i}PS^e is now an open-source project, with the support of Cisco Systems.

There are two ways of running ECL^{i}PS^e programs. The first is using an interactive graphical user interface to the ECL^{i}PS^e compiler and system, which is called *tkeclipse*. The second is through a more traditional command-line interface. The graphical user interface provides many useful features such as a tracer (for debugging), a scratch-pad (to experiment with small pieces of code), statistics (e.g., CPU and memory usage), predicate library help, etc.

ECL^{i}PS^e has introduced many specific language constructs to overcome some of the main deficiencies of Prolog, such as: the ability to use named structures with field names, loop/iterator constructs, a multidimensional array notation, pattern matching for clauses, soft cuts, a string data type, etc.

ECL^{i}PS^e allows the use of different constraint solvers in combination. The different solvers may share variables and constraints. This can be done by combining the `eplex` and `ic` solver libraries of ECL^{i}PS^e. The `eplex` library gives access to LP and MIP solvers (linear numeric constraints and integrality constraints), and the `ic` library implements typical constraint propagation algorithms (finite domain constraints and interval constraints). In a hybrid model, the `ic` solver communicates new tightened bounds to the `eplex` solver. These tightened bounds have typically been deduced from nonlinear constraints (e.g., global constraints) and thus the linear solver benefits from information which would not otherwise have been available to it. On the other hand, the `eplex` solver often detects inconsistencies which would not have been detected by the `ic` solver. Moreover, it returns a bound on the optimization function that can be used by the `ic` constraints. Finally, the optimal solution returned by `eplex` to the "relaxed" problem comprising just the linear constraints can be used in a search heuristic that can focus the `ic` solver on the most promising parts of the search space. Other useful libraries are `colgen` (support for column generation) and `repair` (helps propagate solutions generated by a linear solver to other variables handled by the domain solver).

To illustrate the ECL^{i}PS^e syntax, let us consider a simple example. We are given a time instant t, and three tasks such that the running time of tasks 1 and 2 are, respectively, 3 and 5 time units. We want to find start times for each task such

```
01. :- lib(ic), lib(eplex).
02. overlap(S,D,T,B) :- ic:(B #= ((T $>= S)and(T $=< S+D-1))).
03. hybrid(Time,[S1,S2,S3],Obj) :-
04.     ic:(Obj $:: -1.0Inf..1.0Inf),
05.     ic:([S1,S2,S3]::1..20),
06.     overlap(S1,3,Time,B1), overlap(S2,5,Time,B2),
07.     ic:(B1+B2 #= 1),
08.     eplex:(S1+3 $=< S3), eplex:(S2+5 $=< S3),
09.     eplex:eplex_solver_setup(min(S3),Obj,
            [sync_bounds(yes)],[ic:min,ic:max]),
10.     labeling([B1,B2,S1,S2]).
```

Fig. 2 ECL^{i}PS^e model illustrating solver integration

that two constraints are satisfied: exactly one of tasks 1 and 2 is running at time t, and both tasks 1 and 2 precede task 3. The corresponding ECL^{i}PS^e code appears in Fig. 2. Line 01 loads the `ic` and `eplex` libraries. Line 02 defines a predicate that associates a boolean variable `B` with whether a task with start time `S` and duration `D` is running at time `T`. Lines 04 and 05 give initial domains to the `Obj` variable, which stores objective function bounds, and to the start time variables `S1`, `S2`, `S3`. To enforce the first constraint, we state in line 07 that exactly one of the boolean variables `B1` and `B2` has to be equal to 1. Note that the prefix `ic:` indicates that all of the above constraints are posted to the `ic` solver. We ask the `eplex` solver to handle the second constraint in line 08, while line 09 sets up the objective (minimize `S3`), and passes `ic` bounds to the linear solver before the problem is solved (`[sync_bounds(yes)]`). The statement `[ic:min,ic:max]` triggers the LP solver in case of bound changes on the `ic` side. We search for a solution by labeling variables in line 10. To send a constraint, say $x + 2 \geq y$, to both the `eplex` and `ic` solvers at once, we could write `[eplex,ic]:(X + 2 $>= Y)`.

Three decomposition techniques that are amenable to hybridization are column generation, Benders decomposition, and Lagrangian relaxation. All three have been implemented in ECL^{i}PS^e and used to solve large problems.

The current release of ECL^{i}PS^e (6.0) can interface with the following LP/MIP solvers: CPLEX, Xpress-MP (versions developed until 2005), and the COIN-OR LP solver CLP.

ECL^{i}PS^e's data sheet appears in Table 3.

Coming soon to ECL^{i}PS^e: Integration with the Gecode [56] constraint solver library; improved propagator support; new and more efficient global constraints such as cardinality, 2-D alldifferent, sequence, etc.; search annotations and search tree display. See http://www.eclipse-clp.org/reports/roadmap.html for a road map of future ECL^{i}PS^e releases.

Table 3 ECL^{i}PS^e data sheet

Feature	Description
Tool name	ECL^{i}PS^e
User interface	1a (Saros prototype under development), 1b, 1c
Solver support	2a, 2c
Relaxation support	3a, 3c, 3d
Inference mechanisms	4c
Search mechanisms	5a, 5b, 5c, 5d
Decomposition mechanisms	6b, 6c
Search control	7a, 7b, 7c
Written in	ECL^{i}PS^e/Prolog
Supported platforms	Linux, Mac OS X, Solaris, Windows
Developers	See list at http://sourceforge.net/projects/eclipse-clp
Availability	Freely available at http://www.eclipseclp.org
References	[7, 61]

6 G12

The G12 project [55] started by National ICT Australia (NICTA) seeks to develop a software platform for solving large-scale industrial combinatorial optimization problems. The core design involves three languages: Zinc, Cadmium, and Mercury (group 12 of the periodic table). Zinc [13,28] is a declarative modeling language for expressing problems independently of any solving methodology. Cadmium [25] is a mapping language for mapping Zinc models to underlying solvers and/or search strategies, including hybrid approaches. Finally, the existing Mercury language [54] will be extended as a language for building extensible and hybridizable solvers. The same Zinc model, used with different Cadmium mappings, allows the user to experiment with different complete, local, or hybrid search approaches for the same problem. Cadmium mappings also enable the application of preprocessing steps (e.g., bounds tightening) and transformation steps (e.g., linearization of subproblems) to the original Zinc model. Some of the features provided by Zinc are: mathematical notation-like syntax (overloading, iterations, sets, arrays), expressive constraints, support for different kinds of problems (including preferences, i.e., soft constraints), interface to hybrid solvers (like lazy FD), separation of data from model, high-level data structures and data encapsulation, extensibility (user defined functions), reliability (type checking, assertions), and *annotations* which allow non-declarative information (such as search strategies) and solver-specific information (such as variable representations) to be layered on top of the declarative model. Another G12 component is MiniZinc [47], which is a subset of Zinc that provides many modeling capabilities while being easier to implement. The target language of MiniZinc is FlatZinc, a low-level solver input language that offers an easier way for solver writers to provide reasonable MiniZinc support.

To illustrate the Zinc language, let us consider the following trucking example from [50]. We are given `T` trucks, each having a cost `Cost[t]` and an amount of

```
01. int: P;                          type Per = 1..P;
02. int: T;                          type Trucks = 1..T;
03. array[Per]    of int: Dem;       array[Trucks] of int: Cost;
04. array[Trucks] of int: Load;      array[Trucks] of int: K;
05. array[Trucks] of int: L;         array[Trucks] of int: U;
06. array[Per]    of var set of Trucks: x;

07. constraint forall(p in Per)(
      sum_set(x[p],Load) >= Dem[p]);
08. constraint forall(t in Trucks)(
      sequence([bool2int(t in x[p])|p in Per],L[t],U[t],K[t]));

09. solve minimize sum(p in Per)(sum_set(x[p], Cost));
```

Fig. 3 Zinc model for the trucking

material it can load `Load[t]`. We are also given `P` time periods. In each time period `p`, a given demand of material (`Dem[p]`) has to be shipped. Each truck `t` also has constraints on usage: in each consecutive `K[t]` time periods, it must be used at least `L[t]` and at most `U[t]` times. The Zinc model of this problem appears in Fig. 3. In each time period `p`, we need to choose which trucks to use in order to ship enough material and satisfy the usage limits. Hence, variable `x[p]` represents the subset of trucks used in period `p`. In line 07, the `sum_set`(S, f) function returns $\sum_{e \in S} f(e)$; in line 08, `sequence`$([y_1, \ldots, y_n], \ell, u, k)$ constrains the sum of each subsequence of y variables of length k to be between ℓ and u. As it stands, this model is directly executable in a finite domain solver which supports set variables. In Zinc, we can control the search by adding an annotation on the `solve` statement. For example,

```
solve :: set_search(x,"first_fail","indomain","complete")
   minimize sum(p in Per)(sum_set(x[p], Cost));
```

indicates that we label the `x` variables with smallest domain first (`first_fail`) by first trying to exclude an unknown element of the set and then including it (`indomain`) in a complete search.

To use Dantzig–Wolfe decomposition and column generation on the trucking model, we need to annotate the model to explain what parts define the subproblems, which solver is to be used for each subproblem, and which solver is to be used for the master problem. The annotations would look as follows

```
array[Per] of var set of Trucks: x :: colgen_var;

constraint forall(p in Per)(
sum_set(x[p],Load) >= Dem[p] ::
        colgen_subproblem_constraint(p,"mip"));

solve :: colgen_solver("lp") :: lp_bb(x, most_frac, std_split)
minimize sum(p in Per)(sum_set(x[p], Cost));
```

which means that variables `x` will be used in column generation. The subproblems numbered 1 through `P` are defined in terms of their constraints and their solver (`mip`). Finally, the solver for the master problem and the search specification

Table 4 G12 data sheet

Feature	Description
Tool name	The G12 Project
User interface	1b, 1c
Solver support	2a, 2c, 2d
Relaxation support	3a, 3d
Inference mechanisms	4a, 4b, 4c, 4d
Search mechanisms	5a, 5b, 5c, 5d
Decomposition mechanisms	6a, 6b
Search control	7a, 7b, 7c
Written in	Mercury and C++
Supported platforms	Linux, Mac OS X, Windows
Developers	See list at http://www.nicta.com.au/research/projects/ /constraint_programming_platform (full URL was split in two lines)
Availability	See http://www.g12.csse.unimelb.edu.au
References	[13, 28, 47, 50, 55]

(branch-and-bound selecting the most fractional variable first and performing a standard split) are attached to the `solve` statement. Since column generation is to be used, the transformation must linearize the master constraints and objective function. This can be done in Zinc by giving linear definitions to the `sum_set` and `sequence` constraints. For the complete details, see [50].

The G12 facilities have been successfully used for a variety of computational experiments involving problems such as radiation, cumulative scheduling, and hoist scheduling. Another hybrid algorithm supported by G12 is *Lazy-FD*, which is a tight integration of FD and SAT that has shown consistently good performance on FD problems [48].

Zinc/MiniZinc (as of version 0.9) has been interfaced to a number of finite domain solvers (including Gecode [56] and ECL^{i}PS^{e} [61]), LP solvers (including CPLEX, GLPK, and COIN-OR CLP through COIN-OR OSI), and SAT solvers (including MiniSAT and TiniSAT).

G12's data sheet appears in Table 4.

Coming soon to G12: Propagation-based solving where individual constraints can be annotated with the solver where they should be sent, for example, `x >= y::lp::fd` means that the constraint `x >= y` is sent to both the LP and finite domain solvers; and annotation of models to split the problem into two parts: this-then-that, where the first part is solved and used as input for the second part.

7 IBM ILOG CP Optimizer and OPL Development Studio

The ILOG OPL Development Studio [58] (OPL for short) is designed to support ILOG CPLEX as well as ILOG CP Optimizer [36, 37]. OPL allows users to develop single models in either technology or multimodel solutions that use either or both

technologies combined. A scripting language (ILOG Script for OPL) provides an additional level of control, enabling the user to create algorithms that run multiple models sequentially or iteratively, and to exchange information among them. OPL also includes an integrated development environment (IDE), which is a graphical user interface providing all the typical development facilities and support, including: searching for conflicts between constraints in infeasible mathematical programming (MP) models, visualizing the state of variables at some point during the search for a solution, connecting to a database or to a spreadsheet to read and write data, debugging, etc. The current OPL version as of this writing is 6.0.1.

Since version 2.0, the ILOG CP Optimizer provides a new scheduling language supported by an automatic search that is meant to be robust and efficient (i.e., its default behavior, without much user intervention or fine tuning, should perform well on average). This new scheduling model was designed with the requirement that it should, among other things, be accessible to mathematical programmers, be simple and nonredundant, fit naturally into a CP paradigm, and be expressive enough to handle complex applications. The scheduling language is available in C++, Java, and C#, as well as in OPL itself. The automatic search is based on a self-adapting LNS that iteratively *unfreezes* and *reoptimizes* a selected fragment of the current solution. ILOG CP Optimizer introduces a conditional interval formalism that extends classical constraint programming by introducing additional mathematical concepts (such as intervals, sequences, or functions) as new variables or expressions to capture the temporal aspects of scheduling. For example:

Interval variables: An interval variable a is a decision variable whose domain $\mathrm{dom}(a)$ is a subset of $\{\bot\} \cup \{[s,e) \mid s,e \in \mathbb{Z},\ s \leq e\}$. If $\mathrm{dom}(a) = \bot$, the interval is said to be *absent* (*present* otherwise). An absent interval variable is not considered by any constraint or expression on interval variables in which it is involved. To model the basic structures of scheduling problems, a number of special constraints on interval variables exist, such as precedence constraints and time spanning constraints.

Sequence variables: Sequence variables are a type of decision variable whose value is a permutation of a set of interval variables. They are inspired by problems that involve scheduling a set of activities on a disjunctive resource. Constraints on sequence variables are provided for ruling out illegal permutations (sequencing constraints) or for stating a particular relation between the order of intervals in the permutation and the relative position of their start and end values (no-overlap constraints).

Cumul Function Expressions: Cumul Function Expressions are expressions that are used to represent the usage of a cumulative resource over time. This usage is represented as a sum of individual contributions of intervals (some time intervals make the resource consumption go up, others make it go down).

In a recent study of three problems from the scheduling literature, compact models written in OPL for CP Optimizer produced results that, on average, outperformed state-of-the-art problem-specific approaches for those problems (see [36] for details).

To illustrate the integration capabilities of OPL, CPLEX, and CP Optimizer, we will briefly go over the steps to solve a configuration problem using column generation. We will use the CP Optimizer engine to generate new possible configurations and the CPLEX engine to solve the problem of selecting the best combination of configurations. This configuration problem involves placing objects of different materials (glass, plastic, steel, wood, and copper) into bins of various types (red, blue, green), subject to capacity (bin type dependent) and compatibility constraints (e.g., red bins cannot contain plastic or steel). All objects must be placed into a bin and the total number of bins must be minimized. The idea is to write a model file that uses CP to *generate* bin configurations, a second model file that uses CPLEX to *select* a subset of configurations, and a script file that coordinates the execution of the previous two and passes information from one to the other. The generation model (`generate.mod`) has a color variable to indicate the bin color and object variables indicating how many objects of each material are included in the bin. It then writes the compatibility constraints as logical expressions and imposes the bin capacity constraints. The selection model (`select.mod`) has one variable for each configuration created as input and an objective that minimizes the number of bins produced. The only constraint is to satisfy the demand for each material. The script model appears in Fig. 4. Line 01 associates the internal name `genBin` to the actual generation model, while lines 02–04 set up and start the search. The loop of line 05 asks for all solutions and adds them (line 07) to the `data` structure (`n` is the variable that tells how many objects of each material are in the bin). When no more solutions exist, line 09 ends the search. With all variables now available, line 10 associates the internal name `chooseBin` with the selection model and line 12 passes the data (variables) to it. Lines 14 and 15 solve the problem and wrap up.

```
01. var genBin = new IloOplRunConfiguration("generate.mod");
02. genBin.oplModel.addDataSource(data);
03. genBin.oplModel.generate();
04. genBin.cp.startNewSearch();
05. while (genBin.cp.next()) {
06.   genBin.oplModel.postProcess();
07.   data.Bins.add(genBin.oplModel.newId,
                    genBin.oplModel.colorStringValue,
                    genBin.oplModel.n.solutionValue);
08. }
09. genBin.cp.endSearch();

10. var chooseBin = new IloOplRunConfiguration("select.mod");
11. chooseBin.cplex = cplex;
12. chooseBin.oplModel.addDataSource(data);
13. chooseBin.oplModel.generate();
14. chooseBin.cplex.solve();
15. chooseBin.oplModel.postProcess();
```

Fig. 4 ILOG script for OPL example

Table 5 ILOG CP optimizer/OPL data sheet

Feature	Description
Tool name	ILOG CP Optimizer and OPL Development Studio
User interface	1a, 1b, 1c
Solver support	2a, 2c
Relaxation support	3a, 3c, 3d
Inference mechanisms	4a, 4b, 4c
Search mechanisms	5a, 5b
Decomposition mechanisms	6b, 6c
Search control	7a, 7b, 7c
Written in	C++
Supported platforms	AIX, Linux, Mac OS X, Solaris, Windows (full OPL IDE available only under Windows)
Availability	Commercial, academic and student licenses available. See http://www.ilog.com.
References	[36, 37, 58]
Notes	ILOG CP Optimizer only handles discrete decision variables

This example generated all variables a priori before solving the optimization model. It is possible, of course, to implement an iterative hybrid algorithm that starts with a subset of the variables and transfers dual information (shadow prices) to the generation problem so that it can look for a new (improving) variable to be added to the optimization (master) problem. Then, as usual, this process is repeated until no improving variables exist.

The CP Optimizer/OPL data sheet appears in Table 5.

8 SCIP

Solving Constraint Integer Programs (SCIP [2, 4]) is an implementation of the constraint integer programming (CIP) paradigm, which is an integration of CP, MILP, and SAT methodologies. It can be used either as a black box solver or as a framework by adding user plug-ins written in C or C++. SCIP allows total control of the solution process and access to detailed information from the solver. A number of predefined macros facilitate implementation by encapsulating commonly used function calls into a simpler interface. SCIP comes with more than 80 default plug-ins that turn it into a full solver for MILP and pseudo-Boolean optimization.

A constraint integer program (CIP) is defined as the optimization problem

$$c^* = min\{cx \mid \mathscr{C}(x),\ x \in \mathbb{Z}^I \times \mathbb{R}^{N \setminus I}\}, \tag{4}$$

where $\mathscr{C} = \{C_1, \ldots, C_m\}$ is a set of constraints and $I \subseteq N = \{1, \ldots, n\}$. By definition, the constraint set $\mathscr{C}$ has to be such that once the integer variables have been assigned values, the remaining problem becomes a linear program.

SCIP is a framework for branching, cutting, pricing, and propagation, and its implementation is based on the idea of plug-ins, which makes it very flexible and extensible. Here is a list of the main types of SCIP plug-ins and their roles:

Constraint handlers: These are the central objects of SCIP. Each constraint handler represents the semantics of a single class of constraints and provides algorithms to handle constraints of the corresponding type. The primary task of a constraint handler is to check a given solution for feasibility with respect to all constraints of its type existing in the problem instance. To improve the performance of the solving process, constraint handlers may provide additional algorithms and information about their constraints to the framework, such as: presolving methods to simplify the problem, propagation methods to tighten variable domains, linear relaxations, branching decisions, and separation routines

Domain propagators: Constraint-based domain propagation is supported by the constraint handler concept of SCIP. In addition, SCIP features two dual domain reduction methods that are driven by the objective function, namely objective propagation and root reduced-cost strengthening.

Conflict analyzers: SCIP generalizes conflict analysis to CIP and, as a special case, to MIP [1]. There are two main differences between CIP and SAT solving in the context of conflict analysis: CIP variables are not necessarily binary and the infeasibility of a subproblem in the CIP search tree is usually caused by the LP relaxation of that subproblem. Because it is NP-hard to identify a subset of the local bounds of minimal cardinality that make the LP infeasible, SCIP uses a greedy heuristic approach based on an unbounded ray of the dual LP.

Cutting plane separators: SCIP features separators for a myriad of cuts [10, 29, 34, 39, 41, 49, 52] in addition to $\{0, 1/2\}$-cuts and multi-commodity-flow cuts. For a survey, see [64]. For cut selection, SCIP uses *efficacy* and *orthogonality* (see [6, 11]), and parallelism with respect to the objective function.

Primal heuristics: SCIP has 23 different heuristics, which can be classified into four categories: rounding, diving, objective diving, and improvement [15].

Node selectors and branching rules: SCIP implements most of the well-known branching rules, including reliability branching [5] and hybrid branching [3], and it allows the user to implement arbitrary branching schemes. Several node selection strategies are predefined, such as depth-first, best-first, and best-estimate [27]. The default search strategy is a combination of these three.

Presolving: SCIP implements a full set of primal and dual presolving reductions for MIP problems, such as removing redundant constraints, fixing variables, strengthening the LP relaxation by exploiting integrality information, improving constraint coefficients, clique extractions, etc. It also uses the concept of *restarts*, which are a well-known ingredient of modern SAT solvers.

Additional SCIP features include:

- Variable pricers to dynamically create problem variables;
- Relaxators to provide relaxations and dual bounds in addition to the LP relaxation, e.g., semi-definite or Lagrangian, working in parallel or interleaved;
- Dynamic cut pool management;
- Counting of feasible solutions, visualization of the search tree, and customization of output statistics.

The fundamental search strategy in SCIP is the exploration of a branch-and-bound tree. All involved algorithms operate on a single search tree, which allows for a very close interaction between different constraint handlers. SCIP manages the branching tree along with all subproblem data, automatically updates the LP relaxation, and handles all necessary transformations due to problem modifications during the presolving stage. Additionally, a cut pool, cut filtering, and a SAT-like conflict analysis mechanism are available. SCIP provides its own memory management and plenty of statistical output.

In addition to allowing the user to integrate MILP, CP, and SAT techniques, SCIP can also be used as a pure MILP or as a pure CP solver. Computational results shown on the benchmark web pages of Hans Mittelmann[1] indicate that SCIP is one of the fastest noncommercial MILP solvers currently available.

SCIP uses an external LP solver to handle the LP relaxations. The current release (1.1.0) can interface with CPLEX, Xpress-MP, Mosek, SoPlex, and the COIN-OR LP solver CLP. The ZIB Optimization Suite (http://zibopt.zib.de) offers a complete integrated bundle of SCIP, SoPlex, and ZIMPL (a MILP modeling language). Furthermore, precompiled binaries including SoPlex or CLP can be found on the SCIP web page.

SCIP's data sheet appears in Table 6.

Coming soon to SCIP: interface to the LP solver QSopt, support for FlatZinc [47] models; improved pseudo-boolean performance. Further down the road: exact

Table 6 SCIP data sheet

Feature	Description
Tool name	SCIP – Solving Constraint Integer Programs
User interface	1c
Solver support	2a, 2c, 2d
Relaxation support	3a, 3c, 3d
Inference mechanisms	4a, 4b, 4c, 4d
Search mechanisms	5a, 5b, 5c
Decomposition mechanisms	6b
Search control	7c
Written in	C
Supported platforms	Should compile with any ANSI C compiler
Developers	Tobias Achterberg, Timo Berthold, Stefan Heinz, Thorsten Koch and Kati Wolter
Availability	SCIP is distributed under the ZIB Academic License (see http://zibopt.zib.de/academic.txt). Source code and binaries are available at http://scip.zib.de.
References	[2, 4]
Notes	SCIP is part of the ZIB Optimization Suite. Also available on the NEOS Server.

[1] http://plato.asu.edu/ftp/milpf.html.

integer programming techniques (sound solver without rounding errors); additional global CP constraints; MINLP capabilities (nonlinear and nonconvex constraints).

9 SIMPL

SIMPL [8, 65] is based on two principles: algorithmic unification and constraint-based control. Algorithmic unification begins with the premise that integration should occur at a fundamental and conceptual level, rather than postponed to the software design stage. Optimization methods and their hybrids are viewed, to the extent possible, as special cases of a single solution method that can be adjusted to exploit the structure of a given problem. This goal is addressed with a *search-infer-and-relax* algorithmic framework, coupled with *constraint-based control* in the modeling language. The search-infer-and-relax scheme encompasses a wide variety of methods, including branch-and-cut (B&C) methods for integer programming, branch-and-infer methods for constraint programming, popular methods for continuous global optimization, nogood-based methods as Benders decomposition and dynamic backtracking, and even heuristic methods such as local search and greedy randomized adaptive search procedures (GRASPs) [26].

Constraint-based control allows the design of the model itself to tell the solver how to combine techniques so as to exploit problem structure. Highly-structured subsets of constraints are written as metaconstraints, which are similar to global constraints in constraint programming. Syntactically, a metaconstraint is written much as linear or global constraints are written, but it is accompanied by parameters that specify how the constraint is to be implemented during the solution process. A metaconstraint may specify how it is to be relaxed, how it will filter domains, and/or how the search procedure will branch when the constraint is violated. When such parameters are omitted, a prespecified default behavior is used.

The relaxation, inference, and branching techniques are devised for each constraint's particular structure. For example, a metaconstraint may be associated with a tight polyhedral relaxation from the integer programming literature and/or an effective domain filter from constraint programming. Because constraints also control the search, if a branching method is explicitly indicated for a metaconstraint, the search will branch accordingly. The selection of metaconstraints to formulate the problem determines how the solver combines algorithmic ideas to solve the problem.

To illustrate the above ideas, we consider the following integer knapsack problem with a side constraint (see Chap. 2 of [32]):

$$\begin{aligned} \min\ & 5x_1 + 8x_2 + 4x_3 \\ \text{s.t. } & 3x_1 + 5x_2 + 2x_3 \geq 30 \\ & \textit{alldifferent}(x_1, x_2, x_3) \\ & x_j \in \{1, 2, 3, 4\},\ \forall\ j \end{aligned}$$

```
01. DECLARATIONS
02.    n = 3; limit = 30;
03.    cost[1..n] = [5,8,4]; weight[1..n] = [3,5,2];
04.    discrete range xRange = 1 to 4;
05.    x[1..n] in xRange;
06. OBJECTIVE
07.    min sum i of cost[i]*x[i]
08. CONSTRAINTS
09.    totweight means {
10.       sum i of weight[i]*x[i] >= limit
11.       relaxation = {lp, cp} }
12.    distinct means {
13.       alldifferent(x)
14.       relaxation = {lp, cp} }
15. SEARCH
16.    type = {bb:depth}
17.    branching = {x:first, distinct:most}
```

Fig. 5 SIMPL model for the hybrid knapsack problem

A SIMPL model for the above problem is shown in Fig. 5. The model starts with a `DECLARATIONS` section in which constants and variables are defined. Line 07 defines the objective function. In the `CONSTRAINTS` section, the two constraints of the problem are represented by the (named) metaconstraints `totweight` and `distinct`, and their definitions show up in lines 10 and 13, respectively. The `relaxation` statements in lines 11 and 14 indicate the relaxations to which those constraints should be posted. Both constraints will be present in the LP and in the CP relaxations. Because the `alldifferent` constraint is not linear, submitting it to an LP relaxation means that it will be automatically transformed into a linear approximation (in this case, the convex hull formulation) of the set of its feasible solutions (see [63]). In the `SEARCH` section, line 16 indicates that we will do branch-and-bound (`bb`) with depth-first search (`depth`). The `branching` statement in line 17 says that we will branch on the first of the `x` variables that is not integer (branching on a variable means branching on its indomain constraint). Once all `x`'s are integer, the most violated of the `alldifferent` constraints will be used for branching (`distinct:most`). Initially, bounds consistency maintenance in the CP solver removes value 1 from the domain of x_2 and the solution of the LP relaxation is $x = (2\frac{2}{3}, 4, 1)$. After branching on $x_1 \leq 2$, bounds consistency determines that $x_1 \geq 2$, $x_2 \geq 4$, and $x_3 \geq 2$. At this point, the `alldifferent` constraint produces further domain reduction, yielding the feasible solution $(2, 4, 3)$. Notice that no LP relaxation had to be solved at this node. In a similar fashion, the CP solver may be able to detect infeasibility even before the linear relaxation has to be solved.

In [65], SIMPL was used to model and solve four classes of problems that had been successfully solved by custom implementations of integrated approaches, namely: production planning, product configuration, machine scheduling, and truss structure design. Computational results indicate that the high-level models

Table 7 SIMPL data sheet

Feature	Description
Tool name	SIMPL: A Modeling Language for Integrated Problem Solving
User interface	1b, 1c
Solver support	2a, 2c
Relaxation support	3a, 3d
Inference mechanisms	4a, 4b, 4c
Search mechanisms	5a
Decomposition mechanisms	6c
Search control	7a, 7c
Written in	C++
Supported platforms	Linux
Developers	Ionuţ Aron, John Hooker and Tallys Yunes
Availability	Free for academic use. Preliminary demo version available at http://moya.bus.miami.edu/∼tallys/simpl.php.
References	[8, 65]
Notes	Looking to expand the development group.

implemented in SIMPL either match or surpass the performance of the original special-purpose codes at a fraction of the implementation effort.

SIMPL requires external solvers to handle the problem relaxations. The current release (0.08.22) can use CPLEX as an LP solver and ECL^{i}PS^e as a CP solver. Other solvers can be added by deriving an object class and implementing standard interface methods.

SIMPL's data sheet appears in Table 7.

Coming soon to SIMPL: Interfaces to NLP solvers; additional LP and CP solvers; preprocessing; cutting plane generation. Further down the road: support for SAT solvers; local and incomplete search; branch-and-price; high-level search language; GUI and IDE.

10 Xpress-Mosel

The Xpress-Mosel language [22] (Mosel for short) is a modeling language that is part of the FICO Xpress suite of mathematical modeling and optimization tools[2]. It allows the user to formulate the problem, solve it with a suitable solver engine, and analyze the solution, using a fully-functional programming language specifically designed for this purpose. Mosel programs can be run interactively or embedded within an application. The language is integrated within the Xpress-IVE visual development environment. In addition to the usual features available in standard modeling languages, it provides support for arbitrary ranges, index sets, sparse objects, and a debugger that supports tracing and analyzing the execution of a model.

[2] Originally developed by Dash Optimization under the name Xpress-MP.

Mosel is an open, user-extensible language. The Mosel distribution includes extension libraries (so-called *modules*), one of which provides control of the Xpress-Optimizer (module *mmxprs*), through optimization statements in the Mosel program. Other solver modules give access to formulating and solving nonlinear problems (module *mmnl* handles QCQP, MIQCQP and convex NLP), the Stochastic Programming tool Xpress-SP[3], and the Constraint Programming software Xpress-Kalis (module *kalis*). In Mosel, CP and MIP solving may be used *sequentially*, for instance, employing CP constraint propagation as a preprocessing routine for LP/MIP problems; or *in parallel* as would happen, for example, when CP solving is used as cut or variable generation routine during an MIP branch-and-bound search. We now provide a simplified example of the former type of integration to illustrate Mosel's modeling constructs (taken from [30, 31]). Due to space limitations, we omit the data and variable declarations in our Mosel models.

Consider a project scheduling problem in which a set of tasks (`TASKS`) with a certain default duration (`DUR(i)`, in weeks) have to be executed (e.g., a real estate development project). Precedence constraints exist between given pairs of tasks (`ARC(i,j)`) and, if the manager is willing to spend extra money (`COST(i)`), it is possible to reduce the duration of each task by `save(i)`, up to a certain amount (`MAXW(i)`) (this is sometimes referred to as *crashing*). Assume that we know the earliest project completion date without crashing (`Finish`) and the client is willing to pay `BONUS` dollars for every week the work finishes early. To maximize the manager's profit, we will solve this problem in two stages: first (Fig. 6), we use the

```
01. model "Crashing CP"
02. uses "kalis"
03. declarations
04.   start: array(TASKS) of cpvar
05.   duration: array(TASKS) of cpvar
06.   lbstart, ubstart: array(TASKS) of integer
07. end-declarations
08. forall(j in TASKS) setdomain(start(j),0,Finish)
09. forall(j in TASKS) setdomain(duration(j),
                                 DUR(j)-MAXW(j),DUR(j))
10. forall(i,j in TASKS | exists(ARC(i,j)))
11.   start(i) + duration(i) <= start(j)
12. forall(i in TASKS) do
13.   lbstart(i) := getlb(start(i))
14.   ubstart(i) := getub(start(i))
15. end-do
16. initializations to "raw:"
17.   lbstart as "shmem:lbstart" ubstart as "shmem:ubstart"
18. end-initializations
19. end-model
```

Fig. 6 Mosel CP model for project crashing. File name: crash1.mos

[3] Recently turned into open-source and available from the Xpress website.

```
01. model "Crashing master (CP + LP)"
02. uses "mmxprs", "mmjobs"
03. declarations
04.   CPmodel: Model
05. end-declarations
06. res := compile("crash1.mos"); load(CPmodel,"crash1.bim")
07. run(CPmodel); wait
08. initializations from "raw:"
09.   lbstart as "shmem:lbstart" ubstart as "shmem:ubstart"
10. end-initializations
11. declarations
12.   start: array(TASKS) of mpvar
13.   save: array(TASKS) of mpvar
14. end-declarations
15. Profit := BONUS*save(N) - sum(i in 1..N-1) COST(i)*save(i)
16. forall(i,j in TASKS | exists(ARC(i,j)))
17.   Precm(i,j) := start(i) + DUR(i) - save(i) <= start(j)
18. start(N) + save(N) = Finish
19. forall(i in 1..N-1) save(i) <= MAXW(i)
20. forall(i in 1..N-1) do
21.   lbstart(i) <= start(i); start(i) <= ubstart(i)
22. end-do
23. maximize(Profit)
24. end-model
```

Fig. 7 Mosel LP model for project crashing

CP solver to find bounds on the start times of each task (`start(i)`), and then we use those bounds in an LP optimization model (Fig. 7). In Fig. 6, line 02 indicates we will use the CP module and lines 04 and 05 declare the start time and duration variables. Note that the actual duration of a task is a variable because we do not know a priori by how many weeks each task will be crashed. The calculated bounds on task start times, which are declared in line 06, will be retrieved from the CP solver in lines 12–15, and passed to the LP model through a shared memory space in lines 16–18. Lines 08 and 09 declare the variable domains. Lines 10 and 11 create and post the precedence constraints. In Fig. 7, the master model includes the optimizer module in line 02 and declares a CP model in line 04. In lines 06 and 07, the CP model from Fig. 6 is compiled, loaded, and run. Line 09 retrieves from the shared memory space the bounds on task start times obtained by the CP model. We declare the variables of the optimization model in lines 11–14 and the objective function in line 15. Task number `N` is a virtual task introduced so that its start time represents the completion time of the project. Lines 17 and 18 state the precedence constraints and tie the project completion time to the known duration `Finish`. Line 19 limits the crash amounts, and lines 20–22 use the bounds retrieved from the CP model to tighten the domain of `start(i)`.

Two successful, and more intricate, implementations of hybrid algorithms using Mosel appeared in [17] and [51].

Xpress-Mosel's data sheet appears in Table 8.

Table 8 Xpress-Mosel data sheet

Feature	Description
Tool name	Xpress-Mosel
User interface	1a, 1b, 1c
Solver support	2a, 2b, 2c
Relaxation support	3a, 3b, 3d
Inference mechanisms	4a, 4b, 4c
Search mechanisms	5a, 5b
Decomposition mechanisms	6a, 6b, 6c
Search control	7a, 7b, 7c
Supported platforms	AIX, HP-UX, Linux, Solaris, Windows (Xpress-IVE available on Windows only)
Main developer	Yves Colombani
Availability	Commercial, academic and student licenses available. See http://www.dashoptimization.com or www.fico.com.
References	[21, 22, 30]

Coming soon to Xpress-Mosel: support for multiple problems in the same model file, introducing the concept of *local scoping* i.e., variables and constraints are no longer necessarily global and instead can be defined in the context of a specific subproblem (useful in the implementation of user-defined heuristics); a new feature of Xpress-Kalis that introduces the possibility to work with automatic LP/MIP relaxations of CP constraints (linear constraints and global constraints like alldifferent, occurrence, etc.). The LP/MIP representation is generated automatically and solved by Xpress-Optimizer with numerous configuration options as to how, where, and when to solve the MP problem(s).

11 Other Hybrid Tools

This section covers a few other software tools that include some level of support for integration.

11.1 COIN-OR

Computational Infrastructure for Operations Research (COIN-OR) [40] is a project aimed at spurring the development of open-source software for the OR community. Its objectives include speeding the development and deployment of models, algorithms, and cutting-edge computational research, as well as providing a forum for peer review of software similar to that provided by archival journals for theoretical research. While not being a hybrid tool per se, the COIN-OR initiative can be viewed as a hybrid set of tools that could, in principle, be combined. The COIN-OR

repository is composed of many different projects divided into categories such as developer tools, graph algorithms, abstract interfaces (e.g., OSI), metaheuristics, and optimization packages that can handle distinct types of problems (deterministic linear and nonlinear (both continuous and discrete), deterministic semi-definite continuous, and stochastic). For further details, see http://www.coin-or.org.

11.2 Microsoft Solver Foundation

Recently, the Microsoft Solver Foundation (version 1.1) has added links to a number of well-known solvers through its Solver Plug-in System, namely CPLEX, Xpress-MP, Mosek, and Gurobi. There is also support for CP solvers. For more details, see http://www.solverfoundation.com.

11.3 Prolog IV

Prolog IV [46] is an ISO-compliant replacement for the Prolog III language. It incorporates all the main features of Prolog III with some important changes. It allows programmers to express a wide variety of constraints over real and rational numbers, integers (finite domains), booleans, and lists. In addition to expressing classical linear programming problems on discrete and continuous quantities, it permits, among other things, the use of mixed real/integers problems, and the use of boolean operations to formalize constraint disjunctions. The algorithms include a nonoptimized algorithm for lists (different from Prolog III), Gauss and Simplex algorithms for equations and linear inequalities over rationals, and an interval method for approximate solving of nonlinear constraints over reals. The compiler is integrated into a complete graphic programming environment featuring tools such as a project editor, a multiwindow text editor, grapher, debugger, and online help.

11.4 SALSA

SALSA [38] is a language dedicated to specifying local, global, and hybrid search algorithms. It provides the user with the ability to specify the way the global search tree is explored. SALSA attempts to make the creation of hybrid algorithms a less tedious and less error-prone task by providing a high-level language that offers the ability to perform nonmonotonic operations and hypothetical reasoning. It proposes to consider logic and control separately and because it is not a standalone language, it works in cooperation with a host programming language. SALSA allows the programmer to specify the choice mechanisms that are responsible for generating the

moves of a search algorithm, and it offers primitives for composing the transitions from one state to the next. In global search, goals and constraints describe properties of final states, while in local search, invariants, and neighborhoods describe properties of all states. However, because the basic branching mechanism of global-search and local-search algorithms are very much alike, SALSA allows for hybrid combinations by expressing neighborhoods and choices in the same formalism.

11.5 ToOLS

The purpose of ToOLS [23] (templates of on-line search) is twofold: to help a constraint programmer to build complex customized search algorithms, and to offer readymade search components for engineers, improving algorithm reuse, and capitalization. ToOLS is part of a finite-domain constraint solver library called Eclair, developed in the high-level language Claire [18]. ToOLS divides the description of a search algorithm into three parts (or components): a complete search tree defined by a refinement-based search scheme, a set of conditions restricting the exploration of the tree, and a combination of several partial explorations. Each component can also be reused separately. A search algorithm is a Claire object created by a functional composition of constructors called ToOLS *primitives*. A special function (solve, solveAll or minimize) specifies the goal of the search (satisfaction or optimization) and is applied to a single algorithm object. ToOLS has been successfully used to implement LNS methods applied to satellite observation scheduling and military applications.

12 Conclusion

The availability of software tools for integrated optimization has dramatically increased since the first CP-AI-OR workshop held in 1999. Today, it is possible to exploit the power of hybrid algorithms without having to spend several days (or even months) writing and debugging computer code. Both academia and industry recognize the need for better optimization software that facilitates the integration of solution techniques, and this chapter provides a brief overview of some existing software packages.

Despite the tremendous progress over the past 10 years, there is still much room for improvement. In [60], Wallace, Caseau and Puget mention that “to make an impact, our technology must be made useable so that (1) highly qualified experts can develop solutions very quickly,” and “(2) less expert users can also exploit the technology successfully.” This greater *accessibility* was one of the fundamental forces behind the increased adoption and dissemination of traditional OR techniques. Moreover, the authors also make a case for the need to “make the user’s task more manageable.” We could, for instance, “identify general problem features

that correlate with suitable algorithms. Define simple rules about when to use one problem solving method and when to use another one. Categorize which forms of hybridization work best for which kinds of problems and algorithms." This means that our software tools could greatly benefit from a database of *meta models* (see [19]). Over the years, the OR and CP communities have gathered a wealth of knowledge about what works and what does not work; what kinds of mathematical formulas are more suitable representations of certain real-life phenomena; what kinds of symmetries typically arise in a given class of problems or from a given choice of variables; which cutting planes are effective for a given class of optimization problem; etc. Most of this knowledge resides in the minds of our best modelers when it should be residing in the software itself. The next generation of modeling tools needs to extract more information from the user. The creation of a new model should start with an electronic questionnaire that will collect information about the nature and structure of the problem. It could, for instance, present a tree hierarchy to the user and allow him/her to click on the nodes that have a relationship to that particular problem, as depicted in Fig. 8. To input a problem that combines location, routing, and scheduling aspects, the user simply clicks on more than one node. Similar hierarchies can be used to gather information about the problem domain (e.g., manufacturing, finance, marketing), as well as the nature of the objective function and constraints (e.g., does it include uncertainty/randomness?). In possession of this extra knowledge, the modeling tool will be in a better position to choose its own default parameters, to suggest modeling constructs (constraints, cuts, relaxations, decompositions), and even to flag potentially ineffective choices made by the users as they input the model.

In [33], Hooker argues that "we should take full advantage of the graphical user interface to empower modelers. A metaconstraint should be invoked by opening a window on the computer screen, not by typing a statement. The window should present various options for refining the constraint and importing data. The model as a whole should be depicted graphically, with an opportunity to click on modules for a more detailed look." Once again, it all boils down to accessibility. By making

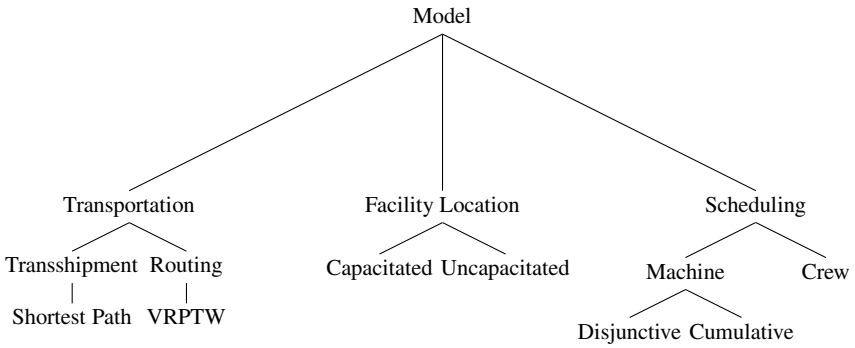


Fig. 8 Hypothetical hierarchy of models. Users click on the nodes that are relevant to their problem

our models easier to build, and therefore easier to understand, we can reach out to a larger audience. Software packages that implement some of these ideas include Visual CHIP [9], CHIP Factory [53], and the computer simulation software Arena [35]. Visualization also plays an important role in solution analysis and model tuning. The visualization capabilities of tools like CHIP and Comet represent an important step in that direction. When combined with the kind of metamodeling information discussed above, such a graphical modeling environment would be able to prepopulate its window with some of the necessary modules. All of this automatic behavior, of course, must be available while also staying out of the way of the expert user.

The development and growth of hybrid modeling tools has created exciting new challenges and numerous research directions. The general trend seems to be one of *unification* rather than separation. Over the next 10 years, our tools will be able to do more, while asking for less of our guidance. It is unlikely that the software expertise will ever substitute the human expertise, but significant improvements over the current state-of-the-art are definitely possible. By being aware of these strengths and weaknesses, we have taken an important step toward a new generation of software tools for integrated optimization.

Acknowledgments The author would like to thank Timo Berthold, Stefan Heinz, Susanne Heipcke, Katya Krasilnikova, Michela Milano, Philippe Refalo, Nick Sahinidis, Kish Shen, Helmut Simonis, Peter Stuckey, Pascal Van Hentenryck, and Mark Wallace for answering technical questions, and for providing feedback on the presentation of the material and on the accuracy of the information about each software tool.

