Laurent Perron
Michael A. Trick (Eds.)

Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems

LNCS 5015

5th International Conference, CPAIOR 2008
Paris, France, May 2008
Proceedings

Springer

Lecture Notes in Computer Science 5015

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Laurent Perron Michael A. Trick (Eds.)

Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems

5th International Conference, CPAIOR 2008
Paris, France, May 20-23, 2008
Proceedings

Springer

Volume Editors

Laurent Perron
ILOG
9, rue de Verdun
94253 Gentilly Cedex, France
E-mail: lperron@ilog.fr

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

Library of Congress Control Number: 2008926947

CR Subject Classification (1998): G.1.6, G.1, G.2.1, F.2.2, I.2, J.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

ISSN 0302-9743
ISBN-10 3-540-68154-X Springer Berlin Heidelberg New York
ISBN-13 978-3-540-68154-0 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12270859 06/3180 5 4 3 2 1 0

Preface

The 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2008) was held in Paris, France May 20–23, 2008.

The purpose of this conference series is to bring together researchers in the fields of constraint programming, artificial intelligence, and operations research to explore ways of solving large-scale, practical optimization problems through integration and hybridization of the fields' different techniques. Through the years, this research community is discovering that the fields have much in common, and there has been tremendous richness in the resulting cross-fertilization of fields.

This year, we allowed submissions of both long (15 page) and short (5 page) papers, with short papers either being original work, a reduced version of a long paper, or an extended abstract of work published elsewhere. We were not surprised by the 69 submissions in the long paper category: this is an active field with many researchers. We were surprised by the 61 short paper submissions. This was far more than predicted. With 130 high-quality submissions, competition for acceptance in this year's program was particularly fierce. In the end, we accepted 18 long papers and 22 short papers for presentation and publication in this volume.

In addition to the selected papers, there were three invited talks. Those speakers were Cindy Barnhart, Professor of Civil and Environmental Engineering at the Massachusetts Institute of Technology, Pascal Van Hentenryck, Professor of Computer Science at Brown University, and François Laburthe, Director of Operations Research and Innovation at Amadeus, a leading information technology firm in the travel industry.

On May 20, a Master Class was held, organized by Cindy Barnhart and Laurent Michel, Assistant Professor of Computer Science and Engineering at the University of Connecticut. The theme of the Master Class was "Modeling Practical Problems: The OR/CP Interface." The Master Class is intended for PhD students, researchers and practitioners.

Thursday afternoon was given over to three workshops: Open-Source Software for Integer and Constraint Programming, organized by Robin Lougee-Heimer of IBM Research and Ionut Aron, formerly of IBM Research, Bin Packing and Placement Constraints, organized by a group led by Nicolas Beldiceanu of EMN Nantes, and Constraint-Based Methods for Bioinformatics, organized by Agostino Dovier of the University of Udine.

This year, the conference organization was divided. While we handled the program, François Fages, Senior Research Scientist INRIA, joined Laurent in the conference organization. François and Laurent, with the help of the colloquium office at INRIA, were responsible for organizing the venue, finding sponsorship

funds, and the million other details that go into running a successful conference. François was also particularly helpful in assisting us with program policy decisions, and we are grateful for his thoughts and experience.

We would particularly like to thank the Program Committee for their efforts. No one expected 130 submissions, and they did a tremendous job of reading, reviewing, and commenting on papers in a timely and insightful fashion.

Finally, we would like to thank the sponsors who make this possible. These include the Association for Constraint Programming, INRIA, Microsoft Research/ INRIA Joint Center, National ICT Australia, ILOG, COSYTEC, Intelligent Information Systems Institute at Cornell, KLS OPTIM, Jeppesen Technology Services and the energy company Total.

May 2008

Laurent Perron
Michael Trick

Organization

Conference Chairs

François Fages, INRIA, France
Laurent Perron, ILOG, France

Program Chairs

Laurent Perron, ILOG, France
Michael Trick, Carnegie Mellon, USA

Master Class Chairs

Cynthia Barnhart, MIT, USA
Laurent Michel, University of Connecticut, USA

Program Committee

Ionut Aron, IBM Research, USA
Philippe Baptiste, École Polytechnique, France
Chris Beck, University of Toronto, Canada
Frédéric Benhamou, University of Nantes, France
Bob Bosch, Oberlin College, USA
Edmund Burke, University of Nottingham, UK
Amedeo Cesta, ISTC-CNR Rome, Italy
John Chinneck, Carleton University, Canada
Emilie Danna, ILOG, USA
Andrew Davenport, IBM, USA
Yves Deville, UCLouvain, Belgium
Hani El Sakkout, Cisco, USA
François Fages, INRIA, France
Bernard Gendron, University of Montreal, Canada
Carla Gomes, Cornell University, USA
Youssef Hamadi, Microsoft Research Cambridge, UK
John Hooker, Carnegie Mellon, USA
Kazuyoshi Inoue, NS Solutions, Japan
Narendra Jussien, École des Mines de Nantes, France
Thorsten Koch, ZIB, Germany
François Laburthe, Amadeus, France
Claude Le Pape, Schneider Electric, France
Janny Leung, Chinese University of Hong Kong, Hong Kong

Andrea Lodi, University of Bologna, Italy
Robin Lougee-Heimer, IBM Research, USA
Anuj Mehrotra, University of Miami, USA
Laurent Michel, University of Connecticut, USA
Michela Milano, University of Bologna, Italy
Yehuda Naveh, IBM Research, Israel
Barry O'Sullivan, University College Cork, Ireland
Gilles Pesant, Polytechnique Montreal, Canada
Jean-François Puget, ILOG, France
Jean-Charles Régin, ILOG, France
Louis-Martin Rousseau, Polytechnique Montreal, Canada
Michel Rueher, University of Nice, France
David Ryan, University of Auckland, New Zealand
Meinolf Sellmann, Brown University, USA
Helmut Simonis, University College Cork, Ireland
Stephen Smith, Carnegie Mellon, USA
Barbara Smith, University of Leeds, UK
Pascal Van Hentenryck, Brown University, USA
Willem-Jan Van Hoeve, Carnegie Mellon, USA
Alkis Vazacopoulos, Fair Isaac, USA
Mark Wallace, Monash University, Australia
Laurence Wolsey, UCLouvain, Belgium
Tallys Yunes, University of Miami, USA

External Reviewers

Tobias Achterberg
Timo Berthold
Simon Boivin
Lucas Bordeaux
Hélène Collavizza
Sophie Demassey
Bistra Dilkina
Ari Freund
Oded Fuhrmann
Jonathan Gaudreault
Carmen Gervet
Frédéric Goualard
Stefan Heinz
Dejan Jovanović
Arnaud Lallouet
Yahia Lebbah
Michele Lombardi
Xavier Lorca
Julien Martin
Jean-Noël Monette
Angelo Oddi
Claude-Guy Quimper
Yossi Richter
Aurélien Rizk
Frederic Saubion
Pierre Schaus
Andrew See
Gil Shurek
Peter Stuckey
Oliver Weide
Alessandro Zanarini

Table of Contents

Invited Talks

Long Papers

Short Papers

Extended Abstract

Airline Scheduling: Accomplishments, Opportunities and Challenges

Cynthia Barnhart

Massachusetts Institute of Technology

Airline scheduling is characterized by numerous complexities, including a network of flights, different aircraft types, limited numbers of gates, air traffic control restrictions, environmental regulations, strict safety requirements, a myriad of crew work rules and complicated payment structures, and competitive, dynamic environments in which passenger demands are uncertain and pricing strategies are complex. This, layered with the airline industry's endemic issues of low profitability, contentious labor issues, and outdated and inadequate infrastructure, poses daunting challenges that have intrigued operations researchers for at least a half-century, and have provided a fertile ground for the development and application of models and algorithms. In this talk, we first briefly summarize the optimization-based accomplishments in this area, highlighting the significant successes and impacts. While impressive, the problem is far from solved today. The focus of this talk, then, is on the many remaining opportunities and challenges, namely:

a) Robust scheduling: A trend in airline scheduling is to generate schedules that are "robust" to the disruptions that plague airline operations. Because airlines have typically constructed schedules with the assumption that every flight departs and arrives as planned, plans are frequently disrupted and airlines often incur significant additional costs beyond those originally planned. A more robust plan can reduce the occurrence and impact of these disruptions.
b) Dynamic scheduling: Stochasticity of passenger demands is a major challenge for the airlines in their quest to produce profit-maximizing schedules. Even using sophisticated optimization tools, many flights upon departure have empty seats, while others suffer a lack of seats to accommodate passengers who desire to travel. One approach to this challenge is to implement [illegible] approaches that re-optimize elements of the flight schedule during the passenger booking process, recognizing that demand forecast quality for a particular date improves as the date approaches.
c) Recovery from irregular operations: We describe approaches designed for use in near real-time mode to adjust operations in response to a variety of disruptions. We present briefly some of the market-based mechanisms being considered to address this problem, with a particular focus on minimizing disruption and delay to passengers.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, p. 1, 2008.

Selected Challenges from Distribution and Commerce in the Airline and Travel Industry

François Laburthe

Operations Research & Innovation, Amadeus

Distribution for the travel industry is about connecting providers (airlines, train companies, car rental companies, hoteliers ... who all have tickets to sell) to customers (business travelers & tourists). Such sales can be done directly or through a travel agent, both offline (during a discussion with an agent) or online (through a web site). Distribution for the travel industry has gone through impressive changes in the past decades with the advent of Global Distribution Systems in 80's, the rise sophisticated pricing & revenue management policies in the 90's and the growth of Internet in the past decade.

Though massive flows of money go through the distribution chain, most actors have small margins, and need efficient commercial policies to be profitable. As reservations are immaterial goods, information is key to business optimization, in order to match supply with demand, price products effectively, and support innovative offerings. In this talk, we will present several challenges for operations researchers & computer scientists related to commerce and distribution systems for the airline industry.

1. **Fare search.** These are the tools that power the web sites of online travel agencies and airlines websites, as well as travel agent desktops. They support the search for travel solutions between two cities at a given date. Such tools return sequences of flights with applicable fares. Whilst one could imagine that they are based on simple database requests, or on shortest paths computation, fare search turns out to be an incredibly combinatorial problem because of the sophisticated business rules governing the validity of a fare. The success of fare search products yields interesting challenges to the operations researcher such as:
   - How fast can one design a graph exploration method, where the pricing rules involve quantified formulas with negation (stating that a fare is applicable if there exists no other travel solution through another airport such that ...)?
   - How can a search domain be partitioned into independent search domains in order to process independent fare search sub-requests in parallel?
   - If one uses a cache to store prices derived from fare rules, how can one predict which are the prices needing to be recomputed after a change in pricing rules?
   - In the case of package search (where the traveler will book flights, hotel stay, car rental & possibly others in a single transaction), how can one index such combinations of products? How can they be efficiently searched for?

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 2–4, 2008.

2. **Inventory & availability management.** Airlines are increasingly moving towards complex commercial policies, with numerous fare classes potentially available on a given flight and where the decision of the actual availability of a fare depends on many characteristics of the request (end-to-end journey, channel through which the request is arriving, membership to loyalty program tier, ...). This trend towards dynamic pricing does stress the inventory management systems which now become solicited not only for performing actual bookings, but also for answering loads of availability requests from website shoppers. The increased look-to-book ratio (with evermore fare search transactions per actual booking) as well as the inability of legacy inventory systems to cope with such transaction volumes calls for much smarter global availability systems. Interesting challenges are:
   - If one caches the availability traffic related to booking attempts (traffic hitting the inventory system), can one reconstruct the availability logic in order to accurately mimic its logic and therefore protect the inventory from too heavy a traffic? In practice, how can one answer availability requests for all availability traffic (including, say, trips from JFK to Rome's FCO airport, by knowing the precise availability information from only a sample of the traffic, including JFK-CDG and LAX-FCO, but not JFK-FCO). The commercial model of the airline is of course proprietary, but the traffic can be sampled, and the availability model could be progressively estimated.
   - How can one predict, based on historical, the evolution of available fares? Is it possible to evaluate a market situation and advise customers on whether it is a good time to buy or not?
   - How can one build a global model of market price for air trips?
3. **Business management solutions for travel agents.** Travel agents, both off-line and on-line ones negotiate special fares & allotments with providers airlines. They make a living both of service fees, and mark-ups (a share of the price of the products they sell). They can optimize their revenues by selling in case of products with similar benefits to the traveler, the product that generates the most profit for them. Examples of challenges related to that issue include:
   - Forecasting the agency's own revenue, based on historical data & the current fee policy,
   - Simulating how the revenue would evolve in case the rules for computing service fees & mark-ups were to change,
   - Deciding what are the proper reachable levels for all providers in order to negotiate commissions based on volume targets
4. **Revenue management solutions for airlines.** The airline industry is probably the industry that for which many of the revenue management concepts have been introduced. Revenue management systems include a forecaster (estimating the future potential sales of tickets from now till the day of departure) and an optimizer (defining the appropriate inventory controls from the forecasted demand). Modern airline revenue management system include end-to-end availability logic (O&D logic) which states that an itinerary

should be available for sale if its yield to the airline is greater than the sum of the opportunity costs on each segment of the trip. Moreover, this network logic is often further tailored by means of fare modifiers which further restrict the availability in cases of hints that the request is originating from a business traveler. Scientific challenges include:

- Understand the real impact of O&D logic in the revenue management. What is the actual positive impact of O&D revenue management compared to simpler tools? Currently, all such questions are answered by simulation. Defining the appropriate simulation protocol to assess such methods is a difficult task.
- Define robust forecasting methods.
- Define incremental optimization methods, ie: methods that evaluate accurately and in real time the actual revenue impact of a ticket sale.

With such a research agenda, IT systems for the airline & travel industry, as they embed more and more operation research components, will continue to offer daunting challenges and tremedous opportunities and will inspire generations of scientists to come.

30 Years of Constraint Programming

P. Van Hentenryck

Brown University, Box 1910, Providence, RI 02912

Abstract. This talk reviews 30 years of constraint programming. It surveys computational and modeling progress, provides some historical perspectives on current research topics, explores some of the challenges faced by constraint-programming technology, and contrasts its development with mixed-integer programming. It also argues that the key strengths of constraint programming will be ubiquitous future optimization systems and describes some of the significant engineering steps to be taken for realizing this vision.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, p. 5, 2008.

Constraint Integer Programming: A New Approach to Integrate CP and MIP

Tobias Achterberg[1], Timo Berthold[2], Thorsten Koch[2], and Kati Wolter[2]

[1] ILOG Deutschland, Ober-Eschbacher Str. 109, 61352 Bad Homburg, Germany
tachterberg@ilog.de
[2] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
berthold,koch,wolter@zib.de

Abstract. This article introduces constraint integer programming (CIP), which is a novel way to combine constraint programming (CP) and mixed integer programming (MIP) methodologies. CIP is a generalization of MIP that supports the notion of general constraints as in CP. This approach is supported by the CIP framework SCIP, which also integrates techniques from SAT solving. SCIP is available in source code and free for non-commercial use.

We demonstrate the usefulness of CIP on two tasks. First, we apply the constraint integer programming approach to pure mixed integer programs. Computational experiments show that SCIP is almost competitive to current state-of-the-art commercial MIP solvers. Second, we employ the CIP framework to solve chip design verification problems, which involve some highly non-linear constraint types that are very hard to handle by pure MIP solvers. The CIP approach is very effective here: it can apply the full sophisticated MIP machinery to the linear part of the problem, while dealing with the non-linear constraints by employing constraint programming techniques.

1 Introduction

In the recent years, several authors showed that an integrated approach of *constraint programming (CP)* and *mixed integer programming (MIP)* can help to solve optimization problems that were intractable with either of the two methods alone [15,25,40]. Different approaches to integrate CP and MIP into a single framework have been proposed, [5,9,14,22,36,37] amongst others.

Most of the existing work follows the concept of augmenting a CP framework with basic MIP techniques, namely LP relaxations and sometimes cutting planes. In contrast, this paper introduces a way to incorporate CP specific solving methods and its strong modeling capability into the sophisticated MIP solving machinery. This is achieved by a low-level integration of the two concepts. The constraints of a CP usually interact through the domains of the variables. Similar to [9,14,36,37], the idea of *constraint integer programming (CIP)* is to offer a second communication interface, namely the LP relaxation. Furthermore, the definition of CIP restricts the generality of CP modeling as little as needed to still gain the full power of all primal and dual MIP solving techniques.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 6–20, 2008.

Therefore, CIP is well suited for problems that contain a MIP core complemented by some non-linear constraints. As an example for such a problem type, the property checking problem is presented in Section 5.

The concept of constraint integer programming is realized in the branch-and-cut framework SCIP. It combines solving techniques for CP, MIP, and *satisfiability problems (SAT)* such that all involved algorithms operate on a single search tree, which yields a very close interaction. A detailed description of the concepts and the software can be found in [2].

The plugins that are provided with the standard distribution of SCIP suffice to turn the CIP framework into a full-fledged MIP solver. In combination with either SoPlex [42] or CLP [17] as LP solver, it is the fastest non-commercial MIP solver that is currently available, see [32] and our results in Section 4. Using Cplex [23] as LP solver, the performance of SCIP is even comparable to the today's best commercial codes.

As a library, SCIP can be used to develop branch-cut-and-price algorithms, and it can be extended to support additional classes of non-linear constraints by providing so-called constraint handler plugins. We present a solver for the chip design verification problem as one example of this usage.

SCIP is freely available in source code for academic and non-commercial use and can be downloaded from `http://scip.zib.de`. The current version 1.00—as of this writing—has interfaces to five different LP solvers and consists of 223 178 lines of C code. The code is actively maintained and extended, and we hope to be able to make further improvements.

The article is organized as follows: in Section 2, we introduce constraint integer programs. Section 3 presents the building blocks of the constraint integer programming framework SCIP. In Sections 4 and 5, we demonstrate the usage of SCIP on two applications. First, we employ SCIP as a stand-alone MIP solver, and second, we use SCIP as a branch-and-cut framework to solve chip verification problems. Computational results are given in the Sections 4 and 5.4.

2 Constraint Integer Programs

Most solvers for CP, SAT, and MIP are based on dividing the problem into smaller subproblems and implicitly enumerating all potential solutions. Because MIP is a very specific case of CP, MIP solvers can apply sophisticated techniques that operate on the subproblem as a whole, for example solving the linear programming (LP) relaxation or generating cutting planes.

In contrast, due to the very general definition of CPs, CP solvers have to rely on constraint propagators, each of them exploiting the structure of a single constraint class. Usually, the only communication between the individual constraints takes place via the variables' domains. An advantage of CP is, however, the possibility to model the problem more directly, using very expressive constraints, which maintain the structure of the problem.

On the other hand, SAT is also a very specific case of CP with only one type of constraints, namely Boolean clauses. Such a clause can easily be linearized,

but the LP relaxation is rather useless, as it cannot detect the infeasibility of subproblems earlier than domain propagation. Therefore, SAT solvers mainly exploit the special problem structure to speed up the domain propagation algorithm.

The hope of integrating CP, SAT, and MIP techniques is to combine their advantages and to compensate for their individual weaknesses. We propose the following slight restriction of a CP, which allows the application of MIP solving techniques, to specify our integrated approach:

Definition. A *constraint integer program* CIP $= (\mathfrak{C}, I, c)$ consists of solving

$$\begin{aligned}(\text{CIP}) \quad c^\star = \min\{c^T x \mid \mathcal{C}_i(x) = 1 \text{ for all } i = 1,\dots,m,\\ x \in \mathbb{R}^n,\ x_j \in \mathbb{Z} \text{ for all } j \in I\}\end{aligned}$$

with a finite set $\mathfrak{C} = \{\mathcal{C}_1,\dots,\mathcal{C}_m\}$ of constraints $\mathcal{C}_i : \mathbb{R}^n \to \{0,1\}$, $i = 1,\dots,m$, a subset $I \subseteq N = \{1,\dots,n\}$ of the variable index set, and an objective function vector $c \in \mathbb{R}^n$. A CIP has to fulfill the following additional condition:

$$\forall \hat{x}_I \in \mathbb{Z}^I\ \exists (A', b') : \{x_C \in \mathbb{R}^C \mid \mathfrak{C}(\hat{x}_I, x_C)\} = \{x_C \in \mathbb{R}^C \mid A' x_C \le b'\} \qquad (1)$$

with $C := N \setminus I$, $A' \in \mathbb{R}^{k \times C}$, and $b' \in \mathbb{R}^k$ for some $k \in \mathbb{Z}_{\ge 0}$.

Restriction (1) ensures that the remaining subproblem after fixing all integer variables is always a linear program. This means that in the case of finite domain integer variables, the problem can be—in principle—completely solved by enumerating all values of the integer variables and then solving the corresponding LPs.

Note, that this does not forbid quadratic or even more involved expressions. Only the remaining part after fixing (and thus eliminating) the integer variables must be linear in the continuous variables. Furthermore, the linearity restriction of the objective function can be compensated by introducing an auxiliary objective variable z that is linked to the actual non-linear objective function with a constraint $z = f(x)$. Analogously, general variable domains can be represented as additional constraints.

Therefore, every CP that meets Condition (1) can be represented as a CIP. Especially, the following proposition holds.

Proposition. The notion of constraint integer programming generalizes finite domain constraint programming and mixed integer programming:

(a) Every CP with finite domains for all variables can be modeled as a CIP.
(b) Every MIP can be modeled as a CIP.

3 The SCIP Framework

SCIP is a framework for constraint integer programming. It is based on the branch-and-bound procedure, which is a very general and widely used method to solve optimization problems.

The idea of *branching* is to successively divide the given problem instance into smaller subproblems until the individual subproblems are easy to solve. The best of all solutions found in the subproblems yields the global optimum. During the course of the algorithm, a *branching tree* is created with each node representing one of the subproblems.

The intention of *bounding* is to avoid a complete enumeration of all potential solutions of the initial problem, which are usually exponentially many. If a subproblem's lower (dual) bound is greater than or equal to the global upper (primal) bound, the subproblem can be pruned. Lower bounds are calculated with the help of a relaxation which should be easy to solve. Upper bounds are found if the solution of the relaxation is also feasible for the corresponding subproblem.

Good lower and upper bounds must be available for the bounding to be effective. In order to improve a subproblem's lower bound, one can tighten its relaxation, e.g., via domain propagation or by adding cutting planes (see Sections 3.2 and 3.4, respectively). Primal heuristics, which are described in Section 3.5, contribute to the upper bound.

The selection of the next subproblem in the search tree and the branching decision have a major impact on how early good primal solutions can be found and how fast the lower bounds of the subproblems increase. More details on branching and node selection are given in Section 3.6.

SCIP provides all necessary infrastructure to implement branch-and-bound based algorithms for solving CIPs. It manages the branching tree along with all subproblem data, automatically updates the LP relaxation, and handles all necessary transformations due to presolving problem modifications, see Section 3.7. Additionally, a cut pool, cut filtering, and a SAT-like conflict analysis mechanism, see Section 3.3, are available. SCIP provides its own memory management and plenty of statistical output.

Besides the infrastructure, all main algorithms of SCIP are implemented as external plugins. In the remainder of this section, we will describe the most important types of plugins and their role for solving CIPs.

3.1 Constraint Handlers

Since a CIP consists of constraints, the central objects of SCIP are the *constraint handlers*. 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. This feasibility test suffices to turn SCIP into an algorithm which correctly solves CIPs with constraints of the supported types. To improve the performance of the solving process, constraint handlers may provide additional algorithms and information about their constraints to the framework, namely

- presolving methods to simplify the problem's representation,
- propagation methods to tighten the variables' domains,

– a linear relaxation, which can be generated in advance or on the fly, that strengthens the LP relaxation of the problem, and
– branching decisions to split the problem into smaller subproblems, using structural knowledge of the constraints in order to generate a well-balanced branching tree.

The distribution of SCIP includes the constraint handler for linear constraints that is needed to solve MIPs. Additionally, some specializations of linear constraints like knapsack, set partitioning, or variable bound constraints are supported by constraint handlers, which can exploit the special structure of these constraints in order to obtain more efficient data structures and algorithms.

3.2 Domain Propagation

Constraint propagation is an integral part of every CP solver [8]. The task is to analyze the set of constraints of the current subproblem and the current domains of the variables in order to infer additional valid constraints and domain reductions, thereby restricting the search space. The special case where only the domains of the variables are affected by the propagation process is called *domain propagation*. If the propagation only tightens the lower and upper bounds of the domains without introducing holes it is called *bound propagation*.

In mixed integer programming, the concept of bound propagation is well-known under the term *node preprocessing*. Usually, MIP solvers apply a restricted version of the preprocessing algorithm that is used before starting the branch-and-bound process to simplify the problem instance (see, e.g., [38] or [20]).

Besides the integrality restrictions, there is only one type of constraints in a MIP, namely the linear constraints. In contrast, CP models can include a large variety of constraint classes with different semantics and structure. Thus, a CP solver usually provides specialized constraint propagation algorithms for every single constraint class.

Constraint based (primal) 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 the *objective propagation* and the *root reduced cost strengthening* [33].

3.3 Conflict Analysis

Current state-of-the-art MIP solvers discard infeasible and bound-exceeding subproblems without paying further attention to them. Modern SAT solvers, in contrast, try to learn from infeasible subproblems, which is an idea due to Marques-Silva and Sakallah [31]. The infeasibilities are analyzed in order to generate so-called *conflict clauses*. These are implied clauses that help to prune the search tree. They also enable the solver to apply so-called *non-chronological backtracking*. A similar idea in CP are *no-goods*, see e.g., [39].

SCIP generalizes conflict analysis to CIP and, as a special case, to MIP. There are two main differences of CIP and SAT solving in the context of conflict analysis. First, the variables of a CIP do not need to be of binary type. Therefore,

we have to extend the concept of the conflict graph: it has to represent bound changes instead of variable fixings, see [1] for details.

Furthermore, the infeasibility of a subproblem in the CIP search tree usually has its reason in the LP relaxation of the subproblem. In this case, there is no single conflict-detecting constraint as in SAT or CP solving. To cope with this situation, we have to analyze the LP in order to identify a subset of the bound changes that suffices to render the LP infeasible or bound-exceeding. Note that it is an $\mathcal{NP}$-hard problem to identify a subset of the local bounds of *minimal cardinality* such that the LP stays infeasible if all other local bounds are removed. Therefore, we use a greedy heuristic approach based on an unbounded ray of the dual LP, see [1].

After having analyzed the LP, we proceed in the same fashion as SAT solvers: we construct a conflict graph, choose a cut in this graph, and produce a conflict constraint which consists of the bound changes along the frontier of this cut.

3.4 Cutting Plane Separators

Besides splitting the current subproblem Q into two or more easier subproblems by branching, one can also try to tighten the subproblem's relaxation in order to rule out the current solution $\check{x}$ and to obtain a different one. The LP relaxation can be tightened by introducing additional linear constraints $a^T x \leq b$ that are violated by the current LP solution $\check{x}$ but do not cut off feasible solutions from Q. Thus, the current solution $\check{x}$ is *separated* from the convex hull of integer solutions Q_I by the *cutting plane* $a^T x \leq b$, i.e., $\check{x} \notin \{x \in \mathbb{R} \mid a^T x \leq b\} \supseteq Q_I$.

The theory of cutting planes is very well covered in the literature. For an overview of computationally useful cutting plane techniques, see [20,30]. A recent survey of cutting plane literature can be found in [27].

SCIP features separators for knapsack cover cuts [10], complemented mixed integer rounding cuts [29], Gomory mixed integer cuts [21], strong Chvátal-Gomory cuts [28], flow cover cuts [35], implied bound cuts [38], and clique cuts [26,38]. Detailed descriptions of the cutting planes algorithms integrated into SCIP and an extensive analysis of their computational impact can be found in [41].

Almost as important as finding cutting planes is the selection of the cuts that actually should enter the LP relaxation. Balas, Ceria, and Cornuéjols [11] and Andreello, Caprara, and Fischetti [6] proposed to base the cut selection on *efficacy* and *orthogonality*. The efficacy is the Euclidean distance of the cut hyperplane to the current LP solution, and an orthogonality bound makes sure that the cuts added to the LP form an almost pairwise orthogonal set of hyperplanes. SCIP follows these suggestions.

3.5 Primal Heuristics

Primal heuristics have a significant relevance as supplementary procedures inside a MIP solver: they help to find good feasible solutions early in the search process, which helps to prune the search tree by bounding and allows to apply

more reduced cost fixing and other dual reductions that can tighten the problem formulation.

Overall, there are 23 heuristics integrated into SCIP. They can be roughly subclassified into four categories:

- *Rounding heuristics* try to iteratively round the fractional values of an LP solution in such a way that the feasibility for the constraints is maintained or recovered by further roundings.
- *Diving heuristics* iteratively round a variable with fractional LP value and resolve the LP, thereby simulating a depth first search (see Section 3.6) in the branch-and-bound tree.
- *Objective diving heuristics* are similar to diving heuristics, but instead of fixing the variables by changing their bounds, they perform "soft fixings" by modifying their objective coefficients.
- *Improvement heuristics* consider one or more primal feasible solutions that have been previously found and try to construct an improved solution with better objective value.

Detailed descriptions of the primal heuristics implemented in SCIP and an in-depth analysis of their computational impact can be found in [12], an overview is given in [13].

3.6 Node Selection and Branching Rules

Two of the most important decisions in a branch-and-bound algorithm are the selection of the next subproblem to process (*node selection*) and how to split the current problem Q into smaller subproblems (*branching rule*).

The most popular branching strategy in MIP solving is to split the domain of an integer variable x_j, $j \in I$, with fractional LP value $\check{x}_j \notin \mathbb{Z}$ into two parts, thus creating two subproblems $Q_1 = Q \cap \{x_j \leq \lfloor \check{x}_j \rfloor\}$ and $Q_2 = Q \cap \{x_j \geq \lceil \check{x}_j \rceil\}$. Methods to select such a fractional variable for branching are discussed in [2,3].

SCIP implements most of the discussed branching rules, especially *reliability branching* which is currently the most effective general branching rule for MIP. Using SCIP, it is possible to implement arbitrary branching schemes such as branchings that create more than two subproblems or branching on constraints.

SCIP offers several node selection strategies as default plugins. *Depth first search* always chooses a child of the current node as the next subproblem to be processed or backtracks to the most recent ancestor with an unprocessed child, if the current node has been pruned. Depth first search is the preferred strategy for pure feasibility problems like SAT. Additionally, it has the benefit that successively solved subproblems are very similar, which reduces the subproblem management overhead.

Best first search aims at improving the global dual bound as fast as possible by always selecting a subproblem with the smallest dual bound of all remaining leaves in the tree. Best first search leads to a minimal number of nodes that need to be processed, given that the branching rule is fixed [1].

Best Estimate search was suggested by Forrest et al. [19]. It estimates the minimum value of a rounded solution in each subproblem and chooses a node with minimal estimate. The aim is to quickly find good feasible solutions. However, this node selection strategy may perform very poor in improving the global dual bound.

The default node selection strategy of SCIP is a combination of these three strategies: it performs depth first search for a few consecutive subproblems after which a node with best estimate is chosen. At a certain frequency, a node with smallest dual bound is selected instead of a node with best estimate.