References

1. Achterberg T (2007) Conflict analysis in mixed integer programming. Discrete Optim 4(1):4–20 Special issue: Mixed Integer Programming
2. Achterberg T (2008) SCIP: solving constraint integer programs. Math Program Comput 1(1):1–41
3. Achterberg T, Berthold T (2009) Hybrid branching. In: van Hoeve WJ, Hooker J (eds) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems, 6th international conference, CPAIOR 2009. Lecture notes in computer science, vol 5547. Springer, Berlin, pp 309–311
4. Achterberg T, Berthold T, Koch T, Wolter K (2008) Constraint integer programming: a new approach to integrate CP and MIP. In: Perron L, Trick M (eds) Proceedings of the conference on integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CP-AI-OR). Lecture notes in computer science, vol 5015. Springer, Berlin, pp 6–20
5. Achterberg T, Koch T, Martin A (2005) Branching rules revisited. Oper Res Lett 33:42–54
6. Andreello G, Caprara A, Fischetti M (2007) Embedding cuts in a branch and cut framework: a computational study with $\{0, 1/2\}$-cuts. INFORMS J Comput 19(2):229–238
7. Apt KR, Wallace M (2007) Constraint logic programming using ECL^{i}PS^{e}. Cambridge University Press, Cambridge
8. Aron ID, Hooker JN, Yunes TH (2004) SIMPL: a system for integrating optimization techniques. In: Rgin J, Rueher M (eds) Proceedings of the conference on integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CP-AI-OR). Lecture notes in computer science, vol 3011. Springer, Berlin, pp 21–36

9. Baader F, Comon H, Smolka G (1997) Visual CHIP: a visual language for defining constraint programs. In: Annual workshop of the ESPRIT working group "constructions of computational logic II" (CCL), Dagstuhl
10. Balas E (1975) Facets of the knapsack polytope. Math Program 8:146–164
11. Balas E, Ceria S, Cornuéjols G (1996) Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. Manage Sci 42:1229–1246
12. Barnhart C, Johnson EL, Nemhauser GL, Savelsbergh MWP, Vance PH (1998) Branch-and-price: column generation for solving huge integer programs. Oper Res 46:316–329
13. Becket R, Brand S, Brown M, Duck GJ, Feydy T, Fischer J, Huang J, Marriott K, Nethercote N, Puchinger J, Rafeh R, Stuckey PJ, Wallace MG (2008) The many roads leading to Rome: solving Zinc models by various solvers. In: Proceedings of the 7th international workshop on constraint modeling and reformulation (ModRef)
14. Benders JF (1962) Partitioning procedures for solving mixed-variables programming problems. Numer math 4:238–252
15. Berthold T (2006) Primal heuristics for mixed integer programs. Master's thesis, Technische Universität Berlin, Berlin
16. Bocca J (1991) Megalog – a platform for developing knowledge base management systems. In: Proceedings of the second international symposium on database systems for advanced applications (DASFAA), Tokyo, Japan
17. Bockmayr A, Pisaruk N (2003) Detecting infeasibility and generating cuts for MIP using CP. In: Fifth international workshop on integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CP-AI-OR), Montréal, Canada, pp 24–34
18. Caseau Y, Josset FX, Laburthe F (1999) CLAIRE: combining sets, search and rules to better express algorithms. In: Proceedings of the international conference on logic programming (ICLP), pp 245–259
19. Caseau Y, Silverstein G, Laburthe F (2001) Learning hybrid algorthms for vehicle routing problems. Theory Pract Log Program 1(6):779–806
20. Clocksin WF, Mellish CS (1981) Programming in prolog. Springer, Berlin
21. Colombani Y, Daniel B, Heipcke S (2004) Mosel: a modular environment for modeling and solving problems. In: Kallrath J (ed) Modeling languages in mathematical optimization. Kluwer Academic, Boston, pp 211–238
22. Colombani Y, Heipcke S (2002) Mosel: an extensible environment for modeling and programming solutions. In: Proceedings of the international workshop on integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CP-AI-OR)
23. de Givry S, Jeannin L (2003) ToOLS: a library for partial and hybrid search methods. In: Fifth international workshop on integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CP-AI-OR), Montréal, Canada, pp 124–138
24. Dincbas M, Van Hentenryck P, Simonis M, Aggoun A, Graf T, Berthier F (1988) The constraint logic programming language CHIP. In: Proceedings of the international conference of fifth generation computer systems, pp 693–702
25. Duck GJ, Stuckey PJ, Brand S (2006) ACD term rewriting. In: Etalle S, Truszczynski M (eds) Proceedings of the 22nd international conference on logic programming (ICLP). Lecture notes in computer science, vol 4079. Springer, Heidelberg, pp 117–131
26. Feo T, Resende M (1995) Greedy randomized adaptive search procedures. J Global Optim 6:109–133
27. Forrest JJ, Hirst JPH, Tomlin JA (1974) Practical solution of large scale mixed integer programming problems with UMPIRE. Manage Sci 20(5):736–773
28. Garcia de la Banda M, Marriott K, Rafeh R, Wallace M (2006) The modelling language Zinc. In: Benhamou F (ed) Proceedings of the 12th international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 4204. Springer, Heidelberg, pp 700–705
29. Gomory RE (1960) Solving linear programming problems in integers. In: Bellman R, Hall JM (eds) Symposia in applied mathematics X, combinatorial analysis. AMS, Providence, RI, pp 211–215

30. Guéret C, Heipcke S, Prins C, Sevaux M (2002) Applications of optimization with Xpress-MP. Dash optimization. Blisworth, UK. http://www.dashoptimization.com/applications_book.html
31. Heipcke S (2005) Hybrid MIP/CP solving with Xpress-Optimizer and Xpress-Kalis. Xpress Whitepaper, FICO. http://www.dashoptimization.com
32. Hooker JN (2000) Logic-based methods for optimization: combining optimization and constraint satisfaction. Wiley-Interscience Series in Discrete Mathematics and Optimization
33. Hooker JN (2007) Good and bad futures for constraint programming (and operations research). Constraint Program Lett 1:21–32 Special Issue on the next 10 years of constraint programming
34. Johnson EL, Padberg MW (1982) Degree-two inequalities, clique facets and biperfect graphs. Ann Discrete Math 16:169–187
35. Kelton WD, Sadowski RP, Sturrock DT (2007) Simulation with arena, 4th edn. McGraw Hill, New York
36. Laborie P (2009) IBM ILOG CP Optimizer for detailed scheduling illustrated on three problems. In: Hooker JN, van Hoeve WJ (eds) Proceedings of the conference on integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CP-AI-OR). Lecture notes in computer science, vol 5547. Springer, Berlin, pp 148–162
37. Laborie P, Rogerie J, Shaw P, Vilím P, Wagner F (2008) ILOG CP Optimizer: detailed scheduling model and OPL formulation. Tech. Rep. 08-002, ILOG. Availableathttp://www2.ilog.com/techreports/
38. Laburthe F, Caseau Y (2002) SALSA: a language for search algorithms. Constraints 7(3–4):255–288
39. Lechford AN, (2002) Lodi a strengthening Chvátal-Gomory cuts and Gomory fractional cuts. Oper Res Lett 30(2):74–82
40. Lougee-Heimer R (2003) The common optimization interface for operations research. IBM J Res Dev 47(1):57–66
41. Marchand H (1998) A polyhedral study of the mixed knapsack set and its use to solve mixed integer programs. Ph.D. thesis, Faculté des Sciences Appliquées, Université Catholique de Louvain
42. McCormick GP (1976) Computability of global solutions to factorable nonconvex programs: Part I – convex underestimating problems. Math Program 10:147–175
43. McCormick GP (1983) Nonlinear programming: theory, algorithms and applications. Wiley-Interscience, New York
44. Meier M, Kay P, Van Rossum E, Grant H (1989) SEPIA programming environment. In: Proceedings of the workshop on PROLOG programming environments NACLP'89, pp 82–86
45. Minton S, Johnson MD, Philips AB (1990) Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In: Proceedings of the eighth national conference on artificial intelligence (AAAI-90), Boston, pp 17–24
46. Narboni G (1999) From prolog III to prolog IV: the logic of constraint programming revisited. Constraints 4(4):313–335
47. Nethercote N, Stuckey PJ, Becket R, Brand S, Duck GJ, Tack G (2007) MiniZinc: towards a standard CP modelling language. In: Bessière C (ed) Proceedings of the 13th international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 4741. Springer, Heidelberg, pp 529–543
48. Ohrimenko O, Stuckey PJ, Codish M (2007) Propagation = lazy clause generation. In: Bessière C (ed) Proceedings of the 13th international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 4741. Springer, Berlin, pp 544–558
49. Padberg MW, van Roy TJ, Wolsey LA (1985) Valid inequalities for fixed charge problems. Oper Res 33(4):842–861
50. Puchinger J, Stuckey PJ, Wallace M, Brand S (2008) From high-level model to branch-and-price solution in G12. In: Perron L, Trick M (eds) Proceedings of the conference on integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CP-AI-OR). Lecture notes in computer science, vol 5015. Springer, Berlin, pp 218–232

51. Sadykov R (2004) A hybrid branch-and-cut algorithm for the one-machine scheduling problem. In: Régin J, Rueher M (eds) Proceedings of the conference on integration of AI and OR techniques in constraint programming for combinatorial optimization problems (CP-AI-OR). Lecture notes in computer science, vol 3011. Springer, Berlin, pp 409–414
52. Savelsbergh MWP (1994) Preprocessing and probing techniques for mixed integer programming problems. ORSA J Comput 6:445–454
53. Simonis H (2000) Finite domain constraint programming methodology. In: Second international conference and exhibition on the practical application of constraint technologies and logic programming (PACLP), Manchester, UK. Tutorial
54. Somogyi Z, Henderson F, Conway T (1996) The execution algorithm of Mercury, an efficient purely declarative logic programming language. J Log Program 29(1–3):17–64
55. Stuckey PJ, Garcia de la Banda M, Maher MJ, Marriott K, Slaney JK, Somogyi Z, Wallace M, Walsh T (2005) The G12 project: mapping solver independent models to efficient solutions. In: van Beek P (ed) Proceedings of the 11th international conference on principles and practice of constraint programming (CP). Lecture notes in computer science, vol 3709. Springer, Berlin, pp 13–16
56. Tack G (2009) Constraint propagation – models, techniques, implementation. Ph.D. thesis, Saarland University, Germany. http://www.gecode.org
57. Tawarmalani M, Sahinidis NV (2004) Global optimization of mixed-integer nonlinear programs: a theoretical and computational study. Math Program 99:563–591
58. Van Hentenryck P, Lustig I, Michel L, Puget JF (1999) The OPL optimization programming language. MIT Press, Cambridge, MA
59. Van Hentenryck P, Michel L (2005) Constraint-based local search. MIT Press, Cambridge, MA, USA
60. Wallace M, Caseau Y, Puget JF (2004) Open perspectives. In: Milano M (ed) Constraint and integer programming: toward a unified methodology. Kluwer, Dordrecht, pp 331–365
61. Wallace M, Novello S, Schimpf J (1997) ECL^{i}PS^{e}: a platform for constraint logic programming. ICL Syst J 12:159–200
62. Warren DHD (1983) An abstract prolog instruction set. Tech. Rep. 309, SRI International. http://www.ai.sri.com/pubs/files/641.pdf
63. Williams HP, Yan H (2001) Representations of the all_different predicate of constraint satisfaction in integer programming. INFORMS J Comput 13(2):96–103
64. Wolter K (2006) Implementation of cutting plane separators for mixed integer programs. Master's thesis, Technische Universität Berlin, Berlin
65. Yunes T, Aron ID, Hooker JN (2010) An integrated solver for optimization problems. Oper Res 58(2):342–356

Connections and Integration with SAT Solvers: A Survey and a Case Study in Computational Biology

Fabien Corblin, Lucas Bordeaux, Eric Fanchon, Youssef Hamadi, and Laurent Trilling

Abstract Boolean constraints play a fundamental rôle in optimization and constraint satisfaction. The resolution of these constraints has been the subject of intense and successful work during the past decade, and SAT solvers have reached a spectacular maturity. This chapter gives a brief overview of the relevant literature on modern SAT solvers and on the recent efforts to better integrate Boolean reasoning with other constraint satisfaction techniques. As a case study that illustrates the use of SAT and CP, we consider an application in computational biology: the task to build gene regulatory networks (GRNs). We report on experiments made on this problem with a combined SAT/CP approach.

1 Introduction

This chapter focuses on Boolean constraints. An important breakthrough in constraint satisfaction over the past decade was the advent of highly scalable solvers for Propositional Satisfiability (SAT). This new technology revolutionized the way Boolean reasoning is done in constraint satisfaction, and has recently generated considerable interest from other areas of optimization. In Constraint Programming (CP), there is naturally a desire to benefit from the techniques developed specifically for the treatment of Booleans, but also to take inspiration from the methods and ideas that were instrumental in the success of SAT solvers.

This chapter gives a brief overview of the breakthroughs related to SAT and of the recent CP work inspired from SAT. Our goal is not to give a detailed introduction to either SAT or CP; two recent handbooks will be useful references for readers interested in such introductions: [11] for SAT and [93] for CP. There is no shortage of literature on the two fields: for instance, a book chapter by [45] gives another, accessible overview of SAT; [12] give a comparative survey of SAT and CP.

F. Corblin (✉)
TIMC-IMAG, Grenoble, France
e-mail: Fabien.Corblin@imag.fr

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_13,
© Springer Science+Business Media, LLC 2011

To illustrate the modeling and problem resolution approaches in SAT and CP, we consider in this survey the task to build gene regulatory networks (GRNs), called in our specific context *GRN deciphering*, and use it as a case study. This general problem is interesting because it is representative of the novel applications of constraints in computational biology, and because approaches based on both SAT and CP have been used to tackle it, mainly in the PhD thesis of the first author. Part of this chapter is therefore based on prior work of which a more detailed presentation can be found in [23, 24]; the focus here differs from this prior work in that we mainly describe and compare the SAT and CP approaches used for the problem.

2 Boolean Constraints as Part of the Bigger Picture

Boolean constraints play an important rôle in optimization and constraint satisfaction as they are ubiquitous in applications. We distinguish between the following classes of applications:

Problems with Boolean decisions. Propositional logic plays a fundamental rôle in computer science, and in some application areas such as hardware verification, some problems are naturally expressed in a *purely Boolean* form: this happens for instance when we directly encode digital circuit to verify their equivalence. Many other problems are not purely propositional but at least part of their decisions are Boolean. For instance in a portfolio optimization problem, we may have one binary decision for each investment: whether or not it should be included in the portfolio.

Problems with a Boolean structure. By Boolean structure, we mean the combination of (non-Boolean) constraints by logical connectives such as disjunctions, negations, or implications. A simple example of Boolean structure is the constraint $x - y \geq 8 \vee y - x \geq 8$ stating that the distance between two points x and y on the real line is at least 8. Even the problems in which decisions are not Boolean often exhibit a rich Boolean structure, and their resolution involves Boolean reasoning. For instance in software verification [22, 30, 67] problems tend to be formulated as complex Boolean combinations of simple (*e.g.,* numerical) constraints: the Boolean structure encodes the control flow of the program, and the numerical relations encode the basic operations (increment, assignment of integer variables, addition, *etc.*). But disjunctions, implications, and other logical combinations of constraints are in fact present in all other areas of applications: from disjunctive scheduling and resource allocation to configuration or computational biology.

The fundamental rôle of Booleans has long been recognized in all areas of optimization and constraint satisfaction. This is true for technologies related to the whole range of problems mentioned above. Let us mention for instance:

- The practical resolution of the *SAT problem* (satisfiability of a propositional logical formula, typically in Conjunctive Normal Form) is a very well-studied problem. The solvers GRASP [74] and then Chaff [78] paved the way for a generation of highly scalable complete SAT solvers. These solvers have quickly

become a widely used and fundamental building block originally used by many verification tools: see *e.g.,* [18] for an influential early work, [17, 18, 73] for a recent overview, and [17] for a general reference on Model-Checking. But it was also soon recognized that these solvers were versatile enough to be used in other application areas, for instance, AI planning [64].

- *CP* systems have traditionally provided support for Boolean constraints (see *e.g.,* [20, 31]). Boolean decisions are a particular case of Finite-Domain variables and are naturally integrated in the constraint propagation engine used by CP tools. It is also natural to encode problems with a Boolean structure in CP thanks to *reified constraints*, which are provided by most systems: if we have a constraint C a reified version of C is the constraint $b \leftrightarrow C$ where an extra Boolean variable b captures the truth value of C; the variable b can in turn be subject to arbitrary constraints, thereby naturally expressing logical combinations of constraints. It is, however, well-known by now that this type of integration of Booleans in CP solvers performs poorly. Recent work has therefore attempted to handle Booleans in CP more effectively using the dedicated techniques proposed in SAT. We discuss some of this work in the next Section.
- In the *Operations Research* literature, Booleans have been widely studied for decades. In particular, a whole sub-field of Integer Programming is dedicated to so-called *pseudo-Boolean* variables, whose value is in $\{0, 1\}$. Linear constraints on this type of variables obviously represent a formalism that is close to SAT, and slightly more general. This formalism is also appropriate for some optimization problems, whereas SAT is restricted to constraint satisfaction. Surveys of Pseudo-Boolean optimization are, *e.g.,* [13] for a mathematical programming viewpoint and [95] for a SAT approach. Some recent work has naturally experimented with Pseudo-Boolean solvers based on modern SAT solvers, for instance [34].
  OR has focused not only on Boolean decisions but also on Boolean structure: disjunctions, implications, and other Boolean combinations of constraints are often found in OR applications. Some authors such as John Hooker have argued that logical combinations of linear constraints should in fact be at the heart of a principled approach to modeling in Mixed Integer Linear Programming [52]. A survey on Booleans in OR is beyond the scope of this chapter; we simply mention that there is a large body of results on the relaxation and inference on disjunctions and other Boolean combinations of linear constraints; see for instance [51].
- In *Theorem Proving* and Automated Reasoning, an interesting recent trend largely motivated by the rise of modern SAT solvers is Satisfiability Modulo Theories (SMT). In the classical, "purist" approach to first-order theorem proving, concepts such as the integers or the reals are described by a theory (logical axioms) rather than natively supported. In contrast, the basic idea in SMT is to use dedicated solvers for some important theories: linear arithmetic or uninterpreted functions, but also arrays, or bit-vectors, *etc.* The latter theories may appear exotic to an audience specialized in optimization but they play an important role in verification problems. Note also that the diversity of SMT theories is reminiscent of the early CLP(X) literature [56]. Although the work in CP was

overwhelmingly specialized on Finite Domain constraints during the last decade, it is good to remember that early CLP systems included constraints on such various domains as lists [21], functions [50], or strings and languages [112], as well as dedicated numerical solvers–for instance for linear constraints [65] or non-linear constraints [19] on the reals. Some of the recent work in SMT is heavily inspired from this research and is in a sense reviving it.
SAT solvers play a central role in modern SMT solvers: the whole search process is guided by the resolution of the Boolean structure of the problem, an SAT solver is used for that purpose, and this solver somehow "orchestrates" the calls to the other theory solvers. This approach is particularly suited for problems where the Boolean structure is very rich (*e.g.,* in software verification). The modern, SAT-based approach to SMT is perhaps best presented under the DPLL(X) framework [80]. A recent reference on the decision procedures used for several interesting SMT theories is [67]. The leading SMT tool is at the time of this writing the Z3 solver [28].

3 Brief Overview of SAT and its Integration in CP

In the following, we give a general but concise overview of the complementary approaches of SAT and CP with respect to a number of aspects: community, techniques, and approach to problem solving. On the way, we survey some proposals of *integration of SAT techniques* into CP. We conclude the Section by an overview of the *integration of SAT solvers* into CP solvers.

3.1 Community and Approach to Research

SAT and CP over Finite Domains are of course not fundamentally different technologies: it is striking for instance that for both of them the most widely used algorithms are backtracking techniques using inference, including forms of constraint propagation that are in essence very similar. The connections between SAT and CP have indeed been noticed for some time and have generated a substantial literature (see *e.g.,* [38, 113]). The view, expressed by some members of the CP community, that SAT is essentially a sub-field of CP, is not without justification; the vision defended by the CP-AI-OR conference and some authors (in particular [51]) is that techniques and sub-areas should be integrated and unified rather than fragmented.

However, it is also fair to observe that the SAT sub-community has been successful in the last decade largely because it has managed to follow an independent and original approach, with more focus on some techniques and practices that also exist in CP but have somehow been overlooked. One defining feature of the SAT

community is its strong emphasis on *experimental* science. Some authors, *e.g.,* [44], have also defended the view of CP as an empirical science, but it is undeniably SAT that has reached the highest standards of experimental comparison.

There are several factors to that success. First, SAT problems can be easily converted into a Conjunctive Normal Form that is very simple and regular, thus providing a standard input *format* for SAT solvers. Second, the SAT community has been able to collect a large amount of *instances* in this format and make them publicly available. These instances include, in particular, many industrial ones, which allow to gain a better understanding of "real-world" problems. Third, the SAT *competition*[1], now organized every other year, has been successful and influential, and has for the past decade provided a high-quality evaluation of the different techniques. Many solvers enter the SAT competition, because the event itself is attractive, but also because the cost of entering is not unreasonably high: the input format is simple, and competitors have been able to a large extent to reuse the code of some influential SAT solvers including Chaff [78] and more recently MiniSAT [33]. Both were distributed open-source, were fairly compact, and provided high-quality code that could be extended reasonably easily.

One outcome of this heavily empirical approach is that the mix of heuristic and techniques used in modern SAT solvers such as MiniSAT has been thoroughly evaluated and refined. Of course, these solvers do not avoid the NP-completeness barrier and remain of limited scalability for many instances, especially artificial ones (randomly generated, or hand-crafted); but their main strength lies in their ability to solve large industrial problems in a surprisingly robust and efficient way. In comparison, CP solvers lack robustness and are more difficult to use (the default heuristic of a CP solver rarely does the job), which is seen as a restriction to its widespread acceptance [87]. It has also been argued by [87] that unlike SAT with its competition, CP lacks good indicators for measuring the progress achieved in the field. Recent attempts have nevertheless been made to improve this with, in particular, a proposal of XML format for CSP instances and a CSP competition based on this format [94], and the proposal of Zinc [75], whose ambition is to become a standard language and which is already used by solvers such as Gecode [99]. Agreement on standards is nonetheless reputedly slow in the CP community. One reason is that CP uses a richer language than SAT by nature–although it must be said that the SMT community has the same problem and yet has rapidly been able to agree on a format, a library[2] and a competition. Another, perhaps more fundamental reason is that the CP approach is not in general a *black-box* approach in which a problem is simply described using some format and solved. We come back to this point later on.

[1] `www.satcompetition.org`.

[2] www.smtlib.org.

3.2 Techniques Used in SAT and CP

Modern SAT solvers are often called DPLL solvers, after the Davis–Putnam–Logemann–Loveland procedure originally proposed for first-order theorem-proving [26, 27]. More specifically the procedure of [27] is a form of first-order resolution method, while the one in [26] is closer to search-based methods. DPLL solvers are of course restricted to the propositional case. They are essentially (depth-first) search-based solvers: in other words they use *backtracking*, just like CP solvers; but they also use forms of resolution. The main components of these modern algorithms are very close to the ones used in mainstream CP solvers such as Ilog Solver [86] or Gecode [99]:

3.2.1 Propagation

The basic form of reasoning done both in SAT and CP tools is propagation. SAT solvers specialize in one type of constraints: clauses. A clause is a disjunction of literals $l_1 \vee \cdots \vee l_m$ (each literal is a variable or its negation), and the role of propagation is to efficiently detect when all literals but one become false, in which case the remaining one must be forced to true. The technique universally[3] used for that purpose in SAT solvers is the *watching* technique introduced with the solver Chaff [78]. For each clause, the basic observation is the following: if we arbitrarily choose two literals of the clause, then as long as those literals are not invalidated nothing can be propagated from the clause. The solver maintains a *dynamic* list of watching clauses for each literal; when a literal is falsified we only need to revise the clauses that are currently watching it (note that they are in general only a *subset* of the clauses in which the literal appears): two new correct (non-invalidated) watched literals are selected or, if such literals cannot be found, a unit clause or conflict is detected. A key feature of the watching technique is that the number of dependencies of each clause is independent of the clause length (we always watch *two* literals), whereas with propagation methods based on static dependency lists we have a number of dependencies equal to the clause length, which do not scale well for large clauses. Another interesting characteristic is that no work is needed to update the watches on backtrack, whereas the data-structures used by most propagation algorithms in CP (*e.g.,* AC algorithms like [6]) need to be backtrackable.

In contrast, in CP propagation is traditionally based on *static* dependency lists. However, watching techniques inspired from SAT have generated some recent CP work: a simple example is given by the *Element* constraint, which is for the form $T[I] = X$, where T is an array of Finite-Domain variables, and I and X are two Finite-Domain variables. [40] remark that the watching technique naturally applies for this constraint: for instance, any value v of the domain of I is supported as long

[3] We have to note, however, that several components of Chaff are being patented and that this may restrict, in particular, the use of watching, at least in industrial tools and applications.

as we have a value w that is common to the domains of X and $T[v]$. Once such a w is found, watches can be used to avoid any work unless w is removed from the domain of either X or $T[v]$. Watching has also been used for other complex constraints including *AllDifferent* [41].

A thorough introduction to the general propagation mechanisms used in CP solvers can be found in [98]. More details on the propagation algorithms for a number of constraints and on the key notion of arc-consistency can also be found in [7].

3.2.2 Execution Tracing and Analysis

Under this slightly unconventional term, we mean that in SAT whenever a literal l is deduced by propagation the solver keeps track of the *reason* of this deduction, *i.e.,* it records the constraint $r(l)$ from which l was deduced. Note that the reason that is recorded is really execution-dependent: it may be the case that l is in fact a logical consequence of several other clauses, but the custom approach is simply to keep track of the one clause that happens to be selected first during the propagation. Note also that $r(l)$ is necessarily of the form $l_1 \vee \cdots \vee l_p \vee l$ where all the other literals l_i are falsified. The data-structure obtained by this book-keeping is usually referred to as the *implication graph*: each literal that is valid in the current context is seen as a node and each clause is in this context seen an oriented (hyper-)arc from $\neg l_1, \cdots, \neg l_p$ to l. The definition of implication graph comes from GRASP [74]; the idea of execution tracing has also been investigated in CP, notably under the name *explanation* [60].

Execution tracing in SAT solvers is used to understand conflicts: when the sequence of decisions and propagations hits a contradiction ("leaf" of the search tree), it is in general not the case that all decisions and propagated literals play a role in this conflict. *Conflict analysis* inspects the trace (implication graph), collects the literals that really play a part in the conflict, and feeds this knowledge to the propagation, heuristic and backtracking components. In SAT, the computation of *first UIP* introduced by [74] has become, since Chaff [117], the standard conflict analysis technique. Some solvers such as MiniSAT [33] include improvements that further simplify the explanations computed by the first UIP scheme, but are nonetheless essentially based on it. The idea of first UIP is to "replay" the execution backward starting from the conflict, to collect the literals that justify the conflict, and to stop as soon as we are left with exactly one literal from the current decision level (first *Unit Implication Point*). We would not describe conflict analysis in details, the reader is referred to [45, 74, 117]. There are three ways in which conflict analysis benefits the search: *clause learning*, *heuristic tuning*, and *non-chronological backtracking*.

- *Clause learning* means that when a conflict is hit, a constraint (clause) is dynamically created which will increase the power of constraint propagation (a property which is called *empowerment* by [82]). The usage of learnt clauses is evaluated during the search and clauses that are not used enough are erased after some time, to keep propagation fast. With clause learning SAT solvers have the ability to produce fairly general resolvents: in a sense modern SAT solvers, although

search-based, have most of the reasoning power of purely deductive, resolution-based SAT solvers. Recent results by [83] shed light on that aspect and show that DPLL solvers are effectively able to produce general resolution proofs.
Learning techniques are also used in CP under the name of no-good recording techniques, see for instance [62, 97]. Note also that resolution in SAT is a particular case of the Cutting Plane generation techniques used in the Operations Research literature for linear constraints [51, 71, 72].

- *Heuristics* in SAT rely on the idea of giving higher priority to the variables that are involved in conflicts; such heuristics are called *activity-based*. An influential activity-based heuristic was the VSIDS heuristic, proposed in Chaff [78], and which is still used with some variations in most SAT solvers. VSIDS associates scores (or activities) to each variable; the branching strategy always selects one of the variables with the highest score (in solvers such as MiniSAT the ordering can be maintained efficiently using data-structures such as heaps [33]); when a conflict is met the scores of the variables involved in it are bumped; scores are also regularly decayed, which means that the search will tend to branch on variables involved in recent conflicts. Since most SAT solvers use random restarts [46], also used in CP, it is interesting to note that activity-based heuristics mix well with restarts: when the solver restarts the heuristic information has been updated by the last conflicts, and the solver has a higher chance to focus on "important" variables, for instance, those that form a "backdoor" [115]. A recent improvement of the branching strategy used in SAT solvers is the caching strategy of [81], which can be seen in CP terms as a value selection heuristic.
  In CP, activity-based heuristics have been proposed recently, notably the DomWdeg heuristic of [14], which is inspired from VSIDS and is considered a good default heuristic used by several CP solvers. Another example of activity-based CP heuristic is the Impact heuristic of [89], which is inspired from Integer Programming techniques.
- *Non-chronological backtracking* means that after a conflict SAT solvers do not simply undo the last decisions but instead backtrack "intelligently" based on the conflict analysis, using the notion of *asserting clause* [45, 74]. In particular, if the conflict analysis revealed that a number of recent decisions are unrelated to the conflict, the backtracking will be able to jump over those decisions, thereby avoiding superfluous branchings were the same conflict would happen again.
  Similar techniques have been experimented with in CP, see for instance [43, 70, 85].

As we have seen, there have been attempts to integrate most of the SAT techniques in CP. An early reference on the use of several similar techniques in CSP is for instance [29]. These techniques have, however, not often become part of mainstream CP solvers. Non-chronological backtracking is to our knowledge not very widely used, perhaps because of the granularity of the representation in CP: the fine-grained representation used in SAT is appropriate for a fine-grained analysis of conflicts, whereas in CP complex constraints sometimes make this more difficult. No-good recording is to the best of our knowledge not very widely used either: one challenge is that adding constraints dynamically in CP is costly, and no-goods

are not as natural to represent in CP as they are in SAT (where a no-good is a clause, which happens to be the type of constraint naturally handled). Execution tracing is in general costly in CP, especially with variables with large domains–in contrast in SAT, every variable is affected by propagation at most once along a search branch; the data-structures needed for book-keeping are simple and of linear size. Activity-based heuristics are perhaps the most successful example of an SAT-inspired technique widely adopted by CP, as they are used in many CP solvers including Abscon [68] and Gecode [99].

We have focussed on *complete*, mainly search-based, SAT and CP techniques. We should mention that other techniques are of interest, for instance, local search techniques. These techniques typically work by generating candidate assignments, for instance at random, and making moves in the assignment space, for instance by changing the value of some variables, guided by the variation in the quality of the assignment. Local search methods are *incomplete*: they will often ultimately be probabilistically guaranteed to find a solution if one exists, but they are not in general able to prove the unsatisfiability of a problem. Until the generation of modern DPLL solvers, local search techniques such as GSAT, WalkSat and its successors [77, 100, 101] were the state of the art in SAT (see [63] for a recent survey of local search in SAT). However, the SAT competition has consistently shown that local search techniques are ineffective for "structured," industrial instances, and currently these methods are essentially competitive for artificial problems such as randomly generated ones. In CP, local search is more widely used because of its ability to solve optimization problems–non-globally but sometimes in a spectacularly scalable way. Comet is an influential recent CP system that integrates local search techniques [109].

3.3 Problem Solving in SAT and CP

We consider how the techniques introduced in the previous sub-section are used to model and solve problems in practical applications of SAT and CP.

- *Modeling.* The input of SAT solvers is a low-level machine-readable format that humans cannot directly use to write real problems; instead people typically use a pre-processor that generates the instances from some other application data. This is in contrast with the high-level approach of CP, whose main strength is global constraints [5, 91, 111] and where modeling languages such as OPL [108] allow to express problems at a very high level, thereby making their structure more explicit.
  A consequence is that SAT, like Mixed Integer Linear Programming, is a technology that is good at solving *large numbers of small constraints*. In contrast CP is good for problems that can be modeled by a *small number of complex constraints*, assuming good propagators are available for these constraints. Traditionally, CP is good for integrating complex, application-specific algorithms, for instance, specialized propagators for scheduling [4]; but it is a weak technology

for reasoning on large numbers of Booleans or linear constraints: the reasoning on Booleans is poor and slow in the absence of dedicated SAT techniques; the reasoning on linear constraints is too local without linear relaxation and dedicated Linear Programming techniques. Of course, a major goal of the CP-AI-OR conference along the past decade has, precisely, been to improve this by combining CP, SAT, and OR techniques.

- *Use of the Solvers.* Another important feature of SAT solvers is that they are *black-box* solvers : users encode a problem and the solver solves it; no heuristic tuning is in general needed (or indeed possible). In contrast, CP libraries and languages rarely obtain a good performance with their default heuristic, and the user is rather asked to use the CP tool to write a dedicated strategy using, for instance, a dedicated programming language [108, 110].
  A consequence is that SAT is a remarkably *autonomous* technology, in the sense discussed in [49]: it is able to solve problems without any intervention from the user. Some members of the CP community believe that the problems that can be solved without help by an SAT solver are problems that are reasonably *easy*–for the more challenging problems, the CP approach requiring the intervention of an expert is often unavoidable. If this is true, the ability of SAT to *solve easy problems easily* is nonetheless a valuable asset: it results in an *ease of use* of the technology for solving the majority of problems; it has been noticed [87] that this ease of use is something that CP lacks.

3.4 Integration of SAT Solvers into CP Solvers

The advantages of using SAT solvers within CP solvers are obvious, as CP solvers can thereby benefit from a better handling of many small constraints (especially if these are almost directly expressed in a Boolean form). We distinguish mainly between two approaches to the use of SAT in CP in the recent literature:

3.4.1 Use of SAT to Solve the Boolean Structure

The most obvious lesson from SAT for CP is that Boolean variables can be better handled in CP by using specialized code and data-structures, rather than by treating them as normal Finite-Domain variables. Improving the handling of Booleans was an original motivation behind the work on the Minion solver [39], which proposed to rethink some traditional assumptions of the design of CP solvers, such as the data-structures used for backtracking. Although no literature is, to our knowledge, published on this aspect, our understanding is that other CP solvers including Comet [109] and Ilog CP optimizer are now using similar ideas.

Further inspiration for improving the way Boolean structure is solved in CP solvers can be taken from SMT. Work has been devoted to reasoning on disjunctions and other Boolean combinations of constraints in CP [3, 69], but these techniques

are not widely used in mainstream CP solvers. Work in SMT has been influenced by the Nelson-Oppen procedure [79] for theory integration and what would be called *solver cooperation* in the CP community. But current SMT solvers are in general SAT-based, and they probably represent the most advanced example of integration of SAT and other constraint solving techniques. In the DPLL(X) schema of [80], the SAT solver is central and is essentially used to solve the Boolean structure of the problem; some Boolean variables essentially correspond to the truth value of some constraints, and when an SAT solution is found, the solver for the theory X is called. A noticeable feature of the DPLL(X) framework is that the interaction between the SAT solver and the theory solvers is rich: the theory solver can be used to check the validity of the constraints, to make some inference (theory propagation), but also to communicate some information back to the SAT solver, mainly in the form of *explanations* of the conflicts. Note that this architecture is the reminiscent of Bender's decomposition techniques familiar to the CP-AI-OR community (see *e.g.,* [51, 54]): the SAT solver plays the role of the master; the solver for theory X is a slave, communication between the slave and the master helps guiding the master when conflicts happen. Another example of use of SAT to solve the Boolean structure of a mixed problem is [36], in which the rest of the constraints are real non-linear.

A particularly relevant theory for CP is Satisfiability Modulo the Theory of *Linear Integer Arithmetic*. It focuses on Boolean combinations of linear constraints, also called *Quantifier-Free Presburger Arithmetic* in the literature. This logical theory has received some attention in the verification literature [66], and state-of-the-art simplex-based techniques used by SMT solvers for (real and integer) linear constraints are described in [32]. It is interesting to note that Quantifier-Free Presburger Arithmetic is a natural logical fragment for expressing combinatorial problems.[4] For instance, most global constraints are defined naturally as Boolean combinations of linear constraints. In fact, the core constraint language of some early CLP(FD) solvers such as GNU-Prolog [31] was mainly Boolean combinations of linear constraints. Quantifier-Free Presburger Arithmetic has also received recent attention in CP, for instance in applications to geometrical reasoning [16].

3.4.2 Translations to SAT

The more radical approach for using SAT is to translate constraints into SAT: the SAT tool solves not only the Boolean structure of a problem, but the whole of it. Encodings of constraints into SAT have been studied for some time, for instance,

[4] There is one difference between combinatorial applications and verification, though: in CP, the variables typically have small domains, whereas in verification and theorem proving the goal is in general to prove properties in Quantifier-Free Presburger Arithmetics for numbers of *arbitrary sizes*. Like for Integer Programming, *small domain properties* can nevertheless be proven [102], so that the decision problem can always be reduced to finite domains. These bounds are, however, often very large, compared to the domains used in classical CP applications.

with the support encoding of binary constraints by [61]; see [84] for a survey. But this subject has received increased attention in recent years, leading to interesting results.

A key CP concept is the notion of *(generalized) arc-consistency* (GAC) (see [7] for a survey of this notion): given a constraint C over n variables $(x_1, \ldots, x_n)$, let $Sol(C)$ represent the set of solutions to C, *i.e.,* the set of tuples $(v_1, \ldots, v_n)$ that satisfy the relation C; a value a of the domain of any variable x_i is said to be *supported* iff there is a tuple $t \in Sol(X)$ that assigns value a to x_i; the domains of all variables are *arc-consistent w.r.t.* C if for each i, each value a in the domain of x_i is supported. Arc-consistency (more precisely the *largest* arc-consistent domains) characterizes the *optimal* domain reduction that can be obtained for the constraint C. An obvious observation is that when we decompose a constraint into more basic constraints–for instance, into Boolean constraints in Conjunctive Normal Form–reaching arc-consistency on the decomposition is often weaker than arc-consistency on the original constraint. As a simple example consider:

$$\mathit{all\_different}(x_1, x_2, x_3) \qquad x_1, x_2, x_3 \in \{0, 1\}$$

One possible SAT translation is the following: we introduce one variable for each value of each domain: one Boolean variable for $[x_1 = 0]$ and one for $[x_1 = 1]$, *etc.* We add constraints to guarantee that each x_i takes exactly one value:

$$\begin{gathered}([x_1 = 0] \vee [x_1 = 1]), (\neg[x_1 = 0] \vee \neg[x_1 = 1]),\\ ([x_2 = 0] \vee [x_2 = 1]), (\neg[x_2 = 0] \vee \neg[x_2 = 1]),\\ ([x_3 = 0] \vee [x_3 = 1]), (\neg[x_3 = 0] \vee \neg[x_3 = 1])\end{gathered}$$

Then, the constraint $\mathit{all\_different}(x_1, x_2, x_3)$ can be decomposed by preventing two variables from taking the same value at the same time, as follows:

$$\begin{gathered}(\neg[x_0 = 0] \vee \neg[x_1 = 0]), (\neg[x_0 = 1] \vee \neg[x_1 = 1]),\\ (\neg[x_0 = 0] \vee \neg[x_2 = 0]), (\neg[x_0 = 1] \vee \neg[x_2 = 1]),\\ (\neg[x_1 = 0] \vee \neg[x_2 = 0]), (\neg[x_1 = 1] \vee \neg[x_2 = 1])\end{gathered}$$

However, this encoding does not allow a good propagation for the SAT solver: if we propagate all the clauses by unit propagation (thereby reaching GAC for each clause), then on this example we obtain no deduction at all. In contrast, GAC on the original constraint should remove all values for all domains, proving inconsistency.

We say that an SAT encoding of a constraint is *GAC-preserving* if unit propagation on the encoding makes the same deductions as a GAC propagator on the original constraint. A major theme in the study of translations from CP into SAT has been the question: *How do we encode constraints to SAT in a GAC-preserving way?* A number of results have recently been obtained on this question. An important class of relations that can be decomposed in a GAC-preserving way are all relations that can be expressed concisely in the Decomposable Negation Normal Form (DNNF) introduced by [25]. The decomposition is discussed, *e.g.,* in [59]; it is also used indirectly

by, *e.g.,* [88], who study the decomposition of grammar constraints. Grammar constraints are an example of constraints that have connections with DNNF[5] and for which an efficient GAC-preserving encoding can be found. [88] use such an encoding for a nurse scheduling problem and show that a pseudo-Boolean solver applied to the decomposition gives very good results.

Other results on GAC-preserving translations have been obtained for instance by [2] for the Table, Regular, Among, and Sequence constraints. It is shown in [9] that for AllDifferent and related constraints a decomposition can be obtained that computes weaker forms of consistency than GAC, namely *bound* and *range* consistency. By adding redundant clauses (or creating those clauses automatically by resolution as shown in [107]), one can reach a stronger global level of consistency than GAC; this idea has been explored for some table constraints by, *e.g.,* [8].

The question, *Which constraints can efficiently be decomposed in a GAC-preserving way?* has only been answered recently by [10]. It had been observed that some propagators, including the well-known GAC propagator for AllDifferent of [90], seemed difficult to decompose into SAT. [10] prove that a concise GAC-preserving decomposition of AllDifferent is indeed impossible. Their result gives a precise characterization of the constraints that can be efficiently decomposed into SAT: a concise GAC-preserving CNF encoding exists only if the GAC property of the constraint can be verified by a concise *monotone* circuit. Results from circuit complexity can therefore be used to show that some constraints are intrinsically non-decomposable as they would require an SAT encoding of exponential size (to this date, AllDifferent is the most significant known example). Compact SAT encodings of these constraints are of course still possible, but they will in general lead to a weaker form of reasoning than GAC.

There are a number of reasons why *decomposing constraints* (into SAT, linear constraints, or a combination of the two) is interesting, and why the results we have just surveyed briefly are therefore exciting.

- First, this approach is simply performant in some cases, as some examples will show in the rest of this paper with our case study. We also note that SAT is one of the possible target solvers that can be used as back-end in systems such as G12 [103], and that other modeling systems in the AI literature are based on a compilation to SAT, see for instance [15] or the literature on Answer Set Programming [37]. Experiments on the systematic encoding of CP problems to SAT are reported by [55].
- Second as noticed by some authors, decomposing constraints in a CP system has some advantages compared to writing a propagator. It is simpler: there is

[5] Similar results on the decomposition of grammars were obtained for another class of graphical models, in probabilistic reasoning: [76] show how the membership problem for context-free grammars can be encoded in a formalism called *Case Factor Diagrams* that can essentially be thought of as the probabilistic equivalent to DNNF; probabilistic reasoning by a simple propagation on these diagrams is shown to be equivalent to global reasoning.

less code to write and maintain as we can rely on a basic, small number of propagators. The incrementality of the propagator is also often guaranteed by the incrementality of the Boolean reasoning. It is interesting to note that the internals of the propagators defined for some global constraints, for instance, the Element propagator of [40], can to a large extent be seen as reimplementing incremental data-structures that more or less mimic the effects which would naturally be obtained by a decomposition into SAT.

- Third, an encoding is something that solvers can reason with more easily than a propagator: an encoding gives a complete view of the internals of the constraint to SAT or LP tools, whereas a propagator is a black box. A large set of tools are directly applicable on a decomposition: (1) conflict analysis can naturally be done (whereas analysing conflicts of global constraints is reputedly challenging); (2) inference techniques such as resolution or cutting plane generation can apply on the variables and constraints introduced by the decomposition, which describe the internals of the constraint; (3) there is a natural notion of linear relaxation that directly applies to an SAT decomposition, whereas defining the interaction between propagators and an LP solver can otherwise be challenging.

An appealing perspective emerging from the results we have surveyed and the view of many researchers in the CP-AI-OR community is that of using CP as a high-level approach for modeling, and a mix of other technologies such as SAT (but also Mixed Integer Linear Programming) for the actual resolution. This approach is to a large extent the one we follow in our case study in the rest of this paper. What the high-level language of CP brings is a modeling framework that is easier from a user viewpoint and which at the same time better reflects the problem structure than low-level languages do, since it allows to explicitly use some patterns or specific high-level (global) constraints. Some recent solvers that follow this direction are SCIP [1] and SIMPL [116].

Because of the *structure-rich* CP encoding, the translation SAT and/or linear programming tools can be optimized and take into consideration the level of propagation obtained. From the results we have surveyed emerges a better understanding of the question: *What is a good SAT encoding?* This question has somehow been overlooked by the SAT community and its answer is greatly clarified by the CP notion of arc-consistency. Note that the art of encoding constraints into SAT in a way that is GAC-preserving has a parallel in Linear Programming: in LP, a good encoding of a constraint is ideally a Convex Hull Relaxation.[6] For literature about Convex-Hull (and weaker) encodings of some constraints, see for instance [51,114]. We also note that [53] goes slightly further than we did in this survey by defining a "good SAT encoding" as one that (1) is GAC-preserving and (2) for which a Convex Hull Relaxation can be obtained.

[6] A Convex Hull Relaxation is the tightest possible linear relaxation that can be obtained for a constraint [51].

4 Application to GRN Deciphering

We now switch to the fast-growing field of systems biology for our case study. This area is a rich source of various types of problems where constraint solving can be very efficient. In recent years, it has become increasingly clear that the right concept to understand the inner working of cells is the concept of network. It is quite rare that an observable property (phenotypic trait) can be explained as the result of the action of a single protein. The general rule is that phenotypes are the result of the action of networks involving many actors interacting one with the other. The interest is thus shifting, with the rise of systems biology, from the study of single molecular objects (genes and proteins) to the study of networks of interacting molecules and genes.

A GRN is a particular kind of such network in which each gene is able to produce a specific regulatory protein at a rate which depends on the cellular context, defined by the concentrations of proteins. In other words, the production rate of a given protein depends on the quantities of proteins in the system. A single gene can be influenced by several proteins in the system, including the protein produced by the single gene itself. The set of all the influences between genes, mediated by the proteins produced, can be represented by an interaction graph. The proteins are generally abstracted away and one speaks of interaction between genes.

These molecular networks have complex behavior because the interactions are highly non-linear and because there are generally many feedback loops. This makes human reasoning on such systems impossible for anything but the smallest networks (two or three nodes), and calls for automated formal methods.

Another aspect is that the data often have a qualitative flavor. It may be sufficient to represent a protein concentration by two (low/high) or three (low/medium/high) levels. The number of levels required rarely exceeds four or five. The formalism which is described in Sect. 4.1 is well suited to the level of knowledge which is generally available in systems biology. All the state variables, as well as all the model parameters, are integers, and simple rules define the evolution of the system.

Modeling is a part of a discovery process which is generally composed of many rounds of experimentation and modeling. The basic question addressed here is how to build a model of a biological phenomenon, based on some partial knowledge about the network architecture and behavior? In this field, an originality of our work is that everything is represented as constraints: the generic evolution rules, the architecture of the particular network at hand, and the knowledge on its behavior.

We describe here how we exploit constraint-based technologies to assist the process of building models of molecular networks using experimental data as constraints. We have developed a tool, GNBox, which provides high-level functionalities to biologists engaged in the modeling process. Among these:

- Proof of inconsistency between the assumed architecture and the behavior
- Inference of model parameters

- Automatic model revision
- Inference of properties (about behaviors, kinetics of reactions, thresholds) shared by all solutions

The GRN deciphering task consists to a use these type of queries in order to build efficiently a GRN.

GNBox makes use of two technologies: CP and SAT. The network evolution rules and the queries describing the above functionalities are first expressed in CP. To increase the efficiency, a translation scheme to SAT has been devised.

Section 4.1 introduces the notion of GRN, describes succintly a particular biological phenomenon, and presents the discrete formalism of R. Thomas which is used here to describe the dynamics of GRNs. This formalism is particularly well suited to a qualitative reasoning on GRNs. We name Thomas-GRNs the GRNs whose dynamics are described by R. Thomas formalism. We explain in Sect. 4.2 how the evolution rules of Thomas-GRNs are modeled in CLP, and how this CLP formulation is translated into SAT. We present in Sect. 4.3 five examples of queries to show the possible tasks of GRN deciphering. Finally, we discuss the performances with CP and SAT solvers and give related works and perspectives of the GRNs deciphering task in Sect. 4.4.

4.1 Thomas-GRNs

A GRN abstracts the *interactions* between several *genes* of a cell. We use in this article, as example, a specific biological phenomenon, the immunity control by the lambda-phage in the bacteria *E. coli*. The GRN involved in this phenomenon, which we now call λGRN for short, has been modeled by [104].

An interaction can be either an *activation* or an *inhibition*. For instance, in the λGRN, the gene *cI* inhibits gene *cII*: it means that it is possible that the concentration of protein *cI* being low (under a certain threshold), the concentration of protein *cII* tends to increase, while it is not the case if *cI* is high (above this threshold). Remember that each gene produces a protein which is specific of this gene. In simple systems like the one considered here, the correspondance between genes and proteins is one to one. We will here make an abuse of language by not always distinguishing genes and proteins. But strictly speaking, a given gene is present in one or a few copies in each cell, whereas regulatory proteins are diffusable molecules whose amounts are defined by concentrations.

In many biological interactions the influence of one gene on another can be represented as a sigmoid function: there is a threshold in the concentration of the protein at which the effect on the production of the target protein changes steeply from efficient to inefficient. In a discrete modeling, sigmoids are approximated by step functions.

The interactions between the genes are traditionally represented by an *interaction graph*: see Fig. 1 for the interaction graph of the λGRN example. In this graph, the

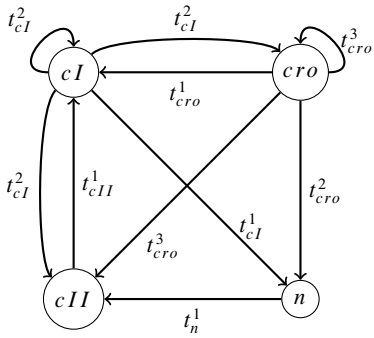


Fig. 1 Interaction graph for the λGRN system. The nodes are the genes of the λGRN. The arrows are interactions, where the label express the threshold t_c^p from which the interaction is effective

nodes represent the genes. An arrow from a node a to a node b models the fact that the concentration of a can influence the production rate of b. Each arrow carries one label: a threshold at which the interaction changes its activity status. These thresholds, noted t_c^p, with c the component (gene) and p the index of a threshold on c, are model parameters. The integer values of the threshold parameters define the order between them, which have sometimes been determined experimentally. In the λGRN, for the gene *cI*, t^1_{cI} is the threshold for the interaction on N and t^2_{cI} is the threshold for the three interactions on *cI* (itself), *cro*, and *cII* (cf. Fig. 1). The order between these two thresholds is assumed to be known and is the following: $t^1_{cI} < t^2_{cI}$. For the gene *cro*, we have: $t^1_{cro} < t^2_{cro} < t^3_{cro}$.

We present informally in the following the asynchronous logical description created by the biologist R. Thomas and his collaborators [106], which relates the interaction graph of a GRN and its dynamical behavior. The main goal of this formalism is to obtain a qualitative understanding of the network dynamics by reasoning on discrete entities. This formalism has been proposed from the beginning of GRN studies [105]. It has largely been applied to the analysis of GRNs, for example, those described in [35, 47, 96, 104] or those described in [57, 58, 92], which use a very similar discrete formalism.