3.7 Presolving

Presolving is a way to transform the given problem instance into an equivalent instance that is (hopefully) easier to solve. The most fundamental presolving concepts for MIP are described in [38]. For additional information, see [20].

The task of presolving is threefold: first, it reduces the size of the model by removing irrelevant information such as redundant constraints or fixed variables. Second, it strengthens the LP relaxation of the model by exploiting integrality information, e.g., to tighten the bounds of the variables or to improve coefficients in the constraints. Third, it extracts information such as implications or cliques from the model which can later be used, for example for branching or cutting plane separation. SCIP implements a full set of *primal* and *dual* presolving reductions for MIP problems, see [1].

Restarts differ from the classical presolving methods in that they are not applied *before* the branch-and-bound search commences, but abort a running search process in order to reapply other presolving mechanisms and start the search from scratch. They are a well-known ingredient of modern SAT solvers, but have not been used so far for solving MIPs.

It is often the case that cutting planes, strong branching [7], and reduced cost strengthening in the root node identify fixings of variables that have not been detected during presolving. These fixings can trigger additional presolve reductions after a restart, thereby simplifying the problem instance and improving its LP relaxation. The downside is that we have to solve the root LP relaxation again, which can sometimes be very expensive.

Nevertheless, the above observation leads to the idea of applying a restart directly after the root node processing if a certain fraction of the integer variables has been fixed during the processing of the root node. In our implementation, a restart is performed if at least 5 % of the integer variables have been fixed.

4 SCIP as a MIP Solver

With the default plugins that are included in the distribution, SCIP can be used as a stand-alone MIP solver. Some of the plugins have been described in Section 3. In this section we evaluate the performance of SCIP for solving MIPs.

We tested SCIP 1.00 running on a 3.00 GHz Intel Xenon with 8 GB RAM and 4 MB cache, using Cplex 11.0 [23] as underlying LP solver. We set a time

Table 1. Results of four MIP solvers on the MIPLIB 2003. If a solver hit one of the limits, we report the primal-dual gap in percent instead of the solving time in seconds.

Name	SCIP /CPLEX Nodes	SCIP /CPLEX Time	CPLEX Nodes	CPLEX Time	SCIP /SOPLEX Nodes	SCIP /SOPLEX Time	CBC /CLP Nodes	CBC /CLP Time
10teams	671	20.3	1	0.4	564	77.7	190	24.9
aflow30a	2353	13.5	3054	7.9	4293	35.6	30577	79.0
air04	334	98.9	263	8.2	159	189.7	565	172.1
air05	384	49.4	467	7.3	314	134.6	548	95.4
cap6000	3455	4.1	4227	0.7	2647	6.4	3390	7.1
disctom	1	85.4	1	6.0	1	64.4	1	4.2
fiber	24	1.1	60	0.2	12	1.3	40	2.2
fixnet6	26	1.6	71	0.6	10	2.8	114	3.4
gesa2-o	108	6.5	482	0.8	155	11.1	5695	32.6
gesa2	132	5.7	147	0.2	251	7.4	275	6.7
manna81	2	5.5	1	0.1	1	5.7	1	0.7
mas74	3275993	783.9	2673089	281.8	3036576	1582.8	4887385	2390.2
mas76	349635	73.4	398167	37.4	313718	118.0	687061	180.3
misc07	19719	15.2	25645	20.2	19831	27.7	29130	64.1
mod011	1751	76.8	54	20.7	2034	636.2	6318	132.4
modglob	21	0.9	183	0.1	3573	50.1	12664	26.3
nw04	457	92.7	283	29.2	49	369.5	22	12.5
p2756	45	2.6	11	0.2	109	3.3	37	1.4
pk1	219292	71.9	186390	81.7	226525	165.5	204094	81.8
pp08a	139	1.3	567	0.4	199	2.5	5087	31.3
pp08aCUTS	77	1.1	1102	1.1	109	2.6	5928	26.5
qiu	12653	76.9	7233	29.3	12973	337.5	31866	295.2
rout	11967	15.3	5260	8.8	10991	36.2	1011908	2219.9
vpm2	297	0.9	1619	0.4	1077	2.2	459	4.3
aflow40b	347845	2067.6	491380	2342.5	427125	2.2 %	1321287	4.0 %
danoint	1158489	4856.1	778939	4975.1	330296	3.5 %	683171	2.0 %
fast0507	1350	395.2	2941	555.0	1380	2407.0	7770	1.6 %
glass4	7335667	79.6 %	8939059	6595.8	322356	125.0 %	1729411	95.8 %
harp2	22481616	<0.1 %	316170	144.8	5732001	0.1 %	2589310	3448.6
mzzv11	3376	547.6	498	90.8	1545	0.6 %	2899	4.8 %
mzzv42z	761	302.9	298	33.5	1369	5243.8	5500	3.9 %
net12	5501	2139.0	2603	28.3 %	1411	—	12191	22.3 %
noswot	1510640	6110.8	8158083	4.7 %	495596	238.4	5713896	2.8 %
opt1217	3833790	16.3 %	1	0.1	3558191	16.6 %	20584953	17.7 %
set1ch	27	1.4	330	0.2	8825	18.9	1317890	0.5 %
tr12-30	909033	2600.7	212451	294.2	1259733	4433.7	506441	1.3 %
Geom. Mean	4101	58.0	2455	11.3	4224	136.5	12609	183.8
Solved Instances		33		34		29		25
$\geq 10\,\%$ faster		–		27		2		5
$\geq 10\,\%$ slower		–		6		30		29

limit of 2 hours and a memory limit of 4 GB. As a comparison we applied the same test with CPLEX 11.0 as stand-alone MIP solver, with SCIP 1.00 using SOPLEX 1.3.2 [42] to solve the LPs, and CBC 2.0 with CLP 1.6 [17] as LP solver. We used the provided default settings for all solvers. As test set we chose the 60 instances of the MIPLIB 2003 [4]. We left out the instances `arki001`, `protfold`, and `timtab1` for which at least one of the solvers returned a wrong answer or reported an error.

Tables 1 and 2 compare the results of the four solvers. The first part of Table 1 lists the instances which were solved to optimality by all solvers, the second part those which were solved by at least one solver, Table 2 those for which all solvers reached a limit. For each instance listed in the "Name" column, the tables show

Table 2. Results of four MIP solvers on the MIPLIB 2003 (continued). For the `markshare` instances we report the upper bound instead of the primal-dual gap; the lower bound is zero in all cases.

	SCIP /CPLEX		CPLEX		SCIP /SOPLEX		CBC /CLP	
Name	Nodes	Gap	Nodes	Gap	Nodes	Gap	Nodes	Gap
a1c1s1	426057	15.8 %	491631	5.7 %	115512	20.7 %	143591	41.0 %
atlanta-ip	11342	5.5 %	4011	8.1 %	10	—	350	—
dano3mip	9911	22.8 %	5565	18.8 %	123	24.1 %	12898	30.5 %
ds	4512	486.6 %	5760	314.2 %	310	511.3 %	456	1482.5 %
liu	3146152	135.4 %	319976	102.1 %	347383	159.3 %	157480	206.4 %
mkc	2396228	1.3 %	140170	0.2 %	1022181	0.9 %	961565	2.5 %
momentum1	6221	20.5 %	23623	18.7 %	1276	—	5158	20.2 %
momentum2	6004	28.7 %	6144	28.7 %	1260	—	5529	152.4 %
momentum3	11	—	140	466.5 %	1	—	1	—
msc98-ip	10301	0.7 %	1996	12.1 %	67	—	324	—
nsrand-ipx	592996	6.5 %	234970	1.1 %	381553	8.8 %	661104	2.0 %
rd-rplusc-21	84288	>10 000 %	35562	>10 000 %	71	—	11795	—
roll3000	1180987	0.6 %	1253352	0.4 %	201728	1.2 %	133378	3.8 %
seymour	103485	2.2 %	146297	1.9 %	2829	11.5 %	33374	5.9 %
sp97ar	86939	3.4 %	210446	0.8 %	36063	4.6 %	180426	2.5 %
stp3d	8	—	20	—	3	—	1	—
swath	429024	19.1 %	262088	19.3 %	257953	26.8 %	2352638	40.7 %
t1717	2665	50.2 %	64721	60.4 %	898	37.0 %	13016	76.9 %
timtab2	3095502	78.4 %	1736172	52.5 %	2420114	63.1 %	639547	102.8 %
markshare1	46 M	5	31 M	4	52 M	6	42 M	6
markshare2	42 M	9	25 M	12	40 M	9	48 M	10

the number of nodes and the time in seconds needed to solve it with each of the four solvers. For instances which could not be solved within the time and memory limit, we report the primal-dual gap in percent instead of the solving time. The primal-dual gap is defined as $\gamma = (\hat{c} - \check{c})/\inf[\check{c}, \hat{c}]$ with $\hat{c}$ being the upper (primal) and $\check{c}$ being the lower (dual) bound. The symbol "—" indicates instances for which no feasible solution was obtained within the limits.

There were 36 instances, given in Table 1, for which at least one solver was able to prove optimality within the time and memory limit. For these instances, the results are summarized at the bottom of the table. The rows "$\geq 10\,\%$ faster" and "$\geq 10\,\%$ slower" give the number of instances for which the solver was at least 10 % faster and at least 10 % slower, respectively, than SCIP-CPLEX. Although SCIP supports the much more general concept of constraint integer programming, it is still competitive to state-of-the-art MIP solvers. On this test set, SCIP-CPLEX can solve only one instance less than CPLEX within the limits.

5 Using SCIP for Property Checking

One of the key technologies in the design of integrated circuits is the verification of the correctness of the design [24]. One important aspect of this process is the so-called *property checking problem*, which means to verify that certain expected inherent properties of the chip design hold.

Today's techniques validate these properties on the so-called *gate level* by transforming the properties into Boolean clauses and hence the property checking

problem into a SAT instance. However, complex arithmetic operations like multiplication lead to SAT instances with quite involved interrelationships between the variables, which are hard to solve for current SAT solvers.

Our approach is to tackle the problem on a higher level, the *register transfer (RT) level*. The property checking problem at RT level can be formulated as CIP on bit vector variables $\varrho \in \{0, \dots, 2^{w_\varrho - 1}\}$ of width w_ϱ. The constraints $r^i = C_i(x^i, y^i, z^i)$ model the circuit operations.

For each bit vector variable ϱ, we introduce single bit variables ϱ_b, $b = 0, \dots, w_\varrho - 1$, with $\varrho_b \in \{0, 1\}$, for which *linking constraints*

$$\varrho = \sum_{b=0}^{w_\varrho - 1} 2^b \varrho_b \tag{2}$$

define their correlation. In addition, we consider the following circuit operations: ADD, AND, CONCAT, EQ, ITE, LT, MINUS, MULT, NOT, OR, READ, SHL, SHR, SIGNEXT, SLICE, SUB, UAND, UOR, UXOR, WRITE, XOR, ZEROEXT with the semantics as defined in [16].

5.1 CP Techniques

For the bit linking constraints (2) and for each type of circuit operation we implemented a specialized constraint handler which includes a domain propagation algorithm that exploits the special structure of the constraint class. In addition to considering the current domains of the bit vectors ϱ and the bit variables ϱ_b, we exploit knowledge about the global equality or inequality of bit vectors or bits, which is obtained in the preprocessing stage of the algorithm.

Some of the domain propagation algorithms are very complex. For example, the domain propagation of the MULT constraint uses term algebra techniques to recognize certain deductions inside its internal representation of a partial product and overflow addition network. Others, like the algorithms for SHL, SLICE, READ, and WRITE, involve reasoning that mixes bit- and word-level information.

5.2 IP Techniques

Because property checking is a pure feasibility problem, there is no natural objective function. However, the LP relaxation usually detects the infeasibility of the local subproblem much earlier than domain propagation.

Table 3 shows the linearizations of the circuit operation constraints that are used in addition to the bit linking constraints (2) to construct the LP relaxation of the problem instance. Very large coefficients like 2^{w_r} in the ADD linearization can lead to numerical difficulties in the LP relaxation. Therefore, we split the bit vector variables into words of $W = 16$ bits and apply the linearization to the individual words. The linkage between the words is established in a proper fashion. For example, the overflow bit of a word in an addition is added to the right hand side of the next word's linearization. The relaxation of the MULT

Table 3. LP relaxation of circuit operations. l_ϱ and u_ϱ are the lower and upper bounds of a bit vector variable ϱ.

Operation	Linearization
r = AND (x,y)	$r_b \le x_b,\ r_b \le y_b,\ r_b \ge x_b + y_b - 1$
r = OR (x,y)	$r_b \ge x_b,\ r_b \ge y_b,\ r_b \le x_b + y_b$
r = XOR (x,y)	$x_b - y_b - r_b \le 0,\ -x_b + y_b - r_b \le 0,$ $-x_b - y_b + r_b \le 0,\ x_b + y_b + r_b \le 2$
r = UAND (x)	$r \le x_b,\ r \ge \sum_{b=0}^{w_x-1} x_b - w_x + 1$
r = UOR (x)	$r \ge x_b,\ r \le \sum_{b=0}^{w_x-1} x_b$
r = UXOR (x)	$r + \sum_{b=0}^{w_x-1} x_b = 2s,\ s \in \mathbb{Z}_{\ge 0}$
r = EQ (x,y)	$x - y = s - t,\ p + q + r = 1,\ p \le s,\ s \le p(u_x - l_y),$ $q \le t,\ t \le q(u_y - l_x),\ s,t \in \mathbb{Z}_{\ge 0},\ p,q \in \{0,1\}$
r = LT (x,y)	$x - y = s - t,\ p \le s,\ s \le p(u_x - l_y),\ r \le t,$ $t \le r(u_y - l_x),\ p + r \le 1,\ s,t \in \mathbb{Z}_{\ge 0},\ p \in \{0,1\}$
r = ITE (x,y,z)	$r - y \le (u_z - l_y)(1 - x),\ r - y \ge (l_z - u_y)(1 - x)$ $r - z \le (u_y - l_z)\,x,\ r - z \ge (l_y - u_z)\,x$
r = ADD (x,y)	$r + 2^{w_r} o = x + y,\ o \in \{0,1\}$
r = MULT (x,y)	$v_{bn} \le u_{y_n} x_b,\ v_{bn} \le y_n,\ v_{bn} \ge y_n - u_{y_n}(1 - x_b),\ v_{bn} \in \mathbb{Z}_{\ge 0}$ $o_n + \sum_{i+j=n} \sum_{l=0}^{L-1} 2^l v_{iL+l,j} = 2^L o_{n+1} + r_n,\ o_n \in \mathbb{Z}_{\ge 0}$

constraint involves additional variables y_n and r_n which are "nibbles" of y and r with $L = \frac{W}{2}$ bits.

No linearization is generated for the SHL, SLICE, READ, and WRITE constraints. Their linearizations are very complex and would dramatically increase the size of the LP relaxation, thereby reducing the solvability of the LPs. For example, a straight-forward linearization of the SHL constraint on a 64-bit input vector x that uses internal ITE-blocks for the potential values of the shifting operand y already requires 30 944 inequalities and 20 929 auxiliary variables.

5.3 SAT Techniques

Conflict Analysis is particular useful on feasibility problems like property checking. By applying reverse propagation, one or more conflict constraints can be extracted from the conflict graph of an infeasible subproblem. In our implementation, we use the *1-FUIP* [43] rule for generating conflict constraints. In addition to the *1-FUIP* conflict constraints we extract clauses from *reconvergence cuts* [43] in the conflict graph to support *non-chronological backtracking* [31].

5.4 Computational Results

We examined the computational effectiveness of the described CIP techniques on industrial benchmarks obtained from verification projects conducted together with INFINEON and ONESPIN SOLUTIONS. The specific chip verification algorithms were incorporated into SCIP 0.90i using CPLEX 10.0.1 [23] as LP solver. All calculations were performed on a 3.8 GHz Pentium-4 workstation with 2 GB

Table 4. ALU properties (time in seconds)

Prop	Meth	register width 5	10	15	20	25	30	35	40
muls	SAT	0.5	—	—	—	—	—	—	—
	CIP	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.3
neg_flag	SAT	0.1	100.0	—	—	—	—	—	—
	CIP	0.8	3.6	11.6	36.3	81.8	136.6	218.4	383.5
zero_flag	SAT	0.0	0.0	0.1	0.1	0.2	0.4	0.5	0.6
	CIP	2.3	0.6	1.6	4.0	6.2	10.7	15.6	379.7

Table 5. Biquad properties

Property	Meth	variant A	B	C
g_checkgpre	SAT	22.2	57.6	29.1
	CIP	14.2	12.3	15.3
g2_checkg2	SAT	—	—	—
	CIP	213.9	204.8	257.6
g25_checkg25	SAT	0.0	2.4	2.5
	CIP	29.7	22.4	24.2
g3_negres	SAT	0.0	0.0	0.0
	CIP	0.7	0.0	0.0
gBIG_checkreg1	SAT	287.2	157.3	159.6
	CIP	170.0	7.0	8.6

Table 6. Multiplier properties (time in seconds)

Layout	Meth	register width 6	7	8	9	10	11	12	13	14
booth	SAT	0.4	3.3	21.0	135.4	935.1	—	—	—	—
signed	CIP	21.3	70.1	318.7	384.2	904.1	1756.2	2883.7	4995.9	3377.9
booth	SAT	0.5	2.5	17.9	102.9	879.0	4360.4	—	—	—
unsgnd	CIP	15.7	51.7	269.1	911.3	1047.6	2117.7	2295.1	4403.4	7116.8
nonbth	SAT	0.4	3.4	21.8	134.1	1344.1	—	—	—	—
signed	CIP	12.8	31.2	100.6	265.9	569.8	690.8	1873.0	1976.3	4308.9
nonbth	SAT	0.3	1.8	16.5	83.1	909.6	5621.5	—	—	—
unsgnd	CIP	3.6	22.4	111.2	214.0	335.4	1040.1	1507.5	2347.7	4500.2

RAM. In all runs, we used a time limit of 2 hours. For reasons of comparison, we also solved the instances with SAT techniques on the gate level. We used MiniSat 2.0 [18] to solve the SAT instances obtained after a preprocessing step.

The experiments were conducted on the valid properties included in the following sets of property checking instances: *ALU* (an arithmetical logical unit which performs ADD, SUB, SHL, SHR, and signed and unsigned MULT operations), *Biquad* (a DSP/IIR filter core obtained from [34] in different representations), and *Multiplier* (gate level net lists for Booth and non-Booth encoded architectures of signed and unsigned multipliers).

Tables 4–6 compare the results of MiniSat and our CIP approach on the valid properties. For each property or layout and each input register width or variant, the tables show the time in seconds of the two algorithms needed to solve the instance. Results marked with '—' could not be solved within the time limit. The experiments show that our approach outperforms SAT techniques for proving the validity of properties on circuits containing arithmetics. For invalid properties, which are not shown in the tables, our algorithm is usually inferior to SAT for finding counter-examples. This is due to the much more involved procedures employed in the CIP approach.

References

1. Achterberg, T.: Conflict analysis in mixed integer programming. Discrete Optimization 4(1), 4–20 (2007) (Special issue: Mixed Integer Programming)
2. Achterberg, T.: Constraint Integer Programming. PhD thesis, Technische Universität Berlin (2007), http://opus.kobv.de/tuberlin/volltexte/2007/1611/
3. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Operations Research Letters 33, 42–54 (2005)
4. Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Operations Research Letters 34(4), 1–12 (2006), http://miplib.zib.de
5. Althaus, E., Bockmayr, A., Elf, M., Jünger, M., Kasper, T., Mehlhorn, K.: SCIL – symbolic constraints in integer linear programming. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 75–87. Springer, Heidelberg (2002)
6. Andreello, G., Caprara, A., Fischetti, M.: Embedding $\{0, \frac{1}{2}\}$-cuts in a branch-and-cut framework: A computational study. INFORMS Journal on Computing 19(2), 229–238 (2007)
7. Applegate, D.L., Bixby, R.E., Chvátal, V., Cook, W.J.: The Traveling Salesman Problem. Princeton University Press, Princeton (2006)
8. Apt, K.R.: Principles of Constraint Programming. Cambridge University Press, Cambridge (2003)
9. Aron, I.D., Hooker, J.N., Yunes, T.H.: SIMPL: A system for integrating optimization techniques. In: Régin, J.-C., Rueher, M. (eds.) CPAIOR 2004. LNCS, vol. 3011, pp. 21–36. Springer, Heidelberg (2004)
10. Balas, E.: Facets of the knapsack polytope. Mathematical Programming 8, 146–164 (1975)
11. Balas, E., Ceria, S., Cornuéjols, G.: Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. Management Science 42, 1229–1246 (1996)
12. Berthold, T.: Primal heuristics for mixed integer programs. Master's thesis, Technische Universität Berlin (2006)
13. Berthold, T.: Heuristics of the branch-cut-and-price-framework SCIP. ZIB-Report 07-30, Zuse Institute Berlin, Operations Research 2007 (to appear, 2007)
14. Bockmayr, A., Kasper, T.: Branch-and-infer: A unifying framework for integer and finite domain constraint programming. INFORMS Journal on Computing 10(3), 287–300 (1998)
15. Bockmayr, A., Pisaruk, N.: Solving assembly line balancing problems by combining IP and CP. In: Sixth Annual Workshop of the ERCIM Working Group on Constraints (June 2001)
16. Brinkmann, R., Drechsler, R.: RTL-datapath verification using integer linear programming. In: Proceedings of the IEEE VLSI Design Conference, pp. 741–746 (2002)
17. Computational infrastructure for operations research, http://www.coin-or.org
18. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
19. Forrest, J.J., Hirst, J.P.H., Tomlin, J.A.: Practical solution of large scale mixed integer programming problems with UMPIRE. Management Science 20(5), 736–773 (1974)
20. Fügenschuh, A., Martin, A.: Computational integer programming and cutting planes. In: Aardal, K., Nemhauser, G.L., Weismantel, R. (eds.) Discrete Optimization. Handbooks in Operations Research and Management Science, ch. 2, vol. 12, pp. 69–122. Elsevier, Amsterdam (2005)

21. Gomory, R.E.: Solving linear programming problems in integers. In: Bellman, R., Hall, J.M. (eds.) Combinatorial Analysis, Symposia in Applied Mathematics X, pp. 211–215, Providence, RI, American Mathematical Society (1960)
22. Hooker, J.N., Osorio, M.A.: Mixed logical/linear programming. Discrete Applied Mathematics 96-97(1), 395–442 (1999)
23. ILOG CPLEX. Reference Manual, http://www.ilog.com/products/cplex
24. International technology roadmap for semiconductors (2005), http://public.itrs.net
25. Jain, V., Grossmann, I.E.: Algorithms for hybrid MILP/CP models for a class of optimization problems. INFORMS Journal on Computing 13(4), 258–276 (2001)
26. Johnson, E.L., Padberg, M.W.: Degree-two inequalities, clique facets, and biperfect graphs. Annals of Discrete Mathematics 16, 169–187 (1982)
27. Klar, A.: Cutting planes in mixed integer programming. Master's thesis, Technische Universität Berlin (2006)
28. Letchford, A.N., Lodi, A.: Strengthening Chvátal-Gomory cuts and Gomory fractional cuts. Operations Research Letters 30(2), 74–82 (2002)
29. Marchand, H.: 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 (1998)
30. Marchand, H., Martin, A., Weismantel, R., Wolsey, L.A.: Cutting planes in integer and mixed integer programming. Discrete Applied Mathematics 123/124, 391–440 (2002)
31. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions of Computers 48, 506–521 (1999)
32. Mittelmann, H.: Decision tree for optimization software: Benchmarks for optimization software, http://plato.asu.edu/bench.html
33. Nemhauser, G.L., Wolsey, L.A.: Integer and Combinatorial Optimization. John Wiley & Sons, Chichester (1988)
34. Opencores, http://www.opencores.org
35. Padberg, M.W., van Roy, T.J., Wolsey, L.A.: Valid inequalities for fixed charge problems. Operations Research 33(4), 842–861 (1985)
36. Refalo, P.: Tight cooperation and its application in piecewise linear optimization. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 375–389. Springer, Heidelberg (1999)
37. Rodosek, R., Wallace, M.G., Hajian, M.T.: A new approach to integrating mixed integer programming and constraint logic programming. Annals of Operations Research 86(1), 63–87 (1999)
38. Savelsbergh, M.W.P.: Preprocessing and probing techniques for mixed integer programming problems. ORSA Journal on Computing 6, 445–454 (1994)
39. Stallman, R.M., Sussman, G.J.: Forward reasoning and dependency directed backtracking in a system for computer-aided circuit analysis. Artificial Intelligence 9, 135–196 (1977)
40. Timpe, C.: Solving planning and scheduling problems with combined integer and constraint programming. OR Spectrum 24(4), 431–448 (2002)
41. Wolter, K.: Implementation of cutting plane separators for mixed integer programs. Master's thesis, Technische Universität Berlin (2006)
42. Wunderling, R.: Paralleler und objektorientierter Simplex-Algorithmus. PhD thesis, Technische Universität Berlin (1996)
43. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in a boolean satisfiability solver. In: ICCAD, pp. 279–285 (2001)

New Filtering for the *cumulative* Constraint in the Context of Non-Overlapping Rectangles

Nicolas Beldiceanu[1], Mats Carlsson[2], and Emmanuel Poder[1]

[1] École des Mines de Nantes, LINA UMR CNRS 6241, FR-44307 Nantes, France
{Nicolas.Beldiceanu,Emmanuel.Poder}@emn.fr
[2] SICS, P.O. Box 1263, SE-164 29 Kista, Sweden
Mats.Carlsson@sics.se

Abstract. This paper describes new filtering methods for the *cumulative* constraint. The first method introduces bounds for the so called *longest cumulative hole* problem and shows how to use these bounds in the context of the *non-overlapping* constraint. The second method introduces *balancing knapsack constraints* which relate the total height of the tasks that end at a specific time-point with the total height of the tasks that start at the same time-point. Experiments on tight rectangle packing problems show that these methods drastically reduce both the time and the number of backtracks for finding all solutions as well as for finding the first solution. For example, we found without backtracking all solutions to 66 perfect square instances of order 23-25 and sizes ranging from 332×332 to 661×661.

1 Introduction

The utility of *cumulative* constraints in the context of non-overlapping rectangles has been advocated for 15 years in the context of constraint programming [1]. The two main reasons for this utility are: first, it allows to come up with necessary conditions for *non-overlapping* which reuse classical filtering algorithms for *cumulative* like task intervals [2,3] and compulsory parts [4]; second, it reduces in practice the combinatorial aspect by dividing by a factor of two the number of decision variables of the problem.[1] More recently, *cumulative* constraints have been used by OR people [7] in the context of rectangle packing problems for the reasons we have just mentioned. Knapsack constraints were also used, by both OR [8,7] and CP [9] people, to solve the subset-sum problem in the context of scheduling and packing.

In the context of tight rectangle placement problems one can observe that standard filtering methods for the *cumulative* constraint are in fact rather weak. A first reason is that they do not explicitly completely integrate the slack (i.e., the difference between the available place and the total area of the rectangles to place) within the filtering process. A second reason is that they relax too much the *cumulative* constraint by allowing to split the tasks in small squares of size one. Based on these observations, we decided to

[1] Experiments have shown [1,5] that, once all coordinates of the rectangles in one dimension are fixed, it is usually straightforward to extend the partial solution to a full solution even if there exist examples [6] where this is not possible at all.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 21–35, 2008.

develop new filtering methods that consider the slack and/or the fact that tasks should not be split in too many small pieces.

The paper is organized as follows. Section 2 recalls the definitions of the *cumulative* and the *non-overlapping* constraints. Section 3 introduces the longest cumulative hole problem, shows its use in the context of the *cumulative* constraint, and provides bounds for this problem. Section 4 presents a new knapsack model of the *cumulative* constraint which considers the available slack. Section 5 evaluates the contribution of the two methods on two types of tight placement problems. Finally, Section 6 concludes.

2 Background

The *cumulative* constraint was introduced in [1] in order to model scheduling problems where one has to deal with a resource of limited capacity. It has the following definition:

$$cumulative(T, L)$$

where for a task $t \in T$, $t.\mathrm{s}$, $t.\mathrm{d}$ and $t.\mathrm{h}$ denote respectively its start, duration and height. They all correspond to integer variables[2], while L is a non-negative integer corresponding to the capacity of the resource. The constraint holds if the following condition is true:

$$\forall i \in \mathbb{N}, \sum_{t|t.\mathrm{s} \leq i \leq t.\mathrm{s}+t.\mathrm{d}-1} t.\mathrm{h} \leq L$$

In the context of a *cumulative* constraint, the *compulsory part* [4] of a task t is the intersection of all feasible instances of t. It can be computed by making the intersection between the task positioned at its earliest start and the task positioned at its latest start. Then the *compulsory part profile* is the aggregation of all compulsory parts of the different tasks of a *cumulative* constraint. When all tasks that have a non-empty compulsory part are completely fixed, the compulsory part profile is simply called the *cumulative profile*.

The *diffn* constraint was introduced in [10] in order to handle multi-dimensional placement problems. It has the following definition:

$$diffn(B)$$

where for a box $b \in B$, $b.\mathrm{o}_k$ and $b.\mathrm{s}_k$ $(0 \leq k \leq n-1)$ are integer variables that respectively denote the origin and size of b in dimension k. The constraint holds when, for each pair of boxes b, b', there exist at least one dimension k where their projections do not overlap.

$$\forall b, b' \in B\ (b \neq b'), \exists k \in [0, n-1] \mid b.\mathrm{o}_k \geq b'.\mathrm{o}_k + b'.\mathrm{s}_k \vee b'.\mathrm{o}_k \geq b.\mathrm{o}_k + b.\mathrm{s}_k$$

In the context of this paper we focus on the two-dimensional case ($n = 2$), and assume that all the rectangle sizes are fixed. However note that most of the results of this paper can be used when we have more than two dimensions, as it is actually the case for our current implementation.

[2] An *integer variable* V ranges over a finite set of integers denoted by $\mathcal{D}(V)$. The extremal values in $\mathcal{D}(V)$ are denoted by $\underline{V}$ and $\overline{V}$.

3 The Longest Cumulative Hole Problem

This section first introduces the *longest cumulative hole problem* and then shows how it can be used in the context of a non-overlapping constraint. Finally, it provides different ways for evaluating an upper bound of the longest cumulative hole.

3.1 Defining the Longest Cumulative Hole

Given a $cumulative(T, L)$ constraint that involves a set of tasks T and a resource limit L, let σ denote the difference between the available space and the needed space (i.e., $\sigma = (\max_{t \in T}(\overline{t.\mathrm{s}} + t.\mathrm{d}) - \min_{t \in T}(\underline{t.\mathrm{s}})) \cdot L - \sum_{t \in T} t.\mathrm{d} \cdot t.\mathrm{h}$). Now, given an integer $\epsilon \in [1, L]$ and the subset of tasks T' of T for which the resource consumption is at most ϵ, the *longest hole problem*[3] is to find the largest integer $lmax_{\sigma}^{\epsilon}(T')$ such that there exist a cumulative placement of maximum height ϵ involving a subset of tasks of T' where, on one interval $[i, i + lmax_{\sigma}^{\epsilon}(T') - 1]$ of the cumulative profile, the area of the empty space does not exceed σ.[4]

Example 1. First, consider seven tasks of respective size 11×11, 9×9, 8×8, 7×7, 6×6, 4×4, 2×2. Part (A) of Figure 1 provides a cumulative profile corresponding to the longest hole problem according to $\epsilon = 11$ and $\sigma = 0$. The longest hole $lmax_{0}^{11}(\{11 \times 11, 9 \times 9, 8 \times 8, 7 \times 7, 6 \times 6, 4 \times 4, 2 \times 2\}) = 17$ since:

- The task 8×8 can not contribute since a gap of 3 cannot be filled by the unique candidate the task 2×2.
- The task 6×6 can also not contribute since a gap of 5 cannot be completely filled by the candidates 4×4 and 2×2.

Second, consider a task of size 3×2. Part (B) of Figure 1 provides a cumulative profile corresponding to the longest hole problem according to $\epsilon = 11$ and $\sigma = 20$. The longest hole $lmax_{20}^{11}(\{3 \times 2\}) = 2$. □

Note that when the gap ϵ is equal to the resource capacity L, the problem of checking whether or not a $cumulative$ constraint has a solution coincides with the longest hole problem so the longest hole problem is clearly NP-hard. Consequently, our goal is to find upper bounds for the longest hole problem as well as easy cases which can be solved in polynomial time.

3.2 Using the Longest Cumulative Hole for Filtering

The main advantage of the longest cumulative hole problem is that it can be used in quite a number of different ways in the context of a two-dimensional non-overlapping constraint, where the slack σ is the difference between the available and the needed space:

[3] A related problem when the slack σ is equal to 0 in the context of rectangles non-overlapping (but not in the context of a *cumulative* constraint) is called the *length of the longest flat surface* in `http://www.stetson.edu/%7Efriedma/mathmagic/1099.html`.

[4] When the set of tasks T is empty we have that $lmax_{\sigma}^{\epsilon}(T) = \lfloor \frac{\sigma}{\epsilon} \rfloor$.

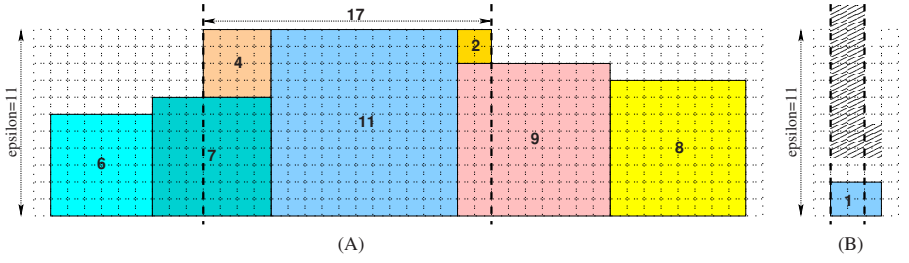


Fig. 1. Two examples for illustrating the longest hole problem

- First, as depicted by Part (A) of Figure 2, it can be used for making an initial pruning of the origin coordinates of the rectangles in order to avoid creating too small holes that cannot be filled enough, with respect to the slack σ, between the border of a rectangle $R1$ and the border of the placement space. For instance, Part (A) illustrates the fact that if, for a given distance $\epsilon \in \mathbb{N}$ between the lower border of a rectangle to place and the lower border of the placement space, the quantity $lmax^{\epsilon}_{\sigma}(\mathcal{R})$ is strictly less than the width of $R1$, then $R1$ cannot start at the corresponding position. $\mathcal{R}$ corresponds to the set of rectangles for which the height does not exceed ϵ (i.e., the rectangles that can fit within the gap). Finally, doing an initial pruning of the origins of the rectangles is important for the knapsack constraints that will be presented in the next section.
- Second, while fixing both origin coordinates of a rectangle $R1$ during the search, it can also be used to check that the vis-à-vis between $R1$ and each rectangle that is already completely fixed[5] can be filled enough with respect to σ. This is illustrated by Part (B) of Figure 2.
- Finally, it can also be directly used within the two *cumulative* constraints, which are well-known necessary conditions for a non-overlapping constraint. For this purpose, consider the highest peak of the compulsory part profile that does not reach the resource capacity (i.e., the difference between the resource capacity and the height of the peak is equal to a strictly positive integer ϵ). Again, we can use the longest cumulative hole problem in order to check that we can fill enough the gap on top of the highest peak. This is illustrated by Part (C) of Figure 2.

3.3 Evaluating the Longest Cumulative Hole

This section shows how to evaluate an upper bound of $lmax^{\epsilon}_{\sigma}(T)$. It assumes that we already know:

- An upper bound of $lmax^{e}_{\sigma}(T)$ for all non-negative integers e that are strictly less than ϵ.

[5] Two fixed rectangles have a *vis-à-vis* if and only if (1) they intersect in one dimension, and (2) if there is a non-empty gap between them in the other dimension.

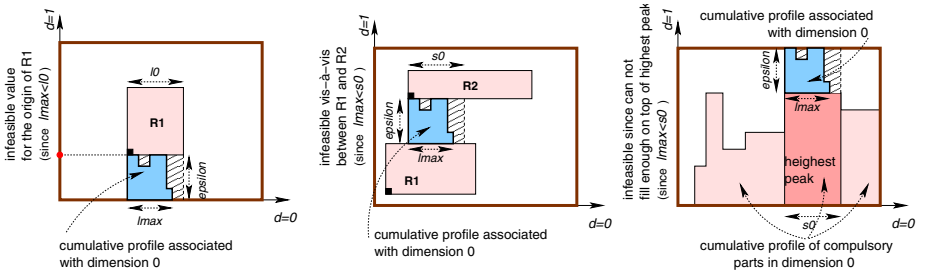


Fig. 2. Three ways of using the longest cumulative hole for filtering a two-dimensional non-overlapping constraint

– An upper bound of $lmax^{e}_{\sigma}(T \setminus \{t\})$ for all non-negative integers e that are strictly less than ϵ and for all $t \in T$ for which $t.\mathrm{h} \leq \epsilon$.[6] This quantity will be used for checking what can be put on top of a task t without reusing t.

We first present three rules that simplify the problem by removing some tasks, one rule that reduces the length of some tasks[7], and a rule that computes an exact value of $lmax^{e}_{\sigma}(T)$ when a specific condition on the heights of the tasks holds. Finally, we present two upper bounds of $lmax^{e}_{\sigma}(T)$ and show that they are incomparable. In the following, a task t of length $t.\mathrm{d}$ and height $t.\mathrm{h}$ will be denoted by $t.\mathrm{d} \times t.\mathrm{h}$.

Simplification 1. Let t be a task of T such that $t.\mathrm{h} > \epsilon$. We have that $lmax^{\epsilon}_{\sigma}(T) = lmax^{\epsilon}_{\sigma}(T \setminus \{t\})$.

Proof. By definition of the longest hole problem, a task of height strictly greater than ϵ cannot be used. □

Example 2. Consider the set of tasks $T = \{2 \times 2, 4 \times 4, 6 \times 6\}$ and assume that we want to compute $lmax^{1}_{3}(T)$. Using Simplification 1, we have that $lmax^{1}_{3}(T) = lmax^{1}_{3}(\emptyset)$, which means that we can only use the slack of 3 to cover a gap of height 1. Consequently, $lmax^{1}_{3}(T) = 3$. □

Simplification 2. Let T_ϵ denote the set of tasks of T for which the heights are equal to ϵ. We have that $lmax^{\epsilon}_{\sigma}(T) = \sum_{t \in T_\epsilon} t.\mathrm{d} + lmax^{\epsilon}_{\sigma}(T \setminus T_\epsilon)$.

Proof. Since the tasks of T_ϵ completely fill the height ϵ, they can be considered separately. □

Example 3. Consider the set of tasks $T = \{2 \times 2, 4 \times 4, 6 \times 6\}$ and assume that we want to compute $lmax^{6}_{0}(T)$. Using Simplification 2, we have that $lmax^{6}_{0}(T) = 6 + lmax^{6}_{0}(T \setminus \{6 \times 6\})$. □

[6] If we don't want to explicitly evaluate an upper bound of $lmax^{e}_{\sigma}(T \setminus \{t\})$, we can take advantage of the fact that $lmax^{e}_{\sigma}(T)$ is an upper bound of $lmax^{e}_{\sigma}(T \setminus \{t\})$.

[7] If, as we will see later, a task cannot contribute on its full length to the longest cumulative hole.

Simplification 3. Let t be a task of T such that $t.\mathrm{h} < \epsilon$ and $lmax_{\sigma}^{\epsilon - t.\mathrm{h}}(T \setminus \{t\}) = 0$. We have that $lmax_{\sigma}^{\epsilon}(T) = lmax_{\sigma}^{\epsilon}(T \setminus \{t\})$.

Proof. When $lmax_{\sigma}^{\epsilon - t.\mathrm{h}}(T \setminus \{t\})$ is equal to 0, this means that no gap of height $\epsilon - t.\mathrm{h}$ can be filled by the tasks of $T \setminus \{t\}$ without creating an empty space greater than the slack σ. Consequently, if we use task t, we cannot fill enough any gap on top of task t.□

Example 4. Consider the set of tasks $T = \{2 \times 2, 3 \times 3\}$ and assume that we want to compute $lmax_0^3(T)$. Assume that we already know that $lmax_0^1(T \setminus \{2 \times 2\}) = 0$. Then, we have that $lmax_0^3(T) = lmax_0^3(T \setminus \{2 \times 2\})$. In other words, we can eliminate task 2×2, since we cannot cover any gap of height 1 on top of task 2×2. □

Shrinking 1. Consider a task t of T such that $t.\mathrm{h} < \epsilon$, $t.\mathrm{d} > lmax_{\sigma}^{\epsilon - t.\mathrm{h}}(T \setminus \{t\})$ and $lmax_{\sigma}^{\epsilon - t.\mathrm{h}}(T \setminus \{t\}) > 0$. We have that $lmax_{\sigma}^{\epsilon}(T) \leq lmax_{\sigma}^{\epsilon}(T \setminus \{t\} \cup \{lmax_{\sigma}^{\epsilon - t.\mathrm{h}}(T \setminus \{t\}) \times t.\mathrm{h}\})$.

Proof. Similar to Simplification 3. We have an inequality since reducing the lengths of more than two disjunctive tasks (i.e., two tasks for which the total height is strictly greater than ϵ) can lead to an overestimation of $lmax_{\sigma}^{\epsilon}(T)$. This stems from the fact that at most two disjunctive tasks can be reduced (and the other disjunctive tasks have to be discarded since they would have to be completely included within the interval corresponding to the longest cumulative hole).- □

Example 5. Consider the set of tasks $T = \{2 \times 2, 4 \times 4, 6 \times 6\}$ and assume that we want to compute $lmax_0^6(T)$. Suppose we already know that $lmax_0^2(T \setminus \{4 \times 4\}) = 2$. Then we have that $lmax_0^6(T) = lmax_0^6((T \setminus \{4 \times 4\}) \cup \{2 \times 4\})$. In other words, the length of task 4×4 is reduced to 2 (i.e., its maximum intersection in time with the longest cumulative hole cannot exceed 2) since, for a gap of 2, we can cover at most a length of 2 without exceeding the slack $\sigma = 0$. □

In the following, all simplification and shrinking rules previously presented are systematically tried before applying the next rule and before evaluating any upper bound.

Termination rule. Given a set of tasks $T = \{t_1.\mathrm{d} \times t_1.\mathrm{h}, t_2.\mathrm{d} \times t_2.\mathrm{h}, \ldots, t_n.\mathrm{d} \times t_n.\mathrm{h}\}$ such that $t_i.\mathrm{h} \geq t_{i+1}.\mathrm{h}$ and $t_i.\mathrm{h} = t_{i+1}.\mathrm{h} \Rightarrow t_i.\mathrm{d} \geq t_{i+1}.\mathrm{d}$ $(1 \leq i < n)$, let $T_{disj} = \{t_1.\mathrm{d} \times t_1.\mathrm{h}, t_2.\mathrm{d} \times t_2.\mathrm{h}, \ldots, t_{ndisj}.\mathrm{d} \times, t_{ndisj}.\mathrm{h}\}$, where $ndisj$ is the largest integer that satisfies $ndisj = 1$ or $t_{ndisj-1}.\mathrm{h} + t_{ndisj}.\mathrm{h} > \epsilon$, be the non-empty subset of disjunctive tasks of T. If the total height of the tasks in $T \setminus T_{disj}$ plus the maximum height of the tasks in T_{disj} (i.e., $t_1.\mathrm{h}$) is at most ϵ, then the quantity $lmax_{\sigma}^{\epsilon}(T)$ can be directly evaluated by using the construction depicted by Figure 3.[8]

The intuition of the first upper bound is to consider the total area of the tasks as well as the slack. However, to get a sharper bound we take into account the fact that at most two disjunctive tasks can partially overlap a given interval.

[8] Assuming that the tasks were sorted, a direct algorithm implementing this construction has the complexity of $O(n)$ where n is the number of tasks.

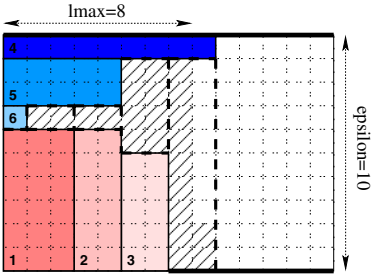


Fig. 3. Illustration of the easy case on six tasks 3×6, 2×6, 2×5, 9×1, 5×2 and 1×1 with $\epsilon = 10$ and a slack σ of 23. Tasks $\{3 \times 6, 2 \times 6, 2 \times 5\}$ correspond to disjunctive tasks, while tasks $\{9 \times 1, 5 \times 2, 1 \times 1\}$ correspond to small tasks that can be put on top of the disjunctive tasks. Then the slack $\sigma = 23$ is positioned as early as possible between the disjunctive tasks and the small tasks, which gives a value of 8 for $lmax_{23}^{10}(\{3 \times 6, 2 \times 6, 2 \times 5, 9 \times 1, 5 \times 2, 1 \times 1\})$.

Upper bound 1. Given a set of tasks $T = \{t_1.\mathrm{d} \times t_1.\mathrm{h}, t_2.\mathrm{d} \times t_2.\mathrm{h}, \ldots, t_n.\mathrm{d} \times t_n.\mathrm{h}\}$ such that $t_i.\mathrm{h} \geq t_{i+1}.\mathrm{h}$ and $t_i.\mathrm{h} = t_{i+1}.\mathrm{h} \Rightarrow t_i.\mathrm{d} \geq t_{i+1}.\mathrm{d}$ $(1 \leq i < n)$, let $T_{disj} = \{t_1.\mathrm{d} \times t_1.\mathrm{h}, t_2.\mathrm{d} \times t_2.\mathrm{h}, \ldots, t_{ndisj}.\mathrm{d} \times t_{ndisj}.\mathrm{h}\}$, where $ndisj$ is the largest integer that satisfies $ndisj = 1$ or $t_{ndisj-1}.\mathrm{h} + t_{ndisj}.\mathrm{h} > \epsilon$, be the non-empty subset of tasks of T. Moreover, let $T'_{disj} \subseteq T_{disj}$ be the subset of tasks of T_{disj} for which the lengths were reduced by rule Shrinking 1. If T'_{disj} contains more than two tasks then let $area\_max_1$ and $area\_max_2$ respectively denote the two largest areas of the tasks of T'_{disj}, and let $area\_rest$ denote the total area of the tasks in $T \setminus T'_{disj}$. We have that $lmax_\sigma^\epsilon(T) \leq \lfloor \frac{area\_rest + area\_max_1 + area\_max_2 + \sigma}{\epsilon} \rfloor$.

Proof. Given a fixed interval $[low, up]$ and a set of disjunctive tasks T_{disj}, at most two tasks of T_{disj} can partially overlap interval $[low, up]$. Note that disjunctive tasks for which the lengths were reduced cannot be completely included within interval $[low, up]$ (i.e., they either overlap one border of interval $[low, up]$, or they don't overlap at all interval $[low, up]$). Consequently, if we reason in terms of areas, we can only consider the two largest areas of the disjunctive tasks for which the lengths were reduced. □

Example 6. Figure 4 illustrates the computation of the first upper bound. From the set of tasks T we can construct the set $T_{disj} = \{2 \times 5, 2 \times 5, 2 \times 5, 2 \times 5, 1 \times 4, 3 \times 3\}$ of disjunctive tasks, since for any pair of tasks in T_{disj} we have that their total height exceeds $\epsilon = 6$. By hypothesis, $T'_{disj} = \{2 \times 5, 2 \times 5, 2 \times 5, 2 \times 5\}$ and $area\_max_1 = area\_max_2 = 10$. Finally, the total area of the tasks in $T \setminus T'_{disj}$, $area\_rest$, is equal to $4 + 9 + 4 + 1 = 18$. Consequently, $lmax_3^6(T) \leq \lfloor \frac{18+10+10+3}{6} \rfloor = 6$. □

The intuition of the next upper bound is not to reason any more just in terms of area, but to take into account the fact that disjunctive tasks cannot be piled up. We sort the disjunctive tasks by decreasing height and try now to reduce their length according to the tasks (and the slack) that can be effectively placed on top of the disjunctive tasks.

Upper bound 2. Given a set of tasks $T = \{t_1.\mathrm{d} \times t_1.\mathrm{h}, t_2.\mathrm{d} \times t_2.\mathrm{h}, \ldots, t_n.\mathrm{d} \times t_n.\mathrm{h}\}$ such that $t_i.\mathrm{h} \geq t_{i+1}.\mathrm{h}$ and $t_i.\mathrm{h} = t_{i+1}.\mathrm{h} \Rightarrow t_i.\mathrm{d} \geq t_{i+1}.\mathrm{d}$ $(1 \leq i < n)$, let $T_{disj} =$

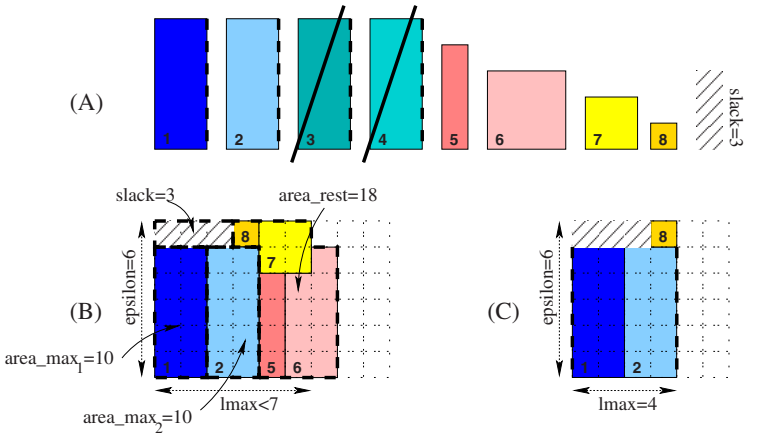


Fig. 4. (B) Illustration of the first upper bound on eight tasks (A) $T = \{2 \times 5, 2 \times 5, 2 \times 5, 2 \times 5, 1 \times 4, 3 \times 3, 2 \times 2, 1 \times 1\}$, where the length of the first four tasks was reduced by applying rule Shrinking 1 (this reduction is depicted by a dashed line along the right border of a task), with $\epsilon = 6$ and a slack σ of 3. (C) A placement giving the exact value for $lmax_3^6(T)$. Note that a task for which the length was reduced can only be put at one of the two extremities of the placement; consequently, we cannot add task 5 and task 7 to gain an extra unit for $lmax_3^6(T)$ (since the lengths of tasks 1 and 2 were reduced, tasks 1 and 2 have to be kept at one of the two extremities).

$\{t_1.\mathrm{d} \times t_1.\mathrm{h}, t_2.\mathrm{d} \times t_2.\mathrm{h}, \ldots, t_{ndisj}.\mathrm{d} \times t_{ndisj}.\mathrm{h}\}$, where $ndisj$ is the largest integer that satisfies $ndisj = 1$ or $t_{ndisj-1}.\mathrm{h} + t_{ndisj}.\mathrm{h} > \epsilon$), be the non-empty subset of tasks of T. Let $t_1, t_2, \ldots, t_{ndisj}$ denote the tasks of T_{disj} sorted by decreasing height and for any h let $area_h$ denote the total area of the tasks of T that have a height less than or equal to h. If there is an $i \in [1, |T_{disj}|]$ such that:

- $\forall j \in [1, i-1], \sum_{k=1}^{j} t_k.\mathrm{d} \cdot t_k.\mathrm{h} + area_{\epsilon - t_j.\mathrm{h}} + \sigma \geq \sum_{k=1}^{j} t_k.\mathrm{d} \cdot \epsilon$,
- $\sum_{k=1}^{i} t_k.\mathrm{d} \cdot t_k.\mathrm{h} + area_{\epsilon - t_i.\mathrm{h}} + \sigma < \sum_{k=1}^{i} t_k.\mathrm{d} \cdot \epsilon$,

then the length of task t_i can be reduced to $\lfloor \frac{area_{\epsilon - t_i.\mathrm{h}} + \sigma - \sum_{k=1}^{i-1} t_k.\mathrm{d} \cdot (\epsilon - t_k.\mathrm{h})}{\epsilon - t_i.\mathrm{h}} \rfloor$.

Now let T'_{disj} be the set of tasks derived from T_{disj} by considering their reduced length and by discarding the tasks for which the reduced length is equal to 0. Let $area = \sum_{t \in T - T'_{disj}} t.\mathrm{d} \cdot t.\mathrm{h} + \sigma$ and let $t'_1, t'_2, \ldots, t'_{|T'_{disj}|}, t'_{|T'_{disj}|+1}$ denote the tasks of T'_{disj} sorted by decreasing height, where $t'_{|T'_{disj}|+1}$ stands for an additional task of height 0 and length $\lceil \frac{area}{\epsilon} \rceil$. In this context, let i be the smallest integer such that $\sum_{k=1}^{i} t'_k.\mathrm{d} \cdot (\epsilon - t'_k.\mathrm{h}) \geq area$. We have that $lmax_\sigma^\epsilon(T) \leq \sum_{k=1}^{i-1} t'_k.\mathrm{d} + \lfloor \frac{area - \sum_{k=1}^{i-1} t'_k.\mathrm{d} \cdot t'_k.\mathrm{h}}{\epsilon - t'_i.\mathrm{h}} \rfloor$.

Example 7. Figure 5 provides an example of application of the second upper bound on a set of tasks $T = \{3 \times 5, 2 \times 4, 2 \times 4, 5 \times 3, 3 \times 3, 2 \times 2, 1 \times 1\}$ under the hypothesis that we have a slack $\sigma = 3$ and a gap $\epsilon = 6$. The set of disjunctive tasks T_{disj} built from these rectangles is $\{3 \times 5, 2 \times 4, 2 \times 4, 5 \times 3\}$. The length of task t_3 (i.e., $i = 3$) can be reduced since:

- $[j = 1]$: $t_1.\mathrm{d} \cdot t_1.\mathrm{h} + area_{6-5} + \sigma = 3 \cdot 5 + 1 + 3 = 19 \geq 3 \cdot 6$,
- $[j = 2]$: $t_1.\mathrm{d} \cdot t_1.\mathrm{h} + t_2.\mathrm{d} \cdot t_2.\mathrm{h} + area_{6-4} + \sigma = 3 \cdot 5 + 2 \cdot 4 + 5 + 3 = 31 \geq (3+2) \cdot 6$,
- $[i = 3]$: $t_1.\mathrm{d} \cdot t_1.\mathrm{h} + t_2.\mathrm{d} \cdot t_2.\mathrm{h} + t_3.\mathrm{d} \cdot t_3.\mathrm{h} + area_{6-4} + \sigma = 3 \cdot 5 + 2 \cdot 4 + 2 \cdot 4 + 5 + 3 = 39 < (3 + 2 + 2) \cdot 6$.

The length of task $t_3 = 3 \times 4$ is reduced to $\lfloor \frac{area_{\epsilon - t_3.\mathrm{h}} + \sigma - t_1.\mathrm{d} \cdot (\epsilon - t_1.\mathrm{h}) - t_2.\mathrm{d} \cdot (\epsilon - t_2.\mathrm{h})}{\epsilon - t_3.\mathrm{h}} \rfloor = \lfloor \frac{5+3-3 \cdot (6-5)-2 \cdot (6-4)}{6-4} \rfloor = 0$. Consequently, $lmax_3^6(T) = 8$ (instead of 9 if t_3 is not removed). □

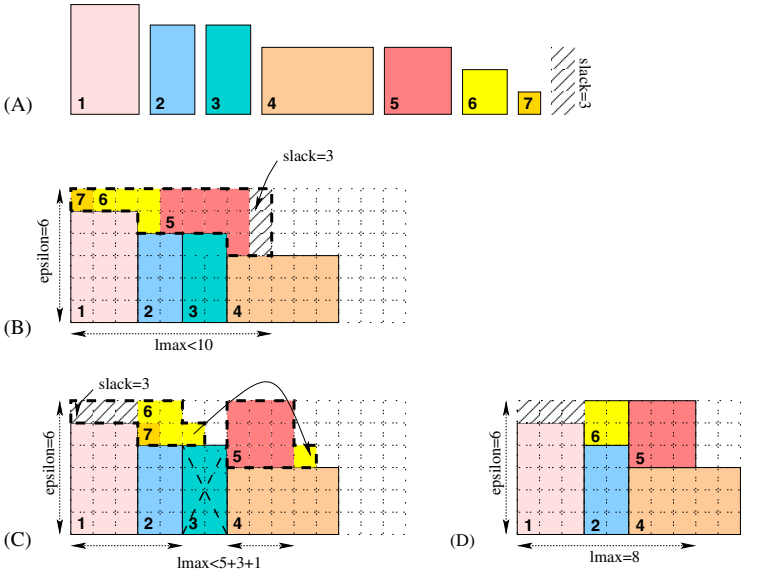


Fig. 5. (A) Seven tasks 3×5, 2×4, 2×4, 5×3, 3×3, 2×2 and 1×1 to place with a slack of 3 and a gap ϵ of 6, (B) An upper bound of 9 obtained without shrinking, (C) A tighter upper bound of 8 obtained by removing the third task, (D) An optimal placement which reaches the bound 8.

This second upper bound can be enhanced by trying to compute a bigger list of tasks in disjunction. A task t_i cannot overlap a task t_j if the sum of their heights, $t_i.\mathrm{h} + t_j.\mathrm{h}$, is greater than ϵ. But we can also use the fact that we have already computed the longest cumulative hole for smaller values of ϵ. Tasks t_i and t_j are also in disjunction if there is a gap g on top of the two tasks (i.e., $g = \epsilon - t_i.\mathrm{h} - t_j.\mathrm{h}$) for which $lmax_\sigma^g(T \setminus \{t_i, t_j\})$ is equal to 0.

3.4 Illustrating the Incomparability of the Two Bounds

This section shows that the two bounds previously described are in fact incomparable. For this purpose, consider the tasks of size 2×2, 4×4, 6×6, 7×7, 8×8, 9×9, 11×11 and 15×15. Let $\mathrm{B1}_\sigma^\epsilon$ and $\mathrm{B2}_\sigma^\epsilon$ respectively denote the upper bounds for $lmax_\sigma^\epsilon$ obtained by the first and the second upper bounds previously introduced. On the one hand, we have that $\mathrm{B1}_0^{12} = 5$ and $\mathrm{B2}_0^{12} = 4$, while on the other hand we have that $\mathrm{B1}_0^{15} = 30$ and $\mathrm{B2}_0^{15} = 32$.

4 Balancing Knapsack Constraints

In the context of a $cumulative(T, L)$ constraint with $|T| = n$ and slack σ, let its timespan be defined as $[u_{\min}, u_{\max}]$ where $u_{\min} = \min\{\underline{t.\mathrm{s}} \mid t \in T\}$ and $u_{\max} = \max\{\overline{t.\mathrm{s}} + t.\mathrm{d} - 1 \mid t \in T\}$. If $\sigma = 0$, then for every time point $b \in [u_{\min}, u_{\max}]$, the total height of tasks intersecting b must equal L. This reasoning can be generalized to non-zero slack and strengthened by considering adjacent pairs of time points $(b-1, b)$ into the following proposition.

Proposition 1. *For a $cumulative(T, L)$ constraint with slack σ and timespan $[u_{\min}, u_{\max}]$, for every time point $b \in [u_{\min}+1, u_{\max}]$, each of the following conditions is a necessary condition for the constraint.*

- *Let H_{b-1} denote the total height of tasks intersecting $b-1$. $H_{b-1} \in [L-\sigma, L]$ must hold.*
- *Let H_b denote the total height of tasks intersecting b. $H_b \in [L-\sigma, L]$ must hold.*
- *$H_{b-1} + H_b \in [2 \cdot L - \sigma, 2 \cdot L]$ must hold.*

Let t_i denote the i^{th} task of T. For every time point $b \in [u_{\min}+1, u_{\max}]$ and task $t_i \in T$, we have four mutually exclusive possibilities. We encode these possibilities as 0-1 variables $S_{ib}, C_{ib}, O_{ib}, N_{ib}$ where $S_{ib} + C_{ib} + O_{ib} + N_{ib} = 1$ and:

$$\begin{array}{l} S_{ib} = 1 \Leftrightarrow t_i \text{ intersects } b \text{ but not } b-1, \text{ that is, } t_i.\mathrm{s} = b \\ C_{ib} = 1 \Leftrightarrow t_i \text{ intersects } b-1 \text{ but not } b, \text{ that is, } t_i.\mathrm{s} = b - t_i.\mathrm{d} \\ O_{ib} = 1 \Leftrightarrow t_i \text{ intersects both } b-1 \text{ and } b, \text{ that is, } t_i.\mathrm{s} \in [b - t_i.\mathrm{d} + 1, b-1] \\ N_{ib} = 1 \Leftrightarrow t_i \text{ intersects neither } b-1 \text{ nor } b, \text{ that is, } t_i.\mathrm{s} \notin [b - t_i.\mathrm{d}, b] \end{array} \tag{1}$$

For a given time point b, the set of tasks T and the above, we can set up the following pseudo-boolean equation system, which essentially captures the above proposition.

$$\begin{array}{l} \forall i \in [1,n] : S_{ib} + C_{ib} + O_{ib} + N_{ib} = 1 \\ H_{b-1} = \sum_{i \in [1,n]} t_i.\mathrm{h} \cdot (C_{ib} + O_{ib}) \in [L-\sigma, L] \\ H_b = \sum_{i \in [1,n]} t_i.\mathrm{h} \cdot (S_{ib} + O_{ib}) \in [L-\sigma, L] \\ H_{b-1} + H_b \in [2 \cdot L - \sigma, 2 \cdot L] \end{array} \tag{2}$$

This equation system can be solved by a dynamic programming method similar to the one described in [11]. Define a function $f(k, l, r)$ equal to 1 if and only if the derived equation system (3) has a solution, and define the dynamic programming recursion as in (4).

$$\begin{array}{l} \forall i \in [1,k] : S_{ib} + C_{ib} + O_{ib} + N_{ib} = 1 \\ \sum_{i \in [1,k]} t_i.\mathrm{h} \cdot (C_{ib} + O_{ib}) = l \\ \sum_{i \in [1,k]} t_i.\mathrm{h} \cdot (S_{ib} + O_{ib}) = r \end{array} \tag{3}$$

$$f(0,l,r) = \begin{cases} 1\ , & \text{if } l = 0 \wedge r = 0 \\ 0\ , & \text{otherwise} \end{cases}$$

$$f(k,l,r) = \max \begin{cases} f(k-1, l, r - t_k.\mathrm{h}) \\ f(k-1, l - t_k.\mathrm{h}, r) \\ f(k-1, l - t_k.\mathrm{h}, r - t_k.\mathrm{h}) \\ f(k-1, l, r) \end{cases} , \text{if } k > 0 \tag{4}$$

Now, intuitively, (2) has a solution if and only if there exist l and r such that $l \in [L-\sigma, L]$, $r \in [L-\sigma, L]$, $l+r \in [2 \cdot L-\sigma, 2 \cdot L]$, and $f(n, l, r) = 1$. One can visualize this as a directed graph with a node for every (k, l, r) for which $f(k, l, r) = 1$ and arcs corresponding to (4). Also, each arc is annotated with the 0-1 variable that is assumed to take the value 1 in that branch of the recursion:

$$(k-1, l, r-t_k.\mathrm{h}) \xrightarrow{S_{kb}} (k, l, r)$$
$$(k-1, l-t_k.\mathrm{h}, r) \xrightarrow{C_{kb}} (k, l, r)$$
$$(k-1, l-t_k.\mathrm{h}, r-t_k.\mathrm{h}) \xrightarrow{O_{kb}} (k, l, r)$$
$$(k-1, l, r) \xrightarrow{N_{kb}} (k, l, r)$$

Among the nodes, let the single *source* node be $(0, 0, 0)$, and let the *sink* nodes be all nodes (n, l, r) where $l \in [L-\sigma, L]$, $r \in [L-\sigma, L]$, and $l+r \in [2 \cdot L-\sigma, 2 \cdot L]$. Then a path from the source to some sink corresponds to a solution to (2). By inspecting the arcs of such paths, we can determine for each 0-1 variable whether it takes the value 1 in some solution to (2). After computing all paths, we inspect each 0-1 variable: if it does not take the value 1 in any solution, the corresponding start time domain is pruned according to the equivalences given in (1). The complexity of this algorithm is $O(nL^2)$ (space and time).