It can be described as follows:

1. The formalism is purely discrete:
   - The thresholds t_c^p take discrete values between 1 and max_c. In the λGRN, for the species *cI* as we have $t^1_{cI} < t^2_{cI}$, then $t^1_{cI} = 1$ and $t^2_{cI} = 2$, and $max_{cI} = 2$. In the general case, note that t_c^p and max_c are not known.
   - The concentration of each protein produced by the gene c in a state (see below) S is modeled by a discrete variable, noted S_c. S_c ranges over $0..\max_c$. For the λGRN, we obtain 3, 4, 2, and 2 discrete values for the concentrations of *cI*, *cro*, *cII*, and *N*, respectively.
   - A *discrete concentration state* S, or just *state*, of the system is represented by an ordered list of discrete values, each value representing the concentration of a protein. For instance, a possible state S of the λGRN is $S = \langle S_{cI} = 0, S_{cro} = 1, S_{cII} = 0, S_N = 0\rangle$, or $S = \langle 0, 1, 0, 0\rangle$ for short. This state

is interpreted as follows: the concentration of proteins *cI*, *cII*, and *N* are all below their lowest threshold, while the concentration of *cro* is between its first and second thresholds. In the λGRN, we have $3 * 4 * 2 * 2 = 48$ different states.

2. The dynamics of a Thomas-GRN is the set of all the possible successions of states according to the *transition rule*. This rule is based on the following notions:
   a. *Continuity*: it is clear that the concentrations evolve continuously. Consequently, a successor of a given state S is necessarily a state adjacent to S.
   b. *Tendency*: in a given state, the system can be thought of as tending to evolve toward a state, called *focal state*. The complex mix of influences between the genes of the system is therefore reduced to abstractions of the form "in state $\langle 0, 1, 0, 0\rangle$, the system tends to evolve toward the focal state $\langle 2, 0, 0, 0\rangle$". The focal state of a state S is noted F_S and the value of each of its components c is denoted $F_{c,S}$.
   c. *Asynchronicity*: in a given transition, the system cannot cross two or more thresholds simultaneously.

   Then, the transition rule can be expressed as follows:

   - If the system's current state S is different from its focal state F_S, then the concentration of only one of its components c will change, by one unit, in the direction indicated by $F_{c,S}$. This is done independently for each component so that a state S can have several successors. We thus obtain a non-deterministic transition system.
   - If the system's current state is equal to its focal state, the system is said to be in a *steady state* and no concentration changes.

   For instance, if the state $\langle 0, 1, 0, 0\rangle$ has $\langle 2, 0, 0, 0\rangle$ as focal state, then the set of possible successors is constituted of $\langle 1, 1, 0, 0\rangle$ (move along the first dimension toward value 2) and $\langle 0, 0, 0, 0\rangle$ (second dimension, towards value 0).
3. This leaves the question of how each component $F_{c,S}$ of the focal state of a state S is defined. This definition is by case: component c is influenced by a number of other components with associated thresholds. These influences split the concentration space in several regions called *cellular contexts* of c. For instance, the gene *cro* is influenced by *cI* with threshold t^2_{cI} and *cro* (itself) with threshold t^3_{cro}; depending on the current discrete values of S_{cI} and S_{cro}, the tendency $F_{cro,S}$ can take different values. We obtain four exclusive possible cases for $F_{cro,S}$ according to the membership of S to one of the cellular context of *cro*:

$$F_{cro,S} = \begin{cases} K^1_{cro} \text{ if } S_{cI} < t^2_{cI} \wedge S_{cro} < t^3_{cro} \\ K^2_{cro} \text{ if } S_{cI} < t^2_{cI} \wedge S_{cro} \geq t^3_{cro} \\ K^3_{cro} \text{ if } S_{cI} \geq t^2_{cI} \wedge S_{cro} < t^3_{cro} \\ K^4_{cro} \text{ if } S_{cI} \geq t^2_{cI} \wedge S_{cro} \geq t^3_{cro} \end{cases} \quad (1)$$

The complete focal equation system for λGRN is given in Appendix 1. The parameters K_c^l (where l is the index of the corresponding cellular context and c is the component) are called *discrete kinetic parameters* (they characterize the kinetics of the system). These parameters are in general not known (as the parameters $\max_c$ and t_c^p). There are, however, constraints on these parameters that are imposed by additional hypotheses about composition of interactions and properties on these compositions. For instance, we add a hypothesis about the observation of inhibition of *cI* on *cro* according to the threshold t_{cI}^2. This hypothesis is modeled by the following *observability* constraint: $K_{cro}^3 < K_{cro}^1 \vee K_{cro}^4 < K_{cro}^2$. In the same manner, the observation of the inhibition of *cro* on itself according to the threshold t_{cro}^3 is modeled by the constraint: $K_{cro}^2 < K_{cro}^1 \vee K_{cro}^4 < K_{cro}^3$. We add also *additivity* constraints which define the composition function of several interactions on a same gene like a sum. The considered constraints in the case of λGRN are given in Appendix 2. For instance, for the two inhibitions (discussed previously) influencing *cro*, we add the following constraints: $K_{cro}^3 \leq K_{cro}^1 \wedge K_{cro}^4 \leq K_{cro}^2$ and $K_{cro}^2 \leq K_{cro}^1 \wedge K_{cro}^4 \leq K_{cro}^3$. Note that we could left undefined this composition function or define it in another manner.

Note that the transition rule (continuity, tendency, and asynchronicity) permits to represent in intension a transition system using the concept of focal state. Such a transition system is constructed using constraints in Sect. 4.2.1.

4.2 Thomas-GRNs Formalization in CLP and SAT

We model the transition rule using a high-level description in terms of multi-valued variables and arbitrary numerical/Boolean constraints in Sect. 4.2.1 to use finite domain solvers. We explain briefly in Sect. 4.2.2 how this formalization in CLP is translated into a Boolean encoding in order to use SAT solvers.

4.2.1 The CLP Encoding of Thomas-GRNs

Let a Thomas-GRN with n components c_i, $i \in 1..n$, whose discrete concentrations take values in $0..\max_{c_i}$ (there are $\max_{c_i}$ interaction from c_i). A model M of this Thomas-GRN is a couple formed by a representation of the focal equation system (in term of cellular contexts and discrete kinetic parameters, see Definition 2 and Constraint 4.2) and a set of additional constraints. The key point is the encoding of the relation between a state $S = \langle S_{c_1}, ..., S_{c_n} \rangle$ and its possible successors $S' = \langle S'_{c_1}, ..., S'_{c_n} \rangle$ for the model M by using variables relative to the focal state $F_S = \langle F_{c_1,S}, ..., F_{c_n,S} \rangle$. The predicate $successor(S, S')$ is true if and only if S' is a possible successor of S for the model M is given in Constraint 4.1. It is

defined with the help of two predicates, on the one hand $focal\_state(M, S, F_S)$ true if and only if F_S is the focal state of S for the model M, on the other hand $transition_{rule}(S, S', F_S)$ true if and only if the transition $S \twoheadrightarrow S'$ complies with the three points a, b, and c given in point 2 of the Sect. 4.1 with F_S the focal state of S.

Constraint 4.1

$$successor(M, S, S') \overset{\text{def}}{\Leftrightarrow} focal\_state(M, S, F_S) \wedge transition\_rule(S, S', F_S)$$

To define $focal\_state(M, S, F_S)$ by Constraint 4.2, we must define the interactions and the resulting cellular contexts for each component of the GRN.

Definition 1. An *interaction* is a triplet $(c', t^p_{c'}, c)$ associated to an edge from node c' to node c in the interaction graph and labeled by $t^p_{c'}$. Each interaction $(c', t^p_{c'}, c)$ onto the target c is associated to a unique index r, $1 \leq r \leq r_c$, where r_c is the number of edges with target c in the interaction graph. We note i^r_c the interaction onto c with index r.

The cellular contexts of each component c are hyper-rectangle regions of the concentration space induced by the set of interactions onto gene c for the model M, $\{i^r_c | 1 \leq r \leq r_c\}$. More precisely, each interaction $(c', t^p_{c'}, c)$ is associated with a threshold $t^p_{c'}$ which defines a hyperplane partitioning the concentration space in two regions. Thus, the r_c influences onto c induce a partition of the space in $l_c = 2^{r_c}$ regions. Indeed, for the viewpoint of c, all the states in a same cellular context of c are on the same influence. Before turning to the formal definition of this notion (Definition 2 below), note that a cellular context is fully specified by the data of r_c Booleans (one for each threshold) specifying the position of this cellular context regarding each threshold (0 if the region is below the threshold, 1 if it is above).

Definition 2. Let c a component with r_c interactions onto it.
Let $V = (V_1, ..., V_r, ..., V_{r_c})$ a list of r_c Booleans. Let l the integer such as $l - 1$ is equal to the decimal value of the binary number composed of component of V: for example, $V = (1, 0)$ corresponds to $l = 3$. The *cellular context* of c characterized by V is the region composed of all the states S checking the condition $Cellc^l_{c,S}$:

$$Cellc^l_{c,S} \overset{\text{def}}{\Leftrightarrow} \bigwedge_{r=1}^{r_c} (i^r_c = (c', t^p_{c'}, c) \quad \wedge \quad V_r \Leftrightarrow S_{c'} \geq t^p_{c'})$$

Constraint 4.2

$$focal\_state(M, S, F_S) \overset{\text{def}}{\Leftrightarrow} \bigwedge_c \bigwedge_{l=1}^{l_c} Cellc^l_{c,S} \Rightarrow F_{c,S} = K^l_c$$

Example 1. For a state S with $S_{cI} = 2$ and $S_{cro} = 1$, we obtain $Cellc^3_{cro,S} = true$ and so by implication $F_{cro,S} = K^3_{cro}$ (cf. (1)).

The predicate $transition\_rule(S, S', F_S)$ is defined by the Constraint 4.3. The o operator is the concatenation of lists and $exactly\_one(L)$ is true if exactly one Boolean of the list L of Booleans is true.

Constraint 4.3

$$\begin{aligned}
&transition_rule(S, S', F_S) \overset{\text{def}}{\Leftrightarrow} \\
&\quad \textstyle\bigwedge_c (LBup_c \Leftrightarrow S'_c = S_c + 1) \ \wedge \\
&\quad \textstyle\bigwedge_c (LBdown_c \Leftrightarrow S'_c = S_c - 1) \ \wedge \\
&\quad (Bst \Leftrightarrow S = S') \ \wedge \\
&\quad \left.\begin{array}{l} \textstyle\bigwedge_c (S_c - 1 \leq S'_c \leq S_c + 1) \ \wedge \end{array}\right\} \ point\ a \\
&\quad \left.\begin{array}{l} \textstyle\bigwedge_c (LBup_c \Rightarrow S_c < F_{c,S}) \ \wedge \\ \textstyle\bigwedge_c (LBdown_c \Rightarrow S_c > F_{c,S}) \ \wedge \\ (Bst \Leftrightarrow S = F_S) \end{array}\right\} \ point\ b \\
&\quad \left.\begin{array}{l} exactly_one(LBup\ o\ LBdown\ o\ [Bst]) \ \wedge \end{array}\right\} \ point\ c
\end{aligned}$$

Several intermediate variables appear to factorize common expressions (such as $S'_c = S_c + 1$) of the formalization of point b and c. $LBup$ and $LBdown$ are lists of Boolean variables, and Bst is a Boolean variable. $LBup_c$ and $LBdown_c$ are the elements of index c of lists $LBup$ and $LBdown$. Each of these variables are associated to one of the $2 * n + 1$ possible directions in concentration space. $LBup_c$, respectively, $LBdown_c$ is true if and only if the transition $S \twoheadrightarrow S'$ increases, respectively decreases, the value of the component c of 1 unit. Bst is true if and only if $S \twoheadrightarrow S'$ is a stationary transition (S equals S').

A rapid examination shows that the number of Booleans necessary for expressing a transition between two states, namely $Cellc^l_{c,S}$, $Bdown_c$, Bup_c, Bst_c stays linear according the number of species. In the case of the $Cellc^l_{c,S}$ one can note that their number grows exponentially according to the inside branching factor of the interaction graph. Fortunately, this factor is rarely above 4. It should be remarked also that a path composed of successive states requires for its definition a number of constraints which is proportional to the size of the path.

The main predicate of our tool GNBox is $path(M, Path, L)$ true if $Path$ is a possible succession of L states for the model M, presented in Constraint 4.4.

Constraint 4.4

$$path(M, Path, L) \overset{\text{def}}{\Leftrightarrow} \bigwedge_{i \in 1..L-1} successor(Path_i, Path_{i+1})$$

Example 2. For the λGRN model M with the focal equation system and constraints in Appendix 1 and 2, and the three following additional constraints:

- All the values of parameters K^l_{cI}, K^l_{cII} and K^l_n are known to be equal to 0 except $K^6_{cI} = 1$, $K^2_{cII} = 1$, $K^1_n = 1$,
- $K^2_{cro} = 3$, $K^3_{cro} = 1$ (K^1_{cro} and K^4_{cro} are not known),
- $path(M, [S1, S2, S3], 3) \wedge S1 = \langle 1, 2, 0, 1\rangle \wedge S2 = \langle 1, 3, 0, 1\rangle$,

we have one unique possible instantiated model possible where $K^1_{cro} = 3$ and $K^4_{cro} = 0$, and two possible states $S3$: $S3 = \langle 0, 3, 0, 1\rangle$ or $S3 = \langle 1, 3, 0, 0\rangle$.

We can check the set of solutions for this very specific (pedagogic) query. The first two additional constraints (values of parameters K^l_c) are consistent with the constraints in Appendix 2. The third additional constraint enforces that $F_{cro,S1} = K^1_{cro}$ (because $Cellc^1_{cro,S1}$ is true for $S1 = \langle 1, 2, 0, 1\rangle$) and $S1_{cro} < F_{cro,S1}$ (because Bup_{cro} is true for the constraint $successor(S1, S2, F_{S1})$). By deduction we have: $2 < K^1_{cro}$ and so the domain of K^1_{cro} is reduced to the singleton $\{3\}$. According to the constraint $((K^2_{cro} < K^1_{cro}) \vee (K^4_{cro} < K^3_{cro}))$ in Appendix 2 and the second additional constraint, we obtain that the domain of K^4_{cro} is reduced to $\{0\}$. Finally, from $S2 = \langle 1, 3, 0, 1\rangle$, $F_{S2} = \langle K^3_{cI}, K^2_{cro}, K^4_{cII}, K^4_n\rangle = \langle 0, 3, 0, 0\rangle$, and $successor(S2, S3, F_{S2})$, we obtain the two possible states $S3$, $S3 = \langle 0, 3, 0, 1\rangle$ and $S3 = \langle 1, 3, 0, 0\rangle$. This is due to the fact that only the Booleans $Bdown_{cI}$ and $Bdown_n$ can be true (tendency with the values of S_c and $F_{c,S}$ known) and only one of the two can be true in a given solution (asynchronicity).

4.2.2 The SAT Encoding

In the following, we present our procedure to replace a multivalued variable by a set of Boolean variables and clauses, and the ideas behind our Boolean encoding of integer constraints.

We chose two manners to encode as Boolean variables the multivalued variables of the model. For every integer variable S_c, $F_{c,S}$, K^l_c, t^p_c and $\max_c$, we introduce systematically as many Boolean variables as the size z of their domains, and a set of clauses specifying that exactly one of these new Boolean variables is true (each of these variables corresponding to a value in the domain of the integer variable). The $exactly\_one$ constraint is the conjunction of a at_least_one (ALO) constraint and a at_most_one (AMO) constraint. Their encoding are those given in [42], where we consider two manners to encode AMO : (1) of [42] for ALO encoding, (2) and (3) for *ladder AMO encoding*, and (4) for *pairwise AMO encoding*. For the pairwise AMO encoding, the number of clauses is quadratic according to z. For the ladder

AMO encoding, $z-1$ *ladder variables* are added and the number of clauses is linear according to z.

We have to encode only relations with at most two multivalued variables (but *exactly_one* constraint) with small domains. In spite of the apparent lack of heavy encoding problems, we cannot use the trivial way, based on the truth table of the initial relation, to obtain a clausal form. We would face an exponential explosion of the number of generated clauses according to the size of the variable domains. By taking into account the fact that exactly one of the Boolean variables for each possible value of a multivalued variable is true, we get a more efficient encoding.

Example 3. Let us consider a relation of the type $B \Leftrightarrow X = Y$, with the domains $\{0,1\}$, $\{0,1,2\}$ $\{1,2,3\}$ for B, X and Y, respectively (such as $Bup_{cI} \Leftrightarrow S_{cI} = S_{cI} + 1$, see definition of the predicate *successor* in Sect. 4.2.1). Let $x0$, $x1$, $x2$, $y1$, $y2$, $y3$, and b be the Boolean variables which are linked to the multivalued variables X, Y, and B (B being considered in CP as a finite domain variable). To encode this relation, we introduce six clauses relative to the conjunction of the two following formulae:

- $B \Rightarrow X = Y$: $(\neg b \vee \neg x1 \vee y1) \wedge (\neg b \vee x1 \vee \neg y1) \wedge (\neg b \vee \neg x2 \vee y2) \wedge (\neg b \vee x2 \vee \neg y2)$
- $X = Y \Rightarrow B$: $(b \vee \neg x1 \vee \neg y1) \wedge (b \vee \neg x2 \vee \neg y2)$

The previous example gives an idea of the encoding algorithm of a relation $B \Leftrightarrow X = Y$. The algorithm loops on the values of B, then those of X and finally those of Y, and produces the implication (clauses) on the values of the different variables.

To do the encoding of our model, we need a procedure for every type of relation: $B1 \vee B2$, $B1 \wedge B2$, $B1 \Leftrightarrow B2$, $X = Y$, $X \neq Y$, $X \leq Y$, $X < Y$, $B \Rightarrow X = Y$, $X = Y \Rightarrow B$, $B \Rightarrow X < Y$, $X < Y \Rightarrow B$ and $B \Leftrightarrow \bigwedge_i B_i$, where the B variables are Booleans and the X and Y variables are potentially multivalued. The size of the CNF representation for all these relations is linear according to the size of the multivalued variable domains (which is the number of species in the system).

4.3 Five Examples of GRN Deciphering Queries

We present five queries relative to the λGRN example and give their results. For all queries, the variable M is the model containing the focal equations of the λGRN given in Appendix 1 and the constraints about hypotheses of interaction compositions and thresholds order given in Appendix 2. The execution times for Queries 2, 3, and 4 are given in Sect. 4.4.

4.3.1 Query 1: Models Consistent with the Existence of One Steady State

The first example of query concerns the imposition of one non specified steady state. Expressing that state S is a steady state means imposing the existence of a

path of two states beginning in the state $\langle S_{cI}, S_{cro}, S_{cII}, S_n \rangle$ and reaching the same state $\langle S_{cI}, S_{cro}, S_{cII}, S_n \rangle$

Query 1

$$path(M, [S, S], 2)$$

The Query 1 leads to a coherence (in about 0.05 second). The CNF formula with the ladder AMO encoding contains 1,727 variables and 4,071 clauses, and with the pairwise AMO encoding, the formula contains 1,651 variables and 3,972 clauses. The number of variables for the ladder AMO encoding is greater because, the ladder variables. The number of clauses for the ladder AMO encoding is surprinsingly greater. This is due to the low number of Booleans used in the *exactly_one* constraint (the ladder AMO encoding beginning to be interesting from 8 Booleans in *at_most_one* constraint).

For the discussion on functionalities (see Sect. 4.4), we note Query 1' the Query 1 where we enumerate on the values of S in order to find all the possible steady states. The Query 1' permits to obtain 25 possible steady states (in about 0.1 s).

4.3.2 Query 2: Models Consistent with Biological Observations

The main query, in the case of λGRN, concerns the existence of models consistent with the two possible observed behaviors in response to an infection of a bacterial cell by a λ-phage virus. These behaviors are (a) the reachability of the *lytic attractor*, represented in the discrete Thomas formalism as a cycle between the two states $\langle 0, 2, 0, 0 \rangle$ and $\langle 0, 3, 0, 0 \rangle$ from the *initial state* $\langle 0, 0, 0, 0 \rangle$; and (b) the reachability of the *lysogenic state* $\langle 2, 0, 0, 0 \rangle$ from the same initial state. The fact that two different attractors can be reached from the same initial state is due to the non-determinism that appears at this level of abstraction. These observations come from [104]. The formal expression of the reachability of the lytic attractor and the reachability of the lysogenic attractor is the following:

Query 2

$$
\begin{aligned}
& S0 = \langle 0, 0, 0, 0 \rangle \ \wedge \\
& S1 = \langle 2, 0, 0, 0 \rangle \ \wedge \\
& L = 48 \ \wedge \\
& path(M, [S1, S1], 2) \ \wedge \ path(M, [S0, ..., S1], L) \ \wedge \\
& S2 = \langle 0, 2, 0, 0 \rangle \ \wedge \\
& S3 = \langle 0, 3, 0, 0 \rangle \ \wedge \\
& S23 = \langle 0, S23_{cro}, 0, 0 \rangle \ \wedge \ S23_{cro} \in 2..3 \ \wedge \\
& path(M, [S2, S3, S2], 3) \ \wedge \ path(M, [S0, ..., S23], L)
\end{aligned}
$$

where $S0$ is the initial state, $S1$ is the representation of the lysogenic attractor, $S2$ and $S3$ the two states of the lytic attractor, and 48 is the number of state for λGRN where all t_c^p are equal to p. The Query 2 leads to a coherence of the set of constraints.

4.3.3 Query 3: Diameter of the Network Taking into Consideration the Biological Observations

The Query 2 is more general than the Query 1 in the sense that it involves paths of states with a given length. The knowledge of this length is necessary to generate a finite system of constraints. A straightforward way to overcome this problem is to consider paths with a length equal to the total number of discrete states (in the case where we enforce one reachability of state – see Query 4 for the case where we enforce multi-reachabilities of states). In the case of λGRN, we have a discrete concentration space of 48 states. But to face efficiently this important issue we must find the maximal length of a path without cycles in models consistent with the behaviors presented in Query 2. This maximal length is the *diameter* of the network, D_{Q2}, relative to the constraints of Query 2. The query to find the diameter of the network is:

Query 3

$$\begin{aligned} &\text{Query 2} \ \wedge \\ &L \in 1..47 \ \wedge \\ &path(M, Path, L) \ \wedge \\ &alldif\_path(Path) \end{aligned}$$

where $alldif\_path(Path)$ is true if all elements of $Path$ are different one for another. The CLP encoding chosen for $alldif\_path(Path)$ is straightforward (without any propagation optimization), and we obtain a quadratic number of disjunctive constraints (enforcing that it exists a component on which two states differ). The enumeration of the values of L in an increasing order allows to identify a maximal value D_{Q2} of L for which this query is coherent. The Query 3 leads by enumeration of L to a coherence for $L = 43$ and an incoherence for $L = 44$. So, the diameter is $D_{Q2} = 43$.

4.3.4 Query 4: Models Consistent with Biological Observations and Induction Hypotheses

From the Query 2, we want to add hypotheses about the ultraviolet (U.V.) perturbation of the cell in lysogenic phase which leads the cell into the lytic phase. This process is called induction. The hypotheses consist in imposing a succession of "checkpoints" through which the system must go. The starting state is $\langle 1, 0, 0, 0 \rangle$

corresponding to a decrease, due to U.V. irradiation, of the concentration of cI in the stationary state of the lysogenic phase. After this, four states must be reach in a specific order, and finally the lytic cycle must be reached. In this query, we want to express the reachability of the two attractors (lytic and lysogenic), with the additional specification of mandatory intermediate states. The formal expression is the following:

Query 4

$$
\begin{array}{l}
\text{Query 2} \ \wedge \\
path(M, Path\_induction, 189) \ \wedge \\
St0 = \langle 1,0,0,0 \rangle \ \wedge \\
St1 = \langle 1,1,0,0 \rangle \ \wedge \\
St2 = \langle 0,1,0,0 \rangle \ \wedge \\
St3 = \langle 0,1,0,1 \rangle \ \wedge \\
St\_lytic = \langle 0, St\_lytic_{cro}, 0, 0 \rangle \ \wedge \ St\_lytic_{cro} \in 2..3 \ \wedge \\
reach([St0, St1, St2, St3, St\_lytic], Path\_induction)
\end{array}
$$

where all the St variables are the "checkpoints," and more precisely $St0$ is the representation of the lysogenic attractor after perturbation by U.V. irradiation and *St_lytic* a state of the lytic attractor. The length 189 for the path of induction *Path_induction* is chosen according to the fact that in the worst case the four paths (multi-reachabilities) included into *Path_induction* can be of length 48 ($189 = 48 * 4 - 3$). The predicate $reach(LS, Path)$ is true if LS is a list of states of $Path$ (reachabilities of each state of LS), and if it is possible to find the states of LS in $Path$ in the same order as in LS (reachabilities in a specific order, in other terms LS is a subsequence of $Path$):

$$
\begin{array}{rcl}
reach(LS, Path) & \overset{\text{def}}{\Leftrightarrow} & length(LS, L1) \ \wedge \\
& & \bigwedge_{i \in 1..L1} \ element(LS_i, Path, Index_i) \ \wedge \\
& & \bigwedge_{i \in 1..L1-1} \ Index_i \leq Index_{i+1}
\end{array}
$$

where $length(List, Length)$ is true if $List$ is a list of length $Length$, and $element(E, Path, I)$ is true if E is the element of the list $Path$ at the index I (by starting to 0):

$$
\begin{array}{rcl}
element(E, Path, I) & \overset{\text{def}}{\Leftrightarrow} & length(Path, L2) \ \wedge \\
& & I \in 0..L2-1 \ \wedge \\
& & \bigwedge_{i \in 0..L2-1} \ i = I \Rightarrow Path_i = E
\end{array}
$$

We give these formalizations to show exactly what are the type of constraints that we use, and that we must translate into CNF to be able to use SAT solvers (see Sect. 4.2.2). The Query 4 leads to a coherence of the set of constraints.

For the performance discussion (see Sect. 4.4), we note Query 4' the Query 4 where the hypotheses of composition of interactions on the gene n are removed. The Query 4' leads to a coherence (obviously because Query 4 is coherent and Query 4' is weaker than Query 4).

4.3.5 Query 5: Inference of Necessary Hypotheses Among the Hypotheses About Composition of Interactions

From the Query 2 with $L = 48$, we want to identify the hypotheses about composition of interactions which must be true. Each of these 20 hypotheses appear in one line (except the first two) of Appendix 2. In order to learn what are the *necessary* hypotheses among them, we reify each of these constraints C_i by introducing Booleans B_i such that $B_i \Leftrightarrow C_i$. We obtain a new set of constraints composed of the constraints of Query 2, minus the constraints C_i, and plus the constraint $\bigwedge_i B_i \Leftrightarrow C_i$.

To infer necessary properties about composition of interactions, we enumerate the possible values of the Booleans B_i and look at the true assigments to identify the necessary hypotheses. In order to avoid redundant solutions (the solution $B_i \wedge B_k$ and the solution $B_i \wedge B_j \wedge B_k$ are redundant for example, because $(B_i \wedge B_k) \vee (B_i \wedge B_j \wedge B_k) \Leftrightarrow (B_i \wedge B_k)$), we enumerate the values of the B_i by increasing value of their sum, and add constraints to avoid redundant solutions.

We use a SAT solver directly called from the CLP program. The CLP program controls, in this case, the enumeration of the integer equal to the sum of the B_i, the rest of the enumeration being done by the SAT solver with a blocking clause mechanism for enumeration.

The Query 5 leads to (in about 1 second) a disjunction which is reduced to only one conjunction of two Booleans that evaluate to *true*: $B_8 \wedge B_{10}$, where $B_8 \Leftrightarrow (K^3_{cro} < K^1_{cro} \vee K^4_{cro} < K^2_{cro})$ is the observability constraint about the 'inhibition' of cI on *cro*, and $B_{10} \Leftrightarrow (K^2_{cro} < K^1_{cro} \vee K^4_{cro} < K^3_{cro})$ is the observability constraint about the 'inhibition' of *cro* on itself.

4.4 Discussion

The Table 1 gives for some selected queries: the coherence of the constraint system (Sol.), the runtime in seconds with the CP version (Perf. CP), and for each type of AMO encoding (ladder and pairwise): the time in seconds to construct the SAT instance with the CLP program (const. CNF), the runtime with MiniSAT (MiniSAT 2.0 beta) and the parallel SAT solver ManySAT [48]. on the generated SAT instance, and the size of the SAT instance in terms of number of variables and number of clauses (size CNF). The runtimes are obtained on an Intel Xeon computer with two quadricores at 2.33 GHz each with 8Gbytes of RAM. The CP and MiniSAT executions use only one CPU, and the Manysat executions use four CPUs.

Table 1 Comparative performances on selected queries

			ladder AMO encoding				pairwise AMO encoding			
Query	Sol.	Perf. CP	const. CNF	Perf. MiniSAT	Perf. ManySAT	size CNF	const. CNF	Perf. MiniSAT	Perf. ManySAT	size CNF
Q.2	true	3.8	0.6	0.2	0.5	20,766	0.5	0.4	0.5	18,790
L = 43						84,696				83,571
Q.2	true	5.1	0.6	0.2	0.5	22,976	0.6	0.3	0.6	20,780
L = 48						94,046				92,801
Q.3	true	43,375	1.1	78.3	34.4	33,678	1.0	74.3	16.1	30,771
$L = 43$						154,699				153,061
Q.3	false	74,666	1.0	510.5	149.8	34,513	1.1	452.5	126.5	31,540
$L = 44$						158,966				157,292
Q.4	true	13.5	2.3	3.9	2.9	73,682	2.5	5.9	2.5	66,563
						310,613				375,131
Q.4'	true	4.2	2.3	33.3	2.9	73,534	2.5	34.8	2.7	66,415
						310,301				374,819

As it was expected on queries with only small domain integer variables, the SAT solvers are more effective (see Query 2 and Query 3 in Table 1). But for Query 4, where we enforce reachability of successive states in a path, it is necessary to introduce variables with large domains (the length of the path). We see in this case that the number of clauses for the ladder AMO encoding is lower than for the pairwise AMO encoding. Between Query 4 and Query $4'$ (where we relax some constraints), the time to obtain consistency for CP solver decreases by a factor approximately equal to 3, but it increases by a factor of about 10 for the MiniSAT solver. This effect is rare but appears in some queries, the propagation being a part of a complex system whose performance is difficult to predict.

When we want to learn properties, such as the identification of the diameter or properties on composition of interactions, the problem turns into the optimization of a parameter p in a formula $F(p)$. An SAT solver can only indicate the satisfiability of $F(p)$. So we must be able to monitor calls to SAT solver on $F(p)$ in terms of the value of p. The hybrid version of the tool (SAT integrated to CLP) enables to achieve this easily to finely control the enumeration of certain variables before the call for SAT (see Query 5).

5 Conclusion

To summarize our overview of SAT, CP, and their use and integration in applications, we look back at a table of [12], in which the authors highlighted distinctive features of SAT and CP. There has been significant interest in SAT from the CP-AI-OR community since the writing of this prior paper, and it is interesting to identify recent or ongoing trends.

On the subject of *methodology*, one thing we have noticed in this paper is how the CP approach, in which modeling is done at a high level and in a structured form, can in fact complement the low-level approach of SAT: our biology application in Sect. 4 illustrates how modeling can be done elegantly using Constraint (Logic) Programming, and then a conversion to other tools including SAT (but also Linear Programming tools) can be used for efficient resolution. This approach relieves the modeler from having to deal directly with an inconvenient, low-level format such as CNF, or matrix representation for an LP. Our case study illustrates a clear recent trend: SAT encodings have been studied quite thoroughly in the CP literature over the last years, as shown in our brief survey of translations into SAT in Sect. 3.4. The advantage of "thinking in CP, solving using SAT" is that the SAT encoding can be informed by the high-level patterns and global constraints that are explicit in the model, so that each constraint is encoded in a way that achieves Generalized Arc Consistency or is experimentally efficient. Another recent trend since the writing of this Table is a shift of CP toward more automatic, general-purpose, heuristic tuning, witnessed for instance by the fact that dynamic heuristics such as DomWdeg [14] are now adopted by many solvers.

In terms of *application area*, it is probably fair to say that SAT is increasingly seen as but one component within more complex constraint solving tools, and that it is therefore expanding beyond its initial application area of hardware verification. The integration of SAT solver techniques is clear in SMT solvers, but also in some recent CP solvers which essentially integrate an SAT solver for the treatment of Boolean structure. Fig. 2 noted the importance of optimization as opposed to simply satisfiability; "pure CP techniques" are by themselves not often very good at optimization but an integrated approach where CP is used as an integration framework for a number of techniques including, for instance, Linear Programming, excels in these problems, and SAT is definitely part of the methods that contribute to a successful integration.

The *architecture* of SAT solvers has not significantly changed since the time of this writing–or indeed since the work on Chaff [78]! It is more difficult to guess how CP software architectures will evolve in a near future. Integration is here again probably a key word, as exemplified by some recent developments such as SCIP [1], SIMPL [116] or Comet [109], in which Local Search, Linear Programming, CP, and SAT techniques cooperate.

Last, in terms of *evolution* of the communities, noticeable efforts have been made recently in the CP community toward better benchmarking practices and a higher measurability of progress. An indication of this is that the CSP 2009 competition attracted up to 14 solvers in some tracks, and that many solvers included some techniques inspired indeed from SAT (restarts, dynamic heuristic, learning in some cases). Fig. 2 noted that the SAT approach has traditionally been "bottom-up" with incremental improvements based on the common DPLL basis. In contrast, [12] described CP's approach as a "top-down" one in which "there is no such thing as

	SAT	CP
Methodology	**low-level** provides an "assembly language" for decision procedures **black box** not meant to be directly used by human; target language for translators **automatic** little room (or need) for informing solver of problem specific information	**high-level** notion of CP *language* with rich set of constructs and constraints **glass box** meant to be used programmatically; direct integration in applications **parameterised** everything can (and, typically, needs to) be tuned
Applications	**focus on decision problems** theorem proving, hardware and software verification; importance of *complete* solvers	**focus on optimisation** scheduling, resource allocation; importance of online optimisation and fast, approximate optima
Architecture	**homogeneous** relatively small programs; unique type of constraints (clauses); very efficient, but hyper-specialised algorithms and data structures	**open** large systems and libraries, open to (possibly user-defined) new constraints; algorithms from OR, graph theory *etc.*, can be integrated
Evolution	**bottom-up approach** incremental improvements of established state-of-the-art DPLL; little room for exotic proposals **good measurability of progress** large set of industrial CNF instances; successful annual competition; state-of-the-art methods well identified	**top-down approach** large set of algorithms provided; no disciplined approach to integrate new algorithms in state-of-the-art **poor measurability of progress** no problem definition format; comparing performance of non-tuned solvers is considered meaningless

Fig. 2 Some distinctive features between SAT and CP

a state-of-the-art algorithm that would be incrementally improved; instead, there is a toolbox that is incrementally enriched." The solvers that participated in the competition seem to exhibit a more mixed approach and show signs that the clear separations drawn between SAT and CP in Fig. 2 are blurring.

Appendix 1: Focal Equations of the λGRN Model

$$F_{cI,S} = \begin{cases} K^1_{cI} & \text{if } S_{cI} < t^2_{cI} \wedge S_{cro} < t^1_{cro} \wedge S_{cII} < t^1_{cII} \\ K^2_{cI} & \text{if } S_{cI} < t^2_{cI} \wedge S_{cro} < t^1_{cro} \wedge S_{cII} \geq t^1_{cII} \\ K^3_{cI} & \text{if } S_{cI} < t^2_{cI} \wedge S_{cro} \geq t^1_{cro} \wedge S_{cII} < t^1_{cII} \\ K^4_{cI} & \text{if } S_{cI} < t^2_{cI} \wedge S_{cro} \geq t^1_{cro} \wedge S_{cII} \geq t^1_{cII} \\ K^5_{cI} & \text{if } S_{cI} \geq t^2_{cI} \wedge S_{cro} < t^1_{cro} \wedge S_{cII} < t^1_{cII} \\ K^6_{cI} & \text{if } S_{cI} \geq t^2_{cI} \wedge S_{cro} < t^1_{cro} \wedge S_{cII} \geq t^1_{cII} \\ K^7_{cI} & \text{if } S_{cI} \geq t^2_{cI} \wedge S_{cro} \geq t^1_{cro} \wedge S_{cII} < t^1_{cII} \\ K^8_{cI} & \text{if } S_{cI} \geq t^2_{cI} \wedge S_{cro} \geq t^1_{cro} \wedge S_{cII} \geq t^1_{cII} \end{cases}$$

$$F_{cro,S} = \begin{cases} K^1_{cro} & \text{if } S_{cI} < t^2_{cI} \wedge S_{cro} < t^3_{cro} \\ K^2_{cro} & \text{if } S_{cI} < t^2_{cI} \wedge S_{cro} \geq t^3_{cro} \\ K^3_{cro} & \text{if } S_{cI} \geq t^2_{cI} \wedge S_{cro} < t^3_{cro} \\ K^4_{cro} & \text{if } S_{cI} \geq t^2_{cI} \wedge S_{cro} \geq t^3_{cro} \end{cases}$$

$$F_{cII,S} = \begin{cases} K^1_{cII} & \text{if } S_{cI} < t^2_{cI} \wedge S_{cro} < t^3_{cro} \wedge S_n < t^1_n \\ K^2_{cII} & \text{if } S_{cI} < t^2_{cI} \wedge S_{cro} < t^3_{cro} \wedge S_n \geq t^1_n \\ K^3_{cII} & \text{if } S_{cI} < t^2_{cI} \wedge S_{cro} \geq t^3_{cro} \wedge S_n < t^1_n \\ K^4_{cII} & \text{if } S_{cI} < t^2_{cI} \wedge S_{cro} \geq t^3_{cro} \wedge S_n \geq t^1_n \\ K^5_{cII} & \text{if } S_{cI} \geq t^2_{cI} \wedge S_{cro} < t^3_{cro} \wedge S_n < t^1_n \\ K^6_{cII} & \text{if } S_{cI} \geq t^2_{cI} \wedge S_{cro} < t^3_{cro} \wedge S_n \geq t^1_n \\ K^7_{cII} & \text{if } S_{cI} \geq t^2_{cI} \wedge S_{cro} \geq t^3_{cro} \wedge S_n < t^1_n \\ K^8_{cII} & \text{if } S_{cI} \geq t^2_{cI} \wedge S_{cro} \geq t^3_{cro} \wedge S_n \geq t^1_n \end{cases}$$

$$F_{n,S} = \begin{cases} K^1_n & \text{if } S_{cI} < t^1_{cI} \wedge S_{cro} < t^2_{cro} \\ K^2_n & \text{if } S_{cI} < t^1_{cI} \wedge S_{cro} \geq t^2_{cro} \\ K^3_n & \text{if } S_{cI} \geq t^1_{cI} \wedge S_{cro} < t^2_{cro} \\ K^4_n & \text{if } S_{cI} \geq t^1_{cI} \wedge S_{cro} \geq t^2_{cro} \end{cases}$$

Appendix 2: Constraints of the λGRN Model

$$t^1_{cI} \leq t^2_{cI} \quad \wedge \quad alldif([t^1_{cI}, t^2_{cI}])$$
$$t^1_{cro} \leq t^2_{cro} \leq t^3_{cro} \quad \wedge \quad alldif([t^1_{cI}, t^2_{cI}, t^3_{cro}])$$

$$\begin{array}{llllllll}
K^1_{cI} \leq K^5_{cI} & \wedge & K^2_{cI} \leq K^6_{cI} & \wedge & K^3_{cI} \leq K^7_{cI} & \wedge & K^4_{cI} \leq K^8_{cI} \\
K^1_{cI} < K^5_{cI} & \vee & K^2_{cI} < K^6_{cI} & \vee & K^3_{cI} < K^7_{cI} & \vee & K^4_{cI} < K^8_{cI} \\
K^3_{cI} \leq K^1_{cI} & \wedge & K^4_{cI} \leq K^2_{cI} & \wedge & K^7_{cI} \leq K^5_{cI} & \wedge & K^8_{cI} \leq K^6_{cI} \\
K^3_{cI} < K^1_{cI} & \vee & K^4_{cI} < K^2_{cI} & \vee & K^7_{cI} < K^5_{cI} & \vee & K^8_{cI} < K^6_{cI} \\
K^1_{cI} \leq K^2_{cI} & \wedge & K^3_{cI} \leq K^4_{cI} & \wedge & K^5_{cI} \leq K^6_{cI} & \wedge & K^7_{cI} \leq K^8_{cI} \\
K^1_{cI} < K^2_{cI} & \vee & K^3_{cI} < K^4_{cI} & \vee & K^5_{cI} < K^6_{cI} & \vee & K^7_{cI} < K^8_{cI}
\end{array}$$

$$\begin{array}{llll}
K^3_{cro} \leq K^1_{cro} & \wedge & K^4_{cro} \leq K^2_{cro} \\
K^3_{cro} < K^1_{cro} & \vee & K^4_{cro} < K^2_{cro} \\
K^2_{cro} \leq K^1_{cro} & \wedge & K^4_{cro} \leq K^3_{cro} \\
K^2_{cro} < K^1_{cro} & \vee & K^4_{cro} < K^3_{cro}
\end{array}$$

$$\begin{array}{llllllll}
K^5_{cII} \leq K^1_{cII} & \wedge & K^6_{cII} \leq K^2_{cII} & \wedge & K^7_{cII} \leq K^3_{cII} & \wedge & K^8_{cII} \leq K^4_{cII} \\
K^5_{cII} < K^1_{cII} & \vee & K^6_{cII} < K^2_{cII} & \vee & K^7_{cII} < K^3_{cII} & \vee & K^8_{cII} < K^4_{cII} \\
K^3_{cII} \leq K^1_{cII} & \wedge & K^4_{cII} \leq K^2_{cII} & \wedge & K^7_{cII} \leq K^5_{cII} & \wedge & K^8_{cII} \leq K^6_{cII} \\
K^3_{cII} < K^1_{cII} & \vee & K^4_{cII} < K^2_{cII} & \vee & K^7_{cII} < K^5_{cII} & \vee & K^8_{cII} < K^6_{cII} \\
K^1_{cII} \leq K^2_{cII} & \wedge & K^3_{cII} \leq K^4_{cII} & \wedge & K^5_{cII} \leq K^6_{cII} & \wedge & K^7_{cII} \leq K^8_{cII} \\
K^1_{cII} < K^2_{cII} & \vee & K^3_{cII} < K^4_{cII} & \vee & K^5_{cII} < K^6_{cII} & \vee & K^7_{cII} < K^8_{cII}
\end{array}$$

$$\begin{array}{llll}
K^3_{n} \leq K^1_{n} & \wedge & K^4_{n} \leq K^2_{n} \\
K^3_{n} < K^1_{n} & \vee & K^4_{n} < K^2_{n} \\
K^2_{n} \leq K^1_{n} & \wedge & K^4_{n} \leq K^3_{n} \\
K^2_{n} < K^1_{n} & \vee & K^4_{n} < K^3_{n}
\end{array}$$

Acknowledgement This work was supported by Microsoft Research through its European PhD Scholarship Programme.