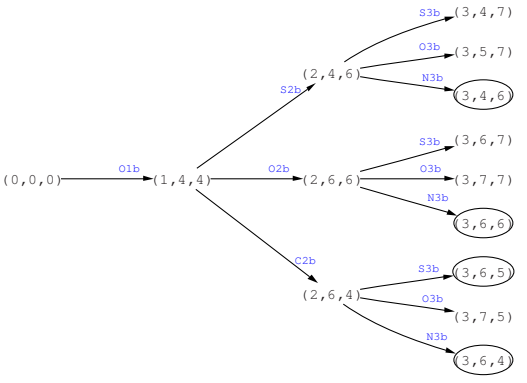
Example 8. Consider a $\mathit{cumulative}(\{t_1, t_2, t_3\}, 6)$ constraint with tasks as defined in Figure 6. Let us apply the method for $b = 4$ and $\sigma = 4$ (the slack has been tightened by other, fixed tasks that have been omitted in the example). The method explores the digraph shown in Figure 6. The four sink nodes are denoted by ellipses. As there is no arc annotated with O_{3b} on a path reaching a sink, we conclude that t_3 cannot intersect both 3 and 4, hence the value 3 can be removed from $\mathcal{D}(t_3.\mathrm{s})$. In this example, the digraph is a tree, which is not generally the case. □

4.1 Strengthening the Method

The method can be strengthened by adding more knapsack constraints, e.g., constraints that capture the fact that the height of the cumulative profile must not exceed L. This can be done as follows:

- Identify subsets $T_l \subseteq T$ and $T_r \subseteq T$ such that the following properties hold:
  - For each $t_i \in T_l$, both 0 and 1 are feasible values for O_{ib}, and $O_{ib} = 0$ would create a compulsory part of t_i to the left of b.
  - If $O_{ib} = 0$ for all $t_i \in T_l$, the cumulative profile would exceed L.
  - For each $t_i \in T_r$, both 0 and 1 are feasible values for O_{ib}, and $O_{ib} = 0$ would create a compulsory part of t_i to the right of b.
  - If $O_{ib} = 0$ for all $t_i \in T_r$, the cumulative profile would exceed L.
- Add the knapsack constraint $\sum_{t_i \in T_l} O_{ib} \geq 1$ for every such subset T_l found.
- Add the knapsack constraint $\sum_{t_i \in T_r} O_{ib} \geq 1$ for every such subset T_r found.

Our implementation includes this idea, using at most one subset T_l and at most one subset T_r.



$\mathcal{D}(t_1.\mathrm{s}) = \{3\}, \quad t_1.\mathrm{h} = 4,\ t_1.\mathrm{d} = 3$
$\mathcal{D}(t_2.\mathrm{s}) = \{2,3,4\},\ t_2.\mathrm{h} = 2,\ t_2.\mathrm{d} = 2$
$\mathcal{D}(t_3.\mathrm{s}) = \{1,3,4\},\ t_3.\mathrm{h} = 1,\ t_3.\mathrm{d} = 2$

Fig. 6. Tasks and digraph explored by dynamic programming for $b = 4, \sigma = 4, L = 6$

4.2 Learning Solutions

Let the *pre-signature* of a $cumulative(T, L)$ constraint and time point b be the set of 0-1 variables for which the value 1 is feasible according to (1) prior to solving the equation system. Similarly, let its *post-signature* be the set of 0-1 variables for which the value 1 is still feasible after solving the equation system.

It is worth noting that, given fixed T, L, σ, the pseudo-boolean equation system is totally abstracted away from the chosen b as well as from the variable domains. It is totally determined by its pre-signature. Thus having solved an equation system, it makes sense to record its pre- and post-signatures. Later on, if an equation system with the same pre-signature arises, we can retrieve the associated post-signature instead of recomputing it. Experience shows that this idea saves about 75% of the computational effort.

5 Performance Evaluation

All the new filtering methods described in this paper were integrated into our *geost* kernel [12] in order to strengthen the sweep-based filtering associated with non-overlapping constraints. The experiments were run in SICStus Prolog 4 compiled with gcc -02 version 4.0.2 on a 3GHz Pentium IV with 1MB of cache. All benchmarks were run with the following four phases search procedure, where at each phase, rectangles are considered by decreasing area. Let $o.x$ denote the X coordinate variable of the rectangle o:

1. For each rectangle o, narrow by binary search the domain of $o.x$ until it has a compulsory part that is at least half the length of o.
2. For each rectangle o, fix $o.x$ by binary search.
3. Repeat steps 1-2 for the Y coordinates.

Wanting to get an idea of their performance on perfect packing problems (i.e., 0% slack), we considered the *perfect square problem* [1,9]. A *perfect square of order* n is a square that can be tiled with n smaller squares where each of the smaller squares has

a different integer size. We used the data available (i.e., the size of the small squares to pack) from the catalogue [13] and tested the corresponding 207 instances. On the one hand, 66 problems were completely solved (i.e., finding all solutions and proving that no other solution exists) without a single backtrack. On the other hand, 36, 84, 20, resp. 1 problems were solved by using 1-10, 11-100, 101-1000, resp. 1001-1438 backtracks. This is an improvement by two orders of magnitude over [12]. From a time point of view, 35, 169 resp. 3 problems were solved in less than 10, 100 resp. 200 seconds.

Set	G_{hk}	G_h	G_k	G
Butone N	170730 (29637)	275445 (25013)	520050 (94205)	1383888 (107317)
Squares N	10043 (4168)	96213 (10951)	17417 (5470)	1006336 (86080)
Squares 1	996 (1557)	9730 (1203)	1817 (1840)	151905 (11813)

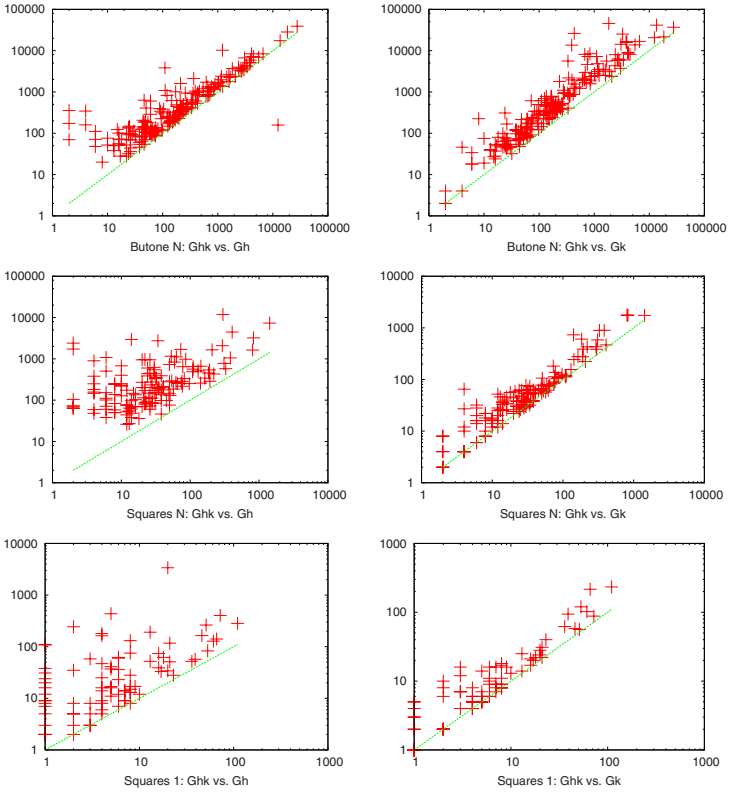


Fig. 7. Top: performance summary as total backtracks (seconds) per benchmark set. Bottom: scatter plots of number of backtracks per instance. X coordinate values correspond to problem instances with both methods enabled. In the left hand column, balancing knapsack constraints were knocked out in the Y coordinate values. In the right hand column, the longest cumulative hole method was knocked out in the Y coordinate values. G_{hk}, G_h, G_k and G denote respectively the *geost* kernel with both methods, with longest cumulative holes only, with balancing knapsack constraints only, and with neither method added.

In order to evaluate our method on non-perfect packing problems (with non-zero slack), we took 202 out of the same 207 perfect square instances, removing in each instance the smallest square to place. Five instances were excluded because they exceeded the time limit.

To evaluate the effectiveness of the two methods described in this paper, Figure 7 summarizes per benchmark set the performance. **Square 1** denotes searching for the first solution of a perfect square with symmetry breaking, whereas **Square N** denotes searching for all (8 or 16) solutions of a perfect square instance with no symmetry breaking, and **Butone N** denotes searching for all solutions of a perfect square instance with the smallest square removed, also with no symmetry breaking. The figure also contains six scatter plots. Each dot corresponds to a problem instance. Its X coordinate equals the number of backtracks to solve it with both methods enabled. Its Y coordinate equals the number of backtracks to solve it with only one method enabled.

On the perfect square instances, we find that both methods sharply decrease the number of backtracks, balancing knapsack constraints having the strongest effect. On the non-perfect packing instances, the results suggest that the effectiveness of balancing knapsack constraints degrades somewhat more rapidly with increasing slack than that of the longest cumulative hole method. In both cases, we find a nice multiplicative effect from combining the two methods. However, when we tried the methods on the 2D orthogonal packing instances proposed by Clautiaux et al. [7], the two methods did not significantly decrease the number of backtracks on non-perfect packing instances, whereas on instances with zero slack they did. So the results should be treated with caution for non-perfect packing problems.[9]

6 Conclusion

This paper introduces two new filtering methods that can be used in the context of the *cumulative* as well as the *non-overlapping* constraints.

1. The *longest cumulative hole problem* can be used to detect early that some specific space can not be filled enough.
2. The *balancing knapsack constraint* relates the total height of the tasks that end at a given time point to the total height of the tasks that start at the same time point.

As demonstrated by our benchmarks, these two methods are complementary, especially when the slack is very small. In such contexts, they reduce significantly the number of backtracks and even allow to completely enumerate the search space for a significant number of instances without any backtrack. An open issue is to come up with more efficient methods for proving infeasibility when the slack is not so small.

Acknowledgements

This research was conducted under European Union Sixth Framework Programme Contract FP6-034691 “Net-WMS”.

[9] Unlike the squares instances, it is worth noting that Clautiaux instances contain rectangles that are long in one dimension and short in the other dimension.

References

1. Aggoun, A., Beldiceanu, N.: Extending CHIP in order to solve complex scheduling and placement problems. Mathl. Comput. Modelling 17(7), 57–73 (1993)
2. Caseau, Y., Laburthe, F.: Cumulative scheduling with task intervals. In: Joint International Conference and Symposium on Logic Programming (JICSLP 1996), MIT Press, Cambridge (1996)
3. Mercier, L., Van Hentenryck, P.: Edge-finding for cumulative scheduling. INFORMS Journal on Computing 20(1) (2008)
4. Lahrichi, A.: Scheduling: the notions of hump, compulsory parts and their use in cumulative problems. C.R. Acad. Sci., Paris 294, 209–211 (1982)
5. Clautiaux, F., Jouglet, A., Carlier, J., Moukrim, A.: A new constraint programming approach for the orthogonal packing problem. Computers and Operation Research 35(3), 944–959 (2008)
6. Biró, M.: Object-oriented interaction in resource constrained scheduling. Information Processing Letters 36(2), 65–67 (1990)
7. Clautiaux, F., Carlier, J., Moukrim, A.: A new exact method for the two-dimensional orthogonal packing problem. European Journal of Operational Research 183(3), 1196–1211 (2007)
8. Lesh, N., Marks, J., McMahon, A., Mitzenmacher, M.: Exhaustive approaches to 2d rectangular perfect packings. Information Processing Letters 90(1), 7–14 (2004)
9. Van Hentenryck, P.: Scheduling and packing in the constraint language cc(FD). In: Zweben, M., Fox, M. (eds.) Intelligent Scheduling, Morgan Kaufmann Publishers, San Francisco (1994)
10. Beldiceanu, N., Contejean, E.: Introducing global constraints in CHIP. Mathl. Comput. Modelling 20(12), 97–123 (1994)
11. Trick, M.A.: A dynamic programming approach for consistency and propagation for knapsack constraints. Annals of Operations Research 118(1-4), 73–84 (2003)
12. Beldiceanu, N., Carlsson, M., Poder, E., Sadek, R., Truchet, C.: A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 180–194. Springer, Heidelberg (2007)
13. Bouwkamp, C.J., Duijvestijn, A.J.W.: Catalogue of simple perfect squared squares of orders 21 through 25. Technical Report EUT Report 92-WSK-03, Eindhoven University of Technology, The Netherlands (November 1992)

Multi-stage Benders Decomposition for Optimizing Multicore Architectures

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

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

Abstract. Software optimization for multicore architectures is one of the most critical challenges in today's high-end computing. In this paper we focus on a well-known multicore platform, namely the Cell BE processor, and we address the problem of allocating and scheduling its processors, communication channels and memories, with the goal of minimizing application execution time.

We have developed a complete optimization strategy based on Benders' decomposition. Unfortunately, a traditional two-stage decomposition produces unbalanced components: the allocation part is difficult, while the scheduling part is much easier. To address this issue, we have developed a multi-stage decomposition, which is a recursive application of standard Logic based Benders' Decomposition (LBD). Our experiments demonstrate that this approach is very effective in obtaining balanced sub-problems and in reducing the runtime of the optimizer.

1 Introduction

Multicore architectures on a single chip are emerging as the most significant paradigm shift in high-end computing platforms in the last twenty years. From the technology viewpoint, multicores are a necessity: a single processor cannot meet the ever increasing performance requirements of applications within a reasonable power budget. Moreover, multicore architectures are intrinsically more robust to variations and hardware failures that characterize current and future silicon technologies.

The Cell Broadband Engine (BE), jointly designed by IBM, Toshiba and Sony, is probably one of the most visible examples of high-end multicore architecture. These industry-leading companies have invested several hundred million dollars in its development, and the Cell BE is now a strategic component for embedded computing (i.e. game consoles) as well as for general-purpose high-performance computing.

The shift toward multicores has pushed to the center stage the critical issue of programming these highly parallel architectures, and more in general, the need for optimally exploiting the available resources in time and space. Cell is a pivotal example also in this area: even though its hardware capabilities are impressive, it is extremely difficult to program it effectively, mostly because software designers cannot manually allocate and schedule processors, communication channels and storage resources in an optimal way.

For this purpose, we have developed a programming infrastructure embedding constraint and integer programming optimization technology for the allocation and

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 36–50, 2008.

scheduling of embedded applications on the Cell BE architecture. We have an application modeled as a task graph. The application workload is partitioned into computation sub-units denoted as tasks, which are the nodes of the graph. Graph edges connecting any two nodes indicate task dependencies due, for example, to communication and/or synchronization. Tasks communicate through queues and each task can handle several input/output queues. We have to allocate tasks to processors, memory requirements and input/output queues to memory devices and schedule the overall application in order to minimize the application execution time (i.e., the schedule makespan).

We have previously solved similar applications [1], [2] via Logic-based Benders Decomposition [7], by facing allocation via Integer Linear Programming and scheduling via Constraint Programming, and the method was proved to be effective. In this case, however, a similar approach scales poorly. The main problem is that for the problem at hand the two-stage decomposition produces two unbalanced components. The allocation part is extremely difficult to solve while the scheduling part is indeed easier.

We have experimented a multi-stage decomposition, which is actually a recursive application of standard Logic based Benders' Decomposition (LBD), that aims at obtaining balanced and lighter components. An extensive set of experimental results confirms that the multi-stage decomposition pays off in terms of efficiency and in the quality of the solutions provided, when the proof of optimality cannot be completed in the available time. Also, we analyze the impact of Benders cuts and number of iterations in the traditional Benders' approach and in the variant we propose.

2 Problem Description

2.1 The Architecture

In this section we give a brief overview of the the Cell hardware architecture, focusing on the features that are most relevant for our optimization engine. Cell is a non-homogeneous multicore processor [11] which includes a 64-bit PowerPC processor element (PPE) and eight synergistic processor elements (SPEs), connected by an internal high bandwidth Element Interconnect Bus (EIB) [10]. Figure 1 shows a pictorial overview of the Cell Broadband Engine Hardware Architecture. The PPE is dedicated to the operating system and acts as the master of the system, while the eight synergistic processors are optimized for compute-intensive applications. The PPE is a multi-threaded core and has two levels of on-chip cache, however, the main computing power of the Cell processor is provided by the eight SPEs. The SPE is a compute-intensive coprocessor designed to accelerate media and streaming workloads [5]. Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC). The MFC includes a DMA controller, a memory management unit (MMU), a bus interface unit, and an atomic unit for synchronization with other SPUs and the PPE.

Efficient SPE software should heavily optimize memory usage, since the SPEs operate on a limited on-chip memory (only 256 KB local store) that stores both instructions and data required by the program.

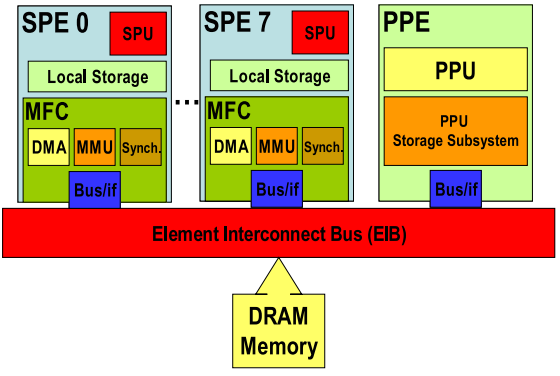


Fig. 1. Cell Broadband Engine Hardware Architecture

2.2 The Target Application

The target application to be executed on top of the hardware platform is input to our methodology, and for this purpose it must be represented as a task graph. This latter consists of a graph pointing out the parallel structure of the program. The application workload is therefore partitioned into computation sub-units denoted as tasks, which are the nodes of the graph. Graph edges connecting any two nodes indicate task dependencies due, for example, for communication and/or synchronization. Tasks communicate through queues and each task can handle several input/output queues. Task execution is modeled and structured in three phases: all input communication queues are read (Input Reading), task computation activity is performed (Task Execution) and finally all output queues are written (Output Writing). Each phase consists of an atomic activity. Each task also has two kinds of associated memory requirements:

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

Both these memory requirements can be either allocated on the local storage of each SPE or in the shared memory (DRAM in figure 1). Clearly the duration of the reading and writing phases are related to the corresponding queue allocation and the duration of the execution is related to the corresponding program data allocation. Remote memory allocation requires a bus access and the time spent is greater than the one for the local memory access. Tasks do not have deadlines, but these constraints could be easily handled by our method.

2.3 Problem Definition

The problem we have to solve is the allocation of tasks to SPE processors, the allocation of program data and communication queues of each task either on the local memory or on the remote DRAM, and the corresponding schedule. For the overall scheduling

problem with alternative resources we have to minimize the total application execution time (i.e., the makespan).

3 Multi-stage Benders Decomposition

The problem we have to solve is a scheduling problem with alternative resources and allocation dependent durations. A good way of facing these kind of problems is via Benders Decomposition, and its Logic-based extension [7]. Previous papers have shown the effectiveness of the method for similar problems. Hooker in [8] and [9] has shown how to deal with several objective functions in problems where tasks allocated on different machines are not linked by precedence constraints. Similar problems have been faced by Jain and Grossmann [6], Bockmayr and Pisaruk [4] and Sadykov and Wolsey [12], the latter comparing this approach with branch and cut and column generation. Many of these approaches consider multiple *independent* subproblems: that is, once the master problem is solved, then many decoupled subproblems result which can be solved in an independent fashion. The same approach is used by Tarim and Miguel [16] to solve stochastic problems with complete linear recourse.

The allocation is in general effectively solved through Integer Linear Programming, while scheduling is better faced via Constraint Programming. In our case, the scheduling problem cannot be divided into disjoint single machine problems since we have precedence constraints linking tasks allocated on different processors. We have implemented such an approach, similarly to [1], [2], and experimentally experienced a number of drawbacks. The main problem is that for the problem at hand a two stage decomposition produces two unbalanced components. The allocation part is extremely difficult to solve while the scheduling part is indeed easier. We will see in section 4 that this approach scales poorly.

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

In figure 2 at level one the SPE assignment problem (SPE stage) acts as the master problem, while memory device assignment and scheduling as a whole are the subproblem. At level two (the dashed box in figure 2) the memory assignment (MEM stage) is the master and the scheduling (SCHED stage) is the correspondent subproblem. The first step of the solution process is the computation of a task-to-SPE assignment; then, based on that assignment, allocation choices for all memory requirements are taken. Deciding the allocation of tasks and memory requirements univocally defines task durations. Finally, a scheduling problem with fixed resource assignments and fixed durations is solved.

When the SCHED problem is solved (no matter if a solution has been found), one or more cuts (labeled A) are generated to forbid (at least) the current memory device allocation and the process is restarted from the MEM stage; in addition, if the scheduling problem is feasible, an upper bound on the value of the next solution is also posted. When the MEM & SCHED subproblem ends (either successfully or not), more cuts (labeled B) are generated to forbid the current task-to-SPE assignment. When the SPE

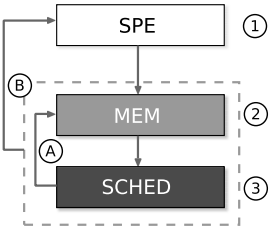


Fig. 2. Solver architecture: two level Logic based Benders' Decomposition

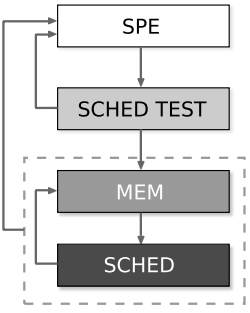


Fig. 3. Solver architecture with schedulability test

stage becomes infeasible the process is over converging to the optimal solution for the problem overall.

We found that quite often SPE allocation choices are by themselves very relevant: in particular, a bad SPE assignment is sometimes sufficient to make the scheduling problem infeasible. Thus, after the task to processor allocation, we can perform a first schedulability test as depicted in figure 3. In practice, if the given allocation with minimal durations is already infeasible for the scheduling component, then it is useless to complete it with the memory assignment that cannot lead to any feasible solution overall.

3.1 SPE Allocation

The computation of a task-to-SPE assignment is tackled by means of Integer Linear Programming (ILP). Given a graph with n tasks, m arcs and a platform with p processing elements the ILP model we adopted is very simple: this a first visible advantage of the the multi-stage approach. We introduce a decisional variable $T_{ij} \in \{0, 1\}$ such that $T_{ij} = 1$ is task i is assigned to PE j. The model to be solved is:

$$\begin{aligned} &\min \ z \\ &s.t. \ z \geq \sum_{i=0}^{n-1} T_{ij} \qquad \forall j = 0, \ldots, p-1 \qquad (1) \\ &\sum_{j=0}^{p-1} T_{ij} = 1 \qquad \forall i = 0, \ldots, n-1 \qquad (2) \\ &T_{ij} \in \{0, 1\} \qquad \forall i = 0, \ldots, n-1, \ \forall j = 0, \ldots p-1 \end{aligned}$$

Constraints (2) state that each task can be assigned to a single SPE; constraints (1) are needed to express the objective function. The makespan objective function depends only on scheduling decision variables. Here we adopt an objective function that tends to spread tasks as much as possible on different SPEs, which often provides good

makespan values pretty quickly. Constraints (1) force the objective variable z to be greater than the number of tasks allocated on any PE.

Constraints on the total duration of tasks on a single SPE were also added to a priori discard trivially infeasible solutions; this methodology in the LBD context is often referred to as "adding a subproblem relaxation", and is crucial for the performance of the method. In practice the model also contains the constraints:

$$\sum_{i=0}^{n-1} dmin(i)T_{ij} \leq dline \quad \forall j = 0, \ldots, p-1$$

Where $dmin(i)$ is the minimum possible duration of task i (reading and writing phases included), and $dline$ is a deadline. Since tasks have no deadline in the present problem, we impose as deadline the makespan of the best solution found so far.

Since the SPE are symmetric resources, the allocation model also features quite standard symmetry breaking ordering constraints to remove SPE permutations.

3.2 Schedulability Test

We modified the solver architecture by inserting a schedulability test between the PE and the MEM stage, as depicted in figure 3.

In practice, once a SPE assignment is computed, the system checks the existence of a feasible schedule using model of section 3.4, with all activity durations (execution, read, write) set to their minimum. If no schedule is found cuts that forbid (at least) the last SPE assignment are generated. Once a feasible schedule is found, the task-to-SPE assignment is passed to the memory device allocation component.

3.3 Memory Device Allocation

Once tasks are assigned to processing elements, their memory requirements and communication buffers must be properly allocated to storage devices. We tackled the problem by means of Mixed Integer Linear Programming, devising a model with a relatively simple "core".

Given a task-to-SPE assignment, for each task we introduce a boolean variable M_i such that $M_i = 1$ if t_i allocates its computation data on the local memory of the SPE it is assigned to (let this be $pe(i)$). Similarly, for each arc/communication queue $a_r = (t_h, t_k)$, we introduce two boolean variables W_r and R_r such that $W_r = 1$ if the communication buffer is on SPE $pe(h)$ (that of the producer), while $R_r = 1$ if the buffer is on SPE $pe(k)$ (that of the consumer).

$$M_i \in \{0,1\} \quad \forall i = 0, \ldots, n-1$$
$$W_r \in \{0,1\}, R_r \in \{0,1\} \quad \forall r = 0, \ldots, m-1$$

Note that, if for an arc $a_r = (t_h, t_k)$ it holds $pe(h) \neq pe(k)$, then either the communication buffer is on the DRAM, or it is local to the producer or local to the consumer; if instead $pe(h) = pe(k)$, than the communication buffer is either on the DRAM, or it is local to both the producer and the consumer. More formally, for each arc $a_r = (t_h, t_k)$:

$$R_r + W_r \leq 1 \qquad \text{if } pe(h) \neq pe(k) \tag{3}$$

$$R_r = W_r \qquad \text{if } pe(h) = pe(k) \tag{4}$$

Constraints on the capacity of local memory devices can now be defined in terms of M, W and R variables. When a task executes it always works on local data, therefore everything it needs (input and output buffers, internal data) is copied to the local device when the task starts. At the end of the execution all data allocated in DRAM are copied back, while all locally allocated requirements are left on the local device.

Therefore, in order to state memory capacity constraint we first define:

$$base\_usage(j) = \sum_{\substack{a_r = (t_h, t_k) \\ pe(k) = j}} comm(r)R_r + \sum_{pe(i)=j} mem(i)M_i + \sum_{\substack{a_r = (t_h, t_k) \\ pe(h) = j \\ pe(h) \neq pe(k)}} comm(r)W_r$$

Where $mem(i)$ is the amount of memory required to store internal data of task i and $comm(r)$ is the size of the communication buffer associated to arc r. The$base_usage(j)$ expression is the amount of memory needed to store all data permanently allocated on the local device of processor j. Then we can post the constraints:

$$\forall j = 0, \ldots, p-1, \ \forall i \text{ such that } pe(i) = j :$$
$$base\_usage(j) + \sum_{a_r=(t_h,t_i)} (1 - R_r)comm(r) +$$
$$(1 - M_i)mem(i) + \sum_{a_r=(t_i,t_k)} (1 - W_r)comm(r) \leq C_j$$

As in the previous stage, we also add to the model a scheduling subproblem relaxation; again, the two basic ideas are that the length of the longest path and the total duration of tasks on a single SPE must be lower than any current deadline. However, since memory allocation choices influence task duration, the relaxation is much more complex than that used in the SPE stage. Details on the relaxation can be found in [3].

The use of multistage Benders decomposition enables the complex resource allocation problem to be split into the drastically smaller SPE and MEM models. However, adding a decomposition step hinders the definition of high quality heuristics in the allocation stages and makes the coordination between the subproblems a critical task. We tackle these issues by devising effective Benders' cuts and using poorly informative, but very fast to optimize objective functions in the SPE and MEM stages. In practice the solver moves towards promising part of the search space by learning from its mistakes, rather than taking very good decisions in the earlier stages. Some preliminary experimental results showed how in our case this choice pays off in terms of computation time, compared to using higher quality (but harder to optimize) heuristics, or less expensive (but weaker) cuts.

3.4 Scheduling Subproblem

The scheduling subproblem is modeled and solved with ILOG Scheduler. In particular, we introduce an activity for each execution phase ($exec_i$) and buffer reading/writing

operation (rd_r, wr_r). Task are not preemptive, thus all activities regarding a single task execute without interruption in a pre-specified sequence. Suppose $rd_{r_0} \dots rd_{r_{h-1}}$ are the reading activities of task t_i and $wr_{r_h}, \dots, wr_{r_{k-1}}$ its writing activities, then:

$$\begin{gathered}
\forall l = 0, \dots, h-2 \quad end(rd_{r_l}) = start(rd_{r_{l+1}}) \\
end(rd_{r_{h-1}}) = start(exec_i) \\
end(exec_i) = start(wr_{r_h}) \\
\forall l = h, \dots, k-2 \quad end(wr_{r_l}) = start(wr_{r_{l+1}})
\end{gathered}$$

Each communication buffer must be written before it can be read. Thus for each pair of tasks t_h, t_k linked via a precedence constraint $a_r = (t_h, t_k)$ in the task graph we impose:

$$\forall r = 0, \dots, m-1 \quad end(wr_r) \leq start(rd_r)$$

Processing elements are modeled as unary resources, and all activities regarding task t_i use SPE of index $pe(i)$. Task durations are fixed and depend on memory allocation; in particular, a local memory requirement allocation always yields smaller durations. The objective function to minimize is the makespan.

In the previous papers on similar problems [1,15] we introduced a bus model using cumulative constraints. Here the applications we face are not communication intensive and the Cell platform provides plenty of communication bandwidth. We therefore did not impose such a constraint on the bus capacity.

3.5 Benders Cuts

Benders cuts are used in the Logic Based Benders Decomposition to control the iterative solution method and are of extreme importance for the success of the approach.

In first place, cuts are generated at each iteration yielding an infeasible subproblem in order to forbid (at least) the current master problem solution; when, after a number of iterations, the master problem becomes infeasible the solution process ends. The efficiency and the effectiveness of those cuts have therefore a strong influence on the total solution time.

Second, whenever a feasible complete solution is found, a new deadline constraint is added to the makespan requiring the forthcoming solutions to be better than the current one; then, cuts for the master problem are generated as in the previous case. In principle, the effectiveness of the method could be further improved by analyzing the last feasible solution to deduce cost bounds for not yet explored master problem assignments. Unfortunately, devising effective bounds of that kind is tricky in our case, due to the presence of precedence relations between tasks on different SPEs: we therefore decided to focus on generating strong feasibility cuts.

In a multi stage Benders Decomposition approach we have to define Benders cuts for each level. Here we have to specify both level 1 and level 2 cuts: we start from the level 2 Benders cuts, between the SCHED ad the MEM stage (“A” cuts in figure 2).

Let σ be a solution of the MEM stage, that is an assignment of memory requirements to storage devices. If X is a variable, we denote as $\sigma(X)$ the value it takes in σ. The level 2 cuts we used are:

$$\sum_{\sigma(M_i)=0} M_i + \sum_{\sigma(R_r)=0} R_r + \sum_{\sigma(W_r)=0} W_r \geq 1 \tag{5}$$

This forbids the last solution σ and all solutions one can obtain from σ by remotely allocating one or more requirements previously allocated locally: this would only yield longer task durations and worse makespan. In practice we ask for at least one previously remote memory requirement to be locally allocated.

Similarly, level 1 cuts ("B" cuts in figure 2), between the SPE and the MEM & SCHED stage must forbid at least the last proposed SPE assignment. Again, let σ be such a (partial) solution. Since the processing elements are symmetric resources, we can forbid together with the last assignment all its possible permutations. This is done by means of a polynomial size family of cuts.

For each processing element j we introduce a variable $S_j \in \{0, 1\}$ such that $S_j = 1$ iff all and only the tasks assigned to SPE j in σ are on a single SPE in a new solution. This is enforced by the constraints:

$$\forall j, k = 0, \ldots, p-1 \qquad \sum_{\sigma(T_{ij})=1} (1 - T_{ik}) + \sum_{\sigma(T_{ij})=0} T_{ik} + S_j \geq 1 \tag{6}$$

We can then forbid the assignment σ and all its permutations by posting the constraint:

$$\sum_{j=0}^{p-1} S_j \leq p - 1 \tag{7}$$

The level 1 and level 2 cuts we have just presented are sufficient for the method to work, but they are too weak to make the solution process efficient enough; we therefore

Algorithm 1. Refinement procedure

1: let X be the set of all master problem decisional variables in the original cut
2: sort the X set in nonincreasing order according to a relevance score
3: set $lb = 0$, $ub = |X|$, $n = lb + \lfloor \frac{ub-lb}{2} \rfloor$
4: **while** $ub > lb$ **do**
5: feed subproblem with current MP solution
6: relax subproblem constraints linked to variables $X_{i_n}, X_{i_{n+1}}, \ldots, X_{i_{|X|-1}}$
7: solve subproblem to feasibility
8: **if** $feasible$ **then**
9: set $lb = n + 1$
10: **else**
11: set $ub = n$
12: **end if**
13: restore relaxed subproblem constraints
14: **end while**
15: return lb

need stronger cuts. For this purpose we have devised a refinement procedure (described in Algorithm 1) aimed at identifying a subset of assignments which are responsible for the infeasibility. We apply this procedure to (5), (6) and (7).

Algorithm 1 refines a cut produced for the master problem, given that the correspondent subproblem is infeasible with the current master problem solution; an example is shown in figure 4, where $X_{i0}, \ldots X_{i5}$ are variables involved in the Benders cut we want to refine.

First all master problem variables in the original cut (let them be in the X set) are sorted according to some relevance criterion: least relevant variables are at the end of the sequence (figure 4-1). The algorithm iteratively updates a lower bound (lb) and an upper bound (ub) on the number of decisional variables which are responsible for the infeasibility; initially $lb = 0$, $ub = |X|$. At each iteration an index n is computed and all subproblem constraints linked to decisional variables of index greater or equal to n are relaxed; in figure 4-1 $n = 0 + \lfloor \frac{0+6}{2} \rfloor = 3$. Then, the subproblem is solved: if a feasible solution is found we know that at least variables from X_{i_0} to X_{i_n} are responsible of the infeasibility and we set the lower bound to $n + 1$ (figure 4-2). If instead the problem is infeasible (see figure 4-3), we know that variables from X_{i_0} to $X_{i_{n-1}}$ are sufficient for the subproblem to be infeasible, and we can set the upper bound to n. The process stops when $lb = ub$. At that point we can restrict the original cut to variables from X_{i_0} to $X_{i_{n-1}}$.

When we apply the Algorithm 1 to level 2 cuts the X set contains all M, R and W variables in the current cut (5); the relevance score is the difference between the current duration of the activity they refer to in the scheduling subproblem (resp. execution, buffer reading/writing) and the minimum possible duration of the same activity. Relaxing constraints linked to M, R and W variables means to set the duration of the corresponding activities to their minimum value.

Level 1 cuts are more tricky to handle: the X set contains tasks (ranked by their minimum duration) rather than decisional variables, and to relax the constraints we have to: A) set to the minimum the duration of all activities related to the considered task; B) remove all related (3) and (4) constraints in the memory allocation subproblem

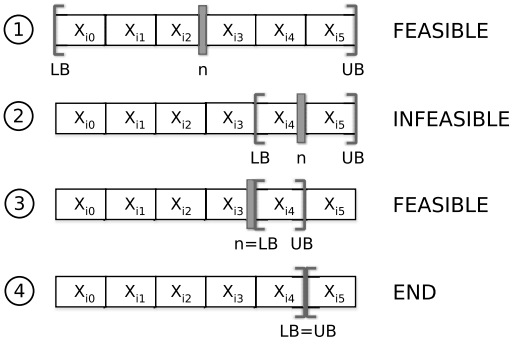


Fig. 4. Refinement procedure: an example

and set to 0 the memory requirement associated to all the corresponding M, R and W variables in the capacity constraints.

This cut refinement method has some analogies with what is done in Cambazard and Jussien [17], where explanations are used to generate logic based Benders cuts.

Note that refinement of level 2 cuts requires to repeatedly solve (relaxed) scheduling problems, which are by themselves NP-hard; the situation is even worse for level 1 cuts, since the subproblem is in this case the couple MEM & SCHED, which is iteratively solved. Therefore generation of refined cut is very expensive: the question is how much effort is worthwhile to spend in generating strong cuts. This is an issue which will be considered in the section about experimental results.

Finally, the described refinement procedure finds the minimum set of consecutive variables in X which cause the infeasibility of the subproblem, *without changing the order of the sequence*. Note however that is possible that some of the variables from X_{i_0} to $X_{i_{n-1}}$ are not actually necessary for the infeasibility. To overcome this limitation Algorithm 1 can be used within the iterative conflict detection algorithm described in [13], [14] to find a minimum conflict set. We implemented such an iterative procedure to generate even stronger (but of course more time consuming) cuts.

4 Experimental Results

Our approach has been implemented using the state of the art solvers ILOG Cplex 10.1 and Scheduler/Solver 6.3. We tested the approach on 200 task graphs representing realistic applications. All graphs were randomly generated by means of a specific instance generator designed to produce realistic task graphs. All instances feature high parallelism and complex precedence relations; durations and memory requirements are randomly generated, but based on values taken from real applications. The Cell configuration we used for the tests has 6 available SPEs.

Table 1 compares performance results for the traditional two stage logic based Benders decomposition approach referred to as BD, and the three stage that we propose in this paper, referred to as TD. In the two level solver, the master problem performs allocation of tasks to SPEs and memory requirements to storage devices through Integer Linear Programming while the subproblem is a scheduling problem and is solved via Constraint Programming. Instances are grouped by number of tasks; each group contains 20 instances, for which the minimum and maximum number of arcs is also reported. The table reports the average number of SPE, MEM iterations for the three-stage approach and the average number of iterations between the master and subproblem in the two stage approach (we refer to this quantity as PM iterations). In the time columns we report the average solution time for both solvers. All tests were run with a cutoff time of 1800 seconds: the last three columns report the number of instances (out of 20) for which: 1) both TD and BD exceed the time limit (TD $\wedge$ BD); 2) BD exceeds the time limit ad TD does not ($\neg$ TD $\wedge$ BD); 3) TD exceeds the time limit and BD does not (TD $\wedge \neg$ BD).

Note that in general TD is much more efficient than BD. Starting from group $20-21$, the high number of timed out instances makes the average execution time a less relevant index; by looking at the last three columns, however, one can easily see how in many

Table 1. Performance tests

		TD			BD		Timed out		
ntasks	narcs	SPE it.	MEM it.	time	PM it.	time	TD ∧ BD	¬ TD ∧ BD	TD ∧ ¬ BD
10-11	4-11	12	13	3.67	73	73.30	0	0	0
12-13	8-14	17	15	11.19	46	151.31	0	1	0
14-15	8-15	19	28	10.25	9	144.49	0	0	0
16-17	11-17	30	41	29.53	101	387.24	0	2	0
18-19	13-19	47	73	158.93	122	814.75	1	4	0
20-21	16-22	90	129	403.20	114	1291.90	2	10	0
22-23	19-26	87	132	571.88	95	1686.00	3	15	0
24-25	20-29	107	162	920.00	79	1639.00	9	7	0
26-27	23-29	88	187	837.50	30	1706.50	6	12	0
28-29	25-35	109	224	1218.50	24	1721.00	9	10	0

large instances TD can still find the optimal solution, while BD is not able to provide it within the time limit (column ¬ TD ∧ BD); note also that the opposite never occurs (column TD ∧ ¬ BD). Of course as the number of nodes and arcs grows the number of instances for which both solvers exceed the time limit also increases (column TD ∧ BD).

Note that TD has a lower execution time, despite it generally performs more iterations than BD. This suggest that the two solvers have in practice a very different behavior: TD tends to work by solving many easy subproblems, while BD performs fewer and slower iterations.

This is more clearly shown in table 2, which reports for each instance group the average number of SPE, MEM, SPE & MEM (PM) and SCHED subproblems solved by both solvers. For each solver the average time to solve a single subproblem of every type is reported.

One can see how TD solves thousands of problems (mostly to generate cuts), while BD faces fewer of them. On the other hand TD subproblems are very easy; note that the difference between the number of SPE, MEM and SCHED subproblems for the TD solver is around one order of magnitude, while the time to solve each subproblem type follows an analogous, inverse trend: once again this suggest that the TD solver has a quite balanced behavior. On the contrary, the resource allocation stage for the BD solver is instead often very time consuming compared to the scheduling; moreover, the gap becomes larger as the size of the instance increases.

Going more deeply, it is interesting to observe the distribution of the solution time between the problem components in the instances solved within the time limit and in those which are not.

Figure 5 reports histograms that show the distribution of the allocation/scheduling time ratio for the TD solver (where "allocation" means SPE + MEM). The X axis is divided into intervals, the Y axis counts the number of instances which fall in each interval.

Intuitively, in a balanced three stage decomposition strategy, the resource allocation is expected to take around 2/3 of the total solution time. One can see how the distribution for the instances solved within the time limit roughly follows a bell-shaped curve, with a peak around 0.7-0.8, slightly more than 2/3. The solution time for instances not solved within the limit appears to be more unbalanced with most of the time absorbed by the

Table 2. Number of subproblems solved and their difficulty

	TD #probs			TD time			BD #probs		BD time	
ntasks	SPE	MEM	SCHED	per SPE	per MEM	per SCHED	PM	SCHED	per PM	per SCHED
10-11	12	177	484	0.0362	0.0046	0.0013	12	165	2.1314	0.0010
12-13	17	285	573	0.0954	0.0078	0.0013	13	195	4.8076	0.0014
14-15	19	389	1312	0.0291	0.0083	0.0016	14	201	6.0836	0.0016
16-17	30	692	2304	0.0656	0.0141	0.0019	18	302	35.5924	0.0017
18-19	47	1463	6014	0.1266	0.0270	0.0028	26	495	84.7409	0.0024
20-21	90	2764	12641	0.7690	0.0549	0.0030	23	428	246.3311	0.0037
22-23	83	2707	12010	0.7709	0.0585	0.0988	19	448	270.8062	0.0049
24-25	107	3807	20877	1.4909	0.0860	0.0077	10	203	773.3269	0.0055
26-27	88	3959	24692	0.6456	0.0824	0.0087	5	87	1088.9167	0.0205
28-29	109	4731	31267	1.4714	0.1091	0.0104	5	140	1080.7726	0.0099

allocation. This suggests that for the TD solver more time could be spent in scheduling, for example to generate stronger cuts for the MEM stage.

This differentiated behavior between timed out and not timed out instances is not observed for the BD solver where substantially all the process time is spent in solving allocation subproblems (see figure 6).

Since most instances in the last two groups were not solved to optimality by both the approaches, we now want to compare the solution quality when optimality is not proved. In these cases the TD solver always finds the best solution and the average improvement is around 9%.

Finally, we considered the impact of strong Benders cuts on the TD solver. We disabled the strong cut refinement system in the TD solver: instead of finding a minimum conflict at each iteration we only remove some non relevant elements, using Algorithm 1. Table 3 reports the number of SPE and MEM iterations, the average solution time and the number of instances not solved within the time limit for the first three groups, without and with strong cut refinement. Note how disabling the refinement process causes a drastic performance breakdown: the weak refinement procedure is therefore not strong enough. Tuning the effort to be spent in cut generation remains an open problem.

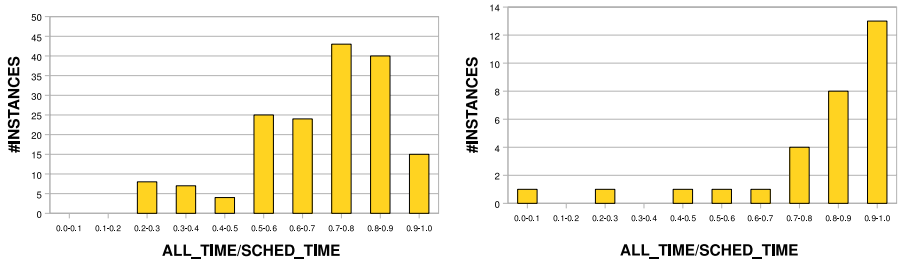


Fig. 5. TD execution time distribution for instances solved within the time limit (on the left) and not solved within the time limit (on the right)

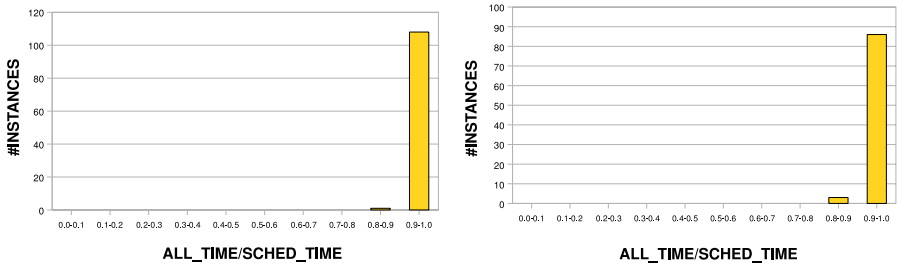


Fig. 6. BD execution time distribution for instances solved within the time limit (on the left) and not solved within the time limit (on the right)

Table 3. Performance results for the TD solver with and without strong cut refinement

	Without strong ref.				With strong ref.			
ntasks	**SPE it.**	**MEM it.**	**time**	**> TL**	**SPE it.**	**MEM it.**	**time**	**> TL**
10-11	192	90	497.90	5	12	13	3.67	0
12-13	386	295	1144.21	11	17	15	11.19	0
14-15	410	539	1181.24	12	19	28	10.25	0

5 Conclusion and Future Works

In this paper we have shown how to optimally solve allocation and scheduling of embedded applications modeled as task graphs on the Cell BE architecture. We have proposed a multi-stage logic based Benders decomposition approach featuring different components interleaved through Benders cuts. Experimental results show that the multi-stage is more efficient than the traditional two stage approach. Open problems remain: the issue concerning how to tune the strength of Benders cuts is extremely important for improving our approach. Also, the question of whether it is possible to determine the optimal number of stages for Benders decomposition based approaches is still open.

Finally, the choice of Benders' decomposition was motivated by the successful application of such method to similar problems with different objective functions; however, the use of other solution techniques, possibly better suited to makespan minimization (such as pure CP or heuristics methods), is of great interest and is subject of current research.

References

1. Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation and scheduling for MPSOCs via decomposition and no-good generation. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, Springer, Heidelberg (2005)
2. Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation, Scheduling and Voltage Scaling on Energy Aware MPSoCs. In: Beck, J.C., Smith, B.M. (eds.) CPAIOR 2006. LNCS, vol. 3990, Springer, Heidelberg (2006)

3. Benini, L., Lombardi, M., Mantovani, M., Milano, M., Ruggiero, M.: Multi-stage Benders Decomposition for Optimizing Multicore Architectures. Technical Report LIA-008-07
4. Bockmayr, A., Pisaruk, N.: Detecting infeasibility and generating cuts for MIP using CP. In: Int. Workshop Integration AI OR Techniques Constraint Programming Combin. Optim. Problems CP-AI-OR 2003, Montreal, Canada (2003)
5. Flachs, B., et al.: A streaming processing unit for a cell processor. In: Solid-State Circuits Conference. Digest of Technical Papers. ISSCC. 2005 IEEE International, pp. 134–135 (2005)
6. Grossmann, I.E., Jain, V.: Algorithms for hybrid milp/cp models for a class of optimization problems. INFORMS Journal on Computing 13, 258–276 (2001)
7. Hooker, J.N., Ottosson, G.: Logic-based benders decomposition. Mathematical Programming 96, 33–60 (2003)
8. Hooker, J.N.: A hybrid method for planning and scheduling. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 305–316. Springer, Heidelberg (2004)
9. Hooker, J.N.: Planning and scheduling to minimize tardiness. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 314–327. Springer, Heidelberg (2005)
10. Kistler, M., Perrone, M., Petrini, F.: Cell multiprocessor communication network: Built for speed. IEEE Micro 26(3), 10–23 (2006)
11. Pham, D., et al.: The design and implementation of a first-generation cell processor. In: IEEE International Solid-State Circuits Conference ISSCC 2005, vol. 1, pp. 184–592 (2005)
12. Sadykov, R., Wolsey, L.A.: Integer Programming and Constraint Programming in Solving a Multimachine Assignment Scheduling Problem with Deadlines and Release Dates. INFORMS Journal on Computing 18(2), 209–217 (2006)
13. de Siqueira, N.J.L., Puget, J.F.: Explanation-Based Generalisation of Failures. In: European Conference on Artificial Intelligence, pp. 339–344 (1988)
14. Junker, U.: QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. In: Proc. of the Nineteenth National Conference on Artificial Intelligence - AAAI 2004, San Jose, California, USA, July 2004, pp. 167–172. AAAI Press / The MIT Press (2004)
15. Lombardi, M., Milano, M.: Stochastic Allocation and Scheduling for Conditional Task Graphs in MPSoCs. In: Proc. of the Intl. Conference in Principles and Practice of Constraint Programming (2006)
16. Tarim, A., Miguel, I.: A Hybrid Benders Decomposition Method for Solving Stochastic Constraint Programs with Linear Recourse. In: Hnich, B., Carlsson, M., Fages, F., Rossi, F. (eds.) CSCLP 2005. LNCS (LNAI), vol. 3978, pp. 133–148. Springer, Heidelberg (2006)
17. Cambazard, H., Jussien, N.: Integrating Benders Decomposition Within Constraint Programming. In: Proc. of the Intl. Conference in Principles and Practice of Constraint Programming, pp. 752–756. Springer, Heidelberg (2005)

Fast and Scalable Domino Portrait Generation

Hadrien Cambazard, John Horan, Eoin O'Mahony, and Barry O'Sullivan

Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland
{h.cambazard,j.horan,e.omahony,b.osullivan}@4c.ucc.ie

Abstract. A domino portrait is an approximation of an image using a given number of sets of dominoes. This problem was first stated in 1981. Domino portraits have been generated most often using integer linear programming techniques that provide optimal solutions, but these can be slow and do not scale well to larger portraits. In this paper we propose a new approach that overcomes these limitations and provides high quality portraits. Our approach combines techniques from operations research, artificial intelligence, and computer vision. Starting from a randomly generated template of blank domino shapes, a subsequent optimal placement of dominoes can be achieved in constant time when the problem is viewed as a minimum cost flow. The domino portraits one obtains are good, but not as visually attractive as optimal ones. Combining techniques from computer vision and large neighborhood search we can quickly improve our portraits to be visually indistinguishable from those found optimally. Empirically, we show that we obtain many orders of magnitude reduction in search time.

1 Introduction

In 1981 Kenneth Knowlton filed for a United States Patent entitled "Representation of Designs" [4] in which he proposed the use of dominoes to render monochrome images. Twenty five years later, at the 2006 Conference on Constraint Programming, Artificial Intelligence and Operations Research (CP-AI-OR 2006), Robert Bosch gave an invited talk on "OptArt", focusing on how optimisation could be used to create pictures, portraits, and other works of art. In that talk, Bosch not only demonstrated the beauty of computer-generated art, but also the technical challenges involved in producing it. A domino portrait is simply a rendering of an image using a given number of sets of dominoes. Generally he uses "double nine" domino sets, which contain all dominoes from the "double blank" to the "double nine", giving fifty five dominoes in all.

The nice property of "double nine" domino sets is that they give a wide range of shades from complete black (the blank domino) to a bright white (the double nine domino). A set of dominoes gives us a constrained palette of monochrome shades, which we can use to produce images. We say that the palette is constrained for two reasons. Firstly, each set of dominoes contains only one domino of each type. Secondly, we are not allowed to break dominoes into two parts, but rather use the entire domino.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 51–65, 2008.

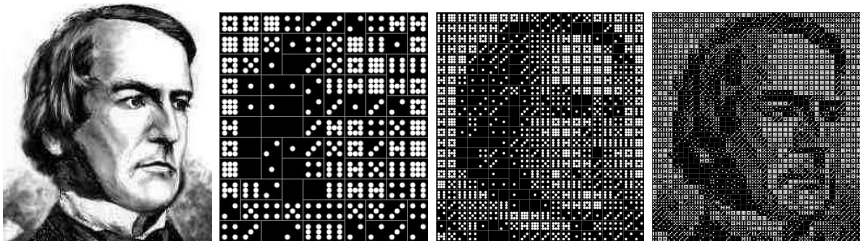


Fig. 1. A well known portrait of George Boole is presented on the left, with a sequence of domino portraits generated from this image using 1, 4 and 16 sets of dominoes as we move to the right, respectively

Several examples of domino portraits based on a well known portrait of George Boole are presented in Figure 1. It is clear that as we increase the number of dominoes we have at our disposal, the domino portrait we obtain is a better approximation of the target input image. In Figure 2 a much larger domino portrait of Boole is presented, which is sufficiently large for the reader to see each of the individual dominoes that comprise the portrait.

A problem with current approaches to generating domino portraits is that they do not scale very well. This is mostly due to the fact that Bosch has been interested in finding optimal domino portraits; we will explain how the notion of optimality is defined later in this paper. We set out to develop a scalable approach to generating domino portraits that would not be concerned with whether the portraits found were optimal or not, but be concerned with whether the portraits were sufficiently good so as to be visually indistinguishable from the optimal ones.

In this paper we present a new approach to building approximations of a target image using a specified number of complete sets of "double nine" dominoes [3,2]. We adopt an approach similar to Knowlton's [4] (and to Knuth's [5]), in which the image is divided up into blank domino outlines to which we assign dominoes. Rather than treating this problem as a traditional assignment problem, which can be solved using the Hungarian Method, and other similar algorithms, we formulate it as a *minimum cost flow*. The advantage is that the assignment step becomes constant time, allowing us to scale to arbitrary sized portraits. However, because we predetermine the orientations of the dominoes, we are unlikely to find an optimal domino configuration. Therefore, we adopt a heuristic approach to identifying regions of the domino placement that, if redesigned, would improve the quality of the resultant portrait. This last step is performed using a *large neighborhood search*. An empirical evaluation demonstrates the utility of our approach.

The remainder of the paper is organised as follows. Section 2 presents the domino portrait problem and explains in detail how it is defined. We then briefly summarise an existing linear model for finding optimal domino portraits in Section 3, as well as other heuristic approaches that have been studied. Section 4 describes the two-step approach we employ here, and our innovation based on a minimum cost flow formulation. In Section 5 we outline a practical improvement

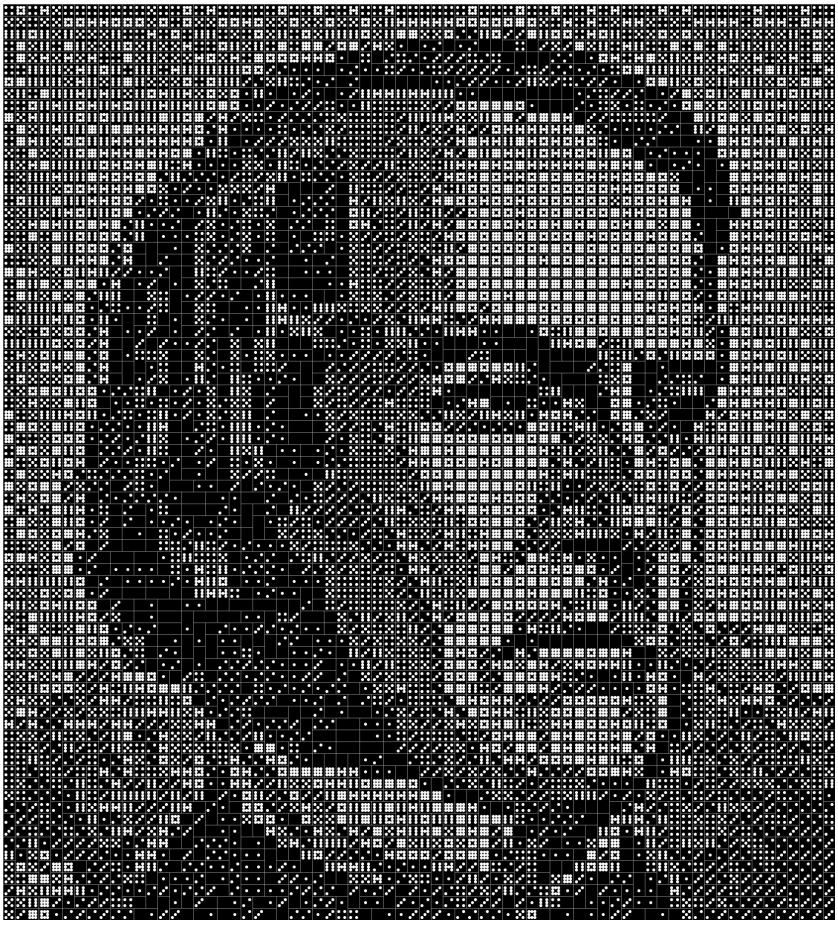


Fig. 2. A domino portrait of George Boole generated by our approach using 49 sets of "double nine" dominoes, i.e. $49 \times 55 = 2695$ individual dominoes

to our basic approach that involves locally perturbing the portrait. Section 6 presents and discusses the results. A number of concluding remarks are made in Section 7.

2 The Domino Portrait Generation Problem

A domino portrait can be generated for any target image. The first step in the process is to convert the target image into a grayscale graphic image using, for example, the UNIX *pgm* command. Each pixel in a grayscale image is given a grayscale value between 0 (black) and 255 (white).

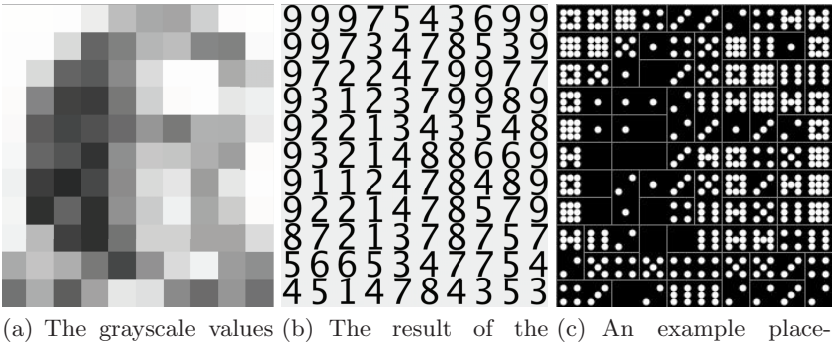
We consider rendering images using sets of "double nine" dominoes. There are 55 dominoes in a complete set of double nine dominoes: 10 dominoes with

equal face values in both halves, i.e. all dominoes with face valuations equal to $(0,0),\ldots,(9,9)$ along with an additional 45 non-equal face dominoes with face values in $\{(v_1,v_2)|v_1 \in \{0,\ldots,8\}, v_2 \in \{v_1+1,\ldots,9\}\}$. The area covered by a single set of dominoes is 110 square units, since we have 55 dominoes each with 2 units. Therefore, given s sets of dominoes, the grayscale image is divided into $11s \times 10s$ cells and for each cell in row r_i and column c_i the mean grayscale value is computed and scaled to an integer between 0 and 9 called g_{ij}. The values in each cell defines the perfect half domino value to place in that cell.

Each domino with equal valued halves has two possible orientations, vertical and horizontal, whereas each non-equal valued dominoes have 4 orientations since such a domino can be flipped along its vertical and horizontal axes. For $k = s^2$ sets of dominoes we can use a canvas of size $11s \times 10s$ to be filled with the $55 \times k$ dominoes, but in practice we can represent any canvas of size $110 \times k$. The following notation will be used throughout the paper:

- k is the number of sets of dominoes, and $N = 55 \times k$ is the number of individual dominoes.
- $d_i = (p_i^1, p_i^2)$ for domino number i, with $p_i^q \in \{0,\ldots,9\}$
- g_{ij} is the grey value of cell (r_i, c_j) between 0 and 9. The whole matrix of grey values is refered to as the grey matrix in the following.