References

1. Achterberg T, Berthold T, Koch T, Wolter K (2008) Constraint integer programming: A new approach to integrate CP and MIP. In: Proceedings of International Conference on Integration of AI and OR Techniques in Constraint Programming (CP-AI-OR), pp 6–20
2. Bacchus F (2007) GAC via unit propagation. In: Proceedings of International Conference on Principles and Practice of Constraint Programming (CP), pp 133–147

3. Bacchus F, Walsh T (2005) Propagating logical combinations of constraints. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), pp 35–40
4. Baptiste P, Le Pape C, Nuijten W (2001) Constraint-Based Scheduling. Springer, Berlin
5. Beldiceanu N, Carlsson M, Demassey S, Petit T (2007) Global constraint catalogue: Past, present and future. Constraints 12(1):21–62
6. Bessière C (1994) Arc-consistency and arc-consistency again. Artif Intell 65(1):179–190
7. Bessiere C (2006) Constraint propagation. In: Handbook of Constraint Programming, chap. 3
8. Bessiere C, Hebrard E, Walsh T (2003) Local consistencies in SAT. In: Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT), pp 299–314
9. Bessiere C, Kastirelos G, Narodytska N, Quimper C-G, Walsh T (2009) Decompositions of all different, global cardinality and related constraints. In: Boutilier C (ed) Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI), Pasadena, California, USA, July 11–17, pp 419–424
10. Bessiere C, Kastirelos G, Narodytska N, Walsh T (2009) Circuit complexity and decompositions of global constraints. In: Boutilier C (ed) Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI), Pasadena, California, USA, July 11–17, pp 412–418
11. Biere A, Heule M, Van Maaren H, Walsh T (2009) Handbook of Satisfiability. IOS Press, Amsterdam
12. Bordeaux L, Hamadi Y, Zhang L (2006) Propositional satisfiability and constraint programming: A comparative survey. ACM Comput. Surv. 38(4)
13. Boros E, Hammer PL (2002) Pseudo-boolean optimization. Discrete Appl Math 123(1-3): 155–225
14. Boussemart F, Hemery F, Lecoutre C, Sais L (2004) Boosting systematic search by weighting constraints. In: Proceedings of European Conference on Artificial Intelligence (ECAI), pp 146–150
15. Cadoli M, Schaerf A (2005) Compiling problem specifications into SAT. Artif Intell 162 (1-2):89–120
16. Carlsson M, Beldiceanu N, Martin J (2008) A geometric constraint over k-dimensional objects and shapes subject to business rules. In: Proceedings of International Conference on Principles and Practice of Constraint Programming (CP), pp 220–234
17. Clarke E, Biere A, Raimi R, Zhu Y (2001) Bounded model checking using satisfiability solving. Formal Methods Syst Des 19(1):7–34
18. Clarke EM, Grumberg O, Peled DA (1999) Model checking. The MIT Press, Cambridge, MA, USA
19. Cleary JG (1987) Logical arithmetic. Future Comput Syst 2(2):125–149
20. Codognet P, Diaz D (1993) Boolean constraints solving using CLP(FD). In: Proceedings of the International Symposium on Logic Programming, MIT Press, Cambridge, MA, USA, pp 525–539
21. Cohen J, Koiran P, Perrin P (1991) Meta-level interpretation of CLP(Lists). In: Benhamou F, Colmerauer A (eds) Constraint Logic Programming, Selected Research, The MIT Press, Cambridge, MA, USA, pp 457–481
22. Collavizza H, Rueher M, Van Hentenryck P (2008) CPBPV: a constraint-programming framework for bounded program verification. In: Proceedings of International Conference on Principles and Practice of Constraint Programming (CP), pp 327–341
23. Corblin F (2008) Conception et mise en œuvre d'un outil dclaratif pour l'analyse des rseaux gntiques discrets. Ph.D. thesis, Universit Joseph Fourier
24. Corblin F, Tripodi S, Fanchon E, Ropers D, Trilling L (2009) A declarative constraint-based method for analyzing discrete gene regulation networks. Biosystems 98(2):91–104
25. Darwiche A, Marquis P (2002) A knowledge compilation map. J AI Res (JAIR) 17:229–264
26. Davis M, Logemann G, Loveland D (1962) A machine program for theorem-proving. Commun ACM 5(7):393–397
27. Davis M, Putnam H (1960) A computing procedure for quantification theory. J ACM 7(3):201–215

28. De Moura LM, Bjorner N (2008) Z3: An efficient SMT solver. In: Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp 337–340
29. Dechter R (1990) Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. Artif Intell 41(3):273–312
30. Delzanno G, Podelski A (2001) Constraint-based deductive model checking. Int J Softw Tools Technol Transf 3(3):250–270
31. Diaz D, Codognet C (2001) Design and implementation of the GNU prolog system. J Funct Log Program 2001(6)
32. Dutertre B, De Moura LM (2006) A fast linear-arithmetic solver for DPLL(T). In: Proceedings of International Conference on Computer-Aided Verification (CAV), pp 81–94
33. Eén N, Sörensson N (2003) An extensible SAT-solver. In: Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT), pp 502–518
34. Eén N, Sörensson N (2006) Translating pseudo-boolean constraints into SAT. J Satisf Boolean Model Comput (JSAT) 2:1–26
35. Fanchon E, Corblin F, Trilling L, Hermant B, Gulino D (2005) Modeling the molecular network controlling adhesion between human endothelial cells: Inference and simulation using constraint logic programming. In: Danos V, Schachter V (eds) Computational Methods in Systems Biology, vol 3082. Springer, Berlin, pp 104–118
36. Franzle M, Herde C, Teige T, Ratschan S, Schubert T (2007) Efficient solving of large nonlinear arithmetic constraint systems with complex boolean structure. J Satisf Boolean Model Comput (JSAT) 1:209–236
37. Gelfond M (2008) Answer sets. In: Handbook of Knowledge Representation, Elsevier, Amsterdam, pp 285–316
38. Génisson R, Jégou P (2000) On the relations between SAT and CSP enumerative algorithms. Discrete Appl Math 107(1-3):27–40
39. Gent IP, Jefferson C, Miguel I (2006) Minion: A fast scalable constraint solver. In: Proceedings of European Conference on Artificial Intelligence (ECAI), pp 98–102
40. Gent IP, Jefferson C, Miguel I (2006) Watched literals for constraint propagation in minion. In: Proceedings of International Conference on Principles and Practice of Constraint Programming (CP), pp 182–197
41. Gent IP, Miguel I, Nightingale P (2008) Generalised arc consistency for the alldifferent constraint: An empirical survey. Artif Intell 172(18):1973–2000
42. Gent IP, Nightingale P (2004) A new encoding of alldifferent into SAT. In: 3rd International Workshop on Modelling and reformulating Constraint Satisfaction Problems (CP2004), pp 95–110
43. Ginsberg ML (1993) Dynamic backtracking. J AI Res (JAIR) 1:25–46
44. Gomes C, Selman B (2007) The science of constraints. Constraint Program Lett 1:15–20
45. Gomes CP, Kautz H, Sabharwal A, Selman B (2009) Satisfiability solvers. In: van Harmelen, Lifschitz, Porter (eds) Handbook of Knowledge Representation, Elsevier, Amsterdam, pp 89–134
46. Gomes CP, Selman B, Kautz HA (1998) Boosting combinatorial search through randomization. In: Proceedings of National Conference on Artificial Intelligence (AAAI), pp 431–437
47. Guespin-Michel J, Bernot G, Comet JP, Mrieau A, Richard A, Hulen C, Polack B (2004) Epigenesis and dynamic similarity in two regulatory networks in pseudomonas aeruginosa. Acta Biotheor 52(4):379–390
48. Hamadi Y, Jabbour S, Sais L (2009) ManySAT: a parallel SAT solver. J Satisf Boolean Model Comput 6:245–262
49. Hamadi Y, Saubion F, Monfroy E, What is Autonomous Search. In: What Is Autonomous Search? pp 357–391
50. Hickey TJ (1991) Functional constraints in CLP languages. In: Benhamou F, Colmerauer A (eds) Constraint Logic Programming, Selected Research, The MIT Press, Cambridge, MA, USA, pp 355–381
51. Hooker J (2006) Integrated Methods for Optimization. Springer, Heidelberg

52. Hooker J (2009) A principled approach to mixed integer/linear problem formulation. In: Chinneck J, Kristjansson B, Saltzman M (eds) Operations research and cyber-infrastructure. Springer, New York, pp 79–100
53. Hooker J (2009) Some observations on boolean logic and optimization. In: Talk, RUTCOR, Rutgers University
54. Hooker J, Ottoson G (2003) Logic-based benders decomposition. Math Program 96:33–60
55. Huang J (2008) Universal booleanization of constraint models. In: Proceedings of International Conference on Principles and Practice of Constraint Programming (CP), pp 144–158
56. Jaffar J, Maher M (1994) Constraint logic programming: A survey. J Log Program (19-20), 503–581
57. de Jong H, Geiselmann J, Batt G, Hernandez C, Page M (2004) Qualitative simulation of the initiation of sporulation in *Bacillus subtilis*. Bull Math Biol 66(2):261–299
58. de Jong H, Gouzé JL, Hernandez C, Page M, Sari T, Geiselmann J (2004) Qualitative simulation of genetic regulatory networks using piecewise-linear models. Bull Math Biol 66(2):301–340
59. Jung JC, Barahona P, Katsirelos G, Walsh T (2008) Two encodings of DNNF theories. In: ECAI workshop on Inference methods based on Graphical Structures of Knowledge
60. Jussien N (2003) The versatility of explanations in constraint programming. Tech. rep., École des Mines de Nantes; Habilitation thesis
61. Kasif S (1990) On the parallel complexity of discrete relaxation in constraint satisfaction networks. Artif Intell 45(3):275–286
62. Katsirelos G, Bacchus F (2005) Generalized nogoods in CSPs. In: Proceedings of National Conference on Artificial Intelligence (AAAI), pp 390–396
63. Kautz H, Sabharwal A, Selman B (2009) Incomplete algorithms. In: Handbook of Satisfiability. IOS Press, Amsterdam
64. Kautz HA, Selman B (1992) Planning as satisfiability. In: Proceedings of European Conference on Artificial Intelligence (ECAI), John Wiley and Sons, New York, NY, USA, pp 359–363
65. Kelly AD, Macdonald AD, Marriott K, Sondergaard H, Stuckey PJ, Yap RHC (1995) An optimizing compiler for CLP(R). In: Proceedings of International Conference on Principles and Practice of Constraint Programming (CP), pp 222–239
66. Kroening D, Ouaknine J, Seshia SA, Strichman O (2004) Abstraction-based satisfiability solving of Presburger arithmetic. In: Proceedings of International Conference on Computer-Aided Verification (CAV), pp 308–320
67. Kroening D, Strichman O (2008) Decision Procedures – An algorithmic Point of View. Springer, Heidelberg
68. Lecoutre C, Tabary S (2006) Abscon 109: a generic CSP solver. In: Proceedings of 2nd International CSP Solver Competition, pp 55–63
69. Lhomme O (2004) Arc-consistency filtering algorithms for logical combinations of constraints. In: Proceedings of International Conference on Integration of AI and OR Techniques in Constraint Programming (CP-AI-OR), pp 209–224
70. Lynce I, Marques Silva J (2002) The effect of nogood recording in DPLL-CBJ SAT algorithms. In: Proceedings of International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP), pp 144–158
71. Manquinho VM, Marques Silva JP (2005) On applying cutting planes in dll-based algorithms for pseudo-boolean optimization. In: Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT), pp 451–458
72. Marchand H, Martin A, Weismantel R, Wolsey LA (2002) Cutting planes in integer and mixed integer programming. Discrete Appl Math 123(1-3):397–446
73. Marques-Silva J (2008) Model checking with Boolean satisfiability. J Algorithms 63(1-3): 3–16
74. Marques Silva JP, Sakallah KA (1999) GRASP: A search algorithm for propositional satisfiability. IEEE Trans Comput 48(5):506–521
75. Marriott K, Nethercote N, Rafeh R, Stuckey PJ, Garcia de la Banda M, Wallace M (2008) The design of the zinc modelling language. Constraints 13(3):229–267

76. McAllester DA, Collins M, Pereira P (2004) Case-factor diagrams for structured probabilistic modeling. In: Proceedings of International Conference on Uncertainty in Artificial Intelligence (UAI), pp 382–391
77. McAllester DA, Selman B, Kautz HA (1997) Evidence for invariants in local search. In: Proceedings of National Conference on Artificial Intelligence (AAAI), pp 321–326
78. Moskewicz MW, Madigan CF, Zhao Y, Zhang L, Malik S (2001) Chaff: Engineering an efficient SAT solver. In: Proceedings of International Design Automation Conference (DAC), pp 530–535. ACM
79. Nelson G, Oppen DG (1979) Simplification by cooperating decision procedures. ACM Trans Program Lang Syst 1(2):245–257
80. Nieuwenhuis R, Oliveras A, Tinelli C (2006) Solving SAT and SAT Modulo Theories: From an abstract davis-putnam-logemann-loveland procedure to DPLL(T). J ACM 53(6):937–977
81. Pipatsrisawat K, Darwiche A (2007) A lightweight component caching scheme for satisfiability solvers. In: Proceedings of International Conference on Theory and Applications of Satisfiability Testing (SAT), pp 294–299
82. Pipatsrisawat K, Darwiche A (2008) A new clause learning scheme for efficient unsatisfiability proofs. In: Proceedings of National Conference on Artificial Intelligence (AAAI), pp 1481–1484
83. Pipatsrisawat K, Darwiche A (2009) On the power of clause-learning sat solvers with restarts. In: Proceedings of International Conference on Principles and Practice of Constraint Programming (CP), pp 654–668
84. Prestwich S (2009) CNF encodings. In: Handbook of Satisfiability, chap. 2, IOS Press, Amsterdam, pp 75–98
85. Prosser P (1993) Hybrid algorithms for the constraint satisfaction problem. Comput Intell 9:268–299
86. Puget JF (1994) A C++ implementation of CLP. Tech. rep., ILOG, inc. ILOG Solver Collected Papers
87. Puget JF (2004) CP's next challenge: simplicity of use. In: Proceedings of International Conference on Principles and Practice of Constraint Programming (CP), p. invited talk. Springer, Heidelberg
88. Quimper CG, Walsh T (2007) Decomposing global grammar constraints. In: CP, Springer, Heidelberg, pp 590–604
89. Refalo P (2004) Impact-based search strategies for constraint programming. In: Proceedings of International Conference on Principles and Practice of Constraint Programming (CP), pp 557–571
90. Regin JC (1994) A filtering algorithm for constraints of difference in csps. In: Proceedings of National Conference on Artificial Intelligence (AAAI), pp 362–367
91. Regin JC (2010) Global constraints. In: Global Constraints: A Survey, pp 63–134
92. Ropers D, de Jong H, Page M, Schneider D, Geiselmann J (2006) Qualitative simulation of the carbon starvation response in *Escherichia coli*. Biosystems 84(2):124–152
93. Rossi F, van Beek P, Walsh T (2006) Handbook of Constraint Programming. Elsevier, Amsterdam
94. Roussel O, Lecoutre C (2009) XML representation of constraint networks: Format XCSP 2.1. arXiv.org
95. Roussel O, Manquinho V (2009) Pseudo-boolean and cardinality constraints. In: Handbook of Satisfiability, chap. 22, IOS Press, Amsterdam, pp 695–734
96. Sànchez L, van Helden J, Thieffry D (1997) Establishment of the dorso-ventral pattern during embryonic development of *Drosophila melanogaster*: a logical analysis. J Theor Biol 187:377–389
97. Schiex T, Verfaillie G (1994) Stubborness: A possible enhancement for backjumping and nogood recording. In: Proceedings of European Conference on Artificial Intelligence (ECAI), pp 165–172
98. Schulte C, Stuckey PJ (2008) Efficient constraint propagation engines. ACM Trans Program Lang Syst (TOPLAS) 31(1)

99. Schulte C, Tack G, Lagerkvist M (2006) Gecode. In: INFORMS Annual Meeting
100. Selman B, Kautz HA, Cohen B (1994) Noise strategies for improving local search. In: Proceedings of National Conference on Artificial Intelligence (AAAI), pp 337–343
101. Selman B, Levesque HJ, Mitchell DG (1992) A new method for solving hard satisfiability problems. In: Proceedings of National Conference on Artificial Intelligence (AAAI), pp 440–446
102. Seshia SA, Bryant RE (2005) Deciding quantifier-free presburger formulas using parameterized solution bounds. Log Methods Comput Sci 1(2)
103. Stuckey PJ, García de la Banda MJ, Maher MJ, Marriott K, Slaney JK, Somogyi Z, Wallace M, Walsh T (2005) The G12 project: Mapping solver independent models to efficient solutions. In: Proceedings of International Conference on Principles and Practice of Constraint Programming (CP), pp 13–16
104. Thieffry D, Thomas R (1995) Dynamical behaviour of biological regulatory networks – ii. immunity control in bacteriophage lambda. Bull Math Biol 57:277–297
105. Thomas R, D'Ari R (1990) Biological Feedback. CRC Press, Boca Raton, FL, USA
106. Thomas R, Kaufman M (2001) Multistationarity, the basis of cell differentiation and memory. ii. logical analysis of regulatory networks in term of feedback circuits. Chaos 11:180–195
107. del Val A (1994) Tractable databases: How to make propositional unit resolution complete through compilation. In: Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR), pp 551–561
108. Van Hentenryck P (1999) The OPL Optimization Programming Language. MIT Press, Cambridge, MA, USA
109. Van Hentenryck P, Michel L (2005) Constraint-Based Local Search. MIT Press, Cambridge, MA, USA
110. Van Hentenryck P, Perron L, Puget JF (2000) Search and strategies in OPL. ACM Trans Comput Log (TOCL) 1(2):285–320
111. Van Hoeve WJ, Katriel I (2006) Global constraints. In: Handbook of Constraint Programming, chap. 6, Elsevier, Amsterdam
112. Walinsky C (1989) Clp(sigma*): Constraint logic programming with regular sets. In: Proceedings of International Conference on Logic Programming (ICLP), pp 181–196
113. Walsh T (2000) SAT v CSP. In: Proceedings of International Conference on Principles and Practice of Constraint Programming (CP), Springer, Heidelberg, pp 441–456
114. Williams HP, Yan H (2001) Representations of the all_different predicate of constraint satisfaction in integer programming. INFORMS J Comput 13(2):96–103
115. Williams R, Gomes CP, Selman B (2003) Backdoors to typical case complexity. In: Proceedings of International Joint Conference on Artificial Intelligence (IJCAI), pp 1173–1178
116. Yunes T, Aron I, Hooker J (2009) An integrated solver for optimization problems. Tech. rep. Working Paper
117. Zhang L, Madigan CF, Moskewicz MW, Malik S (2001) Efficient conflict driven learning in boolean satisfiability solver. In: Proceedings of International Conference on Computer Aided Design (ICCAD), pp 279–285

Bioinformatics: A Challenge to Constraint Programming

Pedro Barahona, Ludwig Krippahl, and Olivier Perriquet

Abstract Bioinformatics is a rapidly growing field at the intersection of biology and computer science. As such, it poses a wealth of problems, opportunities, and challenges for both areas. This paper overviews some of these issues, with an emphasis on those that seem most amenable to constraint programming (CP) approaches and where CP has made some progress. Since bioinformatics is tightly focused on real-life applications, this paper does not expand on theoretical principles but, rather, tries to give an idea of the practical issues. At this light, the paper briefly presents the selected problems together with the solutions found so far, that illustrate the versatility of CP techniques that have been used in this area and the need to integrate them with other complementary techniques to handle realistic applications.

1 Introduction

Bioinformatics arose from the need to manage the large datasets of sequence information generated by molecular biology research. From the outset, this interdisciplinary field was strongly focused on practical applications, and that is still one of its main features. As molecular biology grew from sequence analysis to structural biochemistry, the scope of bioinformatics broadened to include molecular modeling, molecular dynamics, and macromolecular interaction simulations. Nowadays, bioinformatics is still growing, ranging in applications from spectroscopy data processing to biodiversity studies and the modeling of evolutionary processes, and in techniques from simulated annealing to reasoning over ontologies. Of this large field and diverse applications, this article will focus on some problems that seem most amenable to constraint programming (CP) and declarative approaches and on some

P. Barahona (✉)
Centro de Inteligência Artificial, Dep. de Informática, Universidade Nova de Lisboa, 2825 Monte de Caparica, Portugal
e-mail: pb@di.fct.unl.pt

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_14,
© Springer Science+Business Media, LLC 2011

CP, and related, solutions that have been proposed so far. The goal is to provide an overview of some interesting applications of CP in this area and to share the authors' perspective on what may be the main challenges in this endeavor.

1.1 Data Sources

One of the most salient features of current bioinformatics is the abundance of data, which is often overwhelming in both quantity and complexity. Taking bioinformatics in a broad sense, in some areas, the information is proprietary and may be costly to obtain. For example, this is the case in drug design, quantitative structure–activity relationship (QSAR), and other areas that involve large financial investments from private companies. In this paper, we will not address those data sources. Rather, we will focus on freely available data on molecular biology, such as gene sequences, protein structure, and metabolic pathways, as these are accessible to any academic researchers who may wish to explore CP applications to bioinformatics. Still, one should note that there is also a large amount of data and problems in the private sector, mostly with pharmaceutical companies. Where relevant, we shall mention the most accessible and useful sources of data for researchers interested in those areas where free databases are available.

1.2 About This Article

In the following five sections, we summarize aspects of different areas of bioinformatics. Neither the topics chosen nor the aspects on which we focus are meant to cover bioinformatics in a comprehensive manner. Rather, the selection aims at bringing out those problems that the authors feel are more interesting for the CP community. Section 1 covers the analysis of sequence data, such as sequence comparison and pattern matching, but also includes evolutionary models, such as phylogenetic trees, and population genetics problems. Strictly speaking, not all of these involve sequence data. Phylogenetic trees can be calculated from phenotypes and, in the example chosen for population genetics problems, the estimation of genetic diversity from single nucleotide polymorphisms (SNPs), data can be obtained from restriction fragment lengths instead of direct sequences. The decision to group these problems together is not meant to express some clear partition but rather some basic similarities both at the computational and biochemical level.

Section 3 is about modeling RNA structures. Since RNA structure is largely dominated by base pairings, a good portion of this problem can be reduced to more abstract and general problems involving graphs and finite domain variables. Though there are open problems in these areas, there are several well-established and efficient solutions using local search or dynamic programming algorithms. Nevertheless, these problems are interesting because the problem domains are close to

the classical domains where CP is often applied. In fact, CP approaches have been used in RNA structure modeling since even before CP became recognized as an autonomous field.

Sections 4 and 5 address protein structure and interaction. Proteins are crucial parts in life's machinery, involved in most biochemical reactions, and are the expression of the organism's genes. Proteins are thus a focus of interest (and funding) in current bioinformatics and biochemistry research. The main challenge for CP here is probably the integration of CP algorithms with the software and methods used by this research community. The problems are hard to solve, and though there are successful solutions using molecular dynamics, local search and other approaches, CP seems to have a definite contribution to make in this area, if only it can be meshed with the other applications necessary to make the jump from theoretical studies to solving real-life problems. Section 4 is an overview of protein structure prediction and determination problems, and Sect. 5 focuses the problem of modeling interactions between proteins for which the structure is known.

Section 6 is about systems biology. Admittedly, the authors' decision to include systems biology in bioinformatics would not meet consensual approval among either the bioinformatics or the systems biology community. But from the perspective of the CP researcher, systems biology is part of the same broad problem of applying computer science to solving biological problems. In systems biology, these problems are the complex networks of interactions of which life is made, from gene regulation to ecosystems but, in this paper, we focus on metabolic pathways and gene regulation.

Finally, Sect. 7 concludes the paper by highlighting some common points of interest and challenges with the application of CP technology to these diverse and complex problems.

2 Sequence Analysis

The study of gene and protein sequences is ideally suited to CP, both because of the power of CP to solve finite-domain combinatorial problems and because, in this field, it is often possible to separate the abstract problem of processing sequences of symbols from the more concrete, and often messier, issues of biological processes and noisy data collection. However, sequence analysis was the original reason for bioinformatics research, and there are well-established algorithms based on dynamic programming (e.g., the Smith–Waterman algorithm [1] for sequence alignments, and the Sankoff recursions in sequence complementarity matching for RNA structure prediction [2]) that, aided by specialized heuristics, can efficiently solve most problems in this field. As a result, it is hard for CP solvers to compete with algorithms such as FASTA [3] or BLAST [4].

Even so, there are some problems in sequence analysis where the versatility of CP can be a determining advantage relative to existing approaches. For example, whenever one wishes to include the parameters that evaluate substitutions and deletions

into the problem itself [5] or to consider additional constraints based on prior knowledge of conserved regions [6]. There are also specific related problems where the declarative nature and expressiveness of CP can simplify the implementation. For instance, the determination of the optimal dispensation ordering of nucleotides for pyrosequencing, a technique where DNA is sequenced by coupling a light-emitting reaction to the DNA-polymerase reaction. Each base in the sequence is determined by detecting the emission of light when the right nucleotide is dispensed, and the dispensation order is important to make the process both faster and less expensive [7].

Apart from specific applications involving more restricted data sources, such as when working directly with teams sequencing new genes, the main data source for gene and protein sequence analysis would be GenBank, an open-access gene sequence database supported by the NIH [8]. GenBank contains approximately a hundred million gene sequences with an average of a thousand base pairs each, covering 140,000 different organisms. It also provides basic search features based on BLAST, and several specialized adaptations of this algorithm to more specific searches (e.g., primers, conserved domains, particular proteins, and so forth).

2.1 Phylogenetic Trees

Sequence analysis problems in bioinformatics are not restricted to sequence determination or alignment. From that data, much can be inferred about the origin of organisms and species. Phylogenetic trees, which represent the evolutionary relationships between taxonomic groups, are often calculated from gene or protein sequences. Or they can be constructed from phenotype descriptors, sets of states assigned to characters observed in the taxonomic groups being studied, such as the presence or absence of wings or a metabolic pathway. But, even in these cases, the problem, at least computationally, is similar to generating these trees from sequence data, since in both cases the goal is to distribute all sets of attributes in the way that best represents their relationships according to given criteria.

Phylogenetic trees are usually calculated using algorithms that fall outside the scope of this paper, such as Markov chain Bayesian inference or distance matrix methods. However, there are promising results worth noting in the application of CP related techniques to this problem. One example is to use a fundamental property of rooted phylogenetic trees. Since each leaf represents a taxonomic group to classify and each interior node a most recent common ancestor (mrca) of the subtree that starts at that node, by labeling each interior node with a measure of how long ago the mrca lived (or how distant it is from its current descendants) we obtain a min-ultrametric tree, a tree for which the path from the root to any leaf goes through nodes labeled in a strictly increasing order. Using this property as a constraint, [9, 10] approached the computation of phylogenetic trees as a constraint satisfaction problem.

Answer-set programming has also been applied to phylogenetic trees. This form of declarative programming, geared toward processing rules with constraints, was used to enumerate phylogenetic trees for a set of taxa constrained to a maximum specified number of incompatible characters [11]. Characters are considered incompatible with a phylogenetic tree, in the strict sense, if they appear more than once. The reasoning is that it is more parsimonious to assume that all taxa that share a common character do so because they share a common ancestor with that character.

Another example of applying answer-set programming is in assembling phylogenetic trees by combining phylogenetic quartets. Quartet-based phylogeny reconstruction is a phylogenetic method in two steps. First, unrooted phylogenetic trees are estimated for all combinations of four taxa in the set of taxonomic groups to classify. In the second step, these quartets are combined to generate the complete tree. If all quartets are available and correct, the algorithm is polynomial in time. The problem arises when some quartets are ambiguous and thus cannot be generated with confidence or when the topology of some quartets are inconsistent with each other and cannot all fit in the same tree. In this case, the problem becomes one of minimizing the number of quartets that are rejected when assembling the tree, and much harder to solve. This process was implemented using answer-programming by combining the ultrametric tree constraint with the constraint that all used quartets must be compatible [12].

2.2 Haplotypes and SNP

The genetic variation of a population is a bioinformatics area where constraint solving and optimization has been applied. Rather than analyzing the external features of the individuals (their phenotypes), these studies focus instead on their genotypes (the ADN of their chromosomes). More specifically, many studies concentrate on variation in specific positions of the genome sequence, known as SNPs, where mutations are known to have occurred. If not under selection pressure, one of the variant tends to fixate, eventually, in the population, eliminating the polymorphism. Thus, for most SNPs, there are only two different nucleotide bases present in a given population.

Diploid individuals inherit one chromosome (or haplotype) from each of its parents. Denoting by A and a the two alleles (the two different nucleotide bases) of an SNP, an individual may inherit the combinations AA, aa, Aa, or aA. The latter two cases (biallelic or heterozygous SNP) cannot be easily distinguished by current experimental sequencing techniques that may only distinguish the cases AA, aa, and Aa or aA, that we will denote by 0, 1, and 2, respectively.

Given a set of m SNPs sites in some genomic block, all n individuals typically present different genotypes ($n \ll 3^m$). However, when a section of the genotype includes relatively closely spaced SNPs, linkage disequilibrium is higher and recombination is less likely. Although a set of n genotypes is explained by at most $2n$ haplotypes, it can generally be explained also by a much lower number of

haplotypes, and this smaller number of haplotypes is a better measure of population diversity. For example, the six genotypes 21212, 21110, 01112, 11212, 21211 can be explained by a set of only four haplotypes, 01110, 11011, 11110, and 01111, suitably combined. Genotype 21212, for instance, can be explained by the combination of haplotypes 01110 and 11011.

The haplotype inference problem (also known as phasing) can then be stated as follows: given a population of n individuals exhibiting a set of n genotypes find the set of unique haplotypes that exist in the population and find the pair of haplotypes that might exist in each individual, along with the respective probabilities.

The first computational approach to address this problem was Clark's subtraction method [13]. It first creates a set G of all genotypes in the population. The haplotypes that explain the genotypes in G with at most one biallelic SNP can be deterministically inferred and are used to initialize a set H. Then, it selects a haplotype from H and checks whether it can explain any of the genotypes in G. For each of these genotypes, it creates the complementary haplotype that is added to H and eliminates the genotype from G. The method proceeds until the genotype set G becomes empty.

This greedy algorithm aims at keeping small the cardinality of the resulting haplotype set. However, there are many possible orderings in which the haplotypes are selected from set H only upper bound on the cardinality of the set H are obtained, although certain heuristics (e.g., select the haplotype from H that explains more genotypes in G) have been shown to yield tight upper bounds on the minimum number of haplotypes.

In fact, assuming that nature is parsimonious, the haplotype inference problem can be reformulated into a minimization problem. As proposed in [14], the pure parsimonious haplotype inference (or haplotype inference by pure parsimony, HIPP) consists of finding the set of minimal cardinality that explains the genotypes of a population. The author then proposed an integer programming formulation and a technique, referred to as RTIP, that reduces the problem size without jeopardizing optimality.

The problem was shown to be NP-Hard in [15], who proposed a code (SDPHapler) to find approximate solutions (this problem was shown to be APX-hard [16]). Polynomial solutions were proposed, for instances, when all genotypes have at most two heterozygous sites [17], and other "islands of tractability" were investigated in [18]. For the general case, some IP based systems such as PolyIP [19] were developed with optimization techniques (e.g., cutting planes), adapted for the HIPP. Alternatively, Clark's reduction method was adapted to a branch and bound search minimization algorithm [20].

In addition to other IP and branch and bound formulations exploiting different techniques to improve efficiency, a constraint based system, SHIP, was proposed in [21] that formulated HIPP as a SAT problem. The core algorithm basically encodes a solution of the problem as a set of k haplotypes with $O(n^2m)$ constraints and $O(m^2 + nm)$, where n is the population size and m the number of SNPs, and proves whether the problem is satisfiable. The base algorithm obtains an optimal solution by iterating the size k of the haplotype set starting in some lower bound.

The authors present a number of improvements on the model, some of which are common to RTIP to reduce the problem size and others meant to break some symmetries. Of course, the computation of a good lower bound is of great importance in this problem and this is done by an approximate solution to a max-clique in a graph encoding incompatibilities of the genotypes. This lower bound estimate was subsequently improved [22] by solving an SAT problem through local search (based on the SKC variant of WalkSAT). An extensive comparison of HIPP solvers is presented in [23] that also compute tight upper bounds for the HIPP problem by making Clark's reduction algorithm less greedy in the inclusion of haplotypes in set H (by a technique named delayed selection). Moreover, they present a solver, RPoly, based on pseudo-boolean optimization and that uses an encoding similar to that of PolyIP, but with some optimizations that allow a significant reduction on the number of variables. The extensive performance comparison of a number of solvers for the HIPP problem on a set of 329 problem instances clearly states that the ILP approaches are significantly less efficient than the SAT (SHIP) and PBO (RPoly) approaches, by solving much less instances in 1,000 s (30% of the problem instances, against 81% of SHIP and 94% by RPoly). Competitive results were obtained with an answer set programming approach that relies on an underlying SAT solver similar to SHIP, which was also applied to HIPAG, a variation of the HIPP problem that only takes into account biallelic genotypes [24]. Finally, the HIPP problem has been addressed by a two level ant colony optimization approach which reportedly outperforms RPoly in the number of instances solved, but is in general much slower [25].

Although the various approaches seem to show an advantage in solving the HIPP problem with some hybrid techniques, some issues remain regarding the practical use of solutions to the HIPP problem. On the one hand, there may exist a large number of solutions. Lynce et al. [23] computed the set of all solutions of a small instance, SU100kb.25, with 34 genotypes and 15 sites, and found 48 different solutions with 17 haplotypes, 14 of which are common to all solutions. Computing the set of all solutions to the HIPP problem is #P-Complete which makes it very difficult to obtain the correct solution. Moreover, [26] have shown with experimental evidence on true haplotype data (not computationally obtained, but experimentally derived) that three of seven sets have solutions that are not parsimonious (in the worst case, the true solution has 32 haplotypes compared to the 28 haplotypes of the HIPP solution). The authors then propose to focus on the computation of backbones (the set of haplotypes that belong to all solutions found), namely those that are implicit (the explicit backbones are easy to compute, and correspond to the initial haplotypes selected in Clark's reduction method).

3 RNA Structure

RNA molecules are generally transient messengers carrying genetic information from DNA to the ribosome, where it is translated into proteins. For years, this "Central Dogma" was even thought to hold universally, but it is now known that many noncoding RNA (ncRNA) are directly functional in the cell, some even

playing roles similar to those of proteins. The growing number of RNA families being discovered in the last decades [27] led to an increased interest in modeling RNA structures.

RNA folding is not driven by the same forces that guide protein folding [28, 29], being dominated by the hydrogen bonds formed between complementary bases, so its modeling follows different approaches from those deployed for proteins. The base pairing in RNA can be represented by a graph, describing structural elements such as stems and pseudoknots. RNA secondary structure is defined by the graph of pairings between bases in the same RNA molecule. Secondary structure graphs do not exactly cover the whole set of pairings, but a large portion of them that can be drawn in the plane in a tree-like fashion. More precisely, there is a one-to-one correspondence between secondary structures and rooted oriented trees, such a restriction leading to very interesting mathematical properties [30]. Most RNA secondary structure elements involve several bases in a row pairing with another set of contiguous bases running in the opposite direction along the RNA sequence. This arrangement of stacked base pairs is called a stem, or an helix (due to its twisted form in 3D), and one can imagine it as the RNA sequence running to one side, looping and then coming back the other way with the two segments fitting together like a zipper.

Some elements, called pseudoknots, do not fulfill the definition of secondary structure. In a pseudoknot, the two sets of contiguous bases zip together in the same direction, a rarer configuration due to the need to twist the RNA chain in order to accommodate this relative placement. As RNA folding is believed to be partially hierarchical [31], this level of representation is not only useful for algorithmic reasons but also corresponds to the biological assumption that RNA starts to fold driven by these base pairings. Thus, the graphs of nucleotide contacts represent an important aspect of the kinetics of RNA folding, even if they do not give the complete picture of its three dimensional structure. While these graphs of secondary structure – structural elements stabilized by these local interactions between base pairs – are generally computed with combinatorial and discrete methods, a more detailed description of the tertiary structure – the overall spacial configuration of the molecule – is best tackled by geometric and continuous approximations methods.

In both cases, the related problems (structure prediction or display, structural homology finding, etc.) in the most general form are NP-hard and require adequate strategies to be solved. However, only a few of these questions were modeled using constraints, probably because, in the case of secondary structure, efficient polynomial algorithms were found that can solve most instances of this problem in practice, while, in the case of the tertiary structure, the number of known structures only increased recently, and the problem of determining RNA tertiary structure has attracted less interest from the research community in contrast with protein structures.

3.1 Questions Related to Secondary Structure

Secondary structure prediction is probably the problem that received most attention and to which a wider range of techniques were applied (for a rapid and

recent overview of structure prediction at different levels, see [32] for instance). Disregarding the pseudoknot configurations, the ab initio prediction of RNA secondary structure by combinatorial optimization (maximizing the number of pairings under certain rules or minimizing an energy function) is solvable via a low complexity dynamic programming algorithm for a single sequence [33,34], or even for a set of aligned sequences using stochastic context-free grammars [35] or other similar approaches. As these solutions are very efficient, a CP formulation of the problem would not be an improvement unless more is required.

In the case of SAPSSARN [36], which was one of the first attempts to introduce constraints for RNA secondary structure prediction, the additional feature is interactivity. The authors propose a dynamic treatment of constraints during structure prediction, where the computation of each predicted structure is interactive with the user, who may add or remove constraints. The interaction allowed by a constraint formulation of the question would not be possible during the computation if using a dynamic programming approach. In a similar perspective, the same authors proposed later, with RNASEARCH [37], a CP approach for RNA secondary structure display in the 2D space that optimizes the layout of the tree-like secondary structure of RNA – including pseudoknots – by trying to minimize stem overlaps.

Another source of interest in constraints is their expressive style. Their proximity to natural language leads to a direct formulation of a problem that may help avoid being trapped in a rigid algorithmic formulation. This is the case with [38,39] that takes advantage of the declarative nature of constraint network modeling for RNA motif search. A set of conserved RNA features, called a signature, is defined by a series of constraints. A set of template constraint types are defined to handle sequence content, distances, and pair stacking in helices that model the usual structural elements (the approach is not restricted to secondary structure alone). As the problem of finding RNA occurrences that satisfy a given signature is NP-complete for sufficiently general signatures, the previous works traditionally developed two approaches. The grammatical approach models the signature by a context-free grammar, excluding pseudoknots, thus falling into a case where the search can be performed by dynamic programming. Other approaches define the signature as a set of interrelated motifs and perform an exhaustive search using pattern matching techniques. The authors observe that a natural constraint network model formulation emerges from the direct description of the problem: the variables representing the target positions searched in the genomic sequence, the domains being intervals over integers. The specific constraint types and potentially huge domain size call for an adaptation of the usual CP schemes (filtering, backtracking) and for the creation of dedicated reduction operators (a preprocessing of data using specific data structures such as k-factor trees to speed up the search of potential occurrences). Functional RNA are also often interacting with other ligands and the traditional methods are unable to find RNA motifs in interaction with other molecules. The authors simply define and consider a new type of constraint to model the interactions between different molecules. The introduction of a descriptive language, with the scope of the description of new generations of RNA patterns, may have some similarities with an earlier attempt for a programming language dedicated to RNA secondary structure

[40] that could not use at that time all the constraint techniques that were developed during the last years. Both these works take full advantage of the handy declarative nature of constraints.

Constraints may also appear as a heuristic reduction for a polynomial algorithm. When several homologous sequences are available, but are not enough to compute a good starting alignment, a possible strategy is to search for a common secondary structure while aligning the sequences at the same time by maximizing a score which reflects both the structure and the alignment. Such an alignment, respectful of a (previously unknown) consensus structure, is called a structural alignment and when dealing with two sequences, the related problem is usually termed pairwise secondary structure prediction, or pairwise structural alignment. An early dynamic programming solution was provided by Sankoff [2] but the high algorithmic complexity of the exact recursion set he proposed makes them inapplicable on natural sequences and calls for heuristic reductions. The method became popular and stimulated a series of work trying to reduce the complexity. Part of the most recent works [41,42] use alignment constraints, based for instance on nucleotide alignment posterior probabilities. These constraints, defined over the possible structural alignments, drastically reduce the computation requirements but do not really call for dedicated CP techniques.

This reduction on the computationally demanding Sankoff algorithm implicitly suggests that the two approaches – dynamic programming (for which there exists a polynomial algorithm) and CP modeling – could be combined. Apart from the direct use of the constraint technique apparatus, the frequent occurrence of its terminology may lead to unexpected and potentially fruitful connections between remote areas. Secondary structures are, for instance, a special case of outer-planar graphs, which are graphs of low treewidth. Such tree decompositions are actually often used in CP modeling, and it would not be surprising that migrations of these ideas would help bridge CP techniques with different approaches such as dynamic programming.

3.2 Ab Initio 3D Structure Prediction

Less surprising is the use of constraints in three-dimensional structure prediction. Beyond the direct descriptivity of the question by constraints, the CP formalism also allows for the integration of information of very different nature, whatever their origin. Concerning RNA ab initio 3D structure prediction, still not much has been done and this should be related to the little number of available known structures, explained by a rather recent increase of interest for RNA. The McSYM research project, started at the University of Montreal in the 90s [43,45], was the first work addressing that question. MC-Sym builds 3D ribonucleic acid structures from low-resolution data by combining symbolic and numerical computations. The symbolic step generates all-atom sketches of 3D structures, using constraints derived from different sources, such as nuclear magnetic resonance (NMR) spectroscopy data, X-Ray crystallography, chemical modifications, secondary structure information,

and so forth. The conformational search space is defined by spatial relations among RNA bases, which are encoded by transformation matrices that correspond to the transformation of a base referential into another. The inference engine is implemented as a Boolean constraint solver that accepts or refuses a structure whether or not all the given and inferred constraints are satisfied or not. In a context where the efforts are focused on a tighter connection between the different levels of structural description [46], CP appears to be an easily extensible framework that allows for the exploration and the discovery of more general structural rules. Two decades ago, comparative sequence analysis had started to reveal novel tertiary interactions between more than two bases [47], later confirmed with examples from the accumulated knowledge of 3D structures. The elucidation of the relationships between sequences and RNA motifs – in the broadest sense: recurrent structural elements subjects to constraints [48] – becomes one of the current challenges in RNA structure comprehension.

Although CP terminology is more flexible than alternative approaches, it usually implies an NP-hard formulation of the problem. When a polynomial approach exists, as in RNA secondary structure calculations, the benefits then strongly depend on the new types of information the constraints can handle. When, on the contrary, the question is more directly expressible by constraints, it usually calls for dedicated methods that can enrich the corpus of CP techniques while also providing new domains of application.

4 Protein Structure Modeling

Modeling protein structures is a complex problem due to the size and flexibility of these macromolecules, as proteins consist of long polymers of amino acid residues, typically containing thousands of atoms, intricately folded in a structure determined by physical and chemical interactions between these atoms and with the solvent, usually water or a lipid membrane. One can conceive of two different categories of protein modeling problems. One is protein structure prediction, where the structure of the molecule is to be estimated from chemical and physical considerations. The other is the determination of a protein structure given a set of constraints obtained from experimental data, such as NMR spectroscopy.

The most important source of macromolecular structure information is the protein data bank (PDB), which contains nearly 60,000 structures, mostly of proteins but also including nucleic acids and protein/nucleic acid complexes. It is an open access database that can be accessed or downloaded from several organizations (see the Worldwide PDB site at www.wwpdb.org). The structure files include the atomic coordinates, the identification of each atom and monomer or ligand in a compound dictionary that specifies additional structural data (such as chemical bonds), and often specific experimental information such as atom occupancy factors for X-Ray crystallography structures or NMR constraints.

4.1 Structure Prediction

Protein structure prediction is generally seen as an optimization problem, the goal being to find the structural configuration that minimizes the free energy of the system. Since the system includes both the protein and all the solvent molecules surrounding it, and since the free energy includes both enthalpy and the contribution of entropy factors that are difficult to compute, this is a computationally intensive problem. Furthermore, the assumption that the correct structure is at the global energy minimum seems not to hold universally, and may in some cases correspond to a local minimum where the structure is retained during folding due to high energy barriers [49]. Thus, the traditional approach to protein structure prediction relies on molecular dynamics or simulated annealing, based on models of the physical properties of these macromolecules (ab initio structure prediction). It often resorts to using supercomputers or networks to meet the large computational demands, one of the most famous examples being the Folding@Home project [50]. More recently, protein structure prediction has become dominated by methods that rely on identifying structural features that the target protein has in common with the ever increasing set of known protein structures. Even so, there have been advances in the application of CP to these problems using lattice models of protein structure and interaction.

4.1.1 Lattice and HP Models

Despite its importance and the interest it generates, a definitive solution to the problem of predicting protein structures still eludes all research efforts and approaches proposed over the last decades, both because of the difficulty in computing the free energy of the system and the complexity of the structure. But, since proteins are composed of chains of amino acids (more accurately amino acid residues) connected by peptide bonds, the problem can be simplified if rather than considering protein models at an atom level, proteins are modeled at the amino acid level. This way, the variables represent amino acids, either their centers of mass or the alpha carbon in the protein backbone, with the main difference being the way protein chain is considered, either by following the backbone or the average of the atomic positions at each amino acid residue. In both cases, these models are rather simplified representations, since most physical and chemical properties depend on atoms or small chemical groups and are difficult to assign to amino acids abstractions.

Still, a number of simplified models have been proposed (see [51] for an overview) assuming some simplified characterization of the amino acids and placing them in some lattice structure. Among these models, the HP model [52] is worth considering. In this simplified model, amino acids are labeled by their hydrophilic nature, being classified as either hydrophobic (H) or as hydrophilic or polar (P). Since the solvent is water, H amino acids tend to be packed in the interior of the protein. The HP models this tendency indirectly by minimizing an energy function modeled by the (negative) number of contacts between H amino acids that are neighbors in the lattice.

The original problem was formulated for a two dimensional square lattice, but it can easily be extended to a three dimensional cubic grid. In both the square and the cubic lattice, the problem has been shown to be NP-complete (respectively in [53] and [54]).

Although the problem was not formulated as a constraint problem and some algorithms were used to solve it for not using CP technology (e.g., [55–59]), such model can be adequately formulated as a finite domain constraint optimization problem: find the position of the amino acids (in the finite set of vertices of the lattice) that satisfy some constraints (successive amino acids in the protein chain must be neighbors, and no two amino acids can occupy the same position) and optimize the objective function (number of contacts between H amino acids).

As such, [60] were the first to attempt to address this problem as a constraint optimization problem with a cubic lattice. An interesting feature of this problem is that it presents many geometric symmetries (namely rotations). To handle this and other types of symmetries, the authors proposed what was claimed to be the first declarative method that could be applied to arbitrary symmetries [61]. Among other tests, the authors have shown that they could improve the number of search steps and run times of one to two orders of magnitude to find optimal solutions in cubic lattices for proteins of around 30 amino acids.

This model can be improved in two complementary forms, either by changing the energy function or the lattice that is considered. The HPNX model [62] is an extension of the HP model in the first direction. Now in addition to hydrophobic (H) and polar (P) amino acids are classified as negatively charged (N) and neutral hydrophilic (X), and each amino acid pair in contact has a weighted contribution to the energy function (see Table 1, below). These models have been addressed by CP [62], but they do not overcome some important drawbacks of the cubic lattice with respect to structure prediction.

On the one hand, the model prevents, by design, that two amino acids with the same parity in the protein chain establish a contact, which is unreasonable. Moreover, the right angles between amino acids are not very realistic. In fact, [63] has shown that a face-centered cubic lattice, FCC, (a conformation that guarantees optimal packing of spheres [64]) would better approximate the packing of amino acids in a protein, and [65] have shown that the FCC lattice lead to root mean square deviations (RMSDs) of 1.78 Å with respect to the real conformation, rather than the RMSD of 2.84 Å obtained with a cubic lattice.

Table 1 Comparison of the contact scores for the HP (*left*) and HPNX (*right*) models

	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

Obtaining optimal solutions for the HP model on FCC lattices is very hard. In addition to various heuristic approaches (hydrophobic zipper [66], genetic algorithms [67], chain-growth, [59] and approximate algorithms, [68]) a number of CP techniques have been applied with significant success. An interesting model has been proposed in [69] that rather than solving the structure determination problem directly, converts it into a threading problem by adjusting the amino acid sequence to precomputed hydrophobic cores. They showed how to compute such hydrophobic cores for both cubic and FCC lattice models. In particular, they were able to find maximally hydrophobic cores for the FCC lattice for up to 100 H amino acids within seconds.

However, threading the sequences to the hydrophobic cores is still a difficult problem. Again a CP approach was proposed [70], combining path constraints and all-different constraints, to obtain a self-avoiding path constraint that was subsequently used in threading the amino acid sequences to the cores. The results obtained for randomly generated proteins are quite satisfactory for small proteins (100% success to proteins with 25 H amino acids with runs of 15 min) but the success rate decays for longer proteins (only 50% success for proteins with 100 H amino acids).

A different approach to find minimal energy HP models in FCC lattices was proposed in [71] that applied a tabu search metaheuristic to local search. However, the authors have shown in [72] how to use CP techniques to improve good solutions previously found with tabu search in order to exploit such large neighborhoods (LNS – large neighborhood search). For any solution (i.e., a sequence of n amino acids) they randomly select an internal subsequence and perform a systematic search for alternatives, keeping the structure of the prefix and postfix sequences. By adopting a number of relevant modeling decisions (e.g., heuristics and redundant constraints), they obtained quite good results in a set of benchmarks (the Harvard instances). In particular, they describe how the LNS search rapidly improves the tabu search solutions, but it must be noticed that the approach uses very substantial computing power (60 Intel base, dual core, dual processor Dell Poweredge 1885 blade server) running for a few seconds after a tabu search taking a few minutes. However, they cannot find optimal solutions that [69] could find in some randomly generated sequences, finding instead solutions within 3–10% of the optimum in runs of a few hours.

An improvement of the lattice models is explored in [73] where the authors use more information, namely secondary structures and disulfide bonds as additional constraints and subsequently replace the HP amino acid based model with an all atom model to allow an effective measurement of the RMSD between some known proteins and the predicted models (PDB code 1YPA, with RMSD of 9.2A within 116.9 h of computation).

The large computational time required led to the proposal of a specialized solver for lattice models [74]. The authors define special purpose encodings for the domains to improve propagation in a number of specialized constraints useful in these problems (e.g., a constraint, next, to enforce a sequence of amino acids and various constraints that model the spatial distance of amino acids and are useful to handle contacts between spatially near amino acids). Then, the authors present COLA,

a constraint solver that exploits parallel search in constrained optimization problems and assess its performance in the protein structure determination. The authors also address the specification of rigid groups useful for the modeling of secondary structures in protein structures, which were addressed in a later implementation [75]. A pure CP was shown to be improved by its hybridization with local search, and discuss how to improve performance in future work.

Their experiments show significant speed-ups over general purpose constraint solvers such as SICStus and GNU Prolog, but are not directly comparable with the HP model, as they adopt a more comprehensive set of values for any pair of amino acids in contact, without abstracting them into the H and P categories. The comparisons made with protein structures taken from the PDB are also made in terms of the minimal energy solutions obtained, rather than the difference between spatial conformations obtained (e.g., in RMSD) making it hard to assess the results obtained.

HP and lattice models seem a promising way of modeling this problem. One potential advantage of lattice models is the possibility of coupling local search methods in parallel with constraint propagation, improving the enumeration heuristics, and the efficiency with which desired solutions can be found. Experience with PSICO (see below) suggests that without the simplicity and geometric elegance of lattice models this coupling of local search with constraint propagation is not feasible, since the domains of the atomic coordinates are too large at the early stages of the computation to allow the application of any energy function. Thus, without the lattice models, it seems that one must separate the process in two stages, constraint processing followed by local search for refining the structures.

However, and despite the promising results obtained with lattice models, it seems that, by themselves, these models are not competitive in real-life folding problems. The best predictors in the critical assessment of techniques for protein structure prediction (CASP) favor very different approaches, such as threading algorithms [76] or metaservers aggregating predictions from ab initio calculations and ROSETTA fragment insertion [77, 78]. The overview of the first decade of CASP, in 2005, classifies three different types of structural prediction problems [79]. When there is a good sequence similarity between the target protein and a protein, or proteins, with known structure, it is possible to infer the structure of the target by homology modeling followed by suitable refinement with local search. If the target protein is more distantly related to known structures, it is still possible to identify folds, local structures that are stable and common to several proteins, and to use this information to assemble the target structure. Finally, even when no good homology match is found, the large number of structures known makes it possible to choose structure fragments that are good candidates for assembling the target structure, based on secondary structure propensity and sequence compatibility. This is the approach used in the highly successful ROSETTA algorithm, and the current trend in protein structure prediction. Still, prior to this shift to template-based optimization algorithms, there is some mention of lattice models being used in conjunction with ab initio computations to try to predict protein structure [80] that may possibly be useful in practice when integrated with these more informed methods.