The cost of positioning a half-domino p_l^q on a cell (r_i, c_j) is equal to $(p_l^q - g_{ij})^2$. Notice that it is quadratic so that the cost grows faster than the error and large errors are strongly penalised. The problem is to place the dominoes on the canvas so that the overall cost (the sum of the costs of each cell of the canvas) is minimised and every domino is used exactly once. A graphical representation of the process is presented in Figure 3.



(a) The grayscale values are scaled to $0\ldots9$. (b) The result of the scaling process. (c) An example placement of dominoes.

Fig. 3. A summary of the process of generating a domino portrait from an image

3 An Integer Linear Programming Model

Robert Bosch proposed an integer linear programming formulation of the domino portrait generation problem in [3]. His model is based on boolean variables that specify if a given domino is placed with a given orientation with respect to its reference square (the top left corner of each horizontally placed domino in Bosch's model) in a given cell of the canvas. Constraints then stipulate that each domino has to be used exactly once, and that each cell has to be covered by a domino. The resulting integer programs are quite large, with more than one million decision variables and five thousand constraints for $k = 49$, but Bosch reports that they are relatively easy to solve, requiring almost two hours when $k = 49$.

We used this model in our experiments as a baseline, with a very simple improvement not described by Bosch in his papers, but used by Knowlton, which involves keeping only the optimal orientation for each domino. A domino can be placed in two orientations at a given position but one often dominates the other, in terms of cost, and it is only necessary to consider the best orientation; this can be seen as a form of symmetry breaking over individual dominoes. The scalability of this model is, however, very limited and we will present a non-optimal, but much more efficient, approach to generating domino portraits in the next section, and then follow this presentation with an improvement based on large neighbourhood search.

4 A Two-Step Approximation

In his original patent, Knowlton outlined a two-stage process for generating domino portraits. The first step in his approach involved generating an initial arrangement of empty domino holders (rectangles) on the canvas, i.e. pairs of adjacent cells, which were later "filled" using dominoes. In this step he maximised the average *unbalance* of each domino holder by maximising the average difference between the two brightness values it contained. The second step involved assigning dominoes to the holders computed from the first step in order to minimise the error between the brightness provided by a domino and the brightness required in the domino holder computed from the first step. Donald Knuth subsequently recast Knowlton's method as an assignment problem [5], but because the two steps are independent, there is no guarantee the the resulting domino portrait will be close to optimality.

Here, we use another modification of Knowlton's method in which the initial pattern of empty dominoes is generated randomly, the dominoes are then placed into this pattern using an assignment problem formulation. This approach relies on the observation that the problem becomes polynomial if the *pattern* of the dominoes is known, since the assignment step is itself polynomial. This suggests that restricting ourselves to searching over alternative patterns is enough to generate optimal domino portraits. In practice we will show that any random pattern provides a very good upper bound on the cost of the domino portrait. We will present the details of each step in detail.

4.1 Generating the Pattern of Empty Domino Holders

We generate a random packing of empty domino holders on the canvas using Algorithm 1. We refer to this arrangement of empty domino holders as a *pattern*. Generating the pattern can be regarded as a packing problem. An example pattern is presented in Figure 4.

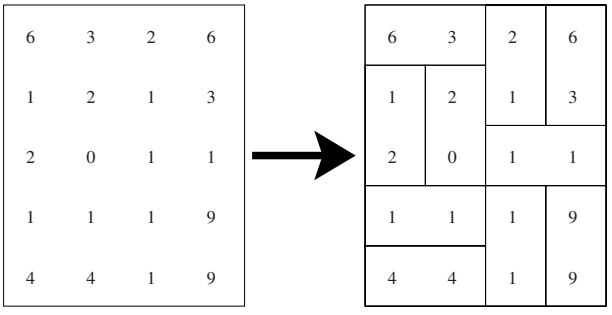


Fig. 4. An example of a pattern on the right that covers the grey matrix on the left

Algorithm 1 proceeds by filling the grid from the bottom to the top, line by line from the left to the right (lines 2 and 3). At each step it randomly assigns a rectangle vertically or horizontally (line 4) before going into a propagation step. Once a cell is surrounded (orthogonally) by three cells already covered by a domino holder, the orientation of the rectangle covering this cell is known and can be propagated (lines 5–6). This is performed until a fixed-point is reached, or a contradiction is met. A contradiction is raised when an odd number of connected cells remains in the grid, since dominoes cover pairs of cells. Each time a contradiction is met a restart step is performed. A small sub-region of the pattern is wiped out by removing a given number of lines.

Algorithm 1. Random pattern generator

1: **while** there exists an empty cell in the grid **do**
2: $i \leftarrow$ the first row containing an empty cell
3: $j \leftarrow$ the first column such that (i, j) is empty
4: Place a rectangle randomly at position $(i, j), (i + 1, j)$ or $(i, j), (i, j + 1)$
5: **while** there exists (i, j), an empty cell with three occupied orthogonal neighbours **and** all regions of empty connected cells are of even size **do**
6: Place a rectangle to cover (i, j) and the empty cell next to (i, j)
7: **end while**
8: **if** there is a region of an odd number of connected empty cells in the grid **then**
9: Wipe out part of the grid
10: **end if**
11: **end while**

This non-deterministic approach to random pattern generation performs very well in practice. In particular, we found this approach much faster than a complete backtracking algorithm for large number of dominoes.

4.2 Solving the Assignment Problem as a Min-Cost Flow

Once the pattern is known, placing the dominoes optimally is a polynomial problem – it is an optimal assignment problem. Figure 5 presents an example of the assignment problem. Notice that the cost $c(d_i, \langle a, b \rangle)$ of assigning a domino $d_i = (p_i^1, p_i^2)$ in a given rectangle of grey values $\langle a, b \rangle$ is defined as the best cost among the two possible orientations of the domino:

$$c(d_i, \langle a, b \rangle) = min((p_i^1 - a)^2 + (p_i^2 - b)^2, (p_i^1 - b)^2 + (p_i^2 - a)^2). \quad (1)$$

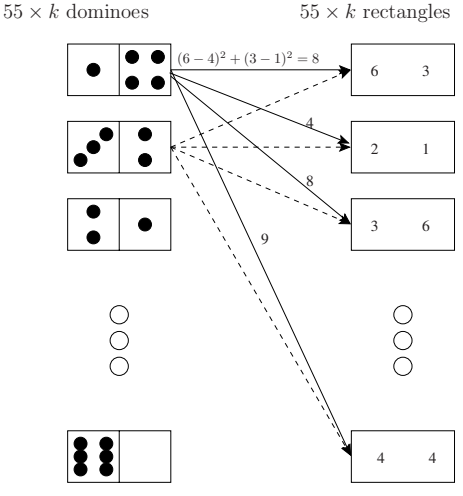


Fig. 5. An example of the assignment problem to be solved once the pattern is known

Solving the assignment problem can be done very efficiently using the Hungarian method in $O(n^3)$. However, in our setting n denotes the number of individual dominoes, which can quickly become very large. A good portrait often requires at least 100 sets of dominoes, giving 5500 individual dominoes. Clearly, the Hungarian method would not scale to those sizes.

We propose a novel formulation of this step as a min-cost flow. Observe that in the bipartite graph in Figure 5, dominoes on the left side are repeated k times and many rectangles on the right side have identical costs. In fact as the number of points varies from 0 to 9 on each square, there is only 55 possible pairs of points (for two adjacent squares) in the portrait. We can take advantage of these symmetries using the following formulation. We define the following notation:

– An *area* is a set of all rectangles with identical pairs of costs in the pattern. Area j corresponds to a rectangle of cost $\langle j_1, j_2 \rangle$ and the number of such

rectangles is denoted $capa_j$. Moreover, the total number of areas is denoted by $nbArea$ and $nbArea \leq 55$.
- x_{ij} is the number of dominoes of *kind* i assigned to area j.
- $c(d_i, j)$ is the cost of assigning a domino of kind d_i into area j. $c(d_i, j)$ is the same cost as previously so that $c(d_i, j) = c(d_i, \langle j_1, j_2 \rangle)$ as defined by Equation 1.

In the pattern given in Figure 4, we would have $nbArea = 6$ where each area would be defined by one of the six rectangles $\{\langle 6,3\rangle, \langle 2,1\rangle, \langle 2,0\rangle, \langle 4,4\rangle, \langle 1,1\rangle, \langle 9,9\rangle\}$. The optimal assignment can be reformulated as follows:

$$\begin{array}{l} Minimize \sum_{i,j} c_{ij} x_{ij} \\ \quad \text{subject to} \\ \quad\quad \sum_j x_{ij} = k, \forall i \leq 55 \\ \quad\quad \sum_i x_{ij} \leq capa_j, \forall j \leq nbArea \end{array} \tag{2}$$

The first constraint of this linear program ensures that exactly k dominoes of each kind are assigned. The second constraint ensures that no more than $capa_j$ dominoes are placed in the same area. In practice, there are exactly $capa_j$ dominoes to fill the area as we have $\sum_j capa_j = 55 \times k$. This problem can be better understood, and more efficiently solved, as a min-cost flow problem on the graph presented in Figure 6, where the x variables can be interpreted as the amount of flow from a domino i to an area j.

There are two key observations to be made about this formulation. Firstly, we only need to know the area where a domino is assigned and not specifically where it is placed in this area. Secondly, we only need to know how many dominoes of each kind are assigned in each area and not where each specific domino is assigned. The min-cost flow formulation takes these symmetries into account and provides a much more efficient way of solving the previous assignment problem. Notice that the size of the graph (number of nodes and edges) supporting the flow is independent of k; only the flow and capacities are increasing, making the approach robust to increases in k.

Once reduced to a min-cost flow formulation the problem can be solved in a variety of ways. It is easy, for example, to formulate it as a linear program (see Model 2). Fortunately this linear program has the quality of integrality, thus only the linear relaxation needs to be solved. Alternatively, there exist many algorithms to solve min-cost flow, e.g. the Successive Shortest Path (SSP) [1] algorithm which sends the largest possible flow along the shortest path from source to sink, found by Dijkstra's algorithm, at each iteration. The complexity of this algorithm with a small optimisation is $O(n \times max_{j \in \{1 \ldots nbArea\}}(capa_j))$ where n is the number of nodes.

An alternate algorithm, the Enhanced Capacity Scaling algorithm [1], is a strongly polynomial improvement on the Successive Shortest Path algorithm. It has a complexity of $O((m \log n)(m + n \log n))$, where n is the number of nodes and m is the number of arcs. This means that for our min-cost flow formulation the algorithm runs in *constant time* as the number of nodes and the number of

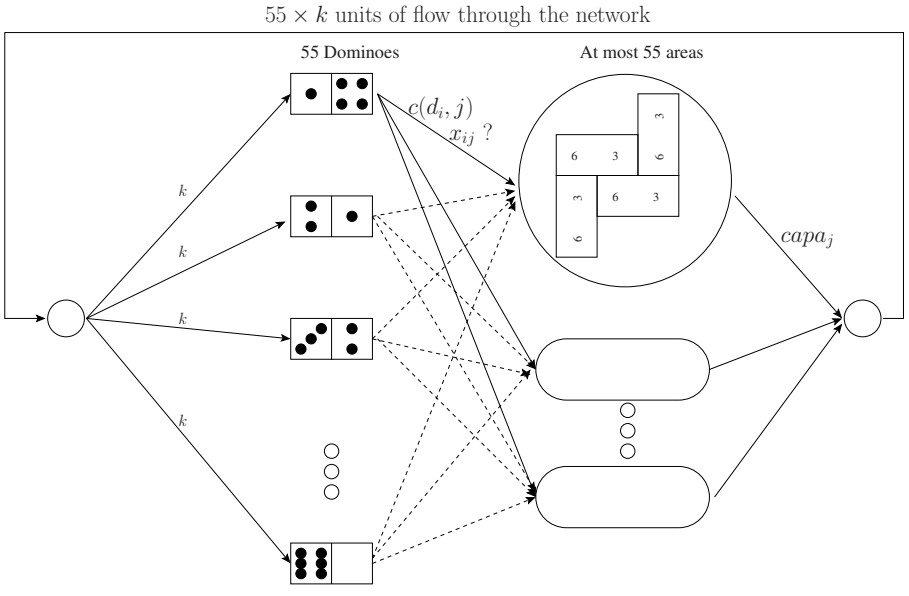


Fig. 6. The assignment problem translated as a min-cost flow problem

arcs are constant and not dependent on the number of sets of dominoes used to generate the portrait.

5 Improving the Pattern Using Local Search

Using the min-cost flow formulation we can solve the assignment step of the domino portrait generation problem in constant time. The only obstacle to generating optimal domino portraits is the choice of pattern to provide to the flow step. Notice that the pattern only matters where the grey values are unbalanced; the pattern in uniform areas has almost no effect on the final cost. In terms of the flow formulation, it means that a change of the pattern that would not affect the size of the areas of the flow graph, the $capa_j$ values, has no effect on the optimal assignment. Therefore, we consider perturbing the pattern slightly in a local search approach to affect the $capa_j$ values in order to improve the flow.

The algorithm we implemented can be described as a Large Neighborhood Search [8] over patterns. It proceeds as follows:

1. Identify the regions of the canvas where the grey values are unbalanced and thus, where the pattern might benefit from improvement. We denote as X the set of points (i, j) corresponding to those regions.
2. Select a point $x \in X$ and remove it from X. If X is empty then select a point randomly.
3. Remove M dominoes around x; x can be seen as the centre of the new empty region.

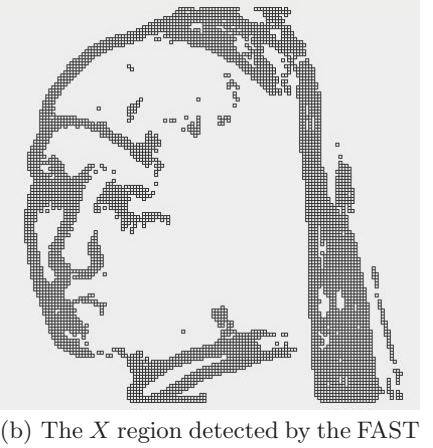
4. Enumerate all possible patterns that can fill the empty region. For each of those patterns incrementally update the $capa_j$ values and compute the corresponding new min-cost flow denoting the cost of the overall resulting pattern. Note that this is a global optimization step as the dominoes that were previously assigned in the region might now be in a completely different place.
5. Return to Step 2 (above) as long as the average improvement over the last 20 iterations remains above a threshold (set very low in practice).

The points in the set X are weighted in such a way that any points of interest adjacent to one already chosen for improvement are less likely to be selected than those that are independent of chosen points. This is to maximize the impact of the improvements during the initial executions and to ensure an overall faster convergence.

The first point is performed using an algorithm from computer vision that performs *corner detection,* or *interest point detection*, to extract certain kinds of features to infer the contents of an image. We used the FAST (Features from Accelerated Segment Test) algorithm from [6,7]. This approach seems very well suited for portraits as it highlights the important characteristics of the face (eyes, mouth, hair etc...) which matter in the final domino portrait. Figure 7(b) shows the result of FAST on the "Girl with a Pearl Earring".

The neighborhood explored is defined by all the possible patterns for a small region of $2 \times M$ squares of the grid ($M = 15$ is the setting used in our experiments). The enumeration is performed using the propagation described in Algorithm 1 in a complete backtracking search. Finally, the problem of finding the optimal flow regarding small changes of $capa_j$ is a sensitivity analysis problem on the min-cost flow and can be performed incrementally [1]. The optimal

(a) Vermeer's "A Girl with a Pearl Earring".



(b) The X region detected by the FAST algorithm for $k = 225$.

Fig. 7. Selecting the interesting region to focus on in the local search step

flow is maintained while performing a local search on the $capa_j$ values reflecting the changes in the pattern. This is possible due to the efficiency of the flow model and its incremental behaviour.

6 Experiments

Robert Bosch proposed an integer linear programming (ILP) approach to solving this problem, which we discussed earlier in the paper, and we used his approach as a baseline in our experiments. We used Vermeer's "A Girl with a Pearl Earring" (Figure 7(a)). All times quoted are the times it takes to generate and solve the respective models; they do not include the time taken to convert the solution into a viewable image. Experiments were run on a 2.8GHz Intel Xeon processor running Linux Fedora Core 2 with 4Gb of RAM.

Firstly, we sought to compare the performance of the min-cost flow and Hungarian method to demonstrate the scalability of the flow algorithm (Table 1). The flow algorithm used is SSP, mentioned previously, which was efficient enough for our purposes and easy to implement. Clearly, the Hungarian method does not scale, while the min-cost flow does very well. While the min-cost flow is constant-time in this setting (although not necessarily so when using the SSP algorithm), there is a small variation for different numbers of sets of dominoes due to the time spent generating the problem.

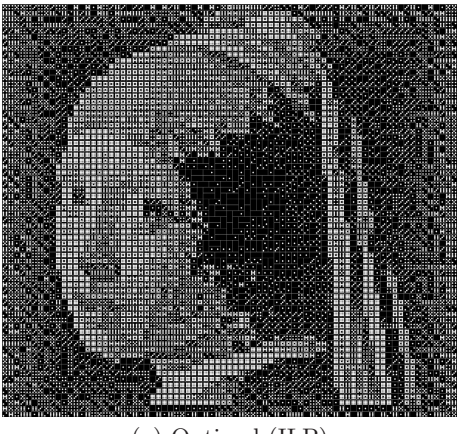
Table 1. Comparing the Hungarian and Min-Cost Flow approaches to solving the assignment phase of domino portrait generation

#Sets of Dominoes	Time (in seconds)	
	Min-Cost Flow	Hungarian
9	0.23	0.47
25	0.15	6.87
49	0.15	50.17
121	0.17	734.69
2,500	0.31	-
10,000	0.63	-

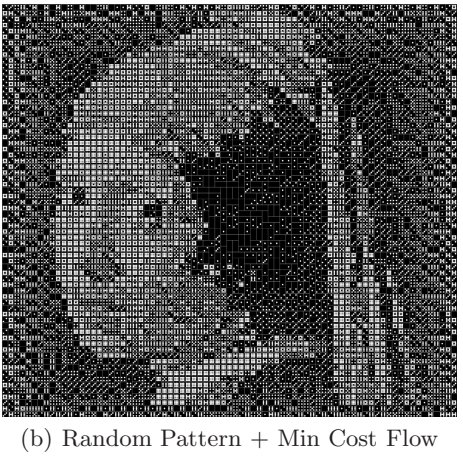
Table 2. Comparing the quality and speed of ILP and random patterns

	ILP		**Two phase (100 runs)**			**Gap (%)**	
k	Cost	Time (s)	Avg Cost	Best Cost	Time (s)	Avg	Best
1	1,192	1.04	1,260	1,222	0.02	5.96	2.52
4	4,844	13.8	5,228	5,139	0.04	7.99	6.09
9	11,255	65.9	12,183	12,013	0.07	8.26	6.73
25	33,673	325.62	36,265	35,998	0.12	7.71	6.90
49	69,585	7,030.29	74,075	73,639	0.13	6.45	5.83
121	171,961	9,797.55	181,768	180,991	0.16	5.72	5.25
225	376,176	44,895.86	386,870	386,326	0.17	2.84	2.69

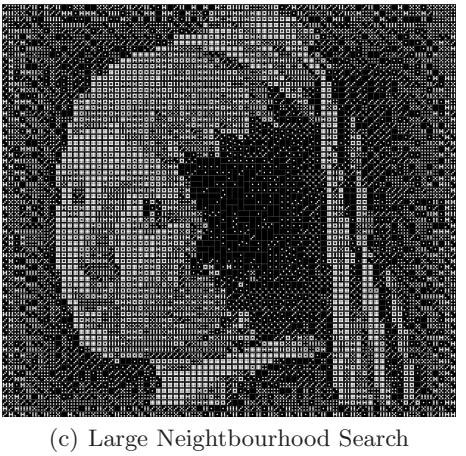
Secondly, we show the average quality of a random pattern (over 100 runs) for different number of sets of dominoes to support our claim that any random pattern provides a good bound on the quality of the domino portrait (see Table 2). The ILP model is solved with CPLEX using Bosch's model discussed in Section 3. In Table 2, the cost is the total cost of the optimal solution; we present both the average and best costs for the random pattern approach. It is interesting to note that as the number of sets of dominoes is increased, the quality of the portrait generated from a random pattern is improving; we can find portraits that are 2.69% worse than the optimal cost found using ILP when using 225 sets of dominoes. A very important difference between methods here, of course, is that the random pattern-based portrait is generated in a fraction of a second, while ILP takes several hours for larger numbers of dominoes.



(a) Optimal (ILP)



(b) Random Pattern + Min Cost Flow



(c) Large Neightbourhood Search

Fig. 8. Comparing the output of the ILP versus our full approach combining min-cost flow and local search

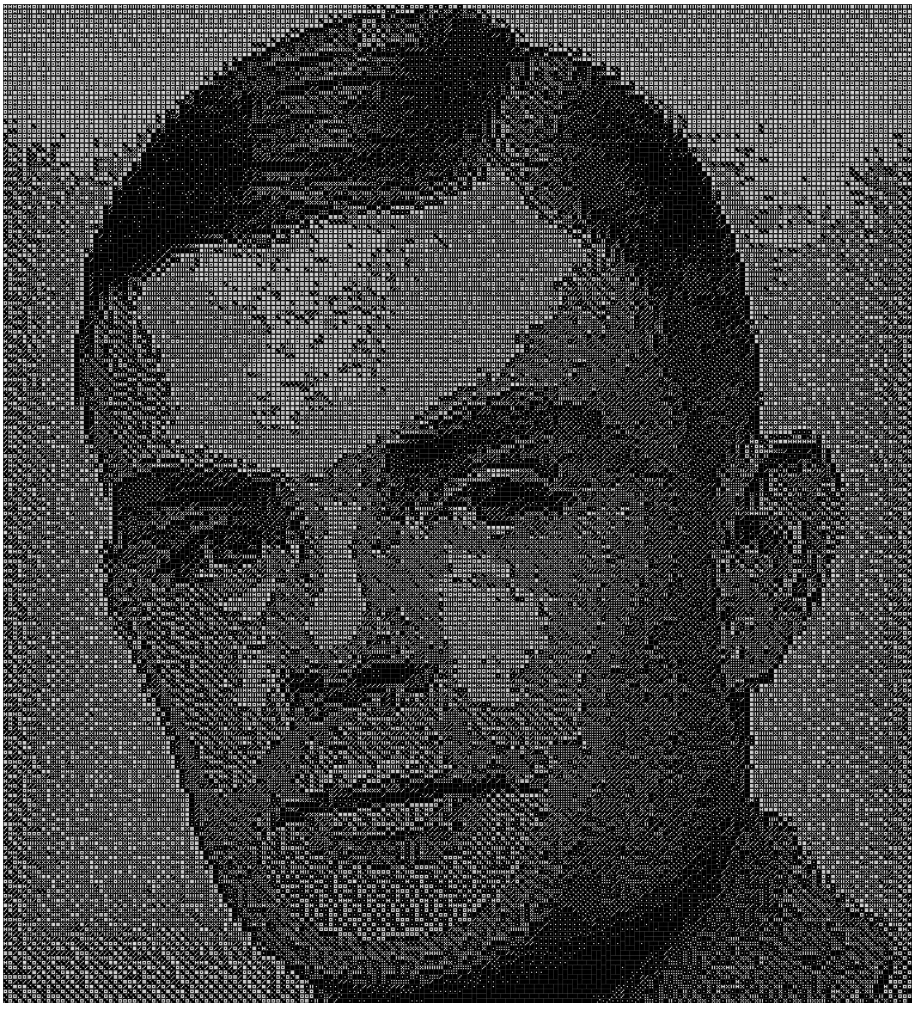


Fig. 9. A domino portrait of Alan Turing generated by our approach using 361 sets of "double nine" dominoes, i.e. 19,855 individual dominoes. The min-cost flow phase for this portrait required 0.194 seconds, and the local search phase required 33.965 seconds. We also generated a much larger portrait using 10,000 sets of dominoes (55,000 individual dominoes), which required 0.539 seconds and 26.285 seconds for the min-cost flow and local search phases, respectively. This portrait is not included in the paper since it would look almost like a standard grayscale image.

Thirdly, in Table 3 we show the results of the ILP formulation and the flow-based approach using local search over patterns which provide very good portraits within a few percent of the optimal value with orders-of-magnitude of speed-up in search time. The resulting images are indistinguishable visually from the optimum for "A Girl with a Pearl Earring" as shown on Figure 8. We show

Table 3. Comparing the quality and speed of ILP against the flow-based approach using local search to improve the pattern

	ILP		LNS patterns		
k	Opt Cost	Time (s)	Cost	Time (s)	Gap (%)
1	1,192	1.04	1,207	8.32	1.26
4	4,844	13.80	4,903	14.00	1.22
9	11,255	65.90	11,512	14.54	2.28
25	33,673	325.62	34,498	15.72	2.45
49	69,585	7,030.29	70,977	17.66	2.00
121	171,961	9,797.55	175,669	27.34	2.16
225	376,176	44,895.86	380,408	32.71	1.13

three portraits using 49 sets of dominoes corresponding to the optimal value obtained by the ILP, random pattern and local search approaches. For interesting sizes (between 9 and 225 sets of dominoes), the local search approach outperforms the ILP model in time without losing any relevant quality in the picture (gap no more than 2.45% and visually irrelevant).

We could not solve the ILP model for portraits requiring more than 225 sets of dominoes because of memory problems. However, even portraits requiring $10,000$ sets of dominoes ($55,000$ dominoes) are not a challenge for our approach. In fact, the larger the number of domino sets we use, the less we need to optimize the pattern using local search. In Figure 9 we show a very complex portrait of Alan Turing generated using 361 sets of dominoes, and report the times required by the min-cost flow and local search phases in its caption. We also report that using $10,000$ sets of dominoes we can generate portraits even faster because we have a much shorter local search step.

7 Conclusion

We have proposed a new solving technique for the domino portrait problem which is based on an original and efficient reformulation of part of the problem as a min-cost flow problem combined with local search. We show that we can obtain several orders-of-magnitude of speed-up to get high quality portraits within a few percent of the optimal value. This approach does not provide optimal solutions but produces high quality solutions within a couple of seconds. It is moreover very robust to the increase of the size of the problem.

Interesting ideas have been explored that might be useful in the context of packing problems with a positioning cost. The packing problem here is easy, as it is only made of rectangles of the same size, but the overall approach might be interesting in more complex and real-life applications where the objects are of different shapes.

Our application involves well known OR algorithms (Hungarian, Min-cost flow and sensitivity analysis of the flow), search techniques (large neighbourhood search, depth first search with constraint propagation) as well as an algorithm

from the computer vision area (FAST) and is, therefore, well suited for teaching Operations Research. It has been used with great success at the Discovery Exhibition 2007 in Cork[1], a science outreach event for pupils aged between 10 and 16.

Acknowledgements

This work was supported by Science Foundation Ireland (Grant No. 05/IN/I886). We thank Robert Bosch for providing the inspiration for tackling this problem and for providing some useful feedback.

References

1. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows - Theory, Algorithms, and Applications. Prentice Hall, Englewood Cliffs (1993)
2. Berlekamp, E., Rogers, T.: The mathemagician and pied puzzler: A collection in tribute to Martin Gardner. AK Peters (1999)
3. Bosch, R.: Constructing domino portraits. Tribute to a Mathemagician, 251–256 (2004)
4. Knowlton, K.C.: Representation of designs. U.S. Patent # 4,398,890 (August 16, 1983)
5. Knuth, D.E.: The Stanford GraphBase: A Platform for Combinatorial Computing. Addison-Wesley, Reading (1993)
6. Rosten, E., Drummond, T.: Fusing points and lines for high performance tracking. In: ICCV, pp. 1508–1515 (2005)
7. Rosten, E., Reitmayr, G., Drummond, T.: Real-time video annotations for augmented reality. In: Bebis, G., Boyle, R., Koracin, D., Parvin, B. (eds.) ISVC 2005. LNCS, vol. 3804, pp. 294–302. Springer, Heidelberg (2005)
8. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Maher, M.J., Puget, J.-F. (eds.) CP 1998. LNCS, vol. 1520, pp. 417–431. Springer, Heidelberg (1998)

[1] http://www.corkcity.ie/discovery/

Gap Reduction Techniques for Online Stochastic Project Scheduling*

Grégoire Dooms and Pascal Van Hentenryck

Brown University, Box 1910, Providence, RI 02912

Abstract. Anticipatory algorithms for online stochastic optimization have been shown very effective in a variety of areas, including logistics, reservation systems, and scheduling. For such applications which typically feature purely exogenous uncertainty, the one-step anticipatory algorithm was shown theoretically to be close to optimal when the stochasticity of the problem, measured by the anticipatory gap, is small. This paper studies the behavior of one-step anticipatory algorithms on applications in which the uncertainty is exogenous but the observations are endogenous. It shows that one-step anticipatory algorithms exhibit a much larger anticipatory gap and proposes a number of gap-reduction techniques to address this limitation. The resulting one-step anticipatory algorithms are shown to outperform significantly the state-of-the-art dynamic-programming approach on an online stochastic resource-constrained project scheduling application.

1 Introduction

Online anticipatory algorithms [8] have been recently proposed to address a wide variety of online combinatorial optimization problems in areas such as logistics, networking, scheduling, and reservation systems. The applications emerged from progress in telecommunication and in information technologies which enable organizations to monitor their activities in real time and collect significant amount of historical data. One-step anticipatory algorithms only rely on two black-boxes: a conditional sampler to generate scenarios consistent with past observations and an offline solver which exploits the combinatorial structure of the application to solve the deterministic version of the problem. Their essence is to transform the multi-stage stochastic optimization application into a 2-stage problem by ignoring all non-anticipativity constraints but those of the current decision. This 2-stage problem is then approximated by sampling, and the approximated problem is solved optimally by computing the offline optimal solutions for all pairs (scenario,decision). One-step anticipatory algorithms were shown to be very effective on a variety of online stochastic combinatorial problems in dynamic fleet management [1,2], reservation systems [8], resource allocation [6], and jobshop scheduling [7]. They were also analyzed theoretically in [4] in terms of the global anticipatory gap (GAG) which is a measure of the stochasticity of the application. The analysis shows that, when the GAG is small, anticipatory algorithms are guaranteed to return high-quality solutions when ran with enough scenarios.

* This research is partially supported by NSF award DMI-0600384 and ONR Award N000140610607.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 66–81, 2008.

This paper examines the behavior of one-step anticipatory algorithms on online Stochastic Resource-Constrained Project Scheduling Problems (S-RCPSP). Such applications are increasingly common for research and development projects, as well as the management of projects in the services industry. Contrary to the above applications in which the uncertainty is purely exogenous, the uncertainty in online S-RCPSP is exogenous but the observations are endogenous: it is necessary to execute an activity to observe its cost, its duration, and its outcome. The endogenous nature of the observations produces a significantly higher GAG, making these applications more challenging.

This paper shows that, despite these difficulties, one-step anticipatory algorithms still outperform the state-of-the-art algorithm proposed in [3] which applies dynamic programming to a heuristically-confined state space (HCDP). Moreover, the paper investigates a number of generic gap-reduction techniques, including a waiting strategy, gap correction, time scaling, and problem pruning. These techniques significantly improve the behavior of one-step anticipatory algorithms which produce an average improvement of about 15% compared to the HCDP algorithm.

The rest of the paper is organized as follows. Section 2 specifies the online S-RCPSP. Section 3 generalizes the generic online algorithm proposed in [8] to accommodate endogenous observations. Section 4 shows how to instantiate the generic algorithm to the online S-RCPSP. Section 5 presents an improved version of the HCDP algorithm from [3]. Section 6 presents the one-step anticipatory algorithm and studies its behavior experimentally. Sections 7, 8, 9, and 10 describe the gap-reduction techniques. Section 11 presents the experimental results and Section 12 concludes the paper.

2 Online Stochastic Project Scheduling

This section describes the online Stochastic Resource-Constrained Project Scheduling Problem (S-RCPSP) from [3]. It starts with the offline (deterministic) problem, presents its stochastic and online versions, and illustrates the problem visually.

The Resource Constrained Project Scheduling. The RCPSP consists of a set of projects (jobs) that must be scheduled on a number of laboratories (machines). Each project consists of a sequence of experiments (activities) which are characterized by their durations and their costs. Each project brings a reward which depends on its completion time. The goal is to schedule the jobs to maximize revenues, i.e., the sum of the project rewards minus the sum of the activity costs. More formally, given a set of labs L, and a set of jobs J, a RCPSP instance ξ consists of a sequence of $n(j,\xi)$ activities $a_{1,j,\xi}, \ldots, a_{n(j,\xi),j,\xi}$ for each job $j \in J$. Activity $a_{i,j,\xi}$ has duration $d_{i,j,\xi}$ and cost $c_{i,j,\xi}$. The reward of project j is given by a function $f_J : \mathbb{N} \to \mathbb{R}$ which, given a completion time t of project j, returns its reward $f_j(t)$. A solution to a RCPSP instance ξ is a schedule σ, i.e., is a partial assignment of activities to labs and starting times ($\sigma : A \to L \times \mathbb{N}$). The schedule typically assigns only a subset of activities but satisfies the constraint that, if an activity is assigned to a lab at a given start time, all the preceding activities of the job must have been assigned to a lab and completed before the start time. The set of activities scheduled in σ is denoted by $dom(\sigma)$; We abuse notations and use $a_{i,j,\xi} \in \sigma$ instead of $a_{i,j,\xi} \in dom(\sigma)$. If $a \in \sigma$, we use $\sigma_s(a)$ to denote the start time of activity a in σ. A project j is scheduled in σ, denoted by $j \in \sigma$, if all its activities are scheduled in

σ and its completion time $ct(j, \sigma)$ is given by $\sigma_s(a_{n(j,\xi),j,\xi}) + d_{n(j,\xi),j,\xi}$. The objective value of a schedule is given by

$$f(\sigma, \xi) = \sum_{j \in \sigma} f_j(ct(j, \sigma)) - \sum_{a_{i,j,\xi} \in \sigma} c_{i,j,\xi}.$$

The S-RCPSP. The S-RCPSP has uncertainty regarding the durations, the costs, and the outcomes of activities. In particular, an activity can now fail, in which case the entire project fails. It may also succeed, in case the project is successful and completed. If the activity neither fails or succeeds, its status is "open". Activities whose outcome is a success or a failure have no successors. Formally, a S-RCPSP is specified by a probability distribution over the set Ξ of RCPSP scenarios. Each scenario $\xi \in \Xi$ specifies a RCPSP instance. Moreover, for each activity $a_{i,j,\xi}$, the scenario specifies an outcome $o_{i,j,\xi} \in \{success, fail, open\}$. A job j is a success in ξ, denoted by $success(j, \xi)$, if its sequence of activities is of the form

$$o_{1,j,\xi} = \ldots = o_{n(j,\xi)-1,j,\xi} = open \;\&\; o_{n(j,\xi),j,\xi} = success.$$

It is a failure otherwise, which means that its sequence is of the form

$$o_{1,j,\xi} = \ldots = o_{n(j,\xi)-1,j,\xi} = open \;\&\; o_{n(j,\xi),j,\xi} = failure.$$

The goal in the S-RCPSP is to find a schedule σ maximizing the objective

$$\mathbb{E}_\xi \left[\sum_{j \in \sigma : success(j,\xi)} f_j(ct(j, \sigma)) - \sum_{a_{i,j,\xi} \in \sigma} c_{i,j,\xi} \right].$$

In [3], the distribution of S-RCPSP scenarios is specified as follows. The number of jobs, labs, and the reward functions of all jobs are the same for all scenarios. The uncertainty on the sequence of activities of each job is modeled using a Markov chain. Each activity $a_{i,j}$ has a set R of potential realizations which are tuples of the form $\langle o_{i,j,r}, c_{i,j,r}, d_{i,j,r} \rangle$ specifying the outcome $o_{i,j,r}$, cost $c_{i,j,r}$, and duration $d_{i,j,r}$ of the activity. The probability to reach a given realization for an activity is conditioned on the realization of its preceding activity. More formally, a transition matrix $P_{i,j}$ defines the conditional probability $p_{i,j,r,r'}$ of activity $a_{i,j}$ having realization r given that activity $a_{i-1,j}$ has realization r', i.e.,

$$p_{i,j,r,r'} = Pr(\langle c_{i,j,\xi}, d_{i,j,\xi}, o_{i,j,\xi} \rangle = \langle c_{i,j,r}, d_{i,j,r}, o_{i,j,r} \rangle \mid \langle c_{i-1,j,\xi}, d_{i-1,j,\xi}, o_{i-1,j,\xi} \rangle = \langle c_{i-1,j,r'}, d_{i-1,j,r'}, o_{i-1,j,r'} \rangle)$$

Figure 2 illustrates such a Markov chain. In the figure, the failing activities are depicted in color, the costs are given inside the activities, and the durations are specified by the length of the tasks. The probability distributions are shown implicitly by the thickness of the transition arrows. For instance, the first activity has a low probability of having a realization with a cost of 400. However, if such a realization happens, it has then a high probability of having a second realization with a cost 250 and a rather long duration.

The Online S-RCPSP. In the online S-RCPSP, the decision maker alternates between scheduling activities and observing the uncertainty. Although the uncertainty about the

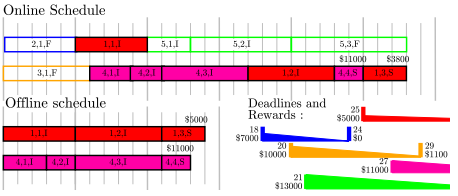


Fig. 1. An Example of Online and Offline Schedules for the S-RCPSP

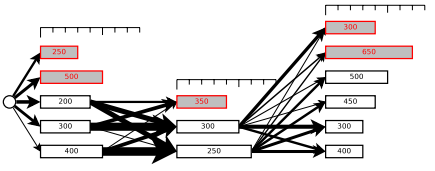


Fig. 2. A Markov Chain Describing the Uncertainty of a Job

projects and their activities is exogenous, the decision maker must schedule an activity to observe its realization, that is its duration, its cost, and its outcome. In particular, its outcome is revealed only when the activity is completed, at which time the decision maker also knows its duration and its cost. The online S-RCPSP is thus of a fundamentally different nature than the online stochastic optimization applications presented in [8]. Indeed, in these applications, the uncertainty is purely exogenous and is about which requests arrive and when: Once a request is placed, its information is fully revealed. In the online S-RCPSP, the decision maker must schedule an activity to reveal its uncertainty, which means that the observation is conditioned to a prior decision (thus it is endogenous). This poses some serious computational issues, partly due to the fact that activities may fail, in which case their project will incur a cost, take scheduling time, and bring no reward.

Illustration. Figure 1 illustrates the concepts visually. It depicts the reward functions of five jobs (bottom right of the figure). The reward f_j of each job is a constant before its first deadline d_j; it then decreases linearly until a second deadline after which it remains constant. For instance, the third job has a reward of 10,000 if it is completed before time 20 and the reward decreases linearly between 20 and 29 to reach 1,100 at the second deadline.

The bottom-left of the figure describes the clairvoyant schedule which has observed all the uncertainty. The solution schedules the first and the fourth job, which finish at times 14 and 13 and yield rewards of $5,000$ and $11,000$ respectively. The inside of each activity specifies the job number, the activity number, and the outcome. The top of the figure describes an online schedule. The online schedule includes activities of failing jobs 2, 3, and 5, with job 5 failing very late. These failed projects push the finish time of job 1 which only brings a reward of $3,800$. Of course, the value of the entire schedule further decreases by the cost of scheduling the activities of the failed projects.

3 The Generic Online Decision-Making Algorithm

Because of the endogenous nature of observations, the online generic algorithm presented in [8] must be generalized to accommodate the concept of observation explicitly. The new generic algorithm is depicted in Figure 3. It receives a decision-making agent $\mathcal{A}$ and a scenario ξ unknown to the decision maker and it maintains the current state of

```
ONLINEDECISIONMAKING(A, ξ)
1  s ← (0, ∅, ∅);
2  while true do
3    d ← A.decide(s);
4    if d = ⊥ then
5      return f(s, ξ);
6    s ← applyDecision(d, s);
7    s ← observe(s, ξ);
```

Fig. 3. The Generic Online Decision-Making Algorithm

decisions and observation s. As long as the decision maker does not decide to terminate (decision $\perp$ in line 4), the online algorithm calls the agent to obtain a decision d (line 3). The decision is applied to the state in line 6 and possible realizations of the uncertainty are observed in line 7. When the decision-maker terminates, the algorithm returns the value of the final state (line 5).

4 Instantiating The Online Decision-Making Algorithm

We now describe how to instantiate the states, the decisions, and the functions *applyDecision* and *observe* for the online S-RCPSP. The rest of the paper will then be concerned with how to make the decision in line 3.

States of the Online Decision-Making Algorithm. The states for the online S-RCPSP are triples $\langle t, \mathcal{C}, \mathcal{R}\rangle$, in which t represents the time, $\mathcal{C}$ the scheduling decisions whose activities have been completed and $\mathcal{R}$ the scheduling decisions whose activities are still running on the labs. The set $\mathcal{C}$ contains tuples of the form $\langle i, j, c_{i,j,\xi}, d_{i,j,\xi}, o_{i,j,\xi}, t_{i,j,\xi}, l_{i,j,\xi}\rangle$, specifying that completed activity $a_{i,j,\xi}$ has cost $c_{i,j,\xi}$, duration $d_{i,j,\xi}$, outcome $o_{i,j,\xi}$, and has been scheduled at time $t_{i,j,\xi}$ in lab $l_{i,j,\xi}$. The set $\mathcal{R}$ contains tuples of the form $\langle i, j, t_{i,j,\xi}, l_{i,j,\xi}\rangle$, specifying that running activity $a_{i,j,\xi}$ has been scheduled at time $t_{i,j,\xi}$ in lab $l_{i,j,\xi}$. For simplicity, we use $a_{i,j,\xi} \in \mathcal{C}$ to denote $\exists c, d, o, t, l : \langle i, j, c, d, o, t, l\rangle \in \mathcal{C}$ and use a similar notation for membership in $\mathcal{R}$. Finally, we use $f(s, \xi)$ to denote the objective value of a state s for scenario ξ.

Decisions. In a first approximation, there are only two types of decisions: scheduling a job in a lab and terminating. Scheduling a job j is feasible in state s, denoted by $Feasible(j, s)$, if none of its activities are currently running and if all its completed activities have open outcomes, i.e., $\neg\exists i : a_{i,j,\xi} \in \mathcal{R} \ \wedge \ \forall a_{i,j,\xi} \in \mathcal{C} : o_{i,j,\xi} = open$. The set of feasible scheduling decisions in state s consists of scheduling feasible jobs in some available lab, i.e.,

$$Feasible(s) = \{schedule(j, l) \mid j \in J \wedge Feasible(j, s) \wedge \neg\exists i, j, t : \langle i, j, t, l\rangle \in \mathcal{R}\}.$$

Terminating is the equivalent of rejecting all requests in request-based online applications and consists in renouncing to schedule all the remaining activities.

Applying a Decision. We are now in position to specify the function *applyDecision* which describes the effect of applying a decision in a state:

$$applyDecision(schedule(j,l),\langle t,\mathcal{C},\mathcal{R}\rangle) = \left\langle t,\mathcal{C},\mathcal{R} \cup \langle next(j,\mathcal{C}),j,t,l\rangle\right\rangle$$

where $next(j,\mathcal{C})$ denotes the next activity of job j to schedule. Scheduling a job on a lab simply inserts the next activity of the job on the lab.

Observations. It remains to specify the *observe* function which returns the next decision state. This happens whenever one of the running activities is completed. For a state s and a scenario ξ, this is given by $\mathcal{NT}(s,\xi) = \min_{a_{i,j,\xi}\in\mathcal{R}} t_{i,j,\xi} + d_{i,j,\xi}$. The completed activities, i.e., $Completed(s,\xi) = \{\langle i,j,t,l\rangle \in \mathcal{R} \mid t + d_{i,j,\xi} \leq \mathcal{NT}(s,\xi)\}$. must then be removed from the running set and transfer, with their observations, to the set of completed decisions, i.e.,

$$Backup(s,\xi) = \{\langle i,j,c_{i,j,\xi},d_{i,j,\xi},o_{i,j,\xi},t,l\rangle \mid \langle i,j,t,l\rangle \in Completed(s,\xi)\}$$

With this at our disposal, the *observe* function can be specified as

$$observe(\langle t,\mathcal{C},\mathcal{R}\rangle,\xi) = \langle \mathcal{NT}(s,\xi),\mathcal{C}\cup Backup(s,\xi),\mathcal{R}\setminus Completed(s,\xi)\rangle$$

We also use $\tau(s,d,\xi) = observe(applyDecision(d,s),\xi)$ to denote the transition obtained by taking decision d in state s and observing ξ in the resulting state.

5 Heuristically-Confined Dynamic Programming

The online S-RCPSP originated from [3] who also proposed an innovative solution technique to approach it: dynamic programming in a heuristically-confined state space (HCDP). Their approach is motivated by the fact that, on their instances, there are 10^9 possible scenarios. Combined with the inherent combinatorics of the offline problem itself, this would generate a gigantic state space, which would preclude the use of dynamic programming techniques.

To tackle this complexity issue, they propose a three-stage algorithm. In the first step, their algorithm applies a set H of heuristics on a set Ξ of scenarios to explore a number of reasonable trajectories in the state space. In the second step, these states are then merged to form a directed acyclic graph that defines a heuristically-confined state space. In the third step, the algorithm uses dynamic programming to obtain the best decision in this state space. The algorithm can be specified as an instantiation of the generic online algorithm as follows. Let $D(s,H,\Xi)$ be the set of decisions taken by the heuristics in H in state s for the set Ξ of scenarios during the first phase of the algorithm and let $C(s,\Xi)$ be the set of scenarios in Ξ compatible with state s, that is the set of scenarios ξ such that there exists a trajectory $s_0 \xrightarrow{d_0} s_1 \xrightarrow{d_1} \dots \xrightarrow{d_{t-1}} s_t = s$ satisfying $s_{i+1} = \tau(s_i,d_i,\xi)$ for all $i < t$. The HCDP policy value of decision d in state s for a set of scenarios Ξ and the set H of heuristics is given by

$$v(s,d,\Xi,H) = \frac{1}{\#C(s,\Xi)} \sum_{\xi\in C(s,\Xi)} Q(\tau(s,d,\xi),C(s,\Xi),H)$$

where the Q-value is defined as follows

$$Q(s, \Xi, H) = \begin{cases} \frac{1}{\#C(s,\Xi)} \sum_{\xi \in C(s,\Xi)} f(s, \xi) & \text{if } s \text{ is a leaf;} \\ \max_{d \in D(s,H,\Xi)} v(s, d, \Xi, H) & \text{otherwise.} \end{cases}$$

We specify the HCDP algorithm as an instance of the online generic algorithm:

HCDP.DECIDE(s)
1 $\Xi \leftarrow \{sample(s) \mid i \in 1..10,000\}$;
2 **return** $\text{argmax}_{d \in D(s,H,\Xi)} v(s, d, \Xi, H)$;

where $sample$ is conditional sampling procedure to generate scenarios of the future compatible with the observation in state s. This implementation is in fact an improvement over [3] because the heuristics and the dynamic program are run for every decision, instead of once at the beginning of the computation. The results improve significantly with this online implementation. Moreover, our actual implementation also uses the fact that the graph is acyclic to improve the runtime performance.

6 The One-Step Anticipatory Algorithm

We now study the use of one-step anticipatory algorithm for the online S-RCPSP. Anticipatory algorithms for online stochastic combinatorial optimization [8] make decisions by generating scenarios of the future, solving these scenarios optimally, and exploiting the resulting optimal solutions to select a decision. They typically use two black-boxes: (1) An optimization algorithm $\mathcal{O}(s, \xi)$ to solve the offline problem associated with state s and scenario ξ and (2) A conditional sampling procedure $sample(s)$ to generate scenarios of the future compatible with the observation in state s. In the S-RCPSP, the offline problem associated with a state s and scenario ξ is the scenario ξ with the additional constraints that all scheduling decisions in state s must be enforced. Note that the uncertainty is completely revealed in this offline problem: the costs and durations of the activities, as well as their outcomes, are known to $\mathcal{O}$. As a result, failed projects and their activities are never scheduled in their optimal solutions.

This paper focuses on the one-step anticipatory algorithm which solves a number of scenarios and selects the best decision with respect to these scenarios. This algorithm was initially proposed for exogenous uncertainty but generalizes naturally to those applications with endogenous observations. Its pseudo-code is depicted in Figure 4. We use the notation $\mathcal{O}^+(s, d, \xi) = \mathcal{O}(s, d, \xi) - f(s, \xi)$, where $\mathcal{O}(s, d, \xi) = \mathcal{O}(applyDecision(d, s), \xi)$, to denote the "future" value of the scenario when decision d is taken. The algorithm first collects the set of possible decisions (line 1) and initializes their scores (lines 2–3). It then generates m scenarios (lines 4–5), which are solved optimally (line 7) for each decision d, whose score is updated accordingly. The decision d with the best score is computed in line 8. The algorithm terminates (decision $\perp$) if the score of the best decision is not positive and returns the best decision otherwise.

This one-step anticipatory algorithm was analyzed for purely exogenous problems in [4]. It was shown that the expected loss of the anticipatory algorithm compared to the

A.DECIDE(s)
1 $D \leftarrow Feasible(s)$;
2 **for** $d \in D$ **do**
3 $score[d] \leftarrow 0$;
4 **for** $i \in 1..m$ **do**
5 $\xi \leftarrow sample(s)$;
6 **for** $d \in D$ **do**
7 $score[d] \leftarrow score[d] + \mathcal{O}^+(s, d, \xi)$;
8 $d \leftarrow \text{argmax}_{d \in D}\, score[d]$;
9 **if** $score[d] > 0$ **then return** d **else return** $\perp$;

Fig. 4. The Basic One-Step Anticipatory Algorithm

Table 1. Experimental Results of the One-Step Anticipatory Algorithm $\mathcal{A}$

	Agr	C2	C5	D.6	D1.5	Reg	P1	P2	P3	P4	R.6	R1.5	Avg
CV	14096	10806	4216	10062	13425	12418	17939	21242	28014	30051	7595	20394	15855
HCDP	12192	4218	0	6432	10333	7318	12638	16114	21657	24084	3571	13605	11013
$\mathcal{A}$	12731	4342	-6197	5712	10441	8115	14347	18602	26587	28851	3906	15096	11878

clairvoyant (i.e., the expected value of the offline problems) is bounded by the global anticipatory gap, which measures the stochasticity of the problem (instance + distribution) and a sampling error which can be arbitrarily small. Moreover, many applications in online routing, scheduling, and resource allocation were shown to have a small global anticipatory gap, explaining the excellent behavior of (approximations) of the one-step anticipatory algorithms. The anticipatory gap of a decision d in a state s is defined as

$$\Delta_g(s) = \mathbb{E}_\xi \left[\max_{d \in D} \mathcal{O}(s, d, \xi) \right] - \max_{d \in D} \mathbb{E}_\xi \left[\mathcal{O}(s, d, \xi) \right]$$

and measures the difference in expectation between being clairvoyant now and after the decision in state s. The global anticipatory gap for an algorithm is simply the sum of the local anticipatory gap for each successive state.

Table 1 gives the expected value $\mathbb{E}_\xi \left[O(s_0, \xi) \right]$ of the clairvoyant (CV) where all the uncertainty is revealed immediately, the expected value of HCDP, and the expected value of the one-step anticipatory algorithm ($\mathcal{A}$) with 200 scenarios (the implemetation and experimental setting is detailed in section 11). The results contain both good and bad news. On the one hand, the one-step anticipatory algorithm performs better in general and in average than the HCDP algorithm, showing the benefit of solving scenarios optimally. This is a very satisfying results, since it means that one-step anticipatory algorithms apply to applications with endogenous observations and outperforms the best method proposed for the online S-RCPSP. On the other hand, the loss of the anticipatory algorithm compared to the clairvoyant is quite substantial and may reach about 10,000 and 6,000 on instances C5 and C2 (These instances are described in detail later in the paper).

The distance between the anticipatory algorithm and the clairvoyant can be explained by the theoretical analysis in [4]. Indeed, Figure 5 depicts the evolution of the local anticipatory gap and the agreement degree over time. The circles in the figure give the

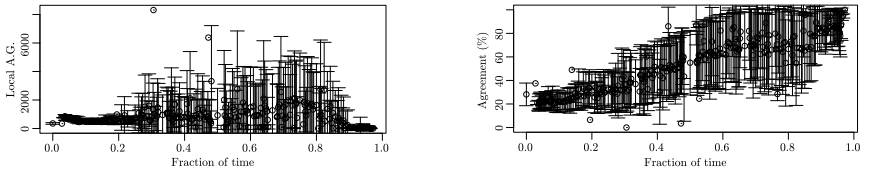


Fig. 5. Local Anticipatory Gap and Agreement Degree as a Function of the Decision Time

mean, while the intervals show one standard deviation around each side of the mean. The left part of Figure 5 shows a significant local anticipatory gap especially during the middle part of the execution. In the early part of the execution, the gap is small, because the algorithm has time to recover from a bad decision. The right part of Figure 5 depicts the agreement degree, i.e., the percentage of scenarios which admit the same optimal decision. Although this agreement is well above 70% in average in applications in routing, packet scheduling, and reservation systems, it is only 20% early in the execution and below 40% for a substantial part of the execution in the online S-RCPSP.

Why is the gap so large and the agreement so low in the online S-RCPSP? One of the main reasons is the endogenous nature of the observations. Indeed, the clairvoyant immediately sees which projects are valuable and does not spend time or incur costs scheduling them. The online algorithm in contrast must execute the project to determine their outcomes. Obviously, the one-step anticipatory algorithms extract from the scenarios which projects are promising, but they still have some significant probability to fail. This explanation is confirmed by instance P4 in which projects have a low probability of failure and only fail early. On this instance, the global loss is small, which directly means that the global anticipatory gap is small. Note also that this difficulty is not only due to the fact that projects may fail: A similar behavior occurs if some project takes an extremely long time. One may also wonder whether all online algorithms will exhibit so large a gap, but this is not the case. For instance, on instance C5, the optimal online policy (in the expected sense) consists of not scheduling any activity, since the expected value of all projects is negative. Yet the one-step anticipatory algorithm has an expected value of -6,197, showing that a significant portion of the gap is due to its behavior. The rest of this paper addresses how to enhance the one-step anticipatory to account for this gap.

7 Gap Reduction through Waiting

Waiting has been proposed for online stochastic vehicle routing (e.g., [2]) and was shown to produce significant improvements in solution quality. Its main benefit is to give the online algorithm more opportunity to observe the uncertainty, thus helping in taking more informed decisions. It is easy to integrate waiting in the online S-RCPSP: It suffices to schedule a dummy activity with no cost, no reward, and duration 1.

We can now show that waiting may be the optimal decision in some instances of the online S-RCPSP. Figure 6 shows a problem instance consisting of job 1 which succeeds and fails fifty percent of the time with respective durations of 5 and 10, as well as two other successful jobs. Job 2 has two activities of duration 2 and job 3 has one activity

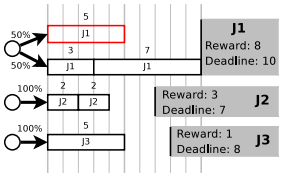


Fig. 6. An S-RCPSP Instance

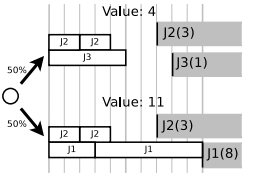


Fig. 7. Clairvoyant Solutions

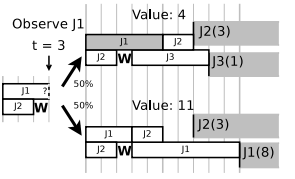


Fig. 8. The Optimal Online Policy with a Wait Action

of duration 5. The deadlines are strict: either the job finishes before its deadline and receives its reward, or it has no reward. The activities have no cost. Figure 7 shows the optimal solutions for both scenarios. Job 1 is not scheduled if it fails and the two other jobs yield their rewards for a total of 4. If job 1 succeeds, it yields its reward of 8 and there is enough room for job 2 which receives a reward of 3, giving a total reward of 11. Overall the expected value of the clairvoyant is thus $\frac{4+11}{2} = 7.5$.

Figure 8 depicts the best online policy which achieves an optimal expected value of 7.5 (the GAG is zero in this case). The policy consists in scheduling jobs 1 and 2 and then waiting one time unit to observe the status of job 1. If the first activity of job 1 is not finished at time 3, it will fail. The best decision then consists in scheduling job 3 then job 2. If the first activity of job 1 is completed at time 3, then the best decision is to schedule its second activity and job 2. By waiting one time unit, the online agent is able to observe the status of job 1 and to select the best schedule. Note that if the agent waits until job 1 finishes to take the next decision and that job fails, it does not have time to schedule job 3 and therefore is sub-optimal. Similarly, if the agent does not wait, it will have to choose between scheduling jobs 2 and 3, which is suboptimal.

8 Gap Reduction through Gap Correction

The one-step anticipatory algorithm uses the offline solution $\mathcal{O}(s, d, \xi)$ as a prediction of the optimal policy $\mathcal{A}^*(s, d, \xi)$ to evaluate each decision d in state s as shown in line 7 of the algorithm of Figure 4. Obviously, replacing $\mathcal{O}$ by $\mathcal{A}^*$ would produce an optimal decision. The basic idea in this section is to correct the evaluation $\mathcal{O}^+(s, d, \xi)$ by estimating the anticipatory gap in a state s: $gap(s, \xi) = \mathcal{O}^+(s, \xi) - \mathcal{A}^{*+}(s, \xi)$ which denotes the loss of the optimal online policy $\mathcal{A}^*$ compared to the clairvoyant on state s and scenario ξ. Note that the expected value of perfect information (EVPI), a fundamental concept in stochastic programming, is simply $EVPI(s) = \mathbb{E}_\xi [gap(s, \xi)]$. Evaluating $gap(s, \xi)$ is difficult however. On the one hand, $\mathcal{A}^*$ is not known: It is the optimal policy that we are trying to approximate. On the other hand, there are a gigantic number of states and scenarios in this problem. Our approach in this paper consists in evaluating the anticipatory gap on a training set and computing the best parameters of a model $\widetilde{gap}(s, \xi)$ approximating $gap(s, \xi)$. This is very natural, since stochastic optimization problems have a stochastic model of the uncertainty as part of their input.

Approximating the Gap using the First Decision. The first difficulty in learning the anticipatory gap can be addressed by learning the expected global loss, i.e., $EGL = \mathbb{E}_\xi \left[\mathcal{O}(\xi) - \mathcal{A}(\xi)\right]$, which provides an upper bound to the EVPI instead of the gap. The second difficulty is addressed by using a set Ξ of training scenarios and measuring

$$\widetilde{EGL} = \frac{1}{\#\Xi} \sum_{\xi \in \Xi} \mathcal{O}(\xi) - \mathcal{A}(\xi).$$

Then the anticipatory gap at state s for scenario ξ can be approximated by

$$\widetilde{Gap}(s, \xi) = \widetilde{EGL} \times \big(1 - CR(s, \xi)\big)$$

where $CR(s, \xi) = \frac{\#\mathcal{C} + \#\mathcal{R}}{\#\xi}$ denotes the completion ratio of s in scenario ξ. The anticipatory algorithm with gap correction $\mathcal{A}_{GC}$ is algorithm $\mathcal{A}$ in which line 7 becomes

$$score[d] \leftarrow score[d] + \mathcal{O}^+(s, d, \xi) - \widetilde{Gap}(applyDecision(s, d), \xi).$$

More Complex Gap Learning models. We also have investigated several finer models for gap learning. These models learn the gap in terms of the completion factors, the offline value of the scenario, and the set of successful jobs. The results were a disappointment as they produce no significant improvement over algorithm $\mathcal{A}$-GC.

9 Gap Reduction through Time Scaling

Although gap correction significantly improves the solution quality of the one-step anticipatory algorithm, if fails to address some of the consequences of the endogenous nature of observations. Indeed, a comparison between offline and online solutions reveals that the clairvoyant is often able to schedule an additional project. This is possible because the clairvoyant does not lose time scheduling failing projects. The online algorithm however needs to schedule them to determine whether they will be successful.

Gap correction is successful in detecting when not to schedule projects whose expected value is negative but is not particularly effective in differentiating potential scheduling decisions. This is due to the fact that the learning phase of gap correction uses algorithm $\mathcal{A}$ which has a low local anticipatory gap early in the search as depicted in Figure 5. This means that, whatever decision is taken at an early step, the clairvoyant has enough time subsequently to reach a high-quality solution, since it does not lose time scheduling unsuccessful projects.

Time scaling is another generic technique to reduce the anticipatory gap: It recognizes that algorithm $\mathcal{A}$ necessarily loses time scheduling activities of unsuccessful projects and compensates by scaling the durations in the offline problems.

Systematic Time Scaling. The simplest time scaling increases the duration globally by a common factor f^{-1} (we use $f < 1$), which, conceptually speaking, amounts to replacing the reward f_j for project j by

$$f_j^-(t) = f_j(\frac{t}{f}).$$

A more dynamic approach consists in scaling the remaining time only, i.e., after the decision time t_d of the current state $s = \langle t_d, \mathcal{C}, \mathcal{R} \rangle$, i.e.,

$$f_j^*(t) = \begin{cases} f_j(t) & \text{if } t < t_d \\ f_j(t_d + \frac{t - t_d}{f}) & \text{otherwise.} \end{cases}$$

Time Scaling by Job Effectiveness. The above proposal scales durations uniformly. It seems more appropriate however to apply scalings tailored to each of the jobs. To obtain such a differentiated scaling, we use job effectiveness, that is, the time spent on successful realizations of a job over the total time spent on the job. This measure can be learned offline like in gap correction and it gives ratios by which the durations should be scaled. Experimental results showed that this ratio was very low and led to drastic deadlines. Averaging the resulting with 1.0 (or equivalently dividing its distance to 1 by 2) led to much better results.