4.2 Protein Structure Determination

From a CP perspective, structure determination from experimental data can be seen as conceptually different from structure prediction, since the latter aims at finding an energy minimum, or simulating folding dynamics, while the former must provide structural models consistent with constraints derived from experimental data. However, the classical approach in the biochemistry community has been to treat structure determination also as an optimization problem, using well established algorithms for simulated annealing and molecular dynamics (e.g., the widely used DYANA/CYANA software [81]). In this approach, the experimental constraints are simply treated as additional factors in the function to optimize. This has the advantage of making the method implicitly more resistant to experimental noise, but at the cost of not using the constraints to narrow down the search space.

Although X-Ray crystallography is the main experimental technique for the determination of macromolecular structures, in this case, the experimental data contain the positions of all atoms, requiring only the deconvolution of the X-Ray diffraction patterns in order to obtain the structure. Computationally, the more interesting problem is with NMR spectroscopy, which accounts for approximately 15% of known protein structures. This technique provides distance constraints between atom pairs and angular constraints on the relative orientation of interatomic bonds, and it is from this set of constraints that the structure must be computed. One example of the application of CP to process these structural constraints is processing structural information with CP and optimization (PSICO). This algorithm considers the atomic coordinate triplets as variables with a continuous domain defined by solid shapes defined by sets of cuboid volumes, due to the convenient property of retaining that shape when intersected with other similarly shaped volumes. With this representation of the atomic coordinate domains, it is possible to propagate interatomic distance constraints efficiently [82], and even include more generic geometric constraints on the relative coordinates of rigid groups or their orientation with respect to a torsion angle [83].

The propagation of distance constraints is easy to illustrate for the simpler case of an upper bound on the distance between two atoms. This is one kind of information that can be obtained from NMR experiments. If two atoms can be, at most, separated by a distance of d, then each atom must be within a neighborhood of distance d of the other atom. These neighborhoods can be computed from the domain of each atom and the constraint propagated by intersecting the domain of each atom with this d neighborhood of the other atom. Lower bounds on the allowed distances can be propagated by adding cuboid volumes contained in the domain of each atom to identify regions from which the atom must be excluded in order to respect these constraints.

This approach led to some promising results. For example, with proteins ranging from 400 to 700 non-Hydrogen atoms and 10,000–15,000 constraints, the solutions found had RMSD values from 2 to 3 Å, taking a few minutes to calculate [82]. In addition, there were promising preliminary results with the propagation of higher order constraints defining rigid groups of atoms and their spatial relations.

These constraints are propagated by calculating which parts of the domain of each atom are inaccessible to that atom due to the constraint imposed by the group and the domains of the other atoms. These groups range from small parts of amino acid residues joined by bonds that are free to rotate to large secondary structure elements such as alpha helices, and the results on randomly generated groups and groups simulating secondary structure elements showed that this algorithm can improve pruning significantly even for small groups. For larger groups, it is even faster than propagating the set of binary distance constraints on those atoms [83].

5 Protein Interaction

Modeling how two proteins interact (protein docking) is a similar problem to the modeling of protein structures in the sense that the goal is to obtain a macromolecular structure. However, the starting point is the known structures of the interacting partners, so the problem is not so much predicting the folding of the molecules involved but, rather, how two known structures best fit together. Although this fit is governed by intermolecular "forces," such as electrostatics and entropy contributions from the solvent, since the interaction is not covalent, it is very weak at any single point, requiring a large surface of contact to stabilize the protein complex. Thus, most protein docking algorithms rely on a geometric filtering stage that identifies those configurations with the largest contact surface.

A widely used class of protein docking algorithms is based on the fast Fourier transform (FFT) computation of correlation matrices [84]. In this approach, each docking partner is represented as a three-dimensional matrix in which numerical values distinguish between the surface regions (e.g., positive value), the core of the molecule (e.g., negative value), and the empty surrounding space (zero). The correlation matrix indicates the total score for each relative placement of these matrices, thus distinguishing the configurations with a large surface contact from those with smaller surface contacts or forbidden overlaps with the core regions. Thought the FFT algorithm is efficient in time, of $O(n^3 log(n)^3)$, the need to represent all grids as numerical matrices results in a significant memory footprint. In practice, a CP approach based on a simpler representation of the protein shapes can perform the calculations in less time and with orders of magnitude lower memory requirements [85].

This algorithm, BiGGER, represents each protein shape as a grid, conceptually, just like in the FFT approach. However, the grids defining core and surface regions for each partner are encoded as sets of line segments that, for each (Z,Y) pair, define the arrays of cells corresponding to surface and core regions along the X axis. Unlike FFT, the search does not involve computing all the correlation matrices but actually searching through the spatial configurations by placing one partner in a position relative to the other. The encoding of the grids makes it easy to restrict the search space by maintaining bounds consistency on the constraints that forbid core–core overlaps, and those requiring a minimum value for the surface–surface

overlaps (branch and bound) or, and more significantly, restrict the search space to configurations consistent with geometric constraints obtained from experimental data.

Empirical results suggest that knowing even a few residues in one partner that must be in contact with the other protein partner may be enough, in most cases, to guide the search to the right complex structure [86], and this can be done efficiently in a flexible manner by enforcing a cardinality constraint on a subset of a given set of potential contacts [85]. The reasoning is that experimental data on residue contacts are usually obtained from either the perturbation of the residue during the interaction (as measured by spectroscopy) or the perturbation in the formation of the complex by mutating or otherwise modifying one residue. Such data suggest that the residue is at the interface, but there are other possibilities that give the same results, such as conformational changes in the protein, either during the formation of the complex or due to the changes in that residue. The ability to specify a set of candidate contacts, each of which can be between one residue of one partner and any residue of the other, and to impose a constraint on how many of those must be verified, allows the user to model quite naturally the uncertainty in the experimental data. For example, given ten potential interface residues, one may require that at least five of those, not specified in advance, must be present at the interface of the model complex. As is characteristic of CP, apart from helping to narrow down the right structures, this also reduces the search time (by about an order of magnitude).

This approach contrasts with the classical approach in the biochemistry community which is, much like with the determination of protein structures, to include the experimental constraints as additional factors in an energy function that is being minimized. HADDOCK [87,88] (High Ambiguity Driven biomolecular DOCKing), for example, is a docking application based on local-search NMR assignment and structure calculation software that predicts protein complexes by minimizing the violation of geometric constraints included in the global function.

As with protein structure modeling, it seems that the potential of CP techniques has not yet been fully appreciated by the biochemistry community. The problem, we feel, lies mostly with the difficulty of crossing the gap between theoretical studies and proof-of-concept of the algorithms, and the actual application to real-life problems, as the latter requires a tighter collaboration with the researchers involved in those problems and the integration of the CP solvers in the often large and complex processing pipeline that goes from the data to the final refined model.

6 Systems Biology

Systems biology is a new area in biology which aims at understanding biological systems at the higher level of the interactions between all components [89]. In particular, biological networks of gene expression and regulation, protein interaction, metabolic pathways, and such processes required for life. The development of computational models plays a key role in systems biology [90], and a number of

techniques have been used to model different levels of abstractions of such systems. In Boolean networks for gene regulation, only the presence or absence of substances is represented and the systems dynamics is modeled by state transitions: variables denote whether a gene is expressed and Boolean functions relate variables in different states [91]. A richer expressiveness is obtained in qualitative networks [92] where multivalued variables are used to represent various degrees of, for example, gene expression. Other computational formalisms developed for systems verification have also been applied to biological systems such as Petri-nets [93], p-calculus [94], and other types of spatial/temporal calculi [95]. Still, the continuous behavior of biological systems is better captured by means of sets of ordinary equations. Various systems exist that exploit one or more of these formalisms, such as BIOCHAM [96] that allows the representation and reasoning about these systems at different levels of detail ranging from Boolean networks, to temporal logics and differential equations.

Despite existing proposals for incorporating differential equations as first order constraints in the CP paradigm [97], its use in modeling biological processes has been limited [98] given the problems of scalability of the approach.

Nevertheless, CP has been applied to specific problems in systems biology. For example, [99] analyses and proposes extensions to the concurrent CP paradigm (CC) to model systems biology problems. They analyze Timed CC, Timed Default CC, and Hybrid CC extensions (Hcc), and show how the latter can express differential equations and initial value problems in addition to algebraic constraints. This allows the modeling of system behavior by means of a sequence of alternating point and interval phases. The authors also show how to model a number of key aspect of biological systems in Hcc constructs, such as Reaching thresholds, time and concentrations, kinetics, gene interaction, and stochastic behavior, and illustrate these concepts in a comprehensive example of cell differentiation for a population of X. Laevis cells.

An alternative to modeling changes is to model steady state behavior of molecular networks. These can now be represented by means of graphs, and reason about the behavior of the networks by finding some patterns of reaction as paths in these graphs. This was a major motivation for the development of CP(Graphs) [100], where a new domain (graphs) was introduced in the CP and a set of (global) constraints where specified (such as path and reachable constraints). Some filtering algorithms where proposed and tested in the analysis of metabolic networks, namely for finding metabolic pathways that are in use in the cell, given a list of reactions detected with DNA chips.

Answer-set programming has also been applied to metabolic pathway and gene expression data, as a declarative approach to processing the rules and constraints that characterize these problems. One example is the check for consistency between metabolic pathway databases, which store the theoretical or presumed knowledge about the reactions occurring inside an organism, and experimental data, such as gene expression data derived from DNA microarrays or direct measures of metabolite levels in the organism [101]. This approach processes the rules implied by the presumed influence of regulator genes and the assumed reaction pathways with the

experimental data about metabolite levels and gene expression. If data and theoretical assumptions are inconsistent, this method also identifies minimal subsets of regulatory influences that can account for the inconsistency.

Another example is an action language designed to describe metabolic pathways or other biological networks, their changes and queries about such systems [102]. This language describes the properties that change and the actions that cause such changes, and has the important property that any description in this language be automatically translated into an answer-set program and take advantage of the efficiency of answer-set solvers.

In essence, systems biology poses several challenges to CP and declarative approaches in general, by addressing system dynamics problems. Some problems fall somewhat outside the main stream of current CP research but open the possibility of addressing new domains and exploit constraint technology (e.g., filtering) to these domains.

As for data sources, a good starting point would be the Kyoto Encyclopedia of Genes and Genomes (KEGG) [103], a knowledge base combining information on metabolic pathways, functional hierarchies that include genes, proteins, drugs, diseases and organisms, and several annotated databases on genes and ligand compounds. From a CP perspective, perhaps the most interesting part of KEGG would be the PATHWAY database, describing metabolic pathways, which are chains of biochemical reactions catalyzed by enzymes and regulated by the activity of the genes that code for each enzyme. The study of metabolic pathways combines graph problems with dynamic systems modeled by differential equations, both fields of interest to CP researchers.

7 Conclusion

This article is an overview of the application of CP technology to the broad area of bioinformatics, an application domain with increasing importance due to the advances in biological and biochemical experiments, and the huge amount of data that must be subsequently processed. In this paper, we have shown that the relationship between CP and bioinformatics holds in both directions. On the one hand, CP programming has shown its potential in many bioinformatics applications, and for some specific models and problems, we have provided examples where it is the most appropriate computational approach to deal with them. Nevertheless, we acknowledge that realistic applications in bioinformatics (as well as in other application areas), given the wealth of questions raised by the ever increasing amounts of experimental data being collected, usually demand a variety of computational techniques. When attempting to solve realistic bioinformatics problems, a comprehensive analysis of these problems should thus be made, both by computer scientists and biologists and biochemists, to determine whether CP can effectively be used, in which subproblems and how these relate to the whole application.

Once such cautions are taken, one must recognize that the computational problems that bioinformatics poses, some of which were addressed in the previous sections, have been a source of inspiration for CP. They have provided new domains where CP can be applied, of which we can refer lattice structures, spatial cuboids, graphs, and even temporal domains (differential equations and timed events). The complexity of such problems demand the exploitation of advanced CP techniques to improve search and a number of them have been already applied, namely various types of global constraints, specialized heuristics, the interaction of CP and local search, and the exploitation of some forms of parallel execution. We can only foresee that, given the increasingly importance of bioinformatics and the rich problems it poses, this trend can only continue in the years to come.

References

1. Smith TF, Waterman MS (1981) Identification of common molecular subsequences. J Mol Biol 147:195–197
2. Sankoff D (1985) Simultaneous solution of the RNA folding, alignment and protosequence problems. SIAM J Appl Math 45:810–825
3. Lipman DJ, Pearson WR (1985) Rapid and sensitive protein similarity searches. Science 227(4693):1435–1441
4. Altschul SF, Gish W, Miller W, Myers EW, Lipman DJ (1990) Basic local alignment search tool. J Mol Biol 215(3):403–410
5. Roland HC Yap (2001) Parametric sequence alignment with constraints. Constraints 6(2–3):157–172
6. Will S, Busch A, Backofen R (2008) Efficient sequence alignment with side-constraints by cluster tree elimination. Constraints 13(1–2):110–129
7. Carlsson M, Beldiceanu N (2004) Multiplex dispensation order generation for pyrosequencing. In: CP'2004 workshop on CSP techniques with immediate application, Toronto, Canada, 27 September 2004
8. Benson DA, Karsch-Mizrachi I, Lipman DJ, Ostell J, Wheeler DL (2004) GenBank: update. Nucleic Acids Res 32(Database issue):D23–D26
9. Gent IP, Prosser P, Smith BM, Wei W (2003) Supertree construction using constraint programming. In: Proc CP2003. Lecture notes in computer science, vol 2833. Springer, Berlin
10. Moore NC, Prosser P (2008) The ultrametric constraint and its application to phylogenetics. J Artif Intell Res 32:901–938
11. Brooks DR, Erdem E, Erdogan ST, Minett JW, Ringe D (2007) Inferring phylogenetic trees using answer set programming. J Automat Reason 39(4):471–511
12. Wu G, You JH, Lin G (2007) Quartet-based phylogeny reconstruction with answer set programming. IEE/ACM Trans Comput Biol Bioinform 4(1):139–152
13. Clark AG (1990) Inference of haplotypes from PCR-amplified samples of diploid populations. Mol Biol Evol 77:111–122
14. Gusfield D (2003) Haplotype inference by pure parsimony. In: 14th Annual symposium on combinatorial pattern matching (CPM03). Springer, Heidelberg, pp 144–155
15. Huang Y-T et al (2005) An approximation algorithm for haplotype inference by maximum parsimony. J Comput Biol 12:1261–1274
16. Lancia G et al (2004) Haplotyping populations by pure parsimony: complexity of exact and approximation algorithms. INFORMS J Comput 16:348–359
17. Cilibrasi R et al (2005) On the complexity of several haplotyping problems. In: 5th Workshop on algorithms in bioinformatics (WABI 2005). Springer, Mallorca, pp 128–139

18. Sharan R et al (2006) Islands of tractability for parsimony haplotyping. IEEE/ACM Trans Comput Biol Bioinform 3:303–311
19. Brown D, Harrower I (2006) Integer programming approaches to haplotype inference by pure parsimony. IEEE/ACM Trans Comput Biol Bioinform 3(2):141–154
20. Wang L, Xu Y (2003) Haplotype inference by maximum parsimony. Bioinformatics 19:1773–1780
21. Lynce I, Marques-Silva J (2006) Efficient haplotype inference with Boolean satisfiability. In: AAAI conference on artificial intelligence, pages 104109, July 2006
22. Lynce I, Marques-Silva J, Prestwich S (2008) Boosting haplotype inference with local search. Constraints 13(1):155–179
23. Lynce I, Graa A, Marques-Silva J, Oliveira AL (2008) Haplotype inference with boolean constraint solving: an overview. In: Proceedings of 20th IEEE international conference on tools with artificial intelligence (ICTAI 08), Dayton, OH, 2008
24. Erdem E, Erdem O, Türe F (2009)In: HAplo-ASP: haplotype inference using answer set programming, LPNMR09. Lecture notes in computer science, vol 5753. Springer, Berlin, pp 573–578
25. Benedettini S, Roli A, Di Gaspero L (2008) Two-level ACO for haplotype inference under pure parsimony. In: ANTS conference, 2008, pp 179–190
26. Climer S, Jäger G, Templeton AR, Zhang W (2009) How frugal is mother nature with haplotypes? Bioinformatics 25(1):68–74
27. Eddy SR (2001) Non-coding RNA genes and the modern RNA world. Nat Rev Genet 2(12):919–929
28. Tinoco I, Bustamante C (1999) How RNA folds. J Mol Biol 293:271–281
29. Moore PB (1999) The RNA folding problem. In: The RNA world, 2nd edn. CSHL Press, Cold Spring Harbor, pp 381–401
30. Waterman MS (1995) RNA secondary structure. In: Introduction to computational biology. Chapman and Hall, London, pp 327–343
31. Wu M, Tinoco I (1998) RNA folding causes secondary structure rearrangement. Proc Natl Acad Sci USA 95:11555–11560
32. Capriotti E, Marti-Renom MA (2008) Computational RNA structure prediction. Curr Bioinform 3(1):32–45
33. Nussinov R, Jacobson AB (1980) Fast algorithm for predicting the secondary structure of single stranded RNA. Proc Natl Acad Sci USA 77:6309–6313
34. Jaeger JA, Turner DH, Zuker M (1989) Improved predictions of secondary structures for RNA. Proc Natl Acad Sci USA 86:7706–7710
35. Knudsen B, Hein J (1999) RNA secondary structure prediction using stochastic context-free grammars and evolutionary history. Bioinformatics 15(6):446–454
36. Gaspin C, Westhof E (1995) An interactive framework for RNA secondary structure prediction with a dynamical treatment of constraints. J Mol Biol 254(2):163–174
37. Gaspin C (2001) RNA secondary structure determination and representation based on constraints satisfaction. Constraints 6(2–3):201–221
38. Thebault P, de Givry S, Schiex T, Gaspin C (2006) Searching RNA motifs and their intermolecular contacts with constraint networks. Bioinformatics 22(17):2074–2080
39. Zytnicki M, Gaspin C, Schiex T (2008) Darn! a weighted constraint solver for RNA motif localization. Constraints 13(1–2):91–109
40. Billoud B, Kontic M, Viari A (1996) Palingol: declarative programming language to describe nucleic acids secondary structures and to scan sequence databases. Nucleic Acids Res 24(8):1395–1404
41. Harmanci AO, Sharma G, Mathews DH (2007) Efficient pairwise RNA structure prediction using probabilistic alignment constraints in dynalign. BMC Bioinformatics 8:130
42. Dowell RD, Eddy SR (2006) Efficient pairwise RNA structure prediction and alignment using sequence alignment constraints. BMC Bioinformatics 7:400
43. Major F, Turcotte M, Gautheret D, Lapalme G, Fillion E, Cedergren R (1991) The combination of symbolic and numerical computation for three-dimensional modeling of RNA. Science 253:1255–1260

44. Shapiro BA, Yingling YG, Kasprzak W, Bindewald E (2007) Bridging the gap in RNA structure prediction. Curr Opin Struct Biol 17(2):157–165
45. Gautheret D, Major F, Cedergren R (1993) Modeling the threedimensional structure of RNA using discrete nucleotide conformational sets. J Mol Biol 229:1049–1064
46. Shapiro BA, Yingling YG, Kasprzak W, Bindewald E (2007) Bridging the gap in RNA structure prediction. Curr Opin Struct Biol 17(2):157–165
47. Gutell RR, Power A, Hertz GZ, Putz EJ, Stormo GD (1992) Identifying constraints on the higher-order structure of RNA: continued development and application of comparative sequence analysis methods. Nucleic Acids Res 20:5785–5795
48. Leontis NB, Lescoute A, Westhof E (2006) The building blocks and motifs of RNA architecture. Curr Opin Struct Biol 16(3):279–287
49. Lazaridis T, Karplus M (2000) Effective energy functions for protein structure prediction. Curr Opin Struct Biol 10(2):139–145
50. Shirts MR, Pande VS (2000) Screen savers of the world, unite! Science 290:1903–1904
51. Dill KA, Bromberg S, Yue K, Fiebig KM, Yee DP, Thomas PD, Chan HS (1995) Principles of protein folding – a perspective of simple exact models. Protein Sci 4:561–602
52. Lau KF, Dill KA (1989) A lattice statistical mechanics model of the conformational and sequence spaces of proteins. Macromolecules 22:3986–3997
53. Crescenzi P, Goldman D, Papadimitriou C, Piccolboni A, Yannakakis M (1998) On the complexity of protein folding. J Comput Biol 5(3):423–466
54. Berger B, Leighton T (1998) Protein folding in the hydrophobic-hydrophilic (HP) model is NP-complete. J Comput Biol 5(3):27–40
55. Yue K, Dill KA (1996) Folding proteins with a simple energy function and extensive conformational search. Protein Sci 5(2):254–261
56. Abkevitch VI, Gutin AM, Shakhnovich EI (1995) Impact of local and non-local interactions oin thermodynamics and kinetics of protein folding. J Mol Biol 252:460–471
57. Unger R, Moult J (1996) Local interactions dominate folding in a simple protein model. J Mol Biol 259:988–994
58. Hinds DA, Levitt M (1996) From structure to sequence and back again. J Mol Biol 258:201–209
59. Bornberg-Bauer E (1997) Chain growth algorithms for HP-type lattice proteins. In: Proceedings of RECOMB97. 1st International conference on Research in computational molecular biology, pp 47–55
60. Backofen R (1998) Constraint techniques for solving the protein structure prediction problem. In: Proceedings of CP98. Lecture notes in computer science, vol 1520, pp 72–86
61. Backofen R, Will S (2002) Excluding symmetries in constraint-based search. Constraints 7(3):333–349
62. Backofen R, Will S, Bornberg-Bauer E (1999) Application of constraint programming techniques for structure prediction of lattice proteins with extended alphabets. Bioinformatics 15(3):234–242
63. Bagci Z, Jernigan RL, Bahar I (2002) Residue coordination in proteins conforms to the closest packing of spheres. Polymer 43:451–459
64. Cipra B (1998) Packing challenge mastered at last. Science 281:1267
65. Park BH, Levitt M (1995) The complexity and accuracy of discrete state models of protein structure. J Mol Biol 249:493–507
66. Cooperativity in protein-folding kinetics. Proc Natl Acad Sci USA 90:1942–1946 (1993)
67. Unger R, Moult J (1993) Genetic algorithms for protein folding simulations. J Mol Biol 231:75–81
68. Agarwala R, Batzoglou S, Dancik V, Decatur SE, Farach M, Hannenhalli S, Muthukrishnan S, Skiena S (1997) Local rules for protein folding on a triangular lattice and generalized hydrophobicity in the HP-model. J Comput Biol 4(2):275–296
69. Backofen R, Will S (2006) A constraint-based approach to fast and exact structure prediction in three-dimensional protein models. Constraints 11(1):5–30
70. Backofen R, Will S (2001) Fast, constraint-based threading of HP-sequences to hydrophobic cores. In: Proceedings of CP01. Lecture notes in computer science, vol 2239, pp 494–508

71. Cebrian M, Dotu I, Van Hentenryck P, Clote P (2008) Protein structure prediction on the face centered cubic lattice by local search. In: Proceedings of AAAI08, pp 241–245
72. Dot I, Cebrin M, Van Hentenryck P, Clote P (2008) Protein structure prediction with large neighborhood constraint programming search. In: Proceedings of CP08. Lecture notes in computer science, vol 5202, pp 82–96
73. Dal Pal A, Dovier A, Fogolari F (2004) Constraint logic programming approach to protein structure prediction. BMC Bioinformatics 5:186
74. Dal Pal A, Dovier A, Pontelli E (2007) A constraint solver for discrete lattices, its paralelization and application to protein structure prediction. Software Pract Ex 37(13):1405–1449
75. Cipriano R, Pal AD, Dovier A (2008) A hybrid approach mixing local search and constraint programming applied to the protein structure prediction problem. In: Proceedings of WCB08, Paris, May 2008
76. Zhang Y (2008) I-TASSER server for protein 3D structure prediction. BMC Bioinformatics 9:40
77. Fischer D (2006) Servers for protein structure prediction. Curr Opin Struct Biol 16:178–182
78. Bonneau R, Tsai J, Ruczinski I, Chivian D, Rohl C, Strauss CE, Baker D (2001) Rosetta in CASP4: progress in ab initio protein structure prediction. Proteins 45(S5)119–126
79. Moult J (2005) A decade of CASP: progress, bottlenecks and prognosis in protein structure prediction. Curr Opin Struct Biol 15:285–289
80. Skolnick J, Kolinski A, Kihara D, Betancourt M, Rotkiewicz P, Boniecki M (2001) Ab initio protein structure prediction via a combination of threading, lattice folding, clustering, and structure refinement. PROTEINS Suppl 5:149–156
81. Gntert P, Mumenthaler C, Wthrich K (1997) Torsion angle dynamics for NMR structure calculation with the new program DYANA. J Mol Biol 273:283–298
82. Krippahl L, Barahona P (2002) PSICO: solving protein structures with constraint programming and optimisation. Constraints 7:317–331
83. Krippahl L, Barahona P (2003) Propagating N-ary rigid-body constraints. In: Francesca Rossi (ed) CP'2003: principles and practice of constraint programming, October 2003. Lecture notes in computer science, vol 2833. Springer, pp 452–465
84. Katchalski-Katzir E, Shariv I, Eisenstein M, Friesem AA, Aflalo C, Vakser IA (1992) Molecular surface recognition: determination of geometric fit between proteins and their ligands by correlation techniques. Proc Natl Acad Sci USA 89(6):2195–2199
85. Krippahl L, Barahona P (2005) Applying constraint programming to rigid body protein docking. In: van Beek P (ed) CP'2005: principles and practice of constraint programming. Lecture notes in computer science, vol 3709. Springer, Berlin, pp 373–387
86. Krippahl L, Moura JJ, Palma PN (2003) Modeling protein complexes with bigger. Proteins 52(1):19–23
87. Dominguez C, Boelens R, Bonvin AMJJ (2003) HADDOCK: a protein–protein docking approach based on biochemical and/or biophysical information. J Am Chem Soc 125:1731–1737
88. de Vries SJ, van Dijk ADJ, Krzeminski M, van Dijk M, Thureau A, Hsu V, Wassenaar T, Bonvin AMJJ (2007) HADDOCK versus HADDOCK: New features and performance of HADDOCK2.0 on the CAPRI targets. Proteins 69:726–733
89. Kitano H (ed) (2001) Foundations of system biology. MIT Press, Camdridge
90. Bower JM, Bolouri H (eds) (2001) Computational modeling of genetic and biochemical networks. MIT Press, Camdridge
91. Kauffman SA (1993) The origins of order. Oxford University Press, New York
92. Thieffry D, Thomas R (1998) Qualitative analysis of gene networks. Pac Symp Biocomput 3:77–88
93. Reddy VN, Mavrovouniotis ML, Liebman ML (1993) Petri net representation in metabolic pathways. Proc Int Conf Intell Syst Mol Biol 1:328–336
94. Regev A, Silverman W, Shapiro E (2001) Representation and simulation of bio-chemical processes using the pcalculus process algebra. Pac Symp Biocomput 6:459–470
95. Cardelli L (2005) Abstract machines of systems biology. Trans Comput Syst Biol 3737: 145–168

96. Calzonne L, Fages F, Soliman S (2006) BIOCHAM. An environment for modeling biological systems and formalizing experimental knowledge. Bioinformatics 22(14):1805–1807
97. Cruz J, Barahona P (2003) Constraint satisfaction differential problems. In: Proceedings of CP03. Lecture notes in computer science, vol 2833, pp 259–273
98. Cruz J, Barahona P (2005) Constraint reasoning in deep biomedical models. Artif Intell Med 34:77–88
99. Bockmayr A, Courtois A (2002) Using hybrid concurrent constraint programming to model dynamic biological systems. In: ICLP02. Lecture notes in computer science, vol 2401, pp 85–99
100. Dooms G, Deville Y, Dupont P (2005) CP (Graph): introducing a graph computation domain in constraint programming. In: Proceedings of CP05. Lecture notes in computer science, vol 3709, pp 211–225
101. Gebser M, Schaub T, Thiele S, Usadel B, Veber P (2008) Detecting inconsistencies in large influence networks with answer set programming. In: International conference on logic programming, 2008
102. Dworschak S, Grell S, Nikiforova VJ, Schaub T, Selbig J (2008) Modeling biological networks by action languages via answer set programming. Constraints 13(1–2):21–65
103. Kanehisa M, Goto S (2000) KEGG: Kyoto encyclopedia of genes and genomes. Nucleic Acids Res 28:27–30

Sports Scheduling

Michael A. Trick

Abstract Sports scheduling has been an extremely active area of research over the past twenty years. One important reason for this is that the computational methods for creating playable sports schedules have improved enough to be useful to real sports leagues. A key aspect to these computational improvements has been the development of hybrid methods that combine two or more of integer programming, constraint programming, metaheuristics, or other core optimization approaches. While there is a broad range of models and applications in sports scheduling, there have been two main classes of problems studied: break minimization problems and travel minimization problems. I illustrate how hybrid methods can be used for these two problems, as well as provide some comments on other, lesser studied problems. I also give some directions for further research in this area.

1 Introduction and Scope

Sports scheduling has been an extremely active area of research over the last decade or so. There have been dozens of papers written describing successes both in theory and in practice. Many of these successes have been on integer or constraint programming approaches, or approaches that combine those methods.

There are a number of reasons for this interest in sports scheduling. First, sports is economically important. The thirty-two teams in the National Football League (US) are each worth more than US$1 billion; the league earns more than US$6 billion per year in television rights fees alone. European football teams can have similar valuations (such as Manchester Uniteds value of US$1.8 billion). Fans pay billions to attend sporting events; television networks pay billions to broadcast them; and sports web pages are some of the most popular on the Internet. While a beautiful

M.A. Trick (✉)
Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA 15213, USA
e-mail: trick@cmu.edu

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_15,
© Springer Science+Business Media, LLC 2011

schedule may not have the same effect as, say, a beautiful goal to win the World Cup, the schedule does have a significant effect on the income of sports teams and leagues as well as the satisfaction of the fans.

Second, sports scheduling research is challenged at roughly the size of practical sports leagues. Most sports leagues have between 8 and 30 teams; most algorithms for creating schedules begin to have difficulty in the 8 to 20 team range. This makes for great interaction between theory and practice: leagues want schedules at exactly the size that we can, with some improvements in our algorithms, provide them. And the relatively small size of challenging instances makes the problem appealing to work on: significant computer resources are not needed just to handle the data.

Third, integer and constraint programming models differ in how they model sports schedules. In an integer program, a typical variable would be `x[i,j,t]`, a binary variable for which 1 corresponds to team i playing at team j in period t. A constraint programming model for this would have variables `plays[i,t]`. The domain for this variable would be all the teams; i playing at j in time t would correspond to having this variable take on value j. With such different variables, integer and constraint programming formulations often take on quite different forms, making it an interesting research question to compare, contrast, and perhaps combine the two models.

And fourth, there is no denying that there is more general interest in a sports schedule than, say, a building construction schedule. For a researcher, there is a great reward when teams are playing due to the outcome of some optimization.

One drawback to sports scheduling is that there are about as many sports scheduling problems as there are sports leagues. Every league has different requirements and objectives. While this might lead to a plethora of algorithms and approaches with no overriding lessons, that has not been the case: there are some general insights that we have gained from the last decade of work.

The purpose of this paper is not to be a complete survey. For that, Kendall et al. [30] have put together an excellent annotated bibliography on scheduling in sports. Other recent surveys include those by Rasmussen and Trick [41], Drexl and Knust [14], and Briskorn [7].

Rather, the purpose of this paper is to highlight the role integer programming, constraint programming, metaheuristics, and combinations thereof has played in advancing the theory and practice of sports scheduling. Practically every approach in this area has appeared in some form in a sports scheduling problem. This survey brings out that.

To illustrate this breadth of technique, I will concentrate on two major sports scheduling problems: The Constrained Break Optimization Problem and the Traveling Tournament Problem. Both of these problems have been the subject of many papers in the last years, and both have proven to be very challenging problems. I will follow those up with a selection of other, generally less studied, problems, which illustrate further the role that IP, CP, and metaheuristics play in this area. In the concluding section, I will outline some challenges and opportunities for the field.

2 Constrained Minimum Break Scheduling

As a basis, we will consider a scheduling problem over n sports teams (n is often assumed to be even). Pairs of teams will meet in *games*. A *round robin tournament* is a tournament where all teams meet all other teams a fixed number of times. For a *single round robin tournament*, this number is one; for a *double round robin tournament*, this number is two; and so on.

Games in the tournament are scheduled in *slots* where each team plays at most one game in a slot. If n is even, then a single round robin tournament can be scheduled in $n - 1$ slots, with each team playing exactly one game per slot; an odd n requires at least n slots. A *compact* schedule is one that uses the minimum number of slots. If a team does not play in a slot, it is said to have a *bye*.

For the problems we look at, each team has a *venue*, and games are played at the venue of one of the two participating teams, called the *home team*. The other team is then the *away team* for this game. Normally, for a double round robin tournament, the schedule is required to be *balanced*: for every pair, one game between them is at one venue and the other is at the other. For a single round robin tournament, balanced means simply that the deviation between the number of home games for a team versus the number of away games for a team is not more than 1.

The sequence of home games, away games, and byes that a team plays during the tournament is known as its *home away pattern*. For many leagues, finding good home away patterns is key to finding good schedules. For some leagues, one key aspect is a *break*. A break occurs when a team plays at home in consecutive slots or plays away in consecutive slots. An ideal schedule would have no breaks: the team would play HAHAHAHA.... It is clear that in a round robin schedule, not every team can have no breaks, since there are only two such home away patterns, one beginning with H and another beginning A, and teams with the same pattern cannot play each other. de Werra [58] showed that $n - 2$ breaks were required for single round robin tournaments, and gave a constructive technique for creating such a schedule.

If the goal were simply to create minimum break schedules, de Werra's work essentially answered the question. His constructions are extremely fast and can be applied to problems of practically any size. Certainly, schedules for thousands of teams could be generated in a matter of seconds.

For most leagues, however, there are additional constraints, even when minimizing breaks is of paramount importance. Rasmussen and Trick [41] outline a number of these constraints, dividing them as follows:

1. *Place constraints.* Constraints ensuring that a team plays home or away in a certain slot.
2. *Top team and bottom team constraints.* Special considerations for teams known or suspected to be particularly strong or weak.
3. *Break constraints.* Limits on where breaks may occur.
4. *Game constraints.* Requirements that particular games occur (or do not occur) in particular slots.

5. *Complementarity constraints.* Requirements that particular teams do not both play at home in a slot.
6. *Pattern Constraints.* Limits on the home away patterns requiring or precluding certain patterns.
7. *Separation Constraints.* Limits on how closely two teams can play against each other.

In addition, there may be costs or profits associated with having a particular game in a particular slot, giving an objective function to be maximized.

Adding these sorts of constraints and objective leads to a constrained minimum break problem (CMBP).

Ideally, one would simply formulate CMBP as an integer or constraint program and solve it. This approach has been explored by a number of researchers, including Henz, Müller and Thiele [27], Trick [50], and Briskorn and Drexl [10]. While some types of problems are solvable with these approaches, current approaches generally require excessive time even for problems with just ten or so teams.

2.1 Multiple Step Approach

To attack the CMBP, then, there have been a number of papers that use a decomposition approach. This sports scheduling problem naturally decomposes into four steps and, although the order of the steps vary and some steps are combined, these four steps are used in almost all solution methods for solving variations of the constrained minimum break problem. The four steps are:

Step 1 Generate patterns.
Step 2 Find a pattern set for placeholders.
Step 3 Find a timetable for placeholders.
Step 4 Allocate teams to placeholders.

To illustrate these phases, and to show how various constraints can be embedded in them, we will go back to the work of Nemhauser and Trick [38] in scheduling a college basketball conference.

At that time, the Atlantic Coast Conference basketball league consisted of nine teams, generally in the south-eastern United States. A perennial powerhouse, the league had lucrative television contracts, a rabid fan base, and coaches with strong views on how the schedule affected the success of their teams. Scheduling up to that time had been done manually, a two-week process that resulted in good schedules. Could schedules be generated faster and at higher quality?

Nemhauser and Trick adopted the four steps given above. They were not the first to do so. Russell and Leung [45] had used a similar approach a few years earlier for a minor league baseball league as did Scheuder [48] for Dutch football.

For the ACC problem, we began by generating all feasible home/away/bye patterns. For this problem, there were many requirements on the schedule (*pattern constraints* in the above list): it needed to be mirrored, teams could not play three

consecutive weekends at home or away, and so on. It turned out that there were only 38 feasible patterns for the nine teams to play. These were found by enumerating the 2,038 possible patterns and discarding the infeasible ones. This completed step 1.

Step 2 involved finding sets of nine patterns that might form a feasible schedule. Since the schedule was compact, each time slot consisted of four teams at home, four away teams, and one team with a bye. While this is not sufficient to define a feasible pattern set, no complete characterization of feasibility is known. This was solved via integer programming, adding "no-good" constraints to generate all pattern sets. There were 17 pattern sets, which completed step 2.

For step 3, we generated all timetables again by integer programming. Here, the variables are binary variables `x[i,j,t]`, for team i plays at j during slot t. These timetables are created for each of the 17 pattern sets. Again, "no-goods" were added to ensure the generation of all timetables. There were 826 timetables found, completing step 3.

The final step is to replace the "placeholders" with actual teams. Up until now, the teams were simply numbered 1–9. In this phase, it was decided that, for instance, team 3 is DUKE, team 4 is UNC, and so on. This was done by complete enumeration of the 9! possibilities for each of the 826 timetables. It is in this phase that most of the idiosyncratic constraints are added. For instance, it was wanted that Duke and UNC not be played consecutively by any other team (both teams were strong at the time, so it was felt to be unfair to have to play both in succession). So any schedule that had the two consecutively was eliminated. With the list of constraints, there were 17 feasible schedules. We chose three to show to the ACC and the ACC then chose one to play.

The complete enumeration in the final step was, perhaps, a strange choice to make. Surely, we could have created an integer program for this, couldn't we? A natural way to model the final step was as a quadratic assignment problem, and that approach was taken by Schreuder [48] years earlier. But for this league's problem, even with a quadratic assignment objective, formulating a number of the constraints was difficult. So, we went with a complete enumeration.

Very shortly after publication of this paper, Martin Henz [25] showed that constraint programming could be used for each of the steps and, in particular, could decrease the computation time for Step 4 from 24 h to just a few seconds. This paper, originally written a month after the Nemhauser and Trick paper, finally appeared in *Operations Research* three years later. Even with this delay, this paper was the first constraint programming paper to appear in the flagship operations research journal. Since that time, there have been a number of constraint programming papers published in *Operations Research*.

The ACC problem is no longer useful as a benchmark. Between faster computers and better algorithms, it is possible to formulate and solve the entire problem as a constraint program, obviating the need for the multiple steps. The Sports Scheduling Group, a company formed by Nemhauser and Trick, use a single constraint program to schedule the ACC and other sports leagues.

There are still many cases where the multiple step approach is appropriate, and it is often used in practical league scheduling ([16] for instance) due to its simplicity and ease of implementation.

On the algorithmic side, there has been much interest in developing alternative techniques to solve either individual steps or multiple steps at once. For instance, Schaerf [47] considered a scheduling problem where constraint programming was used in step 4. Zhang [59] showed how to formulate the steps as satisfiability problems, and applied that approach to a different college basketball conference. Drexl and Knust [14] show how many of the constraints in these sports scheduling problems can be handled as resource constrained scheduling problems, bringing another type of solver into consideration.

Rasmussen and Trick [41] developed an approach that mixed up the steps. Rather than generating all pattern sets, followed by all timetables, and then all schedules, they first found one pattern set and then tried to generate timetables from that. If a timetable was found, then a schedule was developed for that timetable. If, however, no timetable was found, then a constraint was generated to feed back to the previous step. The constraint could be as simple as a "no-good" ("do not use this pattern set") to a more complicated constraint ("Do not use this set of 3 patterns"). In the worst case, this approach is simply the multiple step approach done in a different order. If the constraints are strong enough, however, they can preclude the generation of useless pattern sets or timetables. This is an example of a logic-base Benders approach [28], which we will see again in the next section. This approach proved to be much faster than the standard multiple step approach for a variety of constrained minimum break problems. This approach was used by Rasmussen to schedule the Danish football league [40].

2.2 Schedule then Break

It is also possible to do the steps in a completely different order. When Steps 3 and 4 are done before Steps 1 and 2, the approach is known as a "schedule then break" approach. First, it is known who plays whom when, and then the venues are chosen so as to minimize breaks (or some other objective).

In a "schedule then break" approach, the first phase is to find a schedule without home/away assignments. For the first phase, [27] looked at constraint programming approaches to this phase. Their most significant work was to develop an arc-consistent propagation method based on non-bipartite matching (corresponding to Regin's work on bipartite matching approaches to the `all-different` constraint). Since every time slot in a sports scheduling problem is a non-bipartite matching problem, the improved propagation is needed to effectively generate schedules. Trick [49] compared this approach to an integer programming model. In general, integer programming outperforms constraint programming when an objective is considered, but CP is better at identifying infeasibility.

The second phase of a "schedule then break" approach is to determine the home/away patterns relative to the assigned games. This problem was introduced by

Régin [43] who provided a constraint programming approach to the break problem. This was followed by Trick [49] who showed that an appropriate integer program could solve larger instances than the constraint program. This set off a flurry of activity that well illustrates the range of approaches that can be used in sports scheduling problems. Elf, Jünger, and Rinaldi [19] showed a relationship to the maximum cut problem and applied their excellent integer programming cutting plane system to that problem, solving large instances very quickly. Miyashiro and Matsui linked this problem to semidefinite programming [37] and also showed the polynomiality of the special case of determining whether at most one break per team was possible [36]. On the structural side, Post and Woeginger [39] and Brouwer, Post, and Woeginger [11] were able to give tight bounds on how many breaks might be needed for particular schedules.

A somewhat different approach has been taken in a series of papers by Briskorn and Drexl [8, 9] and Knust and Lücking [31]. In this work, the requirement to have at most one break per team is a hard constraint, and the objective is to minimize a cost function based on when teams play each other.

2.3 Future Directions

Constrained Break Minimization Tournaments have formed the basis for many practical league schedules and for a tremendous amount of research. It has proven to be a good test-bed for the wide variety of methods in integer programming, constraint programming, and combinations thereof. Despite this, there are, I believe, a number of interesting directions for researchers to consider:

1. Non Round Robin Scheduling. Most of the work to date has involved single or double round robin scheduling. Not every league has that structure, so it would be interesting to know what works on more general game numbers.
2. Better results on non-mirrored schedules. de Werra showed that for a mirrored double round-robin tournament $3n - 6$ breaks were required. What if the tournament does not have to be mirrored but there was a separation requirement between the games between any two opponents? If there was no separation requirement, then $n - 2$ breaks suffice, but the value is not known even for separation of 2 (no repeaters in the language of the Traveling Tournament Problem).
3. Better handling of costs. The papers by Briskorn and Drexl and Knust and Lücking have a nice, general cost structure, but the requirement of a minimum number of breaks is very strong, particularly in the context of place constraints. Can this be generalized to allow breaks at a penalty or to have a bound on the number of breaks?
4. Non compact scheduling. Essentially, all the work has been done on compact schedules. There are interesting non-compact scheduling problems (in the United States, both professional basketball and professional hockey have noncompact schedules). What is the right way to handle these sorts of problems?

3 Traveling Tournament Problem

The Traveling Tournament Problem (TTP) was inspired by work done by Trick with Major League Baseball (USA). Trick, together with his partner, baseball executive Doug Bureman, had worked with Major League Baseball since 1996 to create better schedules.

Major League Baseball (MLB) is a league of thirty baseball teams spread throughout the United States and Canada. With more than US$3 billion in annual revenues and an average team value of US$286 million, baseball is a large business in the sporting world. The schedule is also one of the largest of any sport. Each team plays 162 games over the course of the 182 day season. This gives a total of 2,430 games to be scheduled. Fortunately, the effective size of the schedule is somewhat smaller since teams will generally play two to four games against the same opponent on consecutive days in a "series" of games. Teams currently play 52 series in a season, so the league schedule is effective to schedule 780 series. This is somewhat simplified further since MLB has two sub-leagues: the American League with 14 teams and the National League with 16 teams. Except for six series per team, teams play within their leagues. So the National League problem is to schedule 16 teams over 46 time slots, and the American League problem is 14 teams over 46 time slots. Even with these simplifications, these are large scheduling problems.

A key aspect of MLB's schedule is the travel teams undertake. Because the schedule is so full and the United States is so large, teams do not return to their home city in between games. Instead, they stay in the city of the game during a series and then travel to the next city if there are consecutive away series. So if the Pittsburgh team plays 3 games in New York, then 4 games in Miami, and then 3 games in Atlanta, the Pittsburgh team would travel from Pittsburgh to New York to Miami to Atlanta before returning to Pittsburgh. One key objective to MLB is to minimize the total amount of travel. The alternative objective of minimizing deviation of travel among the teams, which might seem fairer, is not liked. Given the geography of the teams, teams in the center of the country (Chicago and that area) can, in minimum distance schedules, travel half the distance of a team on the West Coast, like Seattle. A minimum deviation schedule will simply have Chicago travel extra to bring its travel up to that of Seattle, which makes for an unattractive schedule.

Working against a wish for minimum travel is a need to have good *flow* to the schedule. With few exceptions, each team will play two series in a week. MLB strongly prefers that teams spend not more than two weeks at home or two weeks away due to fan interest, wear-and-tear on the plays, and other concerns. So teams cannot simply embark on an efficient tour of all the other cities: they must return home after at most four series. In fact, MLB has a strong preference that teams spend no more than three consecutive series at home or three consecutive series away. MLB and the teams also do not like to have single series at home or on the road due to the logistical and marketing problems such series cause. Ideally, all homestands would be two or three series, as would be all road trips.

MLB has many other requirements on the schedule. For instance, no team may be at home or on the road three consecutive weekends; teams must play half their

weekends at home; teams cannot begin or end the season with three away series; and so on. Further, individual teams may have requirements and requests. While teams generally own their own stadium or at least have first rights to any date, there are some concerts and other activities that may take precedence, making the stadium unavailable for a home game on particular dates. Teams also have preferences for playing either home or away on major holidays based on other activities in their cities.

This description gives only a hint of the objectives and constraints that go into creating MLB's schedule. For more than twenty years, MLB's schedule was created by a husband and wife team Henry and Holly Stephenson. Starting with the 2005 schedule, MLB's schedule has been created by a small company, the Sports Scheduling Group (SSG), consisting of Bureman, Trick, George Nemhauser, and Kelly Easton. SSG has provided schedules for five of the six seasons 2005–2010 (the remaining season's schedule was provided by the company Optimal Planning Solutions with techniques also based on operations research).

MLB's schedule, while challenging, does not make for a good research benchmark for a number of reasons:

1. MLB has no clear objective function. In its full form, the constraints lead to a clearly infeasible problem. Much of the skill in creating a playable schedule is in trading off violations of constraints, ensuring no team has an unplayable schedule.
2. There is just one instance per year, with little variation in problem size or complexity.
3. MLB would like to keep certain aspects confidential, including the requests and requirements of the teams. Teams may make certain requests for reasons that are not yet public. For instance, if the team knows a star player is to retire, they may request a particular matchup or timing of that player's final home game. Making the full problem public would compromise that confidentiality.

To offset these disadvantages, Easton, Nemhauser, and Trick created a problem class called the Traveling Tournament Problem (TTP). This problem abstracts out the key issues of MLB's problem without including all the detail of the "real" problem.

The TTP is defined as follows:

Input: n, the number of teams; D an n by n integer distance matrix; L, U integer parameters.

Output: A double round robin tournament on the n teams such that

- The length of every home stand and road trip is between L and U inclusive, and
- The total distance traveled by the teams is minimized.
- If i plays at j in slot k, then j does not play at i in slot $k+1$.

The parameters L and U define the tradeoff between distance and pattern considerations. For $L = 1$ and $U = n - 1$, a team may take a trip equivalent to a traveling salesman tour. For small U, teams must return home often, so the distance

traveled will increase. For much of the work in the literature $L = 1$ and $U = 3$, corresponding roughly to the requirements of MLB.

The last constraint is a "no-repeater" requirement: most of the work has included the requirement but some of the published literature does not.

Example 1. NL6 is an instance over $n = 6$ teams. Its distance matrix D is

```
0  745  665  929  605  521
745   0   80  337 1090  315
665  80    0  380 1020  257
929 337  380    0 1380  408
605 1090 1020 1380   0 1010
521  315  257  408 1010   0
```

If we label the teams ATL, NYM, PHI, MON, FLA, and PIT, respectively, (the names come from MLB cities from the year 2000), the optimal TTP solution for $L = 1$ and $U = 3$ is

```
Slot  ATL  NYM  PHI  MON  FLA  PIT

0      FLA @PIT @MON  PHI @ATL  NYM
1      NYM @ATL  FLA @PIT @PHI  MON
2      PIT @FLA  MON @PHI  NYM @ATL
3     @PHI  MON  ATL @NYM  PIT @FLA
4     @MON  FLA @PIT  ATL @NYM  PHI
5     @PIT @PHI  NYM  FLA @MON  ATL
6      PHI @MON @ATL  NYM @PIT  FLA
7      MON  PIT @FLA @ATL  PHI @NYM
8     @NYM  ATL  PIT @FLA  MON @PHI
9     @FLA  PHI @NYM  PIT  ATL @MON
```

with distance traveled of 23,916.

The TTP has proven to be a remarkably challenging problem. Despite significant effort, until 2008 (nine years after the problem was first announced), the instance above was the largest instance solved to provable optimality. Currently, the largest instance (with non-structured D) solved to provable optimality is the 10 team NL10 instance.

There are a number of properties of the TTP that make it a good challenge for computational work:

1. Instances can be generated of whatever size desired.
2. The data requirements for an instance are not extensive.
3. There are different variants that can be explored by varying L and U or ignoring the "no-repeater" requirement.
4. Different structures on D can make the problem easier or harder.
5. While very small (4 team and 6 team) instances are easy, even small problems such as 8 team instances provide a challenge.

The TTP has also been a very good test-bed for experimenting with integer programming, constraint programming, metaheuristics, and various combinations. Since 2001, there has been a repository of solution values at http://mat.tepper.cmu.edu/TOURN that tracks the best upper and lower bounds found to date for the TTP and its variants.

3.1 Exact Approaches

A natural approach to this problem would be to use formulations like that in the previous section: integer programs with binary variables `x[i,j,t]` being 1 if i plays at j in timeslot t or constraint programs with variables `plays[i,t]` taking on value j in that case. Unfortunately, such formulations have not been successful in attacking the TTP. In CPAIOR 2005, Trick [51] reported on experiments with the integer programming formulation. For the NL6 example, the "natural" integer programming formulation for NL6 gives an initial linear relaxation value of 2,186 (the optimal value is more than 10 times that value). Days of computation were needed to close the gap.

In that same paper, Trick reported on a better formulation for this problem that harkened back to branch-and-price approaches. Instead of a variable for every game in every slot, Trick proposed a variable for every road-trip. So `trips3[i,i1,i2,i3,t]` would correspond to a binary variable that is 1 if and only if team i visits team $i1$ in slot t, $i2$ in $t+1$, and $i3$ in $t+2$ (there would be similar variables for length 1 and length 2 road-trips). Such a formulation has far more variables but both the objective and constraints are much simpler than the "natural" formulation. The result is a model that solves much faster. The initial relaxation value for NL6 is 21624.7, and the model is solved in minutes with current optimization codes.

This approach can be seen as a simplification of the first approaches proposed by Easton, Nemhauser, and Trick [17, 18]. In that approach, there is a variable for each "tour" or complete schedule for a team. The large number of tours requires a branch-and-price approach. In this approach, there is a master problem that tries to choose the best set of tours, choosing one tour per team. This is typically an integer program. The subproblem then tries to find better tours to add to the master problem, using the dual values from the linear relaxation to the master problem. For the TTP, the subproblem can be well formulated as a constraint program. More details of this sort of CP-based branch-and-price can be found in the tutorial [18].

For many years, this approach was the only successful complete approach to the TTP. With this approach, the authors were able to solve NL8 (unfortunately without the "no-repeater" constraint, making their work incomparable to other work), albeit on a 20 machine parallel cluster with days of computation time.

Irnich [29] extended and enhanced this approach by exploiting the network structure of the subproblem. By formulating this as a shortest path problem over an expanded network, he is able to generate solutions to the subproblem much faster.

Further enhancements in this careful implementation (with credit given to Ulrich Schrempp) lead to a much faster approach than the one given by Easton, Nemhauser, and Trick and was the first to solve NL8 with the no-repeater constraint.

3.2 Lower Bounds

One key concept introduced in [17] was a lower bound called the *Independent Lower Bound* (ILB). It is possible to calculate, for each team i, the minimum amount of travel i would have to do independent of the other teams' schedules. One way to do this is to generate all roadtrips for i and find the minimum distance set that contains all other teams: this is a straightforward integer or constraint program. The ILB is then the sum of each team's minimum travel. For the problem sizes of interest, the ILB can be calculated in essentially no time.

Other than the bounds one gets from branch-and-bound, there has been little success in improving on this lower bound. Urrutia, Ribeiro, and Melo [55] noted that for some sizes, one can argue that not every team can travel the minimum distance so the lower bound can be slightly improved.

Benoise et al. [6] in CPAIOR 2001 created a single constraint program for the entire problem. By relaxing a single global constraint, the problem broke into pieces equivalent to the problems solved by the independent lower bound. Costs were modified by adopting a lagrangian optimization approach to violations of the relaxed constraint. The modified costs led to different costs for the individual team problems, leading to improved lower bounds.

Cheung [13] further improved the independent lower bound for the mirrored TTP using a combinatorial, or logic-based, Benders decomposition approach [28]. In a mirrored TTP, the schedule is made up of two halves: the first $n - 1$ slots and the last $n - 1$ slots. The schedules in the two halves are identical with the exception of the venue of each game, which is reversed in the second half.

In a Benders approach, the model has two types of variables: master variables (x) and subproblem variables (y). The master problem consists of the x variables along with any constraints that only depend on x. The subproblem consists of variables y along with all constraints that depend on y or on both x and y. The master problem is solved to get a candidate master solution x'. The subproblem is then solved with the x variables fixed to x' to get solution y'. The solution to the overall problem is (x', y'). In addition to y', the subproblem generates a constraint for the master problem that says, essentially, "To find a better solution, x must satisfy this constraint". In traditional Benders' approaches, this constraint is formed from the dual values of the subproblem. In logic based Benders' approach there is much more flexibility in the constraint. A simple constraint is the "no-good" constraint that says $x \neq x'$: in order to get a better solution, choose something other than x'.

In the TTP as formulated by Cheung, the variables in the master problem correspond road trips. For each team, a set of road-trips is generated (minimizing distance traveled), so the initial solution to the master problem is simply the set

of trips that form the independent lower bound. The subproblem then tries to schedule the chosen trips. If the subproblem is able to schedule them, then the resulting solution is the optimal solution to the TTP. Otherwise, a set of mutually infeasible trips is generated, and a constraint is added to the master problem prohibiting that set. This is repeated until the subproblem is feasible. Along the way, each solution to the master problem gives a lower bound on the optimal TTP solution. In this way, Cheung was able to provide excellent lower bounds to the mirrored TTP.

Mitchell, Trick, and Waterer [35] took this idea and applied it to the full TTP. Like Cheung, they noted that the subproblem becomes much more difficult to solve, but they were able to generate some new lower bounds for some of the instances.

Perhaps, the most exciting recent work in this area came in the CPAIOR 2009 conference, where Uthus, Riddle, and Guesgen [57] used DFS* (a variant on depth-first-search where bounds are kept on possible solutions) to attack this problem. In this approach, games are assigned team by team and round by round. The key is to quickly determine a lower bound on a solution based on the current partial assignments. Initially, this bound is just the independent lower bound, but this bound gets updated as assignments get made. The insight of this paper is that these bounds can be precomputed, greatly speeding up the process. This work can find optimal solutions (it is the fastest known way to prove NL8), but also generates good heuristic solutions and excellent lower bounds along the way. By way of example, for the lower bound of NL12, the work of [55] increased the lower bound by 9 above the independent lower bound; the work of [35] increased it by a further 54. Uthus, Riddle, and Guesen increased it by a further 696!

3.3 Feasible Solutions

Given the difficulty in finding optimal solutions to the TTP, it is natural to look for heuristic solutions. The most successful such approaches have been based on local search methods. Anangnostopoulos et al. [1], in the work that first appeared in CPAIOR 2003, gave a local search heuristic based on simulated annealing. The main problem with these sorts of approaches is that most local moves from a TTP solution lead to infeasibilities. If i previously played j in a slot and a local search move causes i to play k, then the schedule must be further modified to have i play j (and not play k) at some other time. To handle this, the authors develop ejection chains from their moves that try to restore feasibility relative to the base round-robin constraints.

This approach was further improved by Vergados and Van Hentenryck [24] in CPAIOR 2006. For many of the instances, the solutions found by this approach are still the best solutions known.

Lim, Rodrigues, and Zhang [32] had a similar approach based on simulated annealing with a similar set of moves. This approach put more emphasis on the assignment of teams to a schedule. Given a solution to the TTP, clearly the names

of the teams can be permuted to get an alternative feasible solution. By judicious renaming, the authors were often able to get better solutions, particularly for instances of the TTP with simpler cost structures.

A standard alternative to simulated annealing is tabu search. Di Gaspero and Schaerf [22] proposed a family of tabu search solvers and analyzed a number of combinations of the resulting neighborhood structures. The approach worked very well on a number of larger instances.

One surprising approach was given by Ribeiro and Urrutia [44] who came up with approaches to the mirrored TTP. In their three-step approach, their first phase created a schedule, the second phase assigned teams, and the third phase set the venues. For the final phase, local search with simple neighborhoods are used to repair any infeasibilities that come up. The interesting thing is that their mirrored solutions were often better than the best known non-mirrored solutions, particularly for larger instances.

This was further improved by Araujo et al. [5] who looked at parallelizing improvements to this approach using grid computing.

One approach that appears promising but that requires more work was introduced by Henz [26] which used large-neighborhood search. In regular local search, small changes are made. So in [1] a typical move would be to swap the home base of a game, or to switch the round in which a game is played. While the ejection-chain approach would then make the neighborhood somewhat larger, there are still relatively few neighboring solutions. In large-neighborhood search, a much larger portion of the solution is relaxed. For instance, all the games in 5 rounds might be released, leading to an optimization problem over those 5 rounds to find the best neighbor. While Henz notes that preliminary results are not encouraging, there has been significant work recently on large neighborhood search that might be applied, suggesting this direction might be worth revisiting.

One final approach for finding good feasible solutions is given by Rasmussen and Trick [42] where they define the *timetable constrained distance minimization problem (TCDMP)*. In the TCDMP, the schedule of games is given without the home/away assignments and the goal is to set the home/away assignments. In keeping with the theme that sports scheduling offers interesting combinations of integer and constraint programming approaches, this paper gave four solutions approaches to this problem: integer programming, constraint programming, a hybrid approach that generated patterns by CP and chose patterns by IP, and a branch-and-price approach with an integer programming master problem and a constraint programming subproblem. The hybrid approach was the best approach. Rasmussen and Trick then used this approach to find good solutions for the TTP by improving initial solutions with moves that reallocate the home/away patterns.

3.4 Further Work and the Future of the TTP

While I have outlined the major papers on the TTP in general, there has been much work on variants and special cases. One special case that attracted a lot of effort is

on the "constant" TTP, where the distance matrix D has $D_{ij} = 1$ for all $i \neq j$ [21, 56]. Other work has been done on the mirrored TTP [12] and the TTP where the venues are fixed [34].

The TTP has inspired work in most of the field represented by integer programming, constraint programming, metaheuristics, and combinations thereof. In the papers listed, we have seen Benders' approaches, simulated annealing, branch-and-price, tabu search, multi-phase approaches, special cases, and much more.

The TTP, despite significant effort by many researchers, is definitely not solved. The fact that NL10, a ten team instance, is still open means that there is still work to do. Here are a few directions that researchers might consider:

1. A more serious effort at an integer programming formulation. Most of the work in integer programming has been the "formulate and put it into CPLEX" version. A more serious effort would involve identification of facets and cuts combined with judicious choice of formulation. Even the obvious odd set constraints that come from the underlying bipartite matching have not been seriously tested.
2. A more serious effort at constraint programming. Since much of the initial testing with constraint programming in the early 2000s, there has been significant work on relevant global constraints such as the stretch constraint and various cardinality constraints. It would be worthwhile determining which of these constraints can help.
3. Improved lower bounds. Essentially all the lower bound work has been based on the independent lower bound. Are there any other lower bounds that can be used?
4. Different values of U. There has been little work except for the $L = 1, U = 3$ case. While it is not likely that $L = 1, U = 4$ will lead to different approaches or insights, $L = 1, U = n$ would be an interesting approach and one that might take research in a different direction.
5. Different D metrics. Are there easy D values (beyond the constant distance matrix)? Even for circular distances, where the points are arranged on a cycle with unit edges, with distances corresponding to the shortest path on the cycle, the instance of size 10 is not solved. Such an instance was designed so that the "Traveling Salesman Problem" aspects would be trivial. Are there any interesting D matrices for which the TTP is easy?

4 Further Models

While Constrained Minimum Break Scheduling and the Traveling Tournament problems are two important problem in sports scheduling, there are many others of interest that illustrate aspects of constraint programming, integer programming, and combinations thereof.

One early work, presented in 1997, in applying constraint programming to a sports problem was the work of McAloon, Tretkoff, and Wetzel [33]. In this problem, the goal is to find a round robin schedule that assigns each game to one of $n/2$

venues in each slot. No team should be assigned more than twice to any venue over the course of the tournament. The paper gives a constraint programming formulation and notes the typical behavior of these sorts of problems that the solvable instances end sharply at approximately 12 teams. Urban and Russell [54] introduced a variant of this problem, inspired by the needs of a football coach to schedule intra-squad workouts on various drill stations. While integer-programming based approaches were limited to 10 teams or so, a followup paper on constraint programming approaches [46] was able to solve 16 team problems. Lim, Rodrigues, and Zhang [32] used simulated annealing to find good feasible solutions quickly for about that size.

In addition to scheduling teams, there has been some interest in scheduling umpires, referees, or other sports officials. Some problems are similar to the Traveling Tournament Problem where the travel of the officials is of paramount importance. Some early work was done by Evans [20] who scheduled the umpires for Major League Baseball using a mixture of matching optimization, heuristic rules, and human intervention. This problem was formalized in the CPAIOR paper by Trick and Yildiz [52] where the Traveling Umpire Problem was defined, allowing, for instances of various sizes. This paper used another variant of Benders cuts to guide a large neighborhood search, further illustrating the ability of sports problems to integrate various optimization approaches. Other work by Trick and Yildiz [53] uses matching optimization to develop an optimized cross over for a genetic programming approach. This work has been applied to schedule the umpires for Major League Baseball for the past five years.

There has also recently been much work on referee assignment where travel is not an issue. The work by Duarte and Ribeiro [15], Duarte, Ribeiro and Urrutia [3] and Duarte, Ribeiro, Urrutia, and Haeusler [4] in particular illustrate the variety of approaches that can be applied to such problems. These problems tend to be somewhat easier than other sports scheduling problems, allowing the optimal or near-optimal solution of instances with up to 500 teams.

5 Conclusions

We have looked at some of the key problems in sports scheduling: break minimization, travel minimization, venue assignment, and referee assignment. This is not close to an exhaustive survey of all of sports scheduling. There are dozens more issues that might be addressed. But it is clear that there is some commonality among these various threads. First, for almost all problems, there is a very sharp divide between solvable and currently unsolvable instances. Second, that divide occurs for seemingly small instances, often in the range of 10 or 12 teams. Third, practically every type of problem can be attacked with a wide variety of techniques: integer programming, constraint programming, metaheuristic approaches, satisfiability, hybrid approaches, and so on. Fourth, the state of the art for our techniques make them applicable to real sports leagues, but not routinely. Many real sports leagues are scheduled using specially-developed software.

In the next few years, I believe there will be increasing interest in sports scheduling. As the excellent annotated bibliography by Kendall, Knust, Riveiro, and Urrutia [30] shows, there has been a steady increase in sports scheduling papers. Since very few problem types have been "solved" in the sense that a definitive result on the best algorithm is known, it seems likely that further improvements will be made in the upcoming years. And, while it is impossible to be certain where these developments will come from, I think there are a few likely trends.

As our techniques get better, it is likely that we will begin combining issues into our instances. For instance, the Traveling Tournament Problem includes travel and break issues, but does not include "carry-over" issues [2, 23] where teams worry about who their opponent played in the previous week. It would be straightforward to formulate a version of the Traveling Tournament Problem with carry-over but such a problem would seem to be very difficult. It would also be possible to combine the Traveling Tournament Problem with the Traveling Umpire Problem to find schedules good for both the teams and the officials.

I think it is also likely that the best techniques in the future will be ones that combine various "pure" approaches. Neither constraint programming nor integer programming have shown themselves to be particularly good at this sort of problem. Short of a miraculous new cutting plane or domain reduction approach, it seems the best approaches will combine multiple approaches.

This field is also likely to have increasing effect on the practice of sports scheduling. Many leagues, including some large and prosperous leagues, are still scheduled essentially by hand. But economic pressures and our field's ability to handle more and more of the needs of the leagues will lead to more real-world implementations.

Overall, sports scheduling has been an excellent test-bed for examining issues in optimization, heuristics, and real-world implementation. It certainly will remain so for years to come.

References

1. Anagnostopoulos A, Michel L, Van Hentenryck P, Vergados Y (2006) A simulated annealing approach to the traveling tournament problem. J Sched 9:177–193
2. Anderson I (1999) Balancing carry-over effects in tournaments. In: Combinatorial designs and their applications, Chapman and Hall, London
3. AR Duarte, Ribeiro CC, Urrutia S (2009) A hybrid ils heuristic to the referee assignment problem with an embedded mip strategy. In: Hybrid metaheuristics. Lecture notes in computer science, vol 4771
4. Duarte AR, Ribeiro CC, Haeusler SU, Haeusler EH (2007) Referee assignment in sports leagues. In: Practice and theory of automated timetabling VI. Lecture notes in computer science. Springer, Heidelberg, pp 158–173
5. Araujo A, Boeres C, Rebello V, Ribeiro C, Urrutia S (2007) Exploring grid implementations of parallel cooperative metaheuristics: a case study for the mirrored traveling tournament problem. In: Metaheuristics: Progress in Compless Systems Optimization, pp 297–322
6. Benoist T, Laburthe F, Rottembourg B (2001) Lagrange relaxation and constraint programming collaborative schemes for traveling tournament problems. In: Proceedings CPAIOR'01, Wye College (Imperial College), Ashford, Kent UK

7. Briskorn D (2008) Sports leagues scheduling: models, combinatorial properties, and optimization algorithms. Springer, Berlin
8. Briskorn D, Drexl A (2009) A branch-and-price algorithm for scheduling sports leagues. J Oper Res Soc 60:84–93
9. Briskorn D, Drexl A (2009) A branching scheme for finding cost-minimal round-robin tournaments. Eur J Oper Res 197:68–76
10. Briskorn D, Drexl A (2009) Ip models for round robin tournaments. Comput Oper Res 36(3):837–852
11. Brouwer AE, Post GF, Woeginger GJ (2008) Note: Tight bounds for break minimization in tournament scheduling. J Combin Theor A 115(6):1065–1068
12. Cheung K (2008) Solving mirrored traveling tournament problem benchmark instances with eight teams. Discrete Optim 5:138–143
13. Cheung K (2009) A benders approach for computing lower bounds for the mirrored traveling tournament problem. Discrete Optim 6:189–196
14. Drexl A, Knust S (2007) Sports league scheduling: Graph- and resource-based models. Omega 35:465–471
15. Duarte A, Ribeiro C (2008) Refeee assignment in sports leagues. In: 19th International Converence on Multiple Criteria Decision Making
16. Durán G, Guajardo M, Miranda J, Sauré D, Souyris S, Weintraub A, Wolf R (2007) Scheduling the chilean soccer league by integer programming. Interfaces 37(6):539–552
17. Easton K, Nemhauser G, Trick M (2001) The traveling tournament problem: Description and benchmarks. In: Walsh T (ed) Principles and practice of constraint programming - CP 2001. Lecture notes in computer science, vol 2239. Springer, Berlin, pp 580–585
18. Easton K, Nemhauser G, Trick M (2003) Solving the traveling tournament problem: a combined integer programming and constraint programming approach. In: Burke E, De Causmaecker P (eds) Practice and theory of automated timetabling IV. Lecture notes in computer science, vol 2740. Springer, Berlin, pp 100–109
19. Elf M, Jünger M, Rinaldi G (2003) Minimizing breaks by maximizing cuts. Oper Res Lett 31:343–349
20. Evans J (1988) A microcomputer-based decision support system for scheduling umpires in the american baseball league. Interfaces 18:42–51
21. Fujiwara N, Imahori S, Matsui T, Miyashiro R (2007) Constructive algorithms for the constant distance traveling tournament problem. In: Practice and theory of automated timetabling VI, pp 135–146
22. Gaspero LD, Schaerf A (2007) A composite-neighborhood tabu search approach to the traveling tournament problem. J Heuristics 13(2):189–207
23. Guedes A, Ribeiro C (2009) A hybrid heuristic for minimizing weighted carry-over effects in round robin tournaments. In: Proceedings of the 4th Multidisciplinary International Conference on Scheduling Theory and Applications
24. Hentenryck P, Vergados Y (2006) Traveling tournament scheduling: a systematic evaluation of simulated annealling. In: Beck J, Smith B (eds) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems. Lecture notes in computer science, vol 3990. Springer, Berlin, pp 228–243
25. Henz M (2001) Scheduling a major college basketball conference - revisited. Oper Res 49:163–168
26. Henz M (2004) Playing with constraint programming and large neighborhood search for traveling tournaments. In: Burke E, Trick M (eds) Proceedings of practice and theory of automated timetabling, 2004, pp 23–32
27. Henz M, Müller T, Thiel S (2004) Global constraints for round robin tournament scheduling. Eur J Oper Res 153:92–101
28. Hooker JN (2005) A hybrid method for planning and scheduling. Constraints 10:385–401
29. Irnich S (2010) A new branch-and-price algorithm for the traveling tournament problem. Eur J Oper Res, 204:218–228
30. Kendall G, Knust S, Ribeiro CC, Urrutia S (2010) Scheduling in sports: an annotated bibliography. Comput Oper Res 37(1):1 – 19

31. Knust S, Lücking D (2009) Minimizing costs in round robin tournaments with place constraints. Comput Oper Res 36(11):2937–2943
32. Lim A, Rodrigues B, Zhang X (2006) A simulated annealing and hill-climbing algorithm for the traveling tournament problem. Eur J Oper Res 174(3):1459–1478
33. McAloon K, Tretkoff C, Wetzel G (1997) Sports league scheduling. In: Proceeding of the 3rd ILOG optimization suite international users conference, Paris, 1997
34. Melo R, Urrutia S, Ribeiro C (2009) The traveling tournament problem with predefined venues. J Sched 12(6):607–622
35. Mitchell S, Trick M, Waterer H (2008) Benders approaches to sports Scheduling, presented at Mixed Integer Programming, New York, NY
36. Miyashiro R, Matsui T (2005) A polynomial-time algorithm to find an equitable home-away assignment. Oper Res Lett 33:235–241
37. Miyashiro R, Matsui T (2006) Semidefinite programming based approaches to the break minimization problem. Comput Oper Res 33(7):1975–1982
38. Nemhauser G, Trick M (1998) Scheduling a major college basketball conference. Oper Res 46(1):1–8
39. Post G, Woeginger G (2006) Sports tournaments, home-away assignments, and the break minimization problem. Discrete Optim 3(2):165–173
40. Rasmussen R (2008) Scheduling a triple round robin tournament for the best Danish soccer league. Eur J Oper Res, 185(2):795–810
41. Rasmussen R, Trick M (2007) A Benders approach for constrained minimum break problem. Eur J Oper Res 177(1):198–213
42. Rasmussen R, Trick M (2009) The timetable constrained distance minimization problem. Ann Oper Res 171(1):45–59
43. Régin JC (2001) Minimization of the number of breaks in sports scheduling problems using constraint programming. DIMACS Series in Discrete mathematics and theoretical computer science, vol 57. Springer, Heidelberg, pp 115–130
44. Ribeiro C, Urrutia S (2007) Heuristics for the mirrored traveling tournament problem. Eur J Oper Res 179(3):775–787
45. Russell R, Leung J (1994) Devising a cost effective schedule for a baseball league. Oper Res 42(4):614–625
46. Russell R, Urban T (2006) A constraint programming approach to the multiple-venue, sport-scheduling problem. Comput Oper Res 33:1895–1906
47. Schaerf A (1999) Scheduling sport tournaments using constraint logic programming. Constraints 4:43–65
48. Schreuder J (1992) Combinatorial aspects of construction of competition dutch professional football leagues. Discrete Appl Math 35:301–312
49. Trick M (2001) A schedule-then-break approach to sports timetabling. In: Burke E, Erben W (eds) Practice and theory of automated timetabling III. Lecture notes in computer science, vol 2079. Springer, Heidelberg, pp 242–252
50. Trick M (2003) Integer and constraint programming approaches for round robin tournament scheduling. In: Burke E, De Causmaecker P (eds) Practice and theory of automated timetabling IV. Lecture notes in computer science, vol 2740. Springer, Heidelberg, pp 63–77
51. Trick M (2005) Formulations and reformulations in integer programming. In: Proceedings CPAIOR'05, Prague, Czech Republic
52. Trick M, Yildiz H (2007) Bender's cuts guided large neighborhood search for the traveling umpire problem. In: Proceedings CPAIOR'07, Brussels, Belgium
53. Trick M, Yildiz H (2009) Locally optimized crossover for the traveling umpire problem. Tepper School Working Paper
54. Urban T, Russell R (2003) Scheduling sports competitions on multiple venues. Eur J Oper Res 148:302–311
55. Urrutia S, Ribeiro C, Melo R (2007) A new lower bound to the traveling tournament problem. In: Proceedings of the IEEE symposium on computational intelligence in scheduling
56. Urrutia S, Ribeiro CC (2006) Maximizing breaks and bounding solutions to the mirrored traveling tournament problem. Discrete Appl Math 154(13):1932–1938

57. Uthus DC, Riddle PJ, Guesgen HW (2009) Dfs* and the traveling tournament problem. In: CPAIOR '09: Proceedings of the 6th international conference on integration of AI and OR techniques in constraint programming for combinatorial optimization problems, pp 279–293
58. de Werra D (1981) Scheduling in sports. In: Hansen P (ed) Studies on graphs and discrete programming. North-Holland, Amsterdam, pp 381–395
59. Zhang H (2003) Generating college conference basketball schedules by a sat solver. In: Proceedings of the fifth international symposium on the theory and applications of satisfiability testing, pp 281–291

Stimuli Generation for Functional Hardware Verification with Constraint Programming

Allon Adir and Yehuda Naveh

Abstract We survey the application of constraint programming techniques for stimuli generation in functional hardware verification, which can be considered the largest and most important industrial application of constraint programming. We provide a thorough introduction to the application domain, aimed at people unfamiliar with this area. We show the sources of constraints and the unique aspects of the constraint satisfaction problems (CSPs) arising in this field. We then present CSP models of a wide variety of stimuli generation problems, as well as the state of the art techniques used to solve them. We also discuss the current challenges in this area, and the prospects of solving them by advancing constraint programming technology beyond the state of the art.

1 Introduction

Over the last 10 years, constraint programming has developed into a mature discipline, both in its academic research and in its application to practical problems. The various chapters of the current book provide an evident manifestation of this development. However, we think that the single-most important industrial application of constraint programming has received very little attention in the past. This application is stimuli generation for functional hardware verification. The purpose of this chapter is to bring this application into the spotlight of the constraint programming community and to share our enthusiasm for applying constraint programming to this domain. In a nutshell, this excitement is the result of a central theme that lies at the basis of the domain: The pressing (and continuously developing) business need for a solution in a multibillion dollar industry, coupled with the unique technical challenges imposed by this need. In the last two decades, this theme has resulted in a constant stream of innovation in applying and enhancing cutting-edge

A. Adir (✉) and Y. Naveh (✉)
IBM Research – Haifa, Haifa University Campus, Haifa 31905, Israel
e-mail: adir@il.ibm.com; nahev@il.ibm.com

M. Milano and P. Van Hentenryck (eds.), *Hybrid Optimization*, Springer Optimization and Its Applications 45, DOI 10.1007/978-1-4419-1644-0_16,
© Springer Science+Business Media, LLC 2011

CP techniques for this domain. More importantly, we have reason to believe that the business pressure will only increase in the coming decade, resulting in more sophistication and ingenuity of the solutions, and hopefully in a much closer involvement of the academic community.

Stimuli generation for functional hardware verification is the process of creating test cases (or 'stimuli') with the intent of revealing unknown bugs in hardware designs, before the design is cast in silicon. We will dive deeper into this sentence in subsequent sections. However, it is important to understand its direct relation to constraint programming at the outset. Given a hardware design, only a very limited number of tests are valid for the design.[1] All other tests do not constitute legal input for the design, and are hence useless. Moreover, almost all valid tests are not interesting in the sense that they do not exercise the hardware design in prone-to-bugs areas. The task of creating a valid and interesting test becomes increasingly difficult when the complexity of the design grows. For modern designs, even of medium complexity, this becomes infeasible when using ordinary procedural techniques. This is where constraint programming comes in. It is possible to state all the constraints that a test must abide by in order to be legal (this set of constraints actually constitutes the *architecture* of the design), as well as the constraints imposed on the test by the current verification goal. A solution to the resulting constraint problem constitutes a valid and interesting test.

All high-end hardware manufacturers use the above concept to produce meaningful stimuli for testing their hardware. Some manufacturers such as IBM(R) [1] rely on proprietary constraint solvers developed in-house to solve this problem. Others such as Intel (R) [2], adapt external off-the-shelf solvers to the stimuli-generation problem. Some manufacturers of less complex designs rely on electronic design automation (EDA) tool vendors (e.g., Cadence (R) and Synopsys (R)) for their stimuli generation needs. Those EDA tools, in turn, are based on internally developed constraint solvers [3, 4].

This mode of operation means that a big portion of the activity in constraint programming, both in the development of constraint-solving tools and their use, takes place at a relatively small number of corporations: either high-end hardware manufacturers or EDA tool vendors. This differs greatly from the more traditional applications of CP such as scheduling, routing, or planning, where the typical users are medium-sized companies with varying needs (e.g., vehicle routing, machine scheduling, or assignment of technical workers). The number of companies using traditional CP applications is much larger than the number of high-end hardware manufacturers and EDA vendors, and each comes with its own scenarios and requirements. This results in higher visibility of their needs, even outside the company, and tend to stimulate discussions on the abstractions of the individual problems and the generic ways to solve them. This is the input that has driven the direction for the application of constraint programming in the last decade.

[1] Take, for example, the design of a processor. Then a test is simply a program, i.e., a sequence of assembly instructions, to be run on the processor. Only a very limited number of instruction sequences (including their parameters) form a valid program.

On the other hand, huge corporations such as IBM, Intel, Cadence, and Synopsys, are able to allocate their own R&D resources in order to create a solution most suitable for their strong business needs. This resulted in increased activity centered around constraint programming for hardware verification in all those corporations. However, the pressing needs and development pressure, possibly coupled with the tendency to remain reserved about new technology, did not support externalization of the technical achievements coming from those R&D organizations, and prevented a long-lasting and broad academic dialog between the companies and academia. Consequently, the importance of stimuli generation based on constraint programming, and its unique potential for innovations, have gone largely unnoticed by the academic community. We hope this chapter will serve to rectify this anomaly and stimulate a broad and fruitful dialog between the industrial stimuli-generation and academic CP communities.

IBM pioneered the research and development in this domain and published the first technical papers, as well as periodic reviews of the topic. The first identification of the potential use of constraint-solving techniques for stimuli generation was published as early as 1992 [5]. Shortly afterwards, the general use of AI techniques in this domain, with an emphasis on a model-based test generation (MBTG) approach, was reported at IAAI in 1994 [6]. In 1995, CP techniques were incorporated into the MBTG framework [7]. The following years focused on intensive adaptation of general CSP methods to stimuli generation, culminating in a 2002 article in the IBM Systems Journal, which reviewed the problem and outlined those adaptations [8]. A more AI-perspective of the subject, including the advancements in AI techniques since 1994, and the central business aspects of the problem, was published at IAAI in 2006 [9], with a longer version in AI Magazine [1].

In the current chapter, we will cover and expand on the main ideas presented in the previous publications. However, in general, we will focus on the problem domain, showing its breadth and depth in a way that emphasizes the fit for constraint-programming techniques. Further details of the technicalities of each of the problems described here can be found in the cited technical references within each section, which are typically oriented for an electrical engineering audience. This chapter also presents some of the more advanced CP techniques we developed, which were only mentioned in passing in the previous publications.

CSP has also been used for software testing, see e.g., [10, 11]. However, this use is inherently different from the use of CSP for hardware verification. In software testing, constraints serve mainly to define parameters to the test so it would reach a particular block of code, some program path, or some synchronization conditions between threads. The constraints are derived directly from the conditional statements on the data at each node in the program path, and in most cases are linear or can be easily linearized. The main challenge is then to perform symbolic calculation on those constraints, possibly with the aid of a linear solver. In contrast, the bulk of constraints in hardware verification serve to enforce validity and quality of the test (defined in more detail in Sect. 3). These constraints are defined over many hardware parameters (not only data), including memory addresses, control registers, and instruction semantics. This results in a much richer and more complex, highly

nonlinear, CSP. At any rate, CSP is only one of a number of methods used for checking software, and in no way does it occupy the central role it has in hardware verification.

In the next section, we introduce the domain of functional hardware verification. In Sect. 3, we show how CSP techniques are used for stimuli generation, and also some of the adaptations of these techniques that are needed for this specific domain. Section 4 discusses general aspects of modeling stimuli generation problems as CSP's, and Sect. 5 then describes specific areas of application within the domain, and the general CSP model for each area. In Sect. 6, we describe some advanced topics and challenges of current research interest. In Sect. 7, we conclude the chapter and look into the future of this field.

2 Functional Hardware Verification

The following is a high-level presentation of the domain of hardware verification and the related terminology referred to in the subsequent discussion.

2.1 Verification Process

Hardware verification [12] is the process of demonstrating that the hardware is implemented correctly according to its specification. Verification is performed as part of the hardware design cycle, starting from the point where the requirements of potential or actual customers are analyzed and a specification for the hardware architecture is compiled, up to the verification of the full physical systems that include the fabricated chips. During the design and verification cycle, the hardware specification is used as a basis for the hardware *design*. There may be several designs, each a refinement of the preceding one. For example, the process can start with a high-level formal definition of the implementation that matches the specification, then move to a detailed implementation of this formal design in terms of logical flow of signals between data elements (the *register transfer level* or *RTL*), and finally to a full physical circuit design ready for manufacturing (also known as *tape-out*) in the fab.

The design is created hierarchically. The basic element is at the designer level and constitutes a block designed by one or very few designers. Blocks are used to construct units with higher-level functions (e.g., a floating-point unit). A chip is a collection of units packaged together. The full system hardware can include multiple chips and devices for handling their connectivity, memory management, and I/O.

The verification follows the path of the hardware design process – checking the preservation of the semantics between the designs in their various abstraction levels and their correspondence to the specification. All elements of the design hierarchy are verified, from the blocks up to the entire system. Manufacture testing is a related process in which every manufactured element is tested to ensure successful fabrication and quality.

In general, the rule is that the later a bug is detected, the higher the cost for its repair [12]. Moving up the design hierarchy, the faults become more complex to detect and fix but still incur a relatively moderate cost. Bugs found after chip fabrication may require an additional tape-out, which can cost millions of dollars and cause a possible delay in product delivery. A bug found after the hardware is delivered to customers can reach hundreds of millions of dollars in complex repair processes and damage to brand image. Consequently, the objective of verification is to try and find the bugs as early as possible. Pre-silicon verification techniques aim at finding as many bugs as possible *before* the chip is fabricated. The target of the verification is the design of the hardware as implemented in a formal hardware description language (HDL), such as VHDL or Verilog. The main method for accomplishing this is simulation-based verification.

Simulation refers to the process of mimicking the behavior of the hardware – as opposed to running the actual hardware. This can be done at very different levels and by various technologies. Design simulation refers to the simulation of the hardware, based on the design of the hardware as implemented in the high-level HDL. Hardware *emulation* is technology for performing this design simulation using dedicated hardware. This speeds the simulation significantly when compared to performing the design simulation in software (the more common and less expensive approach).

Unfortunately, pre-silicon, simulation-based verification often does not succeed in uncovering all the design bugs. Design simulation in software, and even in hardware emulation, is still relatively slow and cannot provide the number of cycles needed to achieve sufficient coverage of the design. Another reason for these *escape* bugs is that not all the verified aspects of the hardware can be accurately modeled and checked through simulation. Thus, post-silicon verification is still required. This starts with the early *bring-up* of the manufactured chip with relatively simple testing and continues with full system testing of the hardware using benchmarks and complex customer applications such as operating systems.

Hardware verification is thus a broad domain that targets a variety of tasks and includes vastly differing technologies. The targets of the verification are also varied – functional verification checks that the hardware performs its functions as specified, performance verification checks that the hardware performs these functions within the planned time limits, and power verification checks that the functions can always be executed within the required power constraints. In this chapter, we concentrate mainly on functional verification.

2.2 Formal Verification

In the current chapter, we are interested in *simulation-based* functional verification of hardware [12], and more specifically in stimuli generation, as presented above. There is, however, a complementary functional verification paradigm that is

being used in parallel with simulation-based verification. This goes under the general name of formal verification, or more specifically theorem proving and bounded model checking (BMC) [13, 14]. In formal verification, the design is modeled as a set of logical expressions, corresponding to the gate-level functionality of the hardware design. Then, generally speaking, theorem proving formally proves that the *design under verification* (DUV) is functionally equivalent (i.e., same inputs result in same outputs) to a reference design, and BMC proves that an assertion representing a bug (typically modeled by a conjunction of the logical expression representing the assertion, with the design logic) can or cannot be reached within a specified number of cycles from some initial state.

The great advantage of formal methods over simulation-based methods is that the former are complete, and can prove that a DUV is functionally correct.[2] However, the big drawback is that today's formal methods can deal only with small designs, typically smaller than a single unit of a processor. Therefore, current industry practice combines the two methods, where formal methods serve to check the smaller design structures, and simulation-based methods check the integrated larger structures. All in all, however, for complex designs, more than 90% of verification resources are spent on simulation-based methods.

Formal verification methods use two major algorithmic schemes at their cores. The first is theorem proving over binary decision diagram (BDD) representations of the design, and the second is Boolean satisfiability (SAT) solving. For example, in BMC, the design logic in conjunction with the bug assertion is modeled as a SAT expression, and a proof that the expression is unsatisfiable constitutes a proof that the bug cannot be reached within the specified number of cycles. In recent years, satisfiability modulo theories (SMT) have been utilized to extend the capabilities of BMC, for example, by allowing verification experts to deal with data and not only control logic [15, 16]. SMT raises the abstraction level at which the design logic can be represented, as it allows, for example, arithmetic and other constraints over integer numbers, and is not limited to the bit-level representation. However, it is still far from the abstraction and expressibility levels of CSP.

The algorithmic basis behind formal methods has been studied and published extensively by the SAT community, whereas in the case of simulation-based methods we are only now starting to see a strong interest from the CSP research community. In the rest of this chapter, we will deal only with simulation-based methods, where constraint programming serves as a core technological basis.

2.3 Simulation-based Verification and Stimuli Generation

The essence of *simulation-based* (or *dynamic*) verification is to test how the design conducts itself when confronted with challenging stimuli. Stimuli generation, in turn, deals with the problem of creating the appropriate stimuli in order to test the

[2] This statement is somewhat weakened in the case of BMC, as the proof is only for a specific set of assertions, and only within a limited number of cycles.

DUV as thoroughly as possible. The nature and abstract level of the stimuli depends on the object being tested and the level of verification. Automatic test-pattern generator (ATPG) tools [17] test the manufacturing of circuits by applying sequences of lowest-level bit-vectors at the circuit's input interfaces. A full processor can be tested by generating test programs in the assembly language of the processor. At the highest abstraction level, system level stimuli can include commands that produce transactions involving multiple system devices.

Good stimuli first needs to be *valid* with respect to the requirements imposed by the DUV. It should also be of high quality in the sense that it tests the behavior of the DUV in some desired circumstances to improve the coverage of the tested behaviors, reach challenging corner cases, and hopefully trigger a bug. Also, the stimuli had better be able to expose the bug if it indeed occurs during the test execution (and not render it unobservable, for example, by masking its effects).

The most basic, technology-free method for generating stimuli is to write them by hand. Surprisingly enough, this is still being done, especially if there are only a few simple directed stimuli that are needed, or when there is no available technology to generate the type of stimuli required. Needless to say, this method is limited in capacity, expensive, error-prone, and often cannot achieve the precise stimuli that is required. A technology for automatic stimuli generation is therefore needed.

2.3.1 Targeted Versus Massive Pseudo-Random Stimuli Generation

There are several stimuli generation technologies for particular verification domains that are designed to create a single targeted test to meet some specific requirement (or declare that the requirement cannot be met at all or within the given time limit). ATPG tools [17], for example, have the ability to generate a stimuli that would manifest a specific fault in the manufactured circuits–for example, observing if some signal line is "stuck-at" some value independent of the inputs. Adir et al. [18] presents a stimuli generator whose test brings the processor to some specified micro-architectural state or declares that state unreachable. Another tool [19] can produce inputs for floating point instructions that cause the arithmetic result to have some specified desired properties.

In all these cases, any single stimuli that meets the requirement would suffice for the task. However, for most verification tasks, even if such analytic tools were available, the verification engineer doesn't know in advance precisely where to look for the bugs. The common approach is then massive generation of different stimuli.

In general, it is impossible to comprehensively test all the possible behaviors of the DUV, but one still wants to test the design with as many tests as can be handled with the available verification platform. The most naive implementation of this massive-generation approach is to create a multitude of random tests. However, random stimuli generation would still need to be constrained to generate only tests which are both valid and that meet any user requests. It must also control the distribution of the generated test instances in order to bias for the more "interesting" tests. This bias should be controlled by the user, or potentially by some internal

expert-knowledge of the stimuli generation tool itself. A good distribution should still not completely rule out any specific valid test since one never knows where the bug may eventually hide.

It is also possible to combine massive pseudo-random and targeted stimuli generation approaches when the verification process follows a plan listing specific tasks that need to be covered by the stimuli. The verification can start with massive testing using relatively random stimuli that are generated with little effort. Coverage measurement tools can keep track of the events that are actually covered by the stimuli. In time, the coverage pace will inevitably decline as only the hard-to-hit events remain. At this point, verification can continue by applying the more directed generation technology to hit these rare (but important) specific events.

2.3.2 Test Specification Language

Stimuli generators generally provide the user with some control over the properties of the tests to be produced. In the trivial (though quite common) case, this amounts to a set of parameters the user can specify to configure the generator. However, more sophisticated stimuli generators support richer languages that describe the desired *test template*.

In order to better exploit the power of automatic massive generation, it makes sense to re-use the same (manually created) test specification to build many different test instances. This implies the need for a test specification language that is abstract enough to describe a multitude of tests and a pseudo-random stimuli generator that can generate a different test each time it is invoked.

For example, the stimuli generator in [20] produces stimuli for a processor in the form of an assembly test program. The test-template language [21] allows the user to describe the desired test program with various levels of abstraction. The user can ask for general high-level processor events such as interrupts or cache misses, or for certain instruction types like a general loading instruction. The user can also be more explicit and ask for a specific load instruction like a load-word or specify the instruction completely along with the register and memory resources to be used. A good test template describes only the properties that are crucial for the test and leaves room for the stimuli generator to "fill-in" the missing details. Such a test template corresponds to many possible concrete tests that are generated according to the biased-random distribution provided by the automatic stimuli generator. The test template can also include directives to control this distribution. This can be done by directly specifying the distribution in the template or by configuring some internal expert-knowledge of the tool.

The template language can support a hierarchy of priorities for the user directives. The crucial properties of the test would be specified to be mandatory and the directives controlling the events that are merely desired in general can be defined in various levels of "softness". As shown in Sect. 5.1, user directives in the test template specification can translate into hard and soft constraints, and can be added to the CSP used in the stimuli generation process.

Test Program Template	**Test Program**
Variable: addr = 0x100	*Resource Initial Values:*
Variable: reg	`R6=8, R3=-25,..., R17=-16`
Bias: Resource-Dependency(GPR) = 30	`100=7, 110=25,..., 1F0=16`
Bias: Alignment(4) = 50	
	Instructions:
Instruction: Load R5 ← ?	`500: Load R5← FF0`
Bias: Alignment(16) = 90	:
Repeat (addr < 0x200)	`504: Store R4 → 100`
Instruction: Store reg → addr	`508: Sub R5 ← R6-R4`
Select	`50C: Store R4 → 110`
Instruction: Add ? ← reg + ?	`510: Add R6 ← R4+R3`
Bias: SumZero	:
Instruction: Sub ? ← ? - ?	`57C: Store R4 → 1F0`
addr = addr + 0x10	`580: Add R9 ← R4+R17`

Fig. 1 Test template and corresponding test

Figure 1 shows an example of a test template and a corresponding generated test. The test template describes a scenario beginning with a load from some unspecified address into the R5 register. A directive attached to the load template specifies a requirement for a strong bias (90) toward addresses aligned to 16 bytes. This bias was met by the stimuli generator, as can be seen by the actual load instruction that starts the test and is shown on the right side of the figure. The template continues by describing a "table-walk" scenario that stores the contents of randomly selected registers into memory addresses ranging from address 0x100 to 0x200, in increments of 16. Each `Store` instruction is followed by either an `Add` or a `Sub` instruction. The first source register used for each `Add` instruction is the same as the source register of the previous `Store`. Additionally, the template requests a number of bias constraints controlling such things as interdependency between instructions and alignment of addresses.

2.3.3 Static Versus Dynamic Stimuli Generation

Static stimuli generation refers to a generation process that is not aware of the state of the DUV. The entire test can be generated offline, that is, independent of the simulation environment and can thus employ general tools that are not available during simulation. It can also potentially employ more complex procedures to generate the test, such as generating the stimuli out of its simulation order.

A *dynamic* stimuli generator, on the other hand, generates the test in simulation order and keeps track of the state of the DUV as the generation progresses. This can be done either by performing the generation *online* (i.e., on the simulation environment itself as the generated test is being simulated) or *offline* by using a *reference model*. A reference model is an application that simulates the hardware according to its functional specification, that is, without any bugs. It can be implemented in software, based directly on the specification and independent of the DUV.

A dynamic test generator can exploit the knowledge of the machine state to more easily maintain the test validity and quality; for example, suppose the generator wants to generate a load instruction in which the address is computed by the addition of a base register value and a displacement field of the instruction. Suppose now that for the load instruction to be valid, it must avoid some reserved memory region. To improve the verification value of the load instruction, the generator's built-in expert-knowledge can recommend using a word-aligned address. In order to meet these constraints, the dynamic generator can refer to the current value of the selected base register and adjust the displacement to avoid the reserved area and make the address word aligned. A static stimuli generator would not be able to do this because it would not know the current values of the registers. It could precede the load with an instruction that loads an appropriate value into the base register – but such *reloading* can interfere with the required test scenario. Section 6.1 shows how the dynamic knowledge of resource values can be handled by a CSP engine.