10 Gap Reduction by Problem Reduction

When studying the results of an online algorithm on a training set, another statistic can be gathered to boost the quality of the algorithm: the job performance. The performance of job j in a schedule σ for scenario ξ is simply $f_j(ct(j, \sigma)) - \sum_{a_{i,j,\xi} \in \sigma} c_{i,j,\xi}$ if j is successfully scheduled in σ and $-\sum_{a_{i,j,\xi} \in \sigma} c_{i,j,\xi}$ otherwise. Obviously removing a job from consideration in the offline problem will decrease the quality of the offline schedule and reduce the anticipatory gap. Moreover, if a job contributes a negative amount in expectation, or a small amount compared to the total reward, the gap reduction will not come at a high cost, since removing the job will not degrade the overall quality of the online algorithm. This is the strategy we experimented with in order to reduce the anticipatory gap: jobs yielding low performance (under a specific threshold like 1% or 5%) are discarded from the whole online policy.

11 Experimental Results

Table 2 gives a summary of the experimental results.

The Instances. The experimental results are based on the reference instance proposed in [3] and a number of derived instances to explore the stochasticity and combinatorial landscape on the online S-RCPSP. The derived instances are obtained by scaling multiple parameters of the instance: the activity costs or durations, the deadlines, and the job rewards. The structure of the instances was also changed by removing activity failures by levels: this is the case of instances P1–P4 which have increasingly fewer failures and whose failures occur earlier. One instance (P1) has no failures in the last activity in the jobs, while other instances (P2–P4) have no failures in the last two, three, or four levels (the latter has no failures at all since the longest jobs have four activities). Finally instance Agr averages the realizations to obtain only two realizations: one success and one failure. This reduces the number of realizations while roughly preserving the length, cost, and success distributions.

Table 2. Experimental Results on Gap Reduction Techniques

	Agr	C2	C5	D.6	D1.5	Reg	P1	P2	P3	P4	R.6	R1.5	Avg
CV	14096	10806	4216	10062	13425	12418	17939	21242	28014	30051	7595	20394	15855
HCDP	12192	4218	0	6432	10333	7318	12638	16114	21657	24084	3571	13605	11013
$\mathcal{A}$	**12731**	4342	-6197	5712	10441	8115	14347	18602	26587	28851	3906	15096	11878
$\mathcal{A}_w$	12414	3544	-3042	5760	9097	7606	13674	18542	**26808**	**28967**	3430	13855	11721
$\mathcal{A}_{GC}$	12696	4804	0	5588	10448	8146	14442	**18760**	26794	28956	3840	15000	12456
$\mathcal{A}_{TS}$	12681	4355	-2066	**7950**	10059	8281	13861	17392	23330	26031	**4451**	14461	11732
$\mathcal{A}_{TI}$	12444	4261	0	6688	9702	8036	14518	18755	26787	28961	3903	14670	12394
$\mathcal{A}_{TE}$	12608	4495	-1659	5467	10435	8337	**14563**	18728	26787	28919	3937	15385	12334
$\mathcal{A}_{TEPR}$	12612	**5375**	0	5737	**10708**	**8432**	14446	18474	26797	28918	4357	**15586**	**12620**

The Algorithms. The experimental results compare a variety of algorithms on the online S-RCPSP. They include the anytime Heuristically-Confined Dynamic Programming algorithm with 10,000 simulations per heuristic, the one-step anticipatory algorithm without ($\mathcal{A}$) and with ($\mathcal{A}_w$) waiting, the anticipatory algorithm with gap correction ($\mathcal{A}_{GC}$, the anticipatory algorithms with the three time-scaling approaches ($\mathcal{A}_{TS}$, $\mathcal{A}_{TI}$, $\mathcal{A}_{TE}$), and the hybrid algorithm combining time scaling by job effectiveness and job pruning ($\mathcal{A}_{TEJP}$). The systematic common scaling factor is 0.8 for $\mathcal{A}_{TS}$. All anticipatory algorithms have been run with 200 scenarios per decision and all learning has been performed on an independent set of scenarios. The results are the average over 1,000 scenarios. Note that using less scenarios decreases the quality of the solutions and introduces much more variability. Using more than 200 scenarios does not lead to significant improvements. The optimization solver used for the anticipatory algorithm is a dedicated branch and bound algorithm whose upper bound relaxes the resource constraints for the remaining tasks (implemented from scratch in C). Elastic relaxations were also tried but provided no additional computational benefits. This branch and bound is very fast and it takes on average less than 1ms for the reference instance.

Gap Reduction Through Waiting. The results about the waiting algorithm $\mathcal{A}_w$ are somewhat mixed since, in average, $\mathcal{A}_w$ produces solutions of slightly lower quality than $\mathcal{A}$. $\mathcal{A}_w$ improves instance C5 significantly, although the global loss on this instance is still significant. It also produces the best solutions on P3 and P4 which are the least stochastic problems. Why is waiting disappointing on the online S-RCPSP? The reason is once again the endogenous nature of observations. When waiting, algorithm $\mathcal{A}_w$ also observes the realization of any activity that algorithm $\mathcal{A}$ would have scheduled and only loses a single time unit for that observation. As a result, in the context of endogenous observations, waiting actually *increases* the anticipatory gap; the algorithm also has a strong tendency to wait, since the gap is larger for this decision. The wait decision gets favored for many scenarios as depicted in figure 9.

Gap Reduction Through Gap Correction. Algorithm $\mathcal{A}_{GC}$ returns better expected values than HCDP on all instances except D.6 and provides a 13% revenue improvement in average, which is quite significant. Gap correction is also very robust as it improves the solution quality of almost all instances. An examination of the traces of algorithm $\mathcal{A}_{GC}$ reveals its main benefits: It terminates schedules early because the overall expected

value of the projects is now negative thanks to the gap correction. It is highlighted on instances C2 and C5: In fact, $\mathcal{A}_{GC}$ now returns the optimal policy on C5. However, as mentioned earlier, gap correction is not effective in differentiating the decisions. This is highlighted on instance D.6 for which its solution quality decreases.

Gap Reduction Through Time Scaling. The static time-scaling algorithm $\mathcal{A}_{TI}$ whose factors are computed for each instance from the expected loss of algorithm $\mathcal{A}$ on the training scenarios is also an effective gap-reduction technique. It returns better expected values than HCDP on all instances except D1.5 (an instance where the deadlines are much looser) and provides a 12% revenue improvement in average, which is quite significant. In contrast to $\mathcal{A}_{GC}$, algorithms $\mathcal{A}_{TI}$ and $\mathcal{A}_{TS}$ are able to improve the solution quality of instance D.6 by removing sub-optimal jobs from consideration. Using job effectiveness is almost similarly effective and it is likely that, with a second learning phase, it would further improve. Scaling durations uniformly on all instances is not sufficient for improving solution quality as highlighted by the overall performance of $\mathcal{A}_{TS}$.

Combining Gap Reduction Techniques. The best algorithm in this experimental study is $\mathcal{A}_{TEPR}$, which combines time scaling by job effectiveness and problem reduction. It returns better expected values than HCDP on all instances except D.6 and provides an expected revenue improvement close to 15% over HCDP and of more than 6% over the one-step anticipatory algorithm.

The Benefits of Gap-Reduction Techniques. The results on the instances P1–P4 confirm the obvious intuition: the bigger the gap, the more effective the gap reduction techniques. In particular, on instances P3 and P4 which are the least stochastic, gap-reduction techniques cause a slight decrease in expected value. Only a fine tightening of the deadlines on P4 and a complex learning model for gap correction (i.e., learn a linear regression of $\mathcal{A}^+(s, d, \xi)$ with respect to $\mathcal{O}^+(s, d, \xi)$ at each depth of decision) managed to improve algorithm $\mathcal{A}$ slightly on this instance. More generally, gap correction, dynamic time scaling, and the hybridization of time scaling and job pruning are robust across all instances and provide significant benefits. None of them however uniformly dominates the others on all instances.

Running-time Comparison. An additional advantage of these gap-reduction techniques is that they do not increase the time of decision-making. Some require offline learning which took 1000 runs of algorithm $\mathcal{A}$. Figure 10 compares two anticipatory algorithms with HCDP in its online and the original version (OHCDP) whose quality is significantly worse. The results gives the time taken to solve 1000 instances of instance Reg. Algorithm OHCDP learns with 450,000 trajectories and the $\mathcal{A}_{\mathcal{TEPR}}$ learns with 1,000 scenarios. These results show that algorithms $\mathcal{A}$ and $\mathcal{A}_{\mathcal{TEPR}}$ outperform the HCDP class of algorithms both in expected value and performance.

Comparison with AMSAA. A companion paper presented another approach to reduce the anticipatory gap: the multi-step anticipatory algorithm AMSAA [5]. AMSAA is guaranteed to converge to the optimal policy, although the convergence result is mostly

$$\begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \text{Avg:} \\ \\ \text{Max} \end{array} \left(\begin{array}{|c|ccc} \hline 25650 & 25650 & 25200 & 25650 \\ 13150 & 13150 & 12750 & 13150 \\ 0 & -500 & -450 & -550 \\ 0 & -250 & -650 & -550 \\ 21750 & 21500 & 21500 & 21750 \\ 9050 & 8550 & 8650 & 9050 \\ 0 & -300 & -400 & -550 \\ 17228 & 18428 & 20194 & 16628 \\ 16250 & 16250 & 15650 & 15700 \\ 0 & -500 & -450 & -600 \\ \hline 10530 & 10289 & 10518 & 10369 \end{array} \right)$$

Fig. 9. Decision matrix in basic expectation, the anticipativity benefits the waiting decision (first column)

	$\mathcal{A}$	$\mathcal{A}_{TEPR}$	$HCDP$	$OHCDP$
Offline	–	303 s	–	772 s
Online	249 s	273 s	227 h	1 s
Total	249 s	576 s	227 h	773 s

Fig. 10. Comparison of Running Times for Solving 1000 scenarios

Table 3. Comparison of Algorithm $\mathcal{A}_{TEPR}$ with AMSAA

	Agr	Cost2	Cost5	D.6	D1.5	Reg	P1	P2	P3	P4	R.6	R1.5	Avg
$\mathcal{A}_{TEPR}$	12612	5375	0	5737	10708	8432	14446	18474	26797	28918	4357	15586	12620
AMSAA-ms	12166	4335	-3229	6143	10378	7856	14218	17879	23066	19058	3671	14502	10837
AMSAA-s	12754	4888	0	6893	10754	8452	14736	19007	26951	29099	4134	15525	12766

of theoretical interest. The following table reports the relative gap in percentage between AMSAA and $\mathcal{A}_{TEPR}$. We compare $\mathcal{A}_{TEPR}$ with AMSAA-31MS in which decisions are given 31ms, for a total time of 611s for 1,000 scenarios and AMSAA-32S which takes 91h to solve those instances.

Table 3 shows that $\mathcal{A}_{TEPR}$ is very competitive with AMSAA: it performs 14% better than AMSAA-31MS in average and is within 1% of the score of AMSAA-32S which cpu time is a factor 1000 greater than ours. On some instances, such as $Cost2$ and $R.6$, $\mathcal{A}_{TEPR}$ even significantly outperforms AMSAA-32S. Note that, on some instances such as $D.6$, $\mathcal{A}_{TEPR}$ has a much larger gap than AMSAA but $\mathcal{A}_{TS}$ in fact performs 15% better than AMSAA-32S on that instance.

12 Conclusion

This paper studied the performance of one-step anticipatory algorithms on the online S-RCPSP. This application is particularly challenging because of the endogenous nature of the observations that produces a significant anticipatory gap. Despite this difficulty, the paper showed that one-step anticipatory algorithms significantly outperform the state-of-art HCDP algorithm. The paper also studied a number of gap-reduction techniques, including waiting, gap correction, time scaling, problem reduction, and their hybridizations. It showed that waiting produces mixed results, typically increasing the anticipatory gap, and often postponing decision too eagerly. The remaining gap-reduction techniques produce significant improvements in solution quality over HCDP, the best algorithm reaching about 15% in average. Gap-reduction techniques are

particularly appropriate in settings in which decisions must be taken under severe time constraints as the gap-reduction techniques do not introduce significant overhead during execution.

References

1. Bent, R., Van Hentenryck, P.: Scenario-Based Planning for Partially Dynamic Vehicle Routing Problems with Stochastic Customers. Operations Research 52(6) (2004)
2. Bent, R., Van Hentenryck, P.: Waiting and Relocation Strategies in Online Stochastic Vehicle Routing. In: IJCAI 2007 (2007)
3. Choi, J., Realff, M., Lee, J.: Dynamic Programming in a Heuristically Confined State Space: A Stochastic Resource-Constrained Project Scheduling Application. Computers and Chemical Engineering 28(6-7), 1039–1058 (2004)
4. Mercier, L., Van Hentenryck, P.: Performance Analysis of Online Anticipatory Algorithms for Large Multistage Stochastic Programs. In: JCAI 2007 (2007)
5. Mercier, L., Van Hentenryck, P.: AMSAA: A Multistep Anticipatory Algorithm for Multistage Stochastic Combinatorial Optimization. In: CPAIOR (submitted, 2007)
6. Parkes, D., Duong, A.: An Ironing-Based Approach to Adaptive Online Mechanism Design in Single-Valued Domains. In: AAAI 2007, pp. 94–101 (2007)
7. Thomas, M., Szczerbicka, H.: Evaluating Online Scheduling Techniques in Uncertain Environments. In: The 3rd Multidisciplinary International Scheduling Conference (2007)
8. Van Hentenryck, P., Bent, R.: Online Stochastic Combinatorial Optimization. The MIT Press, Cambridge (2006)

Integrating Symmetry, Dominance, and Bound-and-Bound in a Multiple Knapsack Solver

Alex S. Fukunaga

Global Edge Institute, Tokyo Institute of Technology, Meguro, Tokyo, Japan
fukunaga@is.titech.ac.jp

Abstract. The multiple knapsack problem (MKP) is a classical combinatorial optimization problem. A recent algorithm for some classes of the MKP is bin-completion, a bin-oriented, branch-and-bound algorithm. In this paper, we propose path-symmetry and path-dominance, which are instances of the symmetry detection by dominance detection approach for pruning symmetric nodes in the MKP branch-and-bound search space. In addition, we integrate the "bound-and-bound" upper bound validation technique used in MKP solvers from the OR literature. We show experimentally that our new MKP solver, which integrates symmetry techniques from constraint programming and bound-and-bound techniques from operations research, significantly outperforms previous solvers on hard instances.

1 Introduction

Consider m containers (bins) with capacities $c_1, ..., c_m$, and a set of n items, where each item has a weight $w_1, ..., w_n$ and profit $p_1, ..., p_n$. Packing the items in the containers to maximize the total profit of the items, such that the sum of the item weights in each container does not exceed the container's capacity, and each item is assigned to at most one container is the *0-1 Multiple Knapsack Problem*, or MKP.

For example, suppose we have two bins with capacities $c_1 = 10, c_2 = 7$, and four items with weights 9,7,6,1 and profits 3,3,7,5. The optimal solution to this MKP instance is to assign items 1 and 4 to bin 1, and item 3 to bin 2, giving us a total profit of 15. Thus, the MKP is a natural generalization of the classical 0-1 Knapsack Problem to multiple containers.

Let the binary decision variable x_{ij} be 1 if item j is placed in container i, and 0 otherwise. Then the 0-1 MKP can be formulated as the integer program below, where constraint 2 encodes the capacity constraint for each container, and constraint 3 ensures that each item is assigned to at most one container.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 82–96, 2008.

$$\text{maximize} \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij} \tag{1}$$

$$\text{subject to:} \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \qquad i = 1, ..., m \tag{2}$$

$$\sum_{i=1}^{m} x_{ij} \leq 1, \qquad j = 1, ..., n \tag{3}$$

$$x_{ij} \in \{0, 1\} \qquad \forall i, j. \tag{4}$$

The MKP has numerous applications, including task allocation among autonomous agents, continuous double-call auctions [7], multiprocessor scheduling [9], vehicle/container loading [1], and the assignment of files to storage devices in order to maximize the number of files stored in the fastest storage devices [9]. A special case of the MKP where the profits of the items are equal to their weights, i.e., $p_j = w_j$ for all j is the *Multiple Subset-Sum Problem* (MSSP).

The MKP (including the special case of the MSSP) is strongly NP-complete.[1] Thus, state-of-the-art algorithms for finding optimal solutions are based on branch-and-bound. Previous work has shown that for problems where the ratio of items to bins is relatively small (i.e., $n/m < 4$), the state-of-the-art algorithm is bin-completion, a bin-oriented branch-and-bound algorithm [6].

The search space explored by bin-completion has many symmetric states. Previous work introduced some techniques for exploiting the symmetry and demonstrated their utility. In this paper, we further investigate methods for exploiting symmetries in the MKP bin-completion algorithm. We propose new techniques that result in significant improvements over the previous state of the art. These techniques are instances of the general symmetry breaking via dominance detection (SBDD) approach [2; 3].

A technique which is responsible for much of the power of previous branch-and-bound MKP solvers in the OR literature is "bound-and-bound" [10; 12], which seeks to prune nodes by heuristically seeking to validate the (optimistic) upper bound on the total profit at each search node. We integrated this technique into our extended bin-completion based MKP solver.

The paper is organized as follows. We start by reviewing the bin completion algorithm (Section 2). Section 3 defines the basic framework we use for symmetry detection and breaking, and reviews previous algorithms for exploiting symmetry in the MKP. We then introduce new, generalized symmetry detection techniques which are more powerful than the previous techniques. We discuss methods for combining various symmetry mechanisms, and compare these methods with related work on symmetry detection and breaking and in the constraint programming literature. We describe the bound-and-bound technique and our integration of bound-and-bound into bin-completion in Section 4. In Section 5,

[1] In contrast, the single-container 0-1 Knapsack problem is weakly NP-complete, and can be solved in pseudopolynomial time using dynamic programming.

```
1 search_MKP(bins, items, sumProfit)
2   if bins==∅ or items == ∅
3     if sumProfit > bestProfit then bestProfit = sumProfit; return
4   ri = reduce(bins,items) /* Pisinger's R2 reduction */
5   if ri ≠ ∅
6     search_MKP(bins, items \ ri, sumProfit)
7     return
8   upperBound = compute_upper_bound(items,bins)
9   if (sumProfit + upperB ≤ bestProfit
10    return /* upper-bound based pruning using SMKP bound */
11  if (validate_upper_bound(upperBound))
12    return /* bound-and-bound */
13  bin = choose_bin(bins)
14  undominatedAssignments = generate_undominated(items,capacity(bin))
15  foreach A ∈ sort_assignments(undominatedAssignments)
16    if not(symmetric(A))
17      assign A to bin
18      search_MKP(bins \ bin, items \ A, sumProfit+∑_{j∈A} p_j)
```

Fig. 1. Bin-completion-based algorithm for the MKP. The top-level call is `search_MKP(bins,items,0)`.

we experimentally evaluate various combinations of symmetry mechanisms, and conclude with a discussion of results and directions for future work.

2 Bin-Completion Algorithm for the MKP

Bin-completion is a branch-and-bound algorithm for finding optimal solutions to multi-container assignment problems including the MKP and bin packing problems [6]. We briefly describe this algorithm. **For simplicity of exposition, in the examples below, we assume (unless stated otherwise) multiple-subsets sum problem (MSSP) instances, where $\forall j, p_j = w_j$. Thus, whenever possible in the description below, we simply refer to an item by its weight.**

A *bin assignment* $B_i = (item_1, ..., item_k)$ is a set of all of the items that are assigned to a given bin i, $1 \leq i \leq m$. Thus, a valid solution to a MKP instance consists of a set of bin assignments, where each item appears in exactly one bin assignment. A bin assignment is *feasible* with respect to a given bin j if the sum of its weights does not exceed the capacity of the bin, c_i. Otherwise, the bin assignment is *infeasible*. We say that a bin assignment S is *maximal* with respect to bin i if S is feasible, and adding any other remaining items would make it infeasible.

The bin-completion algorithm searches a tree where each node at depth d, $1 \leq d \leq m$, represents a maximal, feasible bin assignment. The bin-completion algorithm for the MKP is shown in Figure 1, where each call to `search_MKP` corresponds to a node in the branch-and-bound search tree (e.g., Figure 2).

Nodes are pruned according to an upper bound which is based on a relaxation of the problem by Martello and Toth [11] (Line 8). Pisinger's R2 reduction procedure [12] is applied at each node (Line 4) in order to try to reduce the problem by eliminating some items for consideration. The `choose_bin` function (Line 13) selects the bin b with least remaining capacity. The `generate_undominated` function generates the set of all maximal, feasible assignments for b, with the additional constraint that these assignments are not dominated by any other assignment according to a *dominance criterion*. Given two feasible bin assignments F_1 and F_2, F_1 *dominates* F_2 if the value of the optimal solution which can be obtained by assigning F_1 to a bin is no worse than the value of the optimal solution that can be obtained by assigning F_2 to the same bin. Bin-completion prunes feasible assignments which are dominated according to the following MKP dominance criterion [6], which is based on the Martello-Toth dominance criterion for bin packing [11].

Proposition 1 (MKP Dominance Criterion). *Let A and B be two assignments that are feasible with respect to capacity c. A* dominates *B if B can be partitioned into i subsets $B_1, ..., B_i$ such that each subset B_k is mapped one-to-one to (but not necessarily onto) a_k, an element of A, and for all $k \leq i$, (1) the weight of a_k is greater than or equal to the sum of the item weights of the items in B_k, and (2) the profit of item a_k is greater than or equal to the sum of the profits of the items in B_k.*

The undominated bin assignments are sorted (Line 20) in order of non-decreasing cardinality, and ties are broken in order of non-increasing profit. The `symmetric` function (Line 21) applies one of the symmetry detection strategies described in this paper, and `validate_upper_bound` implements the bound-and-bound strategy described in Section 4. For example, given a bin with capacity 10 and items 9,8,7,3,2, the undominated, feasible bin assignments are (9),(8,2), and (7,3). It is possible for there to be a very large number of undominated bin assignments generated by `generate_undominated`, but this problem can be avoided by processing these in smaller batches, and the only thing we lose is part of the benefits of the value ordering (`sort_assignments`). This is called hybrid incremental branching, and details are in [6]. Figure 2 shows part of an example bin-completion search tree.

3 Exploiting Symmetry

To describe our symmetry breaking mechanisms, which are instances of the general SBDD approach [2; 3], we first introduce some notation and define the notion of a *nogood*, which is central to all of our symmetry exploitation methods.

Let B^d denote a bin assignment which assigns the elements of set B to a bin at depth d. Thus, $(10, 8, 2)^1$ and $(10, 7, 3)^1$ denote two possible bin assignments for a bin at depth 1.

Definition 1 (Nogood). *Let X^d be some node in the bin-completion search tree at depth d. Let $E^1, ..., E^{d-1}$ be ancestors of X^d at depths $1, ..., d-1$, respectively.*

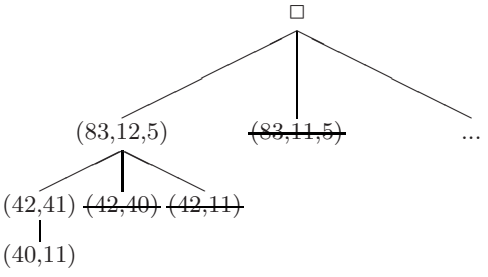


Fig. 2. Part of the bin-completion search tree for a MKP instance with capacity 100 and items with weights {83,42,41,40,12,11,5} ($\forall i, p_i = w_i$) Each node represents a maximal, feasible bin assignment Bin assignments shown with a ~~strikethrough~~, e.g., (83,11,5), are pruned because they are dominated according to the criterion in Proposition 1

For each such ancestor E_i, we say that every sibling of E^i to the left of E^i in the depth-first bin-completion search tree is a nogood *with respect to X^d.*

In Figure 3, $(8,2)^1$ is a nogood with respect to the descendants of $(7,4)^1$. Since bin-completion is a depth-first branch-and-bound algorithm, a nogood denotes a bin assignment (node) whose descendants have been exhaustively searched in the current search tree. The union of all current nogoods is a concise description of the entire portion of the search tree which has been searched so far. This is similar to the use of the term "nogood" in [4].

3.1 Path-Symmetry

Consider the search tree shown in Figure 3. Assume that the capacities for bins 1-4 are 11,11,12, and 10, respectively. Assume that we have already exhaustively searched the subtree under $(8,2)^1$, and we have generated the node $(7,4)^1,(10)^2,(8,3)^3,(6,2,2)^4$. By rearranging the items in bins 1-4, we can obtain a new set of bin assignments: $(8,2)^1,(7,3)^2,(10,2)^3,(6,4)^4$. This is a symmetric rearrangement, as the optimal solution under the first set of bin assignments is the same as the optimal solution under the latter set of assignments. Thus, we can prune the node at $(6,2,2)^4$.

More generally: Given a bin-completion search tree where we are considering a bin assignment for depth d, we define the *current path from depth g to depth d* as the union of bins g,$g+1$,...,d. The *current path items* are the union of all items in the current path. For example, in Figure 3, if we are at node $(6,2,2)^4$, the current path from depth 1 to 4 is the set of bins 1, 2, 3, and 4, and the current path items are $7,4,10,8,3,6,2,2$.

Definition 2 (Path-Symmetry). *Let N^g be a nogood with respect to a candidate bin assignment B^d, and let P be the current path items from depth g to d. we say that there is a* path-symmetry *with respect to nogood N^g if two conditions hold: (1) every item in N^g is a member of P, and (2) it is possible to (a) assign the items from the current path items corresponding to the items of N^g*

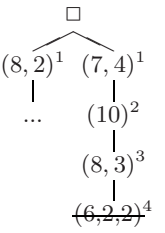


Fig. 3. The bin assignment $(6, 2, 2)^4$ can be pruned by Path-Symmetry. ($c_1 = 11, c_2 = 11, c_3 = 12, c_4 = 10$).

($Items(N^g) \subset P$) to bin g , and (b) assign the remaining items ($P \setminus Items(N^g)$) to bins $g + 1, ..., d$ such that all bins $g, ..., d$ are feasible.

If there is a path-symmetry between B^d and some nogood N^g as defined above, B^d can be pruned. The correctness follows directly from the definition of nogoods.

Checking the first condition of Definition 2 is straightforward. However, checking the second condition efficiently is not as straightforward, because it is essentially the decision version of a bin packing problem,[2] where we attempt to pack the items in $P \setminus Items(N^g)$ into bins with capacities $c_{g+1}, ..., c_d$. We describe several approaches:

In the first approach, we try to directly solve this bin packing problem using a simple backtracking algorithm (BT). The bin packing problem, like the MKP, is strongly NP-complete, and in the worst case, BT will take time which is $O(n^m)$, where n is the number of items and m is the number of bins. It is possible to avoid backtracking and use a standard bin packing heuristic such as first-fit decreasing (FFD), which has a polynomial complexity. Thus our second approach uses FFD to pack the items $P \setminus Items(N^g)$ into bins $g+1, ..., d$. The drawback of heuristics such as FFD is that it is not guaranteed to find a packing of the items into the bins even if one exists. However the symmetry check is still admissible – path-symmetry using a FFD check to test condition (2) may sometimes fail to prune a node that a BT check would have pruned, but will never prune a node that a BT check will not prune.

Another way to approximate the full check for condition (2) for path-symmetry is to limit the set of items that can be swapped among the bins. That is, instead of repacking all of the items $P \setminus Items(N^g)$ into bins $g + 1, ..., d$, we can "lock" some of the items into their current bins and only consider packing the unlocked items. We consider a *limited* packing problem (as opposed to the *full* packing problem without locked items) where we (a) assign the items from the current path items corresponding to the items of $N^g (Items(N^g) \subset P)$ to bin g, and (b) pack the items $P \setminus Items(N^g)$ into bins $g + 1, ..., d$, but in contrast to the full

[2] In the decision version of bin packing, we are given m bins and n items, and the problem is to determine whether all n items can be packed into m bins such that the capacity constraints on all of the bins are not violated.

packing problem, we lock all of the items in $P \backslash Items(N^g)$ except for the items in bin g. In Fig. 3, the unlocked items would be the 7 and 4 from bin 1. The limited packing problem is to pack the 7 and 4 into three bins: bin #2 with remaining capacity 1 (the 10 is locked), bin #3 with remaining capacity 9 (the 8 is moved to bin #1, the original capacity is $c_3 = 12$, and there is a 3 which is locked, so the remaining capacity is 12-3=9), and bin #4 with remaining capacity 2 (one of items with weight 2 has moved to bin 1, but the remaining 6 and 2 are locked). In this case, the packing fails, so limited packing is insufficient, but a full packing (where all current path items were unlocked) would have enabled path symmetry detection. The choice of BT vs. FFD, and the choice of full vs. limited packing are orthogonal choices. Thus, full packing using BT will give us the full pruning power of path-symmetry (albeit at highest cost per node), while limited packing using FFD gives us a weaker (but cheaper) pruning test.

A more restricted version of this test was previously considered in [6]: Given a bin assignment B^d for the bin at depth d, we can prune B^d if there is a nogood N^g with respect to B^d such that (1) B^d includes all the items in N^g, and (2) if we swap the items in N^g from B^d with the items that are currently assigned to the bin at depth g, both resulting bin assignments are feasible. We call this strategy *2-swap-path-symmetry*, because it only considers symmetries that can be detected by swapping items between two particular bins.

3.2 Path-Dominance

Path-dominance is a generalization of path-symmetry. Consider the search tree shown in Figure 4 for an instance where the bin capacities for bins 1-3 are 11, 12, and 13, respectively. Assume that we have already exhaustively searched the subtree under $(8,2)^1$, and we have generated the current path in the search tree, $(7,4)^1, (5,6)^2, (9,2)^3$. By rearranging the items in bins 1-3, we can obtain a new set of bin assignments: $(7,2)^1, (5,6)^2, (9,4)^3$. This is a symmetric rearrangement, since the optimal solution under the first sequence of bin assignments must be the same as the optimal solution the latter sequence of assignments. Thus, we can prune the node $(9,2)^3$, since $(8,2)^1$ dominates $(7,2)^1$. More generally:

Definition 3 (Path-Dominance). *Let N^g be a nogood with respect to candidate bin assignment B^d, and let P be the current path items from depth g to d. We say that there is a* path-dominance symmetry *with respect to nogood N^g*

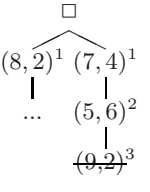


Fig. 4. The bin assignment $(9,2)^3$ can be pruned by Path-Dominance ($c_1 = 11, c_2 = 12, c_3 = 13$)

established at depth g if there exists some $s \subset P$ such two conditions hold: (1) s is dominated by N^g according to the MKP dominance criterion and (2) it is possible to (a) assign s to bin g, and (b) assign