2.3.4 Checking

Checking is the process of detecting that the DUV is not behaving as expected, implying that a bug has been triggered by the stimuli. For example, the environment can monitor the behavior of the DUV as it is being tested by examining internal states or transactions taking place at its external or internal interfaces. The monitored events can then be analyzed for bugs immediately as they occur, or after the simulation ends, based on more complete traces of the simulation. Section 6.4 shows how CSP modeling can assist such off-line checking of a processor's shared memory architecture. This method of checking can be handled by the simulation environment independently of the stimuli generator and thus has the advantage of being applicable to a greater variety of stimuli sources. However, the stimuli generator can also help the checking process, for example, by including self-checks inside the stimuli itself or by avoiding the masking of bugs (i.e. where one part of the stimuli hides the erroneous effects of a bug that occurs in another part of the stimuli).

The stimuli generator can also assist the checking by adding to the test information about the behavior that is expected of the DUV when the test is executed. The stimuli generator can find out this expected behavior by running the stimuli on a reference-model of the DUV. Thus, any inconsistency between the actual behavior of the DUV and the behavior predicted by the reference model indicates a bug in the DUV (or, less likely in the reference model itself).

2.3.5 Test Benches

A test bench is a comprehensive library of code used to generate, monitor, and check predefined stimuli to the design [12]. Once implemented, the test bench is connected to the simulator of the DUV to perform the actual testing. There are

well-established methodologies but no strict rules regarding test bench design. It is commonly made from a collection of independent components performing the various functions in a distributed manner. Nevertheless, some uniformity between different test benches exists as the result of test bench development and deployment environments provided by major EDA vendors. The common paradigm is to code all the test bench components with a language that includes a comprehensive collection of verification-oriented constructs.

SystemVerilog [22] and e [23] are popular test bench languages that have become IEEE standards. Both languages include constructs to support the wide range of functions required, including stimuli generation, checking based on a reference model, monitoring, and coverage measurement and reporting. The test bench developer uses a rich object-oriented typing language (with some aspect-oriented features in the case of the *e* language [24]) to define the structures of the objects that make up the stimuli. The object types and the level of abstraction with which they are defined naturally derives from the verification target and level. Some examples are raw data for block level arithmetic computations, instruction objects on the processor level, or communication packets on the higher system level. The strong relation of test benches and in particular object types to constraint programming is discussed at length in the next section.

3 Constraint Satisfaction for Stimuli Generation

3.1 Introduction and Motivation

As mentioned, the basic problem of stimuli generation for functional hardware verification lies in creating tests that abide by all architectural validity rules, by the current verification scenario, and by a general requirement for the test to be interesting (i.e., reach prone-to-bugs areas, and be substantially different from other tests).

All those requirements can be stated as constraints on the final test-case. Architectural validity constraints are specified in the design specification document. Examples for such constraints are 'load-word instructions must be word-aligned', 'if hypervisor bit is off, access to memory is restricted', 'data written by processor A and read by processor B abides with specified cache-coherency rules', and of course, 'after instruction of type $A + B \rightarrow C$, the values of registers A, B, C are such that $A + B = C$. There are typically several hundred such rules defined in the architecture of a modern processor.

A second source of constraints is the verification scenario targeted by the verification engineer, who is responsible for ensuring that the tests cover all suspected areas of the DUV. The scenario can typically be stated as a set of constraints, such as 'the load instruction must hit a page-boundary' (e.g., see the test-template example in Figure 1). Finally, requirements for the test to be interesting can be formulated as a large set of expert-knowledge constraints, such as 'add instructions must have a sum

of zero', or 'subsequent load instructions should hit the same cache-line'. Obviously, a complete set of expert-knowledge constraints would typically be self-contradicting and contradict the current-scenario constraints. Therefore, expert-knowledge constraints are modeled as soft constraints [25]. The stimuli generator can use these soft constraints to inject interesting random "background noise" to the basic scenario described by the verification engineer. The user can also be given some control over the type and intensity of this background noise.

While the three types of constraints mentioned above are an inherent feature of stimuli generation and exist in all its forms, the exact form of the constraints, as well as the number of variables in the problem, the number of constraints, and the number of soft constraints, varies greatly from problem to problem. We discuss these details below while describing each of the specific stimuli-generation problems.

With this view in mind, it is clear that constraint programming is an ideal option for stimuli generation. On the one hand, using procedural code, it can be extremely difficult, or even impossible in practice, to create a valid and interesting test that would be complex enough to reach deep areas in the design (e.g., a test consisting of 1,000 assembly instructions for a processor). On the other hand, stating the constraints is a relatively straightforward task. The ideal solution is an algorithm that solves the (soft) constraint problem and outputs a valid and interesting test to satisfy the current verification scenario. In fact, the strong dichotomy between the extreme difficulty in creating tests by procedural code, and the elegance of writing the constraint model for this case, makes stimuli generation a prime example of the so-called 'holy grail of computer science' [26]: the user states the problem, the computer solves it.

A second critical reason in support of constraint programming for stimuli generation is the dynamic character of the problem and the business pressure for fast results. It is well recognized that time scales of development between different generations of the same processor decrease drastically. In the current business environment, time to market becomes exceedingly smaller, while design complexity grows. This means that stimuli for new generations of the designs must be created more quickly. Maintaining and changing procedural code between different generations may be a nonsimple, time-consuming, and prone-to-bugs task. In contrast, changing the form of constraints when the design changes can be very simple as one needs only to remove, add, or change some of the constraints of the original model. In Sect. 4, we present the modeling aspects and platforms that support this type of rapid changes in the CSP model.

3.2 Unique Requirements

The above arguments make it clear why constraint programming in general is suitable as a solution scheme for stimuli generation. However, taking a closer look at the problem, we can find inherent peculiarities that are not seen in other application domains for CSP. The unique aspects of CSPs for stimuli generation are discussed

in length in [1, 8]. Here we summarize the most important of those aspects, which add to the technical challenge of this domain and create a broad arena for innovation and creativity.

- *Requirement to randomly sample the solution space* A complete set of architectural, expert-knowledge, and verification task constraints defines a single soft CSP. (In what follows, we refer to this soft CSP as plain 'CSP' even though it contains soft constraints.) However, we expect to obtain many different tests out of this single CSP. Moreover, we want the tests to be distributed as uniformly as possible among all possible tests that conform to the CSP. Essentially, we want to reach a significantly different solution each time we run the solver on the same CSP [27].
- *Constraint hierarchy* Expert knowledge is entered as a set of soft constraints; these constraints may be applied in a multitiered hierarchy (Borning Hierarchy [28]) according to their perceived importance in any specific verification scenario. While constraint hierarchies also appear in other applications, stimuli generation stands out in terms of both the number of soft constraints in the model and the depth of the hierarchy. See the chapter by van Hoeve in this book for a thorough discussion of soft constraints [25].
- *Huge domains* Many of the CSP variables have exponentially large domains. The simplest examples are address and data variables that can have domains of the order of 2^{64} or larger. Handling such variables, and in particular pruning the search space induced by these variables by using constraint propagation, cannot be done using regular methods, that frequently rely on the relative smallness of the domain.
- *Conditional problems* Many of the verification problems are conditional; depending on the value assigned to some variables, extensive parts of the CSP may become irrelevant. (In early literature these problems were also known as 'dynamic CSP' [29]). Conditional problems also occur in other applications (e.g., vehicle configuration). However, we encounter a different flavor of these problems. In one typical scenario, a full problem may consist of several weakly coupled CSPs, where the number of those CSPs is itself a CSP variable. Sections 6.2 and 6.6 describe such scenarios.
- *Generic modeling of domain-specific propagators* Some of the relevant constraint propagators are extremely complex and require months to implement. However, the hardware specification may change on the same time scale, rendering the implementation obsolete. Therefore, we need a generic model of the propagator, where, by changing some parameters of the model, the propagation algorithm becomes suitable for the new problem. One clear example of such a case would be described in Sect. 5.4.
- *Computationally hard propagators* These are abundant in the verification of floating point units. See Sect. 6.7.

The various hardware manufacturers and EDA tool vendors have coped with those problems in different ways, and with various degrees of success. Intel has been adapting the ILOG solver to handle those problems [2]. IBM, Cadence, and

Synopsys have all developed their in-house solver, which was built with those problems in mind. To the best of our knowledge, out of the three, IBM has been the most open about the methods it implemented in its solver. Some of the methods used to overcome the above challenges are presented by Naveh et al. [1] in the section titled "The Generation Core Toolbox". We provide a deeper look at some of the solutions in Sect. 6 below. However, for many of these challenges, we are still far from an ultimate and satisfying solution.

3.3 Alternatives to Constraint Programming and Hybrid Approaches

While constraint programming is a clear candidate for solving the stimuli generation problem, there are other approaches, both declarative and procedural, that are favored for particular cases. Those include integer linear programming (ILP), satisfiability programming (SAT), satisfiability modulo theories (SMT), manual writing of tests, and creation of tests by specific procedural algorithms.

3.3.1 Declarative Alternatives

The first three cases (ILP, SAT, and SMT) have similar advantages to CP, but have limited expressibility compared to CP. Therefore, when the constraints in the problem are not linear or Boolean disjunctions of linear expressions, it is preferable to use CP; otherwise, the model needs to be translated into the appropriate ILP, SAT, or SMT model, and this can create a severe maintenance problem when the design changes. On the other hand, when the problem can be naturally expressed in one of those frameworks, it is certainly advantageous to use the appropriate ILP, SAT, or SMT solver, since their algorithms are tuned for the respective problem in a manner superior to that of a general purpose CSP solver. From our experience, whole verification problems cannot be naturally expressed as ILP, SAT, or SMT problems. However, there are subproblems that can typically be expressed in this manner; in such cases, some form of cooperation between the CSP and the other solver is needed. An example where a SAT solver is used for the stimuli generation for a floating point unit in the context of processor verification is discussed below in Sect. 6.7. Another example, where an ILP solver was used in conjunction with a CSP solver in the context of verification of a system on chip is described by Nahir et al. [30].

3.3.2 Hybrid Approaches

Given the unique advantages of each of the approaches, and given the complexity and possible heterogeneity of the problem, hybridization between the various approaches is called for. However, designing, implementing, and maintaining hybrid

solvers is a difficult task, and to the best of our knowledge only recently have such efforts resulted in working industrial applications in the stimuli generation domain. Still, designing various hybridization schemes is at the front of present research activity, as we describe below.

One relatively simple way of hybridization is the master/slave scheme. Such an approach has proved successful in at least two major cases: floating point unit validation [19], and architectural validation [31]. In both these cases, complex arithmetic scenarios are needed in the general context of processor verification. For example, consider the case where as part of a long test, an assembly instruction implementing $a \times b = c$ needs to be generated with some constraints on a, b, and c [32]. The processor verification problem carries with it the usual bag of complex constraints which are best written as a CSP, and solved by ordinary CP approaches. However, now the additional arithmetic constraints need to be satisfied simultaneously with all processor constraints. Such constraints are hard to propagate and can be a major inhibitor if modeled as regular constraints for the CP solver.

We overcome this problem by incorporating a new object, called a 'restrictor' into the CP solver. A restrictor has the same interface as a constraint, that is, a propagation function. However, it generally returns only a small subset of the mathematically supported domains, and it can return different subsets each time it is called on the same input domains. A restrictor can be added to the CSP at any place a constraint can be added, and the solver treats the restrictor in exactly the same way it treats a constraint, except that once it calls a restrictor and the restrictor does not guarantee to return the full mathematically propagated domains (this is the ordinary case), then the solver can no longer deduce that a CSP is unsatisfiable. In addition, in this case, the solver may call the same restrictor more than once even if no domains were reduced by other constraints or by instantiation steps.

In the case of [19] and [31], restrictors are used to model the hard arithmetic constraints, while ordinary constraint objects model all other processor constraints. The restrictors are implemented in this case by calling a stochastic local search engine which solves the arithmetic subproblem a predefined number of times, each time providing a different solution. Then, the values for each variable are aggregated from all solutions and returned as the output domains of the restrictor. This way, each output domain value is guaranteed to have at least one support for this constraint. Once the restrictor has finished its task, control goes back to the CP engine which continues with an ordinary maintain-arc-consistency (MAC) scheme. Note that the stochastic solver called by the restrictor finds solutions to the arithmetic problem out of the input domains specified to the restrictor. Hence, when the MAC solver returns to this restrictor after reducing domains of any one variable, the restrictor would now find solutions under the reduced domains. Such solutions are known to exist because only supported values were returned in the previous call to the restrictor. In fact, the restrictor may internally save all previous solutions in order to come up faster with a solution in subsequent calls, after domains have been reduced by the MAC solver.

Another important cooperation between a CP solver, a stochastic solver, and general logic algorithms was presented in [33]. In that paper, we presented a framework

for applying logical manipulation of the problem, reminiscent of systematic solving, in order to make the problem more suitable for stochastic local search. A simple example of such manipulation is just calling the systematic CP solver's reach-arc-consistency function before sending the problem into the stochastic solver. This may reduce the state-space for the stochastic solver. However, other techniques presented in the above paper go beyond reaching arc-consistency, and are shown to dramatically affect the state-space (size and topography, e.g., elimination of local minima) in favor of subsequent stochastic search.

Additional hybridization schemes which we are at an early stage of evaluation are: translation of some constraints into SAT or ILP and running a SAT or ILP solver on those constraints (CP/SAT–ILP master/slave scheme). The incorporation of the results into the larger CSP is then done as described above for incorporating restrictor results; Running a SAT solver over the reified truth-values of constraints (SAT/CP master/slave scheme) – this somewhat resembles the mode of operation of an SMT solver, except that the underlying theory is not restricted and has the full expression power of the CP solver; Adapting clause-learning techniques from SAT into the systematic CP solver; And applying exhaustive search methods at critical stages of the stochastic local search solver.

3.3.3 Procedural Alternatives

Procedural algorithms are typically used when the verification problem is very simple, and a basic test can be created by a simple program or a few lines of script. This can occur in a variety of cases. The most obvious case is when the design has a straightforward architecture. Another common case is when the design is in the first stages of development and is still saturated with bugs. Then, even simple tests are sufficient for finding most low-hanging bugs. Note, however, that this stage in the verification is performed by the designer of the module rather than by actual verification teams.

3.3.4 Manual Alternatives

At the other extreme, there is often the need to target a very specific and rare scenario in the design. Here, a generic constraint solver may not be able to solve the highly specific problem, even if the scenario can be expressed as a set of constraints on the design. In this case, there is no alternative but to resort to a highly specific test written manually by a dedicated verification expert.

3.4 Concrete Example

As discussed in Sect. 2.3.5, industrial test benches use object types to describe the various hardware scenarios to be tested. Constraint programming dictates that these

```
struct Packet {

  %sourceAddress : uint (bits : 8);
  %targetAddress  : uint (bits : 8);
  %payloadSize    : uint (bits : 16);
  %payload        : list of uint(bits : 8);
  parityType      : bool;
  %parity         : uint (bits : 8);

  keep sourceAddress != targetAddress;
  keep payloadSize == payload.size();
  keep payloadSize in [4..1024];
  keep (parityType == TRUE) == (parity == calculateParity(payload));
  keep soft parityType == TRUE;
  keep soft payloadSize ==
    select { 20 : 4; 20 : [5..10]; 20 : 1024; 40 : others; };
};
```

Fig. 2 Constrained packet definition in *e*

object types be defined as classes with constraints on the class properties. The constraints can be used to define the valid domains of the properties and to relate the properties of a class among themselves or with relation to properties of other objects. Constraints can be defined as hard (mandatory) or with a hierarchy of softness.

Figure 2 (based on Nahir and Ziv (Private communication)) shows an example of a definition of a packet object type in the *e* language. A packet object has six properties. sourceAddress and targetAddress are 8-bit integers specifying the source and destination of the packet. payloadSize is an integer specifying the size of the payload property, which is in turn a list of 8-bit integers embodying the data being sent in the packet. parityType is a Boolean property indicating whether or not the parity property needs to be computed for the packet.

The constraints on the packet properties follow, each prefixed by the keyword *keep*, or *keep soft* in the case of a soft constraint. Note, the constraint can use a previously defined function like calculateParity. The last constraint defines a desired distribution for the payload sizes between 4 and 1,024 (20% for 4, 20% for sizes between 5 and 10, etc). parityType is not a physical property that would appear in the actual packet, rather it is a virtual property to assist in the definition of the constraints used in the packet generation process. This is indicated by the % symbol, which precedes only the physical properties of the object.

Constraints can also create a relationship between different objects. For example, one can define a stimuli of 1,000 packets, where every packet uses a targetAddress different from the one used in the preceding packet. The constraints can be defined on object types as in the figure but also on specific object instances. One can also import a library of predefined object types and then impose additional constraints upon them as required by the specific application. This enables, for example, a scheme for prioritizing soft constraints where the later a soft constraint appears in the class definition, the higher its priority. Another interesting aspect of CSP modeling for a

test bench with dynamic generation is the ability to configure the constraints dynamically as the stimuli is simulated on the DUV – including the modification, addition, or removal of constraints – all according to the current state of the simulator.

A test bench environment that supports these languages needs to have some CSP capabilities for handling these types of constraints. Various EDA vendors, like Synopsys, Cadence, and Mentor Graphics provide test-bench environments based on SystemVerilog. In addition, Cadence's Specman is also based on *e*. As each EDA vendor provides its own CSP engine, there are a variety of such engines to support the constraint-related aspects of the standards. The actual technique used to solve the CSP is often not entirely hidden from the user. For example, for some time, Specman solved the CSP property-by-property, in the order in which the properties were declared in the object types. This affected the way that the constraints and the order of properties had to be defined if one wanted to cause or avoid biasing the distribution of property combinations or if one simply wanted to improve the success rates of the CSP engine. A recently publicized version of Specman [34] now groups the properties into related sets, where each set is solved together and the solution order for the different sets is automatically determined following dedicated heuristics.

4 Modeling for Verification with CP

4.1 Introduction

With a *model-based* approach to verification tool design, the knowledge of the architecture is kept separate from the control, which is the tool's main engine. Typically the model is kept in some database ontological tool, while the engine is a program written in code. This approach has proved extremely apt for many types of verification tools for various reasons. Typically, many stages of the verification process are performed while the architecture is still evolving. Maintaining the changing knowledge of the architecture is much easier when it is placed in a separate formal declarative knowledge base; rather than hidden, often implicitly, in various parts of the code. The separation of knowledge from control also facilitates using the same tool for different designs or for various generations of the same design. In addition to a description of the relevant aspects of the design, the modeled knowledge can also include heuristic knowledge on how best to verify those design aspects. This is what we term *expert-knowledge*.

The ever-increasing complexity of today's designs raises the demands from the modeling technologies. The modeling language must be (1) expressive enough to describe the complex intricacies of the architecture and its expert knowledge, (2) formal enough to facilitate maintenance, and (3) be easily convertible to the representation that is required for the operational needs of the verification tool. As more and more verification tools are using CP as part of their engine, this internal representation is essentially a CSP.

In the next section, we describe various CSP models pertaining to different verification needs. Once each of the models is defined, it is input into a generic solver with abilities described in Sect. 3.2, and the solution of the CSP is then used to create the stimuli.

4.2 Distributed Model

The CSP model of the hardware design is highly distributed by nature. In a full hardware system, different entities (component types, hardware transactions between the components, configuration) carry different packages of constraints. However, the distribution of the model exists even for single-unit designs and is related to the various business areas responsible for creating different parts of the model.

At least three such areas exist, and are depicted in Fig. 3. First, hardware engineers contribute constraints specifying the architecture of the hardware (in the case of system verification, this actually means several teams of engineers, one per component type). Second, domain experts contribute expert knowledge in the form of soft constraints (those two functions are represented collectively by the 'knowledge engineer' in Fig. 3). Lastly, verification engineers contribute constraints that ensure the current verification task is accomplished.

While all three teams share the ultimate goal of releasing hardware with no bugs, each handles quite a different set of constraints and may use somewhat different terminology. The architecture team deals with hard constraints with one-to-one mapping to the formal specification of the hardware. The expert-knowledge team adds soft constraints that are less exact by nature and retain a level of fluidity in definition. The verification team adds hard constraints on the one hand and tunes the soft constraint parameters on the other hand. The three teams also work according to very different time scales. Expert knowledge is rather robust, major parts of it are valid across designs, and may remain intact for years. Architectural constraints are strictly valid for a single design, but may migrate between different generations of the design. For any new design, constraints must be added, deleted, or rewritten. Verification task constraints are short-lived and may change within weeks.

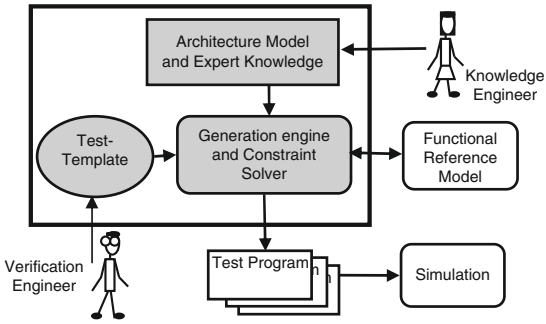


Fig. 3 Distributed sources of constraints

All those distributed parts of the model ultimately form one single CSP, and therefore relate to the same CSP variables. It is up to the tool developer to provide a modeling platform that, on the one hand, would be easily usable by the different teams and in the different time scales, but on the other hand, would share the same variables between all parts of the distributed model. The front end of the platform may appear differently to each user, in each case providing a different modeling language. However, a common back end should recognize the various types of constraints and synchronize between them.

We have found that the architecture and expert-knowledge teams can share the same front end as they are both aware of the same aspects of the design. For example, a constraint specifying that a particular data transfer needs to cross a page boundary can be both an architectural constraint (typically, in this case, the architecture may forbid a page cross), or an expert knowledge constraint, as a page cross stimulates complex hardware mechanisms that may reveal previously unreached behavior. On the other hand, a higher-level language is found more suitable for the verification teams. In the case of system level verification, for example, the task may involve transactions, components, and special instructions. The basic entities of the front end provided to the verification teams are formed from the higher-level entities modeled by the architecture and expert-knowledge team. So a verification engineer can ask for several transactions of a given type (as modeled by the architecture team) to pass through various components, using a given value of data (where the data parameter is again modeled by the architecture team). In addition, the verification engineer can request the test to be more strongly biased by any of the different forms of biasing created by the expert-knowledge team. For example, the engineer may request a specific type of data contention to occur in the test by choosing the relevant data contention entity which was modeled by the expert-knowledge team in terms of constraints forcing re-use of the same memory range in different transactions.

4.3 A CSP-Oriented Modeling Language

IBM has developed a general ontology-based, constraint-oriented, modeling platform called ClassMate. Its main (though not exclusive) deployment is for modeling verification domains for various CSP-based stimuli generators. The modeling language therefore includes first-class constructs particularly appropriate for the subsequent treatment of the modeled information as a CSP, such as soft and hard constraints and value domains. However, no verification-specific constructs are defined at this level.

The ClassMate language is basically an object-oriented typing language. As the first step in modeling a new problem domain, the user defines a meta-language that includes the most basic types of this domain. For example, in the verification domain, the meta-language may include such constructs as ‘component’, ‘register’, ‘instruction’, ‘contention’, and so on.

A *Type* definition is in essence a definition of the collection of the type's *instances* – termed the *extent* of the type. These instances can be explicitly modeled and then enumerated in the type definition, but the language includes basic built-in types and instances such as *integer*, *boolean*, *string*, and others with the implied domain of instances. The instances of the *integer* type, for example, are all the integers. Another method for more finely defining the extent of a type is by means of constraints. For example, the following is the definition of the type SmallEvenFibonacci with an extent including all even Fibonacci numbers between 1 and 100. The constraints appear inside square brackets, and % is the modulo operator:

type SmallEvenFibonacci : *integer*{[1, 100]}
[$*this*%2 == 0; *IsFibonacciNumber*($*this*)]

The above definition of the SmallEvenFibonacci type is an example for defining a type as a *refinement* of another base type, where the refinement here is performed by restricting the extent of the base type. The *integer* type is first refined to the set of integers between 1 and 100, and then further refined to only the even Fibonacci numbers among them.

The language also supports structured types where each field of the structure is, in turn, of some predefined type. The full *extent* of a structured type is the set of tuples resulting from the cross-product of the extents of the field types. This extent can again be further refined by specifying constraints on the allowed tuples. A structured type can also refine a given base type by *inheriting* from the base type and then adding more fields to the structure, or limiting the extent by adding further constraints or domain limitations to the structure fields.

Figure 4 shows the types involved in the modeling of a memory operand used by a CSP-based test program generator such as the one described in Sect. 5.1. The RegisterOperand type is a record with the following fields: index (there are 32 registers indexed 0–31), data (the registers are the size of 8 hex digits, i.e., 32 bits), and length (in bytes; in this case, 32 bits are 4 bytes). The data field is a refinement of the built-in type called bitstream. The bitstream type is similar to the integer type except that integer domains are defined by enumerating integers and integer ranges, whereas bitstream domains are defined with binary or hexadecimal masks that include "dont-care" positions. 0xXXXXXXXX is a bitstream domain including all the numbers that are representable with 8 hexadecimal digits. An example instance of the RegisterOperand type is the tuple $< index : 5, data : 0x000A2300, length : 4 >$.

The MemoryOperand type is defined very similarly, except that the address is more conveniently represented as a 32-bit bitstream and the operand length is left unrestricted. The type HalfWordMemoryOperand is a refinement of MemoryOperand. The domain of the data field is restricted to 16-bit values and the length field is restricted to the value 2. HalfWordMemoryOperand also includes two new fields: baseRegister of the RegisterOperand type and a displacement that is a 16-bit bitstream. An example instance of the HalfWordMemoryOperand could be the tuple $< address : 0x000A2340, data : 0xABCD, length : 2, baseRegister :< index : 5, data : 0x000A2300, length : 4 >, displacement : 0x0040 >$.

```
type RegisterOperand: record
{
 index: integer {[0,31]};
 data:   bitstream { 0xXXXXXXXX };
 length:integer { 4 };
}
type MemoryOperand: record
{
 address: bitstream { 0xXXXXXXXX };
 data:     bitstream;
 length:   integer;
}
type HalfWordMemoryOperand: record MemoryOperand
{
 data:   bitstream { 0xXXXX };
 length:integer { 2 };

 baseRegister: RegisterOperand;
 displacement: bitstream { 0xXXXX };

 [
  $address = $baseRegister.data + $displacement;
  Overflow($baseRegister.data, $address);
 ]
}
```

Fig. 4 A classMate type for a memory operand of an instruction

The constraint $address = \$baseRegister.data + \$displacement$ restricts the extent of the HalfWordMemoryOperand type to only those tuples in which the address field equals the sum of the baseRegister's data field and the displacement (the $ notation is used to access variables defined in the ClassMate record from the constraints.) The example tuple given above is thus a valid instance of the HalfWordMemoryOperand type, that is, it satisfies all constraints.

The object of the *Overflow* constraint that follows is to restrict the extent of the HalfWordMemoryOperand type to only those tuples that correspond to an "overflow" event when the memory operand is actually used in a test program. The precise constraint propagator of the Overflow restriction is presumably implemented by the user in some code library that can be used by the particular test program generator when solving the corresponding CSP. The Overflow constraint is an example of an "expert-knowledge" type of constraint. It does not represent an architectural restriction of memory operands but merely describes the types of memory operands desired for verification purposes. Such expert-knowledge constraints are commonly not mandatory. The ClassMate language enables the modeling person to indicate that constraints are hard or soft and also to prioritize the soft constraints.

It is important to note that ClassMate is only a modeling platform particularly adapted to model CSPs and not a CSP engine that can *solve* the modeled problems.

It provides a platform to model the information and also run-time libraries that provide reflection of the modeled information for any tool that chooses to solve the CSPs using CSP engines. The modeling platform (which also includes a graphical user interface) provides the strength of modeling very different systems (due to the separation between language and meta-language), the agility in quick adaptation to next-generation models (because of the stability of the model language, the ability to just change parameters or add or remove constraints), and the direct formation of a CSP model at its back end (because of the first-class status of constraints in the language).

5 Examples of CP models

5.1 Processor Modeling

Lichtenstein et al. [6] first described a model-based test program generator for processor verification. Adir et al. [20] described how an advanced version of the tool models the processor instructions and related expert-knowledge in a form that is directly translatable to a CSP.

Figure 5 shows a model of a *Load-Word* instruction that loads four bytes from memory. The operands are arranged in a tree structure with the attribute names (in bold) and value domains. The *Load* instruction is modeled with two operands: the source memory and the target register. The source memory operand, in turn, has two sub-operands: the base register and the displacement (immediate field) that are used to compute the loaded address. The arcs in the figure denote relations between attributes and correspond to constraints. These relations are imposed by the architectural definition of the instruction. For example, the source memory address is

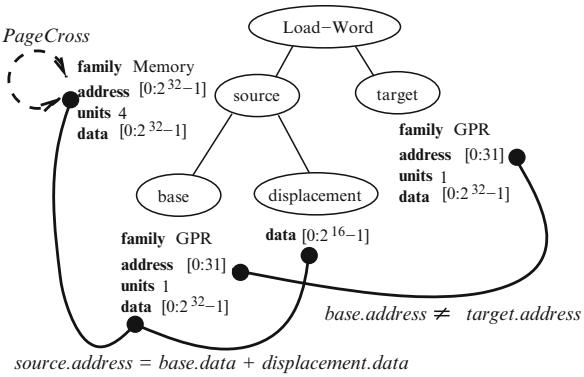


Fig. 5 Model of a load-word instruction

the sum of the value of the base register and the displacement (*source.address* = *base.data* + *displacement.data* in Fig. 5). Relations such as this one, which can be expressed as equations with arithmetic, Boolean, and bitwise operators, are stated directly in the model. The modeling of relations with more complex semantics can refer to an external implementation for the propagation in C++, which the modeling engineer needs to provide.

Instruction-specific expert knowledge can also be modeled as part of the instruction model. As an example, consider the instruction "`add Rt ← Ra+Rb`". The probability of randomly generating an instance of the `add` instruction in which *data*(Ra) + *data*(Rb) = 0 is negligible. Verification engineers may decide to give this combination a higher probability of appearing in tests because it may stimulate complex underflow mechanisms in the processor's arithmetic unit. In this case, they can model a corresponding "expert-knowledge" relation between the attributes *data*(Ra) and *data*(Rb). Figure 5 includes an expert-knowledge constraint represented by the dashed arc, which constrains the `address` attribute so that the word access crosses a page boundary. Such expert-knowledge relations are typically translated into soft constraints that can be adjusted, added, or removed from the actual CSP using user directives according to the specific requirements of the verification engineer.

5.2 *Verifying ADL-based Systems with Random CSPs*

As seen above, many verification tools have a model-based design, enabling knowledge of the hardware architecture being verified to be separate from the main control of the tool. Several vendors provide a hardware architecture modeling framework to automatically create a family of tools for early design exploration and other tasks in the design cycle, including verification tools; and in some cases, even major parts of the design itself. The modeling is performed using what is termed an architecture description language or ADL (although this term is also used in the context of software architecture modeling). CoWare [35] and Target [36], for example, provide platforms where the model can be used to create parts of the design and various tools including simulators, a compiler, assembler, and stimuli generators. Figure 6 shows an example from [37] showing part of the modeling of an *add* instruction with the LISA language [38] supported by CoWare's *Processor Designer* [35] with

```
OPERATION add_d
{
  DECLARE  GROUP Dest, Src1, Src2 =  register ;
  CODING     Dest, Src2, Src1 0b01000 0b1000
  SYNTAX     "ADD" ".D" Src1 "," Src2 "," Dest
  BEHAVIOR Dest = Src1 + Src2;
}
```

Fig. 6 Modeling syntax and semantics in LISA

```
operation MYLOAD
{ out AR Target, in AR Base, in imm8 Displacement }
{ out VAddr, in MemDataIn32 } {
  assign VAddr = Base + Displacement;
  assign Target = MemDataIn32;
}
```

Fig. 7 TIE description of a load instruction

information on its binary encoding, string mnemonic structure, and behavioral semantics. Other parts of the modeling for the instruction provide more details about the intended implementation, which enables cycle-accurate simulation of the instruction.

A related technology is that of *configurable* processors such as the one supported by Tensilica [39] and Arc [40], where the designer can construct the processor according to his or her particular requirements by configuring many aspects of a basic given processor. Tensilica performs this configuration using the TIE language [41].

In such model-based design environments, the main onus of verification falls more on the provider of the environment and less on the designer. The environment must be capable of producing valid designs and tools (when used properly). Rimon et al. [42] describes how such verification can be executed by testing the framework on randomly generated processor architecture models. Just as in any random stimuli generation, these random processor models perform better when biased toward interesting designs and so as to cover many possible design types. A CSP-based processor model–as shown in Fig. 5–is automatically created for the randomly generated configuration and the test generator is then used to verify the design resulting from the random model.

Thus, the processor model, together with its many CSPs, needs to be configurable based on the configuration in the TIE language. For example, the TIE description in Figure 7 of a load instruction results in a CSP similar to the one shown in Fig. 5. Here VAddr and MemDataIn32 are names of TIE *interfaces*. The TIE designer can assign a value to VAddr, then read the data from that memory address on the MemDataIn32 interface.

5.3 Micro-architectural Events

The term *architecture* in this chapter refers to a high-level specification of the design, for example, as required by a software engineer writing an application to be executed on the design. The *micro-architecture* is a much lower-level description of the design, mainly required by the designer as directives on how the design is to implement the architecture. Modern microprocessors have several micro-architectural mechanisms, not necessarily visible to the programmer, which improve

performance but increase the complexity of the design, thereby increasing the risk of bugs. Examples of such mechanisms include multiple execution units supporting concurrent execution, out-of-order execution, pipelining,[3] and caching.

A major problem for test program generators is how to trigger specific desired micro-architectural events by means of an architecture level assembly program. Adir et al. [18] describes a micro-architecture CSP-based approach that can deal with this difficulty. The model proposed includes several customizable building blocks that describe micro-architectural components (such as out-of-order queues or pipeline stages) and mechanisms (such as instruction flushing from the pipeline). These are defined using fixed properties such as the size of the queue or the cache-line replacement policy. The model also includes some dynamic properties of the micro-architectural components that refer to their state as instructions flow through them. The possible test instructions themselves are also modeled in terms of the timing of their passage through the possible execution stages (such as pipeline stage entry and exit time, time of flushing if relevant, etc.).

For example, the model may define that the pipeline property of any load instruction is one of two possible load-store pipelines, depending on which of them has the first stage free when the load is issued for execution. The first stage of this pipeline requires two cycles to complete. The first stage of the pipeline is full if an instruction was issued to the pipeline during the previous cycle, or if the stage was full in the previous cycle and the instruction is stalled. All these dynamic properties are then represented as CSP variables that are constrained, based on the micro-architectural characteristics of the components. The number of instructions included in the test must be decided in advance and all the timing variables of the instructions are added to the CSP and constrained as required. The user can also add constraints that specify some specific desirable micro-architectural event. The CSP solution includes a precise cycle-accurate description of the flow of all the instructions of the test through the micro-architectural components. More importantly, the required architectural test program can also be deduced from the CSP solution.

For example, suppose the user wants to hit the following micro-architectural event: an instruction, I_1, enters stage 1 of a pipeline and stalls there for several cycles. One cycle later, another instruction I_2 enters stage 3 of the pipeline and also stalls, during which time a misalign type exception is generated (by another instruction I_3) that causes I_2 but not I_1 to be flushed (i.e., prematurely exit the pipeline). Note that the event refers to three abstract instructions but the test may need to include many more to cause the precise desired timing. Figure 8 shows the constraints that the user can define to specify this event.

[3] A pipeline is a micro-architecture mechanism that enables the concurrent computation of multiple instructions. The computation of a single instruction is broken into stages that are handled in corresponding stages of the pipeline. An instruction enters the pipeline and goes one by one through all the stages. Each stage can handle just one instruction at a time, but the pipeline can concurrently handle instructions at various stages.

I[1].pipeline = I[2].pipeline
I[1].stage[1].stall_time > 0
I[2].stage[3].stall_time > 0
I[2].stage[3].entry = I[1].stage[1].entry+1
I[3].flush = true
I[3].flush_time ≥ I[2].stage[3].entry
I[3].flush_time ≤ I[2].stage[3].exit
I[1].destiny=finished
I[2].destiny=flushed

Fig. 8 Using constraints to define a micro-architectural event

5.4 Address Translation

Most modern processor architectures provide a virtual address space for their software applications. The operating system maps these virtual addresses to physical hardware resources (such as memory, disk, or I/O ports) using address translation mechanisms. The translation is executed primarily using hardware mechanisms such as translation tables in memory and control registers. Address translation also commonly plays a part in memory protection and caching mechanisms by maintaining related properties for basic translated memory units such as pages or segments. The operating system maintains these resources through software, but the translation itself is executed by the hardware.

Gutkovich and Moss [43] demonstrates the modeling of the address translation mechanism of Intel's IA-32 architecture as a CSP. The logical address is translated first through a segment descriptor table and then possibly also through a page table with possible exceptions occurring on the way. All the relevant translation properties such as the initial, intermediate, and final addresses, the table entries that were used, and their fields, are modeled as CSP variables. These are then constrained according to the architectural definitions and also potentially with user-defined constraints (such as avoiding exceptions).

Adir et al. [44] describes a generic platform for modeling translation mechanisms as flowcharts that are then automatically converted into CSPs. This is an appropriate approach due to the procedural nature of the address translation architecture. The flowchart describes the possible successive stages of the translation process, each linked to its antecedent by a set of conditions. Each stage comes with a set of properties that are related to properties in previous stages. These property transformations and edge traversal conditions are then converted into corresponding constraints in the CSP.

A simplified example is shown in Fig. 9. The initial stage has a *Virtual-Address* attribute providing the virtual address to be translated into a physical address, an *Access-Type* attribute indicating the type of access (e.g., data or instruction fetch), and a *Control-Register* attribute providing the value of a register that controls the translation process. From this stage, the translation may proceed through one of three possible edges, depending on the value of the *Control-Register*.

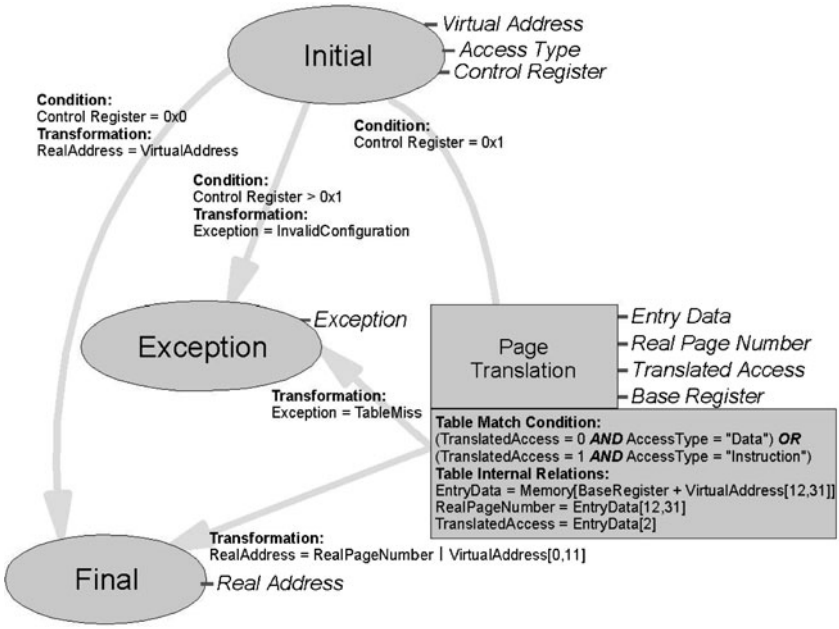


Fig. 9 Translation modeling with CSP

The transformation associated with the corresponding edge links the attributes of the succeeding stage with the attributes of the preceding one. In the figure, $A[i,k]$ designates a bitstream, which is the sub-stream of bitstream A between bit i and bit k. The symbol '|' designates bitwise-or.

Translation tables are widely used in various translation mechanisms. A model of a translation table includes a set of specific properties, such as its size and entry fields. It also provides a *match condition*, which defines whether a table entry translates a given address. In addition, it specifies the transformations between the attributes of the stages connected by the translation table and various additional constraints between table attributes. As the figure shows, there are different transformations and succeeding stages, depending on whether a matching entry is found in the table (i.e., the match condition is satisfied).

The size of the CSP can become quite large; up to 270 variables with 599 constraints for PowerPC address translation, and 218 variables with 493 constraints for address translation on an InfiniBand bus. Note that many of the variables are 64-bit numbers, and the constraints may be complex as they deal with bit indexes and bit fields of those numbers. However, in the solution to the CSP, only one path out of the many possible paths of the flowchart is actually taken. Thus, the whole CSP becomes highly conditional (meaning that large parts of the problem are dropped dynamically, depending on the values of some of the variables in the solution [29]) and a CSP engine with strong support for conditional CSPs is required to enable reasonable performance [45]. Performance can also improve if the constraints that

define the taken path are given propagation precedence over those that relate the properties on the path stages. On the other hand, such a precedence approach may lead to many deep backtracks on hard problems, requiring heuristic tuning of the precedence relations.

5.5 System Modeling

Figure 10 presents a view of a hardware system. A hardware system is composed of various hardware components such as processors, memories, and busses, connected to each other. Verification of the system as a whole can find bugs in the way the components interact with each other, and can find bugs in each of the units by creating system-level stimuli, which is difficult to create in the context of a single component.

The system model of the system level test generator of [47] is composed of three different parts. First, a model of all component types (e.g., processor, memory) that includes all constraints resulting from the specification of this component. For example, the following constraint may be specified for a bus: [*at any time, the accumulated number of bytes flowing into a bus minus the number of bytes flowing out of a bus cannot exceed X*]. This constraint makes sure that the buffers in the bus do not overflow, and it may limit the test case inputs for components even if they are distant from the bus.

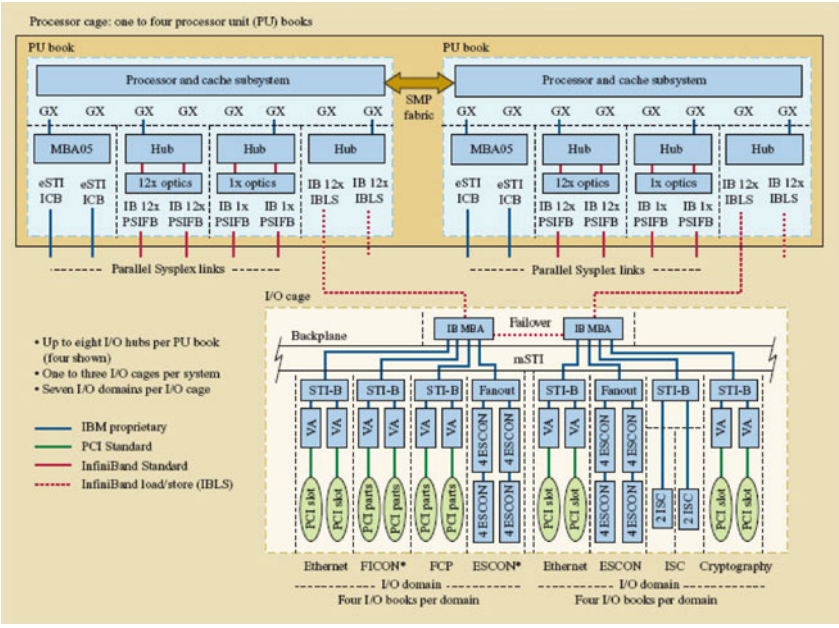


Fig. 10 IBM's system Z. Reproduced with permission from [46]

Second, the configuration is modeled. The number of instances of each component type is specified, as well as the connectivity between the various components.

Third is a model of all possible transactions between the components. This includes the component types that are the initiator and target of the transaction, and the constraints particular to that transaction. For example, a transaction of type 'data-transfer' may specify a processor as the initiating component type, a memory as the target component type, and a set of variables that include the data being transferred, the processor's source register, the memory target address, and more. The following constraint needs to be added to the model to mimic the correct behavior of the system when the transaction takes place: [*After a 'store' instruction from register X in CPU-Y to address Z in MEM-W, the data value of address Z is equal to the data value of register X*].

A CSP for a system interaction to be generated into the test can now be constructed by adding the constraints induced by the component types participating in the interaction (including at least the initiator and target of the transaction) and the constraints related to the particular interaction.

The model can also include expert-knowledge constraints that are specific to a system. For example, the following constraint *all transactions in the system must go through the same bus* ensures traffic congestion through the chosen bus, thus overstressing the bus and increasing the probability of exposing a bug related to data contention and race conditions. As usual, the expert-knowledge constraints are soft.

Lastly, the user may specify constraints related to the verification task at hand. For example the user can request that all transactions would use, at most, two memories, or that a transaction would go through path lengths larger than some number, where the path length is defined as the number of components on the path.

All in all, the final model contains constraints describing each of the component types, constraints describing the valid and interesting interactions between any two component types, and constraints defining what constitutes a valid path (e.g., number on components of a specified type on the path, separation between them). Now, for any transaction, it is up to the CSP solver to find an initiating component, a target component, a path between them, and input stimuli to the initiating component, such that all the constraints are satisfied. Section 6.5 discusses some of the issues involved in solving this kind of CSP.

The separation of the model into separate sets of constraints is necessary for allowing fast maintenance and changes in the model. This is particularly important in system verification because this level of verification is done toward the end of the verification activity when time-lines are inevitably short. Thus, when the design changes at the last minute, only the model of the affected entity needs to be changed, leaving all other areas of the model intact.

5.6 Automatic Test-pattern Generation

The process of manufacturing chips (for a presumably verified design) can be quite error prone in itself. The percentage of error-free elements (termed yield) can get

as low as a few points. Automatic test pattern generation (ATPG) tools [17] look for manufacturing problems by applying sequences of signal vectors at the circuit's input interfaces. The resulting values at the circuit's outputs are then compared with the values expected from a valid circuit.

ATPGs target potential faults in the circuit – the most common are "stuck-at" faults where some internal signal is stuck at 0 or at 1 regardless of the inputs. ATPGs can then be used to create a test suite that covers all observable production faults of this kind. A single test can expose several faults in the circuit but most ATPGs eventually need to target a single stuck-at fault at the output of some specific logical gate. For example, if a specific "AND" gate is suspected of being stuck at 0, then the test would need to ensure that both inputs to the gate are 1. The test would indeed need to select a sequence of inputs to the entire circuit that would eventually justify such a setup at the internal gate's inputs and also that if a 0 indeed results in the output of the gate then this bad value is observable, that is, it propagates to at least one of the circuit's outputs.

The ATPG problem as thus described is NP-complete with regards to the number of binary inputs to the circuits, and most ATPG tools employ dedicated technologies that involve various search strategies and heuristics. Some of these are SAT based, which appears appropriate given the boolean-logical nature of the problem. Abramovici et al. [48], Hentenryck et al. [49], Simonis [50], Brand [51] demonstrate how the problem can be modeled and solved as a CSP. The approach presented in [49] and extended in [51] models the gates as implementing logical functions on variables with domains that include values beyond just 0 and 1. For example, the value "d" represents a signal with value 1 that should really have been 0, and the value "dnot" represents a signal with value 0 that should really have been 1. Thus, for example "0 or d = d" because a wrong value of 1 that propagates through an "OR" gate results in a wrong value of 1 at the output of the gate. Other signal values like "irrelevant" or "uncontrollable" are also proposed. The states of the signals connected as input/outputs to the circuit gates are modeled as CSP variables with these extended logical domains. Every gate is modeled as a constraint between the gate's input and output signals that corresponds to the function of the gate. Other constraints set the input signals to the targeted gate as implied from the targeted fault, and also require that at least one of the circuit output signals has a value of d or dnot. The CSP engine then searches using local propagations at the gate points. This differs from many common ATPG tools that take a more global control strategy. Yet, [51] was able to report reasonable results with various types of logics (i.e., variable domains) on several benchmark sequential circuits.

6 Advanced Topics and Challenges

6.1 Dynamic Test Program Generation

Consider a stimuli generator for a processor that generates an assembly program, one instruction after the other, in program order. The generator is also dynamic in

that it keeps track of the state of the resources, such as registers or memory locations, as they should be when the program (as generated so far) will be executed on the verified processor. This is done using a reference model, during generation time, which simulates the correct behavior of the processor. As seen in the example in Fig. 5, knowing the current values of the resources is useful for maintaining the validity and quality of the generated instructions.

Gutkovich and Moss [43] propose a method of supporting dynamic resource values within the framework of a CSP for the instruction. The CSP for the instruction includes data and address variables for the participating resources (similar to Fig. 5). A "customized" constraint defined on these CSP variables refers to tables maintained in the reference model to determine the current values of the resources and project the address and data variables accordingly. In addition, for the sake of performance, the domain reductions for each of the address and data variables are delayed until enough is known about the other. For example, in the case of a memory resource, the bounding of the data variable can be delayed until the address variable is also bound. Such delaying tactics can be relaxed when the domains are smaller, as is more natural in the case of register resources.

It is possible for the current values of the resources to be still undetermined even in the reference model since the reference model would know only about the resources that were referred to so far in the test. Thus, the bounding of a data variable for some resource can have the side effect of updating that resource in the reference model tables. A dilemma related to such implementation occurs when the CSP includes variables for multiple resources that may become identical. For example, consider a CSP for an Add instruction that sums two memory operands. One needs to ensure that the data variables of the two memory resources are bound to the same values if the addresses are the same. One way of handling this is simply to add a direct constraint that relates these data and address variables. When using MAC-3, another approach is possible. Whenever a data variable of some resource is bound, all the custom constraints that relate the address and data variables of resources of the same type to the reference model are tagged as needing further propagation before the MAC-3 fixed point is reached. This solution also nicely handles the case of overlapping non-identical resources.

6.2 Problem Partitioning

CSPs for many verification problems tend to be relatively large with many complex constraints over a large number of variables with large domains. For example, [18] models a whole test program as a CSP with tens of thousands of variables and constraints. As a result, only short tests can be handled with reasonable performance. The test generator in [18] targets micro-architectural events that can be hit with a relatively short sequence of instructions. However, most stimuli generators aim at tasks that require much longer scenarios [52]. This enables, for example,

hitting more events in a single test and testing the design while its resources are already "warm" with the effects of previous instructions (e.g., full buffers, caches, pipelines).

Thus, various kinds of partitioning techniques are employed to enable the generation of long tests. There are basically two generic approaches to problem partitioning: divide and conquer, and abstraction. With the divide and conquer approach the CSP is split into a set of subproblems with a much higher constraint density inside the subproblems than among them. For example, the test program generator in [20] creates and solves a CSP for every instruction of the test separately, one after the other in program order. The generator of [47] creates a test with system level interactions and similarly solves a separate CSP for every interaction. The obvious problem with the divide and conquer approach is that the CSP is often not so neatly divisible and there are still some (hopefully few) constraints that relate the subproblems, which may lead to ultimate failure if ignored while the subproblems are solved. The simplest solution is backtracking, but this can be of limited value, depending on the type of the interproblem constraints. Bin et al. [8] propose using a separate CSP for the interproblem constraints, which shares the variables with the subproblem CSPs. Then, arc-consistency is reached in the inter-problem CSP before every subproblem CSP is solved.

Bin et al. [8] discuss the application of CSP abstraction [53] to test generation. This is particularly appropriate for cases where the CSP is highly dynamic in the sense that many components of the problem are conditional and would not be relevant in the final solution. First, a separate CSP is constructed and solved to determine the "structure" of the base problem and only then the base CSP is solved with far fewer potentially irrelevant components. For example, a generator that creates an address translation following the flowchart approach described in [44] can start by solving a CSP to decide which translation path to take and then continue with solving a CSP for just the chosen path. The problem with abstraction-type partitioning is that a solution cannot be guaranteed for the lower-level CSP once the higher-level CSP is solved.

Adir et al. [54] shows how CSP abstraction can be used in the verification of various types of parallelism in processor architectures such as Super-scalar, Multi-threaded, and VLIW architectures. Very long instruction word (VLIW) is a performance-enhancing mechanism that considers a short sequence of basic instructions as a single "long-word" instruction and executes the constituent instructions concurrently. Figure 11 shows the model for a VLIW with places (slots) for up to four instructions to be executed concurrently. The VLIW model here (based on STMicroelectronics' ST100 and ST200 [55] architectures) defines four *slot types* each enumerating the valid instructions for a slot (including *Nop* indicating an empty slot). Every physical slot points to one or more slot types. The architecture imposes constraints on the composition of the VLIW such as requiring no more than two loads, or prohibiting more than one instruction of the VLIW to write over the same resource. The proposed solution is to first solve a CSP problem to decide on the composition of the slots that is, what instruction types are to be placed in each of

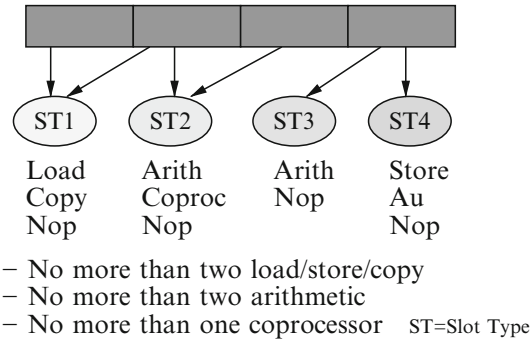


Fig. 11 VLIW modeling

the four slots. Then a second CSP is constructed for deciding on the properties of the four selected instruction types such as resources used and access types, in a manner similar to that shown in Fig. 5.

6.3 Compliance Validation

Verification refers to the entire process of uncovering bugs in a design, while the scope of compliance validation is ensuring that the architecture specification has been correctly interpreted by the designer during the implementation. The importance of validating architecture compliance has been exacerbated in recent years due to a trend for standardization at all design levels. This is partly due to a shift from custom ASICs toward SoC (system on a chip) designs, which include ready-made components with well-established and recognized architectures. Companies responsible for such designs and the respective organizations defining the standards obviously have a strong drive to ensure that the design complies with its specification.

It is important to note that the types of errors that should be targeted here are different from the types of errors caused by a bad implementation of a well-understood specification. However, ensuring that a specification was understood correctly is a much more feasible task than ensuring that it was implemented correctly. Although finite, the specification describes how the design must behave in a potentially infinite number of scenarios. To be confident in the correctness of an implementation, one must either simulate enough test cases until a sufficient level of coverage is reached, or formally prove the correctness of all the scenarios. However, to check against misinterpretations, it should suffice to ensure that each of the finite number of requirements made in the specification was understood correctly.

Adir et al. [31] describes a technique for validating the compliance of a processor design to a processor architecture specification by targeting certain cognitive models of possible human misinterpretations. These can include misreading an ambiguous text, forgetfulness, or a wrong association with a similar but different specification

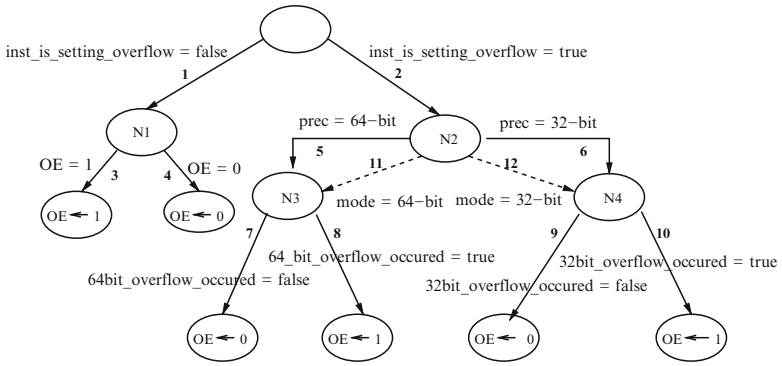


Fig. 12 Flowchart for the overflow setting process

that is familiar or seems reasonable to the reader. The association can be with some other behavior in the same specification or with behavior from another architecture.

The approach here is to model the processes performed by the instructions of the architecture as flowcharts. The edges are labeled with transition conditions, based on the current machine state or instruction properties, and the actions are concentrated at the leaves. To this flowchart are added *false* edges and nodes to mimic behaviors that are not part of the specification.

Figure 12 shows a flowchart that describes the overflow flag (OE) setting by the fixed-point instructions. The architecture distinguishes between instructions that may set the OE bit and those that should not. This distinction is made by the two edges exiting the root node. The sub-tree of node N_1 specifies that the OE bit must keep the same value that it had before the operation. A "precision" field of the fixed-point instructions defines if the operation is to be performed on all 64 bits of the input registers or only on the least significant 32 bits. This is represented by the two edges E_5 and E_6. Finally, the edges exiting nodes N_3 and N_4 lead to the setting or un-setting of the *OE* bit, depending on the occurrence of an overflow.

The architecture also defines a general precision mode of either 32 or 64 bits, configurable in the control register. However, the precision mode is *not* applicable to instructions that have a precision field. One may think that the precision mode *does* have an effect on these instructions, thereby overriding the precision field. This misinterpretation is captured by the false (dashed) edges.

Thus, in a *False edge* misinterpretation, the process goes down a false edge instead of the right true edge (e.g., E_{12} instead of E_5) because of wrongly assuming the existence of the false edge. In a *Wrong edge* misinterpretation, the process goes down a wrong true edge instead of the right true edge coming out of the same node (e.g., E_6 instead of E_5) because of a misinterpretation of the conditions of the true edges. Now, on top of these definitions, one can define various types of coverage models such as for every path of the flowchart test the case where the path is wrongly not followed due to some wrong or false edge on the way. The system proposed in [31] can be used to create a test suite that covers all the misinterpretations targeted by the coverage model.

Variable domains:
$inst\_is\_setting\_overflow = \{true, false\}$
$prec = \{32, 64\}$
$mode = \{32, 64\}$
$32bit\_overflow\_occurred = \{true, false\}$
$64bit\_overflow\_occurred = \{true, false\}$
$Instruction = \{add\_64, add\_32, load, store, \dots\}$
$operand1 = \{0xXXXXXXXX\}$
$operand2 = \{0xXXXXXXXX\}$
Constraints:
Prefix: E_2 $inst\_is\_setting\_overflow = true$
True-edge: E_5 $prec = 64$
False-edge: E_{12} $mode = 32$
True-suffix: E_7 $64bit\_overflow\_occurred = false$
False-suffix: E_{10} $32bit\_overflow\_occurred = true$
Instruction-Semantics:
$(Instruction = add\_64) \rightarrow (inst\_is\_setting\_overflow = true)$
$(Instruction = add\_64) \rightarrow (64bit\_overflow\_occurred \leftrightarrow operand1 + operand2 > 0xFFFFFFFF)$
$(Instruction = add\_64) \rightarrow (32bit\_overflow\_occurred \leftrightarrow operand1[0:31] + operand2[0:31] > 0xFFFF)$
There are also similar constraints for other instructions of the architecture

Fig. 13 CSP for a false-edge misinterpretation

The system works by producing a machine state and an instruction instance that exposes the targeted misinterpretation, if it exists. For example, consider the flowchart in Fig. 12 and the false-edge misinterpretation that takes edge E_{12} instead of E_5. Figure 13 shows the CSP corresponding to this misinterpretation. The resources should be initialized so that the instruction flows through a *prefix* path from the root of the flowchart down to node N_2 and continues on edge E_5. This ensures that if there are no misinterpretations, the flow will pass through E_5. The transition constraint of the false edge E_{12} must also be satisfied. This is because the test must lead the flow along the false edge in the event of a misinterpretation. Finally, the test must also ensure that taking E_{12} by mistake is observable. This is done by finding two *suffix* paths leading down to leaf nodes–one from N_3 and the other from N_4, so that the two leaves at the end of these two paths have different assignments. In our example, the prefix path is E_2 to which are added the true edge E_5 and the false edge E_{12}. The two suffix paths E_7 and E_{10} are appropriate because they end in different assignments.

The flowchart is modeled as a CSP in which variables represent the relevant parts of the machine state (in our example, the status and control registers) and instruction properties (in our example, the precision field and data operands, and the distinction of the instruction as setting the overflow). The transition and conditions are modeled as constraints. The system then searches for appropriate prefix and suffix paths based on the assignments in the leaves and then uses a CSP engine to find an appropriate machine state and instruction field setting to test for the misinterpretation. For our

example, a solution could be an *add_64* instruction (assuming it is of the type that sets the overflow bit and that its precision field is 64) that causes an overflow beyond the 32nd bit but not beyond the 64th bit. In addition, the 32-mode bit in the control register should be set to match the false edge E_{12}.

6.4 Checking the Processor's Memory Model

A memory model (or memory consistency model [56]) is that part of the processor's architecture that specifies how memory appears to behave with respect to applications running on the processor. The strongest and most intuitive memory model for a programmer is sequential consistency [57] with which "...the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." This memory model is the most natural for a programmer to work with. Unfortunately, it would impede many common processor design optimization techniques.

Consider, for example, a *load* instruction that takes many cycles to complete (because it requires a slow memory access) and is followed in the program by some quick arithmetic operation that only manipulates internal registers independent of the *load*. With an out-of-order architecture, the processor can opt to perform the arithmetic operation before it performs the load. Another common optimization is to speculate on the direction to be taken by a conditional branch and to start performing the instructions following the branch in the speculated direction in advance (with hopefully a small probability of paying the penalty for mistaken speculation). It could be very difficult to completely hide the effects of such optimizations from the programmer. Many processor architectures define memory models that are more relaxed than sequential consistency and thus do not completely hide the effects of these optimizations and yet provide a reasonable model for the programmer with which to work. For example, in general, the PowerPC architecture does not allow a process to observe that its own operations were performed out of order but in some cases it does allow a process to observe that *another* process' operations were performed out of order. Thus it is possible for a program running on one process to include two stores and for another process to observe the memory modifications performed by these two stores in reverse order to their original program order.

Checking that a processor observes the rules of the memory model can be carried out by simulating multiprocess programs. While the program is simulated, several monitors accumulate a history of memory operations (loads and stores) as they are actually performed in the system. Once this trace is available, the actual checking can be executed off-line. The inputs to the checker include the program that was executed and the trace information gathered by the monitors while it was executed. The checker's task is then to try to detect violations of the processor's memory model if these indeed exist and are observable in the trace.

In general, checking such traces for memory consistency is known to be an NP complete problem, though additional information like mapping loads to the

stores that created the loaded values together with store order information or limiting the number of participating processors or addresses can make the problem tractable [58, 59]. This calls for various heuristic search methods including SAT-based techniques [60, 61]. The tool presented in [61] can also use constraint-logic-programming with Prolog to check the legality of test execution with respect to a given memory model that is specified using an extended type of predicate logic.

Recent (unpublished) work at IBM Research – Haifa also demonstrated how CSP can be used by a checker that verifies that a processor observes the rules of the memory model while a multiprocess program is executed on it. Several alternative approaches to modeling the problem with CSP were tried. For example, in the sequential consistency model, the trace must be consistent with some total ordering of all the operations. The problem of the CSP is to come up with some such total order if possible (or declare a violation of the memory model if no such order exists). The operations can be modeled as vertices of an ordered graph where the edges represent the view order. The CSP includes a Boolean variable for every potential ordered edge. The constraints of the CSP require that the order be total (handled by N^2 constraints for antisymmetry, N^3 for transitivity, and N^2 for totality). Also, every load operation must be ordered after the store operation that wrote the loaded value and all other stores must be ordered either before the *mapped* store or after the load (i.e., the loaded value is not masked by another store). This requires an additional Boolean CSP variable between all load-store pairs that indicates whether the load is mapped to the store. A solution to the CSP would then provide a fully specified view order of all the observed operations that explains how memory appears to the programmer in a manner consistent with the architectural specification of the memory model.

Figure 14 shows an example of an attempt to solve such a CSP. The dashed arrows correspond to reduced CSP variables. Suppose that the stores are somehow known to be before operation Op1 and the loads to be after operation Op2. In part A of the figure, the CSP engine tries to reduce an order variable so that Op1 is ordered before Op2. This leads to failure because any reduction of the variables ordering the two stores leads to a contradiction in the constraint that prohibits the masking of stores from the observing loads. Part B orders Op2 before Op1, which now enables several possible solutions to the CSP, such as the view order given at the bottom of the figure.

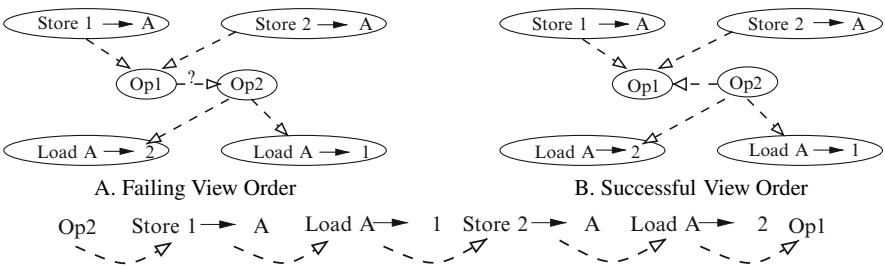


Fig. 14 View order CSP

Other modeling techniques were tried, based on the formal definition of the PowerPC memory model in [62]. Here, the model includes a separate, potentially different, *partial* view order per process. The constraints of the CSP further limit and relate the different view orders according to the particular requirements of the PowerPC memory model. Another variation was to model the graph vertices as integer CSP variables (instead of Boolean variables for the edges). The view order is then derived from the numeric relationship between the variables. This approach reduces the number of constraints but makes their propagation more difficult. Further study is still in progress with these approaches and a publication can be expected.

6.5 *System Verification: Coupling of the Transaction Path CSP with the Unit CSPs*

When generating stimuli for a whole system, a multitude of components need to be taken into account. During regular operation of the system, the components interact among themselves. It is this interaction that needs to be stimulated and verified for correctness of behavior. From a high-level point of view, the verification engineer thinks of such interactions in terms of transactions being executed between the various components (a lower-level approach would be extremely difficult to manage at the system level). So the engineer can request, say, a direct-memory access (DMA) transaction involving data going from the USB to a memory component inside a micro-processor through various busses and bridges (see Fig. 15). The user may specify various additional constraints, such as the required length of the data transferred, the memory addresses of interest, and more. In addition, the user can specify explicit requests at the system level; for example, that the number of components between the USB and the memory components be smaller than (or larger than) some number, that certain components be included or excluded from the transaction, that two neighboring bridges on the path would be of identical type, and so on.

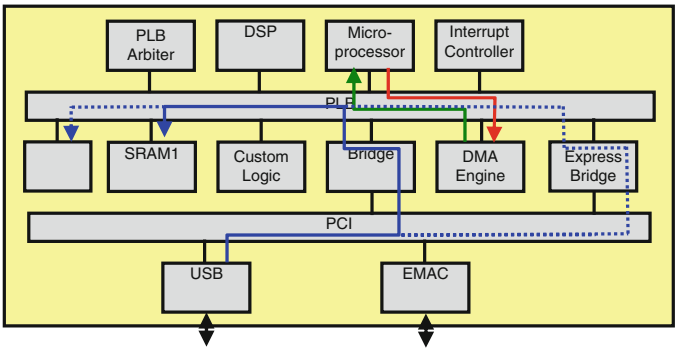


Fig. 15 Schematic view of a direct-memory-access transaction in a hardware system

Once the user specifies the required transaction and additional unit- and system-level constraints, the tool must find a valid transaction satisfying all user and architectural constraints, and as many bias (soft) constraints as possible. A bias constraint can be at the unit level, for example, cross a memory page boundary, or at the system level, for example, choose components that were heavily used already by previous transactions – thus maximizing congestion on some components during simulation.

However, there is another challenge in generating such stimuli. This stems from the fact that the choice of components participating in the transaction determines the set of unit-level constraints to be satisfied. For example, the transaction of Fig. 15 can pass either through a regular 'bridge' or an 'express bridge'. The two bridges have very different behaviors, and therefore impose different architectural constraints. We therefore have a highly conditional problem, where each component introduces a set of constraints into the system-level constraint problem however; this set of constraints is conditional on the component being ultimately part of the solution of the transaction.

It turns out that this conditional problem is much too large to be solved by standard conditional-CSP algorithms. This is mainly because of the scale of the full problem compared to the size of the final solution. For example, a typical 32-way system of processors, their memories and the various interconnecting and I/O components may be composed of several hundred components, while the solution for a single transaction may involve only five or ten components participating in the transaction. Under this scale of conditionality, even sophisticated conditional-pruning techniques [45] fail to be useful.

The most naive approach to tackle this problem is to partition the CSP into two; first, the system-level CSP that includes all constraints relating explicitly to the identity of the components participating in the transaction. For example, the types of components required by the transaction, the connectivities of the components, the user-defined constraints, and the general biases related to the component selection. Once this CSP is solved, we have a solution for the components participating in the transaction. We can now proceed with the second part of the CSP and formulate and solve the lower-level constraints related to the properties of those specific components.

This approach has an obvious drawback in that the full CSP may be unsolvable over the chosen components. For example, a user may request a particular memory region that is nonexistent in the chosen memory, or two components may be chosen such that their lower-level interface is incompatible for the transaction. Of course, there are much more complex examples where the incompatibility of the chosen components results from interaction of two or more low-level constraints.

We solve this problem by doing a static analysis of the system, one time per system. In this analysis, for each transaction type, we choose all possible combinations of 'principal components', where a principal component has a significant amount of logic such that it introduces enough constraints to influence the lower-level CSP. For each such choice of a set of principal components, we run the lower-level (full) CSP up to the first arc-consistency level. We then save the reduced

arc-consistent domains of this CSP. Once we have this static knowledge, we add it in the form of constraints on the component-choice (high-level) CSP each time we generate a transaction of this type in that system. In addition, when the user specifies constraints on the low-level CSP variables, we add those constraints to the component-choice CSP as well. Then, the combination of the reduced-domains constraints and the user-specified constraints result in pruning of incompatible components out of this CSP.

This method proved itself in the sense that it drastically reduced the number of failures of the lower-level CSP because of bad component choices at the higher level. However, it still has one major drawback. For large systems, the static analysis can be very time-consuming, taking hours or even days to analyze all transactions (lazy analysis, done only when a given transaction is requested for the first time, may somewhat reduce the pain here, but means that a transaction which is ordinarily generated in seconds may now take more than an hour to generate). This analysis needs to be done each time the system is changed, even if the change involves only a single repositioning of a component. Hence, the method is viable only at later stages of the system verification phase, when the system being verified is relatively stable.

With the expected persistent growth in complexity of systems, coupled with decreasing time available for verification at the system level, it is clear to us that more powerful techniques that are able to cope with the full-system CSP will need to be developed in the foreseeable future. We think this particular problem can serve as an important test bench for state of the art CSP decoupling and decomposition algorithms. In addition, it can stimulate research in graph-based CSP, where paths on a graph (connected components participating in a transaction in our case) are treated as CSP variables among other variables introduced by each of the nodes of the graph.

6.6 *Complex Data Transfers: Unbounded Vector Constraints*

Figure 16 describes a clustering network in which two or more nodes are connected via a high-performance network. Typically, each node contains a CPU unit responsible for managing the data transfer from the node's side, a memory component where

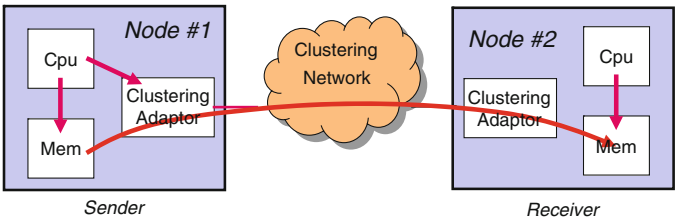


Fig. 16 Sender and receiver nodes in a clustering network

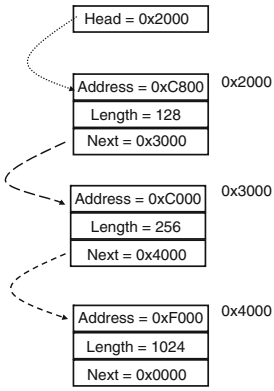


Fig. 17 Linked list in memory, specifying the chunks of data to be transmitted

data is being stored and retrieved, and a high-end clustering adapter responsible for sending or receiving the data in the most efficient way. The InfiniBand protocol is an example of such a modern cluster communication scheme. The most important aspect of this network is the requirement for fast and reliable transfers of huge amounts of data (for example, think of an online video being broadcast to clients).

One of the most common features in all implementations of clustering networks is that data is often transferred from fragmented sources in memory. This means that it is up to the CPU to create an ordered list, or 'vector', of all memory regions from which data is to be retrieved (or to which it should be stored). For example, the CPU can create the linked list, of Fig. 17. In this list, the first element, 'or head' is located in some reserved space in memory recognized by the network adapter, and points to the first actual data-element descriptor. Each such descriptor contains the memory location of the chunk of data to be transferred (start position and length), and a pointer to the next descriptor. The adapter then recognizes all those chunks and sends them to the network (or receives the chunks from the network and places them in the specified locations).

The challenge in verifying this system is that the adapter implements extremely complex mechanisms for fetching the chunks, concatenating them, breaking again, and sending into the network in the most efficient way. These mechanisms are prone to bugs and innovative tests must be designed in order to verify their correctness. In many cases, the verification engineer, who has an intimate understanding of the mechanisms, would require some specific relations between the various chunks of data in memory. The engineer must be provided with the most simple way of specifying those relations, while not limiting all other (unspecified) aspects of the transfer.

For the user to do his or her job in the most natural way, we have come up with a generic modeling structure for vectors as in Fig. 18. Here, the modeler can specify the CSP variables that comprise each of the general vector elements (for example, in the figure, 'address', 'length', 'memory address', and 'next' variables). These variables are duplicated for all elements in the vector. In addition, the user can add specific variables for each of the vector elements. Finally, the user can add

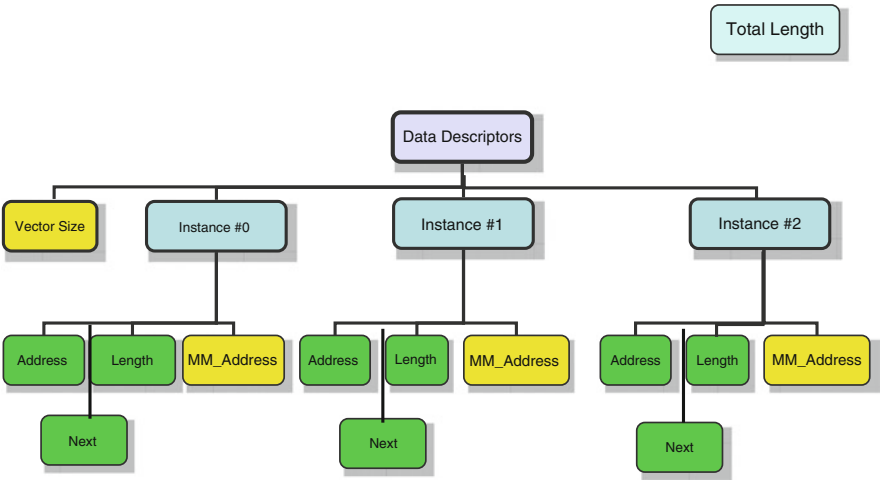


Fig. 18 Vector model of the clustering data transfer. Size of the vector is also a CSP variable

constraints on all elements, for example: "mm-address[i+1] = mm-address[i] + 1", or they can add constraints that are specific to a single element.

One important aspect of this scheme is the question of the length of the vector. Of course, if the user requires a vector of some length, they could just specify so. However, as with all parameters of the test, if there is no reason to specify the length, then it is best left unspecified, and it is then up to the stimuli generator to decide the length. This decision can be influenced by two aspects: first, the length can be biased by the generic expert knowledge – for example, it may be sometimes beneficial to have very long transfers to reach deeply into the adapter. At other times, short transfers may be beneficial because they can stress the adapter by passing more transfers in a shorter period of time. Second, there may be an upper-bound imposed on the length because of constraints that become unsatisfiable for some element n. Then, only a vector with less than n elements can provide a satisfying solution.

When the length is not specified by the user, it needs to be determined by the CSP solver. To this end, we treat the length as a regular CSP variable, with domain extending from 0 to some large number. The algorithmic problem of solving this 'unbounded CSP' in an efficient way, while still enabling solutions with large enough lengths, is still open. The problem stems from the fact that as long as the domain of the length variable is large, the CSP can be thought of as a disjunction of different CSPs, each with a given length out of this domain. Therefore, propagation into the length domain may be very weak (it may require a complete solution to a sub-CSP to determine that a specific length value is infeasible). On the other hand, just choosing a length randomly may result in an infeasible solution, and would require many backtracks on this domain. Note that depending on the nature of the constraints on the length variable, the feasibility or infeasibility of the length may

not be monotonic. It may happen that the problem is infeasible for length k, but feasible for $k + 1$, and we do not want to miss all the large-length results just because we encountered one infeasible length k.

Currently, the test generator of [47] solves the problem by separating it into two CSPs. The first CSP includes all constraints that are likely to affect the length of the vector. Once this CSP is solved, we proceed to solve the complete CSP, including all constraints, with the only difference being that now the domain of the length variable is fixed to a single element. The challenge in this method is to identify all the constraints of importance for fixing the length. This requires expert knowledge, and sometimes fine-tuning of the method. Drawing the line in the wrong place may result in a CSP as hard to solve as the original (with too many constraints identified), or an infeasible choice of length for the full CSP (too few constraints identified).

6.7 Floating Point Unit Verification: A Small and Hard CSP

It is well recognized today that the processor's floating point (FP) unit is one of the most difficult hardware elements to verify. This unit, the workhorse of numerical number crunching, is steadily increasing in complexity to keep track with demand. In addition, it is almost impossible to find acceptable workarounds to bugs shipped in the hardware. While the operating system may avoid a memory niche to circumvent a rare corner case bug in the load-store unit, it is expected that any numerical calculation can be executed with correct answer by the FP unit.

It turns out that many interesting verification scenarios for the FP unit can be targeted by constraining the test with four types of constraints applied simultaneously to three FP numbers [19]. A floating point number is represented by a mantissa and an exponent, as in Fig. 19 in which three 64-bit FP numbers are shown. The four types of constraints are:

- *Operation*: $a \times 2^{\alpha}$ op $b \times 2^{\beta} = c \times 2^{\gamma}$, with 'op' being any of the floating point operations '+', '-', '*', '/', and more.
- *Mask*: a list of bits in each of the six variables a, b, c, α, β, γ that are forced to be 0, and another list of bits forced to be 1.

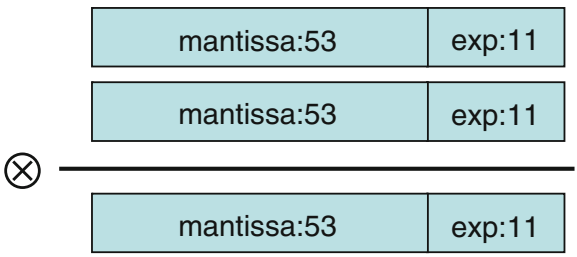


Fig. 19 Exponent and mantissa representation of an IEEE floating point number

- *Range*: each of the six variables constrained to be in some given range
- *Number-of-ones*: the number of bits equal to 1 in each of the six variables constraint to be in some range (e.g., in a, the number of 1's is between 5 and 8).

This CSP is very small: it is composed of six variables, and, at most, 19 constraints (usually fewer, although the 'operation' constraint is always present). However, the CSP is one of the hardest to solve that we have encountered. In particular, consider the case where the domain of the variable c is a single element, where a and b have an unrestricted initial domain, and where the only constraint is 'operation' with the operation being 'multiply'. Then, this problem reduces to a factorization problem of a 53-bit number (or more if dealing with a 128-bit processor). Moreover, the propagation problem here is even harder than simple factorization, and even if computation speed is not a problem, representing propagated domains in memory becomes infeasible. Once the mask, range, and number-of-ones constraints are added, it becomes clear that any MAC-based algorithm is unable to tackle this problem.

We approach the floating point verification problem by first trying to solve it analytically where possible. Some operators (e.g., plus or minus) lend themselves more easily to analytic approaches than others (e.g., multiply or divide). Even in such cases, this approach cannot solve the full problem, so we address it in stages by solving a simplification analytically, and then using CSP or iterative methods to solve the actual problem. We refer to this as the 'semi-analytic' method in Fig. 20.

In cases where analytic solutions are not available, we resort to three different CSP solving routines. First is SAT, in which each bit of the CSP variables is represented by a single SAT variable, and the 'operation' (and other) constraints are transformed into clause constraints on those bits. Second is heuristic search, where simple search methods are adapted for each type of operation or constraint. Third is full-fledged stochastic local search.

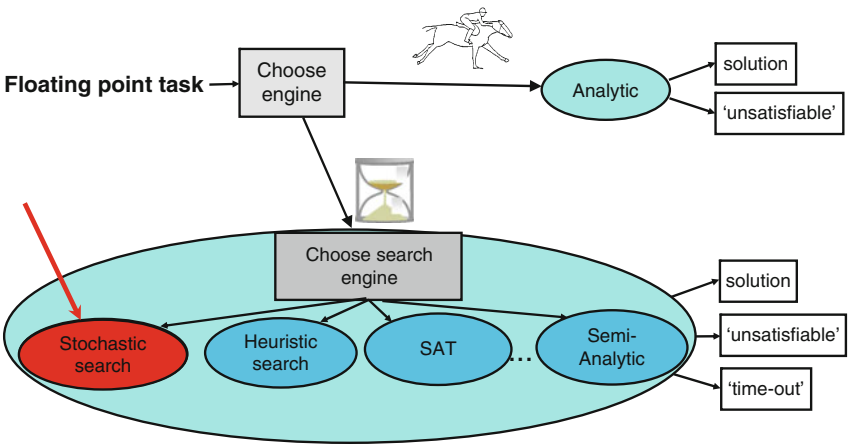


Fig. 20 Toolbox of solvers participating in the creation of stimuli for floating point unit

	ZChaff	SVRH
Maximal length	64 bit	128 bit
Average time per solution	200.5 s	0.97 s
Solution times with the largest ratio	2,861 s	0.3 s
Solution times with the smallest ratio	25 s	5.7 s
Quality of solution (extreme case)	0p0=0x43F0000000000000, 0p1=0xB180000000000000	0p0=0x070E342575271FFA, 0p1=0x9560F399ECF4E191
Reports unsatisfiable	YES	NO

Fig. 21 Comparison of stochastic search algorithm SVRH with ZChaff SAT solver for a floating point verification benchmark

Whenever available, analytic and semi-analytic methods beat all other methods in terms of runtimes by a few orders of magnitude. When we resort to search methods, stochastic local search solves the largest number of problems in the least time [32, 63]. Figure 21 shows such a comparison for a benchmark of 133 hard problems in which the operation is multiply, and constraints are a combination of range, number-of-ones, and mask types. Details of the experiment, including how we modeled the problem as a SAT and a CSP problem, were provided in [63]. We see that not only stochastic search is faster by two orders of magnitude than SAT in average, it is also able to deal with longer floating point numbers (ZChaff's memory exploded on 128-bit problems), and it provides better quality solutions (the trivial solution that ZChaff found and is shown in the figure is not very helpful from a verification point of view). However, unlike SAT and heuristic search methods, stochastic local search is incomplete and cannot report that an instance is unsatisfiable. This is a serious drawback because it is impossible for the verification engineer writing down the constraints to know in advance whether they are asking for too much in terms of the ability to satisfy their request. Finally, we note that we have not conducted recent experiments with newer SAT solvers. However, judging from the magnitude of the difference shown in Fig. 21, we do not expect such experiments to change the conclusions in any drastic way.

7 Conclusions

We have presented the vast and widespread application of stimuli generation for functional hardware verification. It is now widely accepted that the most potent solution technology for this application is constraint programming. While our presentation specifically highlighted the activities done at IBM, it should be recognized that all high-end hardware manufacturers, as well as EDA tool providers, use constraint programming in the most sophisticated ways to create stimuli for verification of hardware. Still, IBM rightfully claims the right of priority for this domain, having worked extensively in the field already in 1994 [5–7].

While the work in the field has been both long-standing and extensive, we can in no way consider it a complete work. Indeed, as time passes and hardware becomes even more complex, with heavy time-to-market pressure and highly demanding users, the challenges of verification become ever greater. Scientists and engineers working in the field feel the pressure to push the technological envelope on a weekly basis. This pressure translates directly into increasing challenges to the constraint solver at the core of the stimuli generation tools. More complex hardware designs require the ability to cope with ever-larger and more entangled CSPs. The requirement to create more tests in less time translates into the necessity to solve those CSPs faster. There are numerous challenges that still need to find their optimal solution. Some were described in Sect. 6 in detail. Others are still at the initial level of comprehension and can, at this stage, only be discussed orally. Either way, it is certain that another lively, challenging, and fruitful decade is waiting for anyone who will be working on the next-generation solutions in applying CP to stimuli generation.

Acknowledgments We are grateful to Amir Nahir, Gil Shurek, and Avi Ziv with whom we held extensive discussions. The material and book [12] for the Verification Course given by them at the Technion, Israel Institute of Technology, formed the basis for many of the ideas presented in Sect. 2. We also thank Eitan Marcus for his contribution to the sections related to checking and to Merav Aharoni, Sigal Asaf, and Yoav Katz for some of the figures in this chapter. The advancements in CP for verification presented here could not have been accomplished without the innovation, talent, and dedication of dozens of researchers and engineers at IBM Research – Haifa, and without the continuous feedback of verification engineers across IBM. The work of all those people is described and cited in many places in this chapter.

References

1. Naveh Y, Rimon M, Jaeger I, Katz Y, Vinov M, Marcus E, Shurek G (2007) Constraint-based random stimuli generation for hardware verification. AI Mag 28:13–30
2. Moss A (2007) Constraint patterns and search procedures for CP-based random test generation. In: Haifa verification conference, pp 86–103
3. Cadence web page (2007) Incisive Enterprise Specman Products. http://www.cadence.com/rl/Resources/datasheets/specman_elite_ds.pdf; We are not aware of an academic publication of Cadence's constraint solver
4. Iyer MA (2003) Race a word-level ATPG-based constraints solver system for smart random simulation. In: Proceedings of the international test conference, 2003, (ITC'03), pp 299–308
5. Chandra AK, Iyengar VS (1992) Constraint solving for test case generation: a technique for high-level design verification. In: Proceedings of IEEE international conference on computer design: VLSI in computers and processors, ICCD'92, pp 245–248
6. Lichtenstein Y, Malka Y, Aharon A (1994) Model based test generation for processor verification. In: Sixth annual conference on innovative applications of artificial intelligence, Menlo Park, USA, 1994. American association for artificial intelligence, pp 83–94
7. Lewin D, Foumier L, Levinger M, Roytman E, Shurek G (1995) Constraint satisfaction for test program generation. In: Proceedings of the IEEE fourteenth annual international phoenix conference on computers and communication, pp 45–48
8. Bin E, Emek R, Shurek G, Ziv A (2002) Using a constraint satisfaction formulation and solution techniques for random test program generation. IBM Syst J 41:386–402

9. Naveh Y, Rimon M, Jaeger I, Katz Y, Vinov M, Marcus E, Shurek G (2006) Constraint-based random stimuli generation for hardware verification. In: Innovative applications of artifical intellegence (IAAI'06).
10. Zhang J, Wang X (2001) A constraint solver and its application to path feasibility analysis. In: Proceedings of international journal of software engineering and knowledge engineering.
11. Godefroid P, Klarlund N, Sen K (2005) Dart: directed automated random testing. In: Proceedings of the ACM SIGPLAN conference on programming language design and implementation, PLDI'05, pp 213–223
12. Wile B, Goss JC, Roesner W (2005) Comprehensive functional verification - the complete industry cycle. Elsevier, UK
13. Kropf T (1999) Introduction to formal hardware verification. Springer, Berlin
14. Clarke EM, Grumberg O, Peled DA (1999) Model Checking. MIT Press, Cambridge
15. Ganai MK, Gupta A (2006) Accelerating high-level bounded model checking. In: Proceedings of the IEEE/ACM international conference on computer-aided design, ICCAD'06, pp 794–801
16. Armando A, Mantovani J, Platania L (2008) Bounded model checking of software using SMT solvers instead of SAT solvers. Int J Software Tool Tech Tran 11(1):69–83
17. Lavagno L, Martin G, Scheffer L (2006) Electronic design automation for integrated circuits handbook. CRC Press, Boca Raton
18. Adir A, Bin E, Peled O, Ziv A (2003) Piparazzi: a test program generator for micro-architecture flow verification. In: High-level design validation and test workshop, 2003. 8th IEEE International, pp 23–28
19. Aharoni M, Asaf S, Fournier L, Koifman A, Nagel R (2003) FPGen - a test generation framework for datapath floating-point verification. In: High-level design validation and test workshop. Eighth IEEE international, 2003, pp 17–22
20. Adir A, Almog E, Fournier L, Marcus E, Rimon M, Vinov M, Ziv A (2004) Genesys-Pro: innovations in test program generation for functional processor verification. IEEE Design and Test of Computers 21:84–93
21. Behm M, Ludden J, Lichtenstein Y, Rimon M, Vinov M (2004) Industrial experience with test generation languages for processor verification. In: Design automation conference, 2004. 41st Proceedings, pp 36–40
22. SystemVerilog web page (2009). http://www.systemverilog.org/
23. IEEE Standard(2008) Functional verification language e 3 Aug 2008
24. Hollander Y, Morley M, Noy A (2001) The e language: a fresh separation of concerns. In: Proceedings of technology of object-oriented languages and systems, TOOLS'01
25. van Hoeve WJ (2009) Over-constrained problems. In: Hybrid Optimization: The Ten years of CPAIOR – edited collection. In: The Ten Years of CPAIOR: A Success Story, Springer, pp 1–9
26. Freuder E (1996) In pursuit of the holy grail. ACM Comput Surv 63
27. Dechter R, Kask K, Bin E, Emek R (2002) Generating random solutions for constraint satisfaction problems. In: 18th national conference on artificial intelligence, Menlo Park, USA. American association for artificial intelligence, pp 15–21
28. Borning A, Freeman-Benson B, Willson M (1992) Constraint hierarchies. Lisp Symbol Comput 5:223–270
29. Mittal S, Falkenhainer B (1989) Dynamic constraint satisfaction problems. In: 8th national conference on artificial intelligence, Menlo Park, USA. American association for artificial intelligence, pp 25–32
30. Nahir A, Shiloach Y, Ziv A (2007) Using linear programming techniques for scheduling-based random test-case generation. In: Proceedings of the Haifa verification conference (HVC'06), pp 16–33
31. Adir A, Asaf S, Fournier L, Jaeger I, Peled O (2007) A framework for the validation of processor architecture compliance. In: Design automation conference, 2007, DAC '07. 44th ACM/IEEE, pp 902–905
32. Naveh Y (2008) Guiding stochastic search by dynamic learning of the problem topography. In: Perron L, Trick MA (eds) CPAIOR. Lecture notes in computer science, vol 5015. Springer, pp 349–354

33. Sabato S, Naveh Y (2007) Preprocessing expression-based constraint satisfaction problems for stochastic local search. In: Hentenryck PV, Wolsey LA (eds) CPAIOR. Lecture notes in computer science, vol 4510. Springer, pp 244–259
34. Cadence web page (2009) The new generation testcase utility. http://www.cadence.com/community/blogs/fv/archive/2009/01/08/the-new-generation-testcase-utility.aspx
35. CoWare web page (2009) Coware processor designer. http://www.coware.com/products/ processordesigner.php
36. Target web page (2009) http://www.retarget.com
37. Pees S, Hoffmann A, Zivojnovic V, Meyr H (1999) Lisa-machine description language for cycle-accurate models of programmable dsp architectures. In: Design automation conference, 1999. 36th Proceedings, pp 933–938
38. Zivojnovic V, Pees S, Meyr H (1996) Lisa-machine description language and generic machine model for hw/sw co-design. In: IEEE workshop on VLSI signal processing IX, pp 127–136
39. Tensilica web page (2009). http://www.tensilica.com
40. ARC web page (2009) ARC configurable CPU/DSP cores. http://www.arc.com/ configurablecores
41. Tensilica web page (2009) Create TIE processor extensions. http://www.tensilica.com/ products/xtensa/extensible/create_tie.htm
42. Rimon M, Lichtenstein Y, Adir A, Jaeger I, Vinov M, Johnson S, Jani D (2006) Addressing test generation challenges for configurable processor verification. In: High-level design validation and test workshop, 2006. Eleventh annual IEEE international, pp 95–101
43. Gutkovich B, Moss A (2006) CP with architectural state lookup for functional test generation. In: High-level design validation and test workshop, 2006. Eleventh annual IEEE international, pp 111–118
44. Adir A, Founder L, Katz Y, Koyfman A (2006) DeepTrans - extending the model-based approach to functional verification of address translation mechanisms. In: High-level design validation and test workshop, 2006. Eleventh annual IEEE international, pp 102–110
45. Geller F, Veksler M (2005) Assumption-based pruning in conditional CSP. In: van Beek P (ed) CP 2005. Lecture notes in computer science, vol 3709. Springer, pp 241–255
46. Chencinski EW, Check MA, DeCusatis C, Deng H, Grassi M, Gregg TA, Helms MM, Koenig AD, Mohr L, Pandey K, Schlipf T, Schober T, Ulrich H, Walters CR (2009) IBM system z10 I/O subsystem. IBM J Res Dev 53:1–13
47. Emek R, Jaeger I, Naveh Y, Bergman G, Aloni G, Katz Y, Farkash M, Dozoretz I, Goldin A (2002) X-Gen: A random test-case generator for systems and socs. In: 7th IEEE international high-level design validation and test workshop, HLDVT-02, pp 145–150
48. Abramovici M, Breuer M, Friedman A (1995) Digital systems testing and testable design. Wiley IEEE Press, New York
49. Hentenryck PV, Simonis H, Dincbas M (1992) Constraint satisfaction using constraint logic programming. In: Artificial intelligence, 58(1-3):113–159
50. Simonis H (1989) Test generation using the constraint logic programming language chip. In: Proceedings of the 6th international conference on logic programming (ICLP '89), pp 101–112
51. Brand S (2001) Sequential automatic test pattern generation by constraint programming. In: CP2001 post conference workshop modelling and problem formulation
52. Hartman A, Ur S, Ziv A (1999) Short vs long: Size does make a difference. In: Proceedings of the high-level design validation and test workshop, pp 23–28
53. Ellman T (1993) Abstraction by approximate symmetry. In: IJCAI'93: Proceedings of the 13th international joint conference on artificial intelligence, Morgan Kaufmann Publishers, San Francisco, 1993, pp 916–921
54. Adir A, Arbetman Y, Dubrov B, Liechtenstein Y, Rimon M, Vinov M, Calligaro M, Cofler A, Duffy G (2005) VLIW - a case study of parallelism verification. In: Design automation conference, 2005. 42nd Proceedings, pp 779–782
55. STMicroelectronics web page (2009) STMicroelectronics demonstrates complex multimedia application on VLIW micro core. http://www.st.com/stonline/press/news/year2002/t1124p.htm

56. Adve SV, Gharachorloo K (1996) Shared memory consistency models: A tutorial. In: Computer,pp 66–76
57. Lamport L (1979) How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans Comput C-28:690–691
58. Gibbons P, Korach E (1992) The complexity of sequential consistency. In: Parallel and distributed processing, 1992. Proceedings of the Fourth IEEE Symposium on 1–4 December 1992, pp 317–325
59. Cantin J, Lipasti M, Smith J (2005) The complexity of verifying memory coherence and consistency. In: IEEE transactions on parallel and distributed systems, pp 663–671
60. Gopalakrishnan G, Yang Y, Sivaraj H (2004) QB or not QB: an efficient execution verification tool for memory orderings. In: Lecture notes in computer science: computer aided verification, pp 401–413
61. Yang Y, Gopalakrishnan G, Lindstrom G, Slind K (2004) Nemos: a framework for axiomatic and executable specifications of memory consistency models. In: Parallel and distributed processing symposium, 2004. Eighteenth international proceedings, pp 26–30
62. Adir A, Attiya H, Shurek G (2003) Information-flow models for shared memory with an application to the powerpc architecture. In: IEEE transactions on parallel and distributed systems, vol 14(5), pp 502–515
63. Naveh Y (2004) Stochastic solver for constraint satisfaction problems with learning of high-level characteristics of the problem topography. In: First international workshop on local search techniques in constraint satisfaction, LSCS'04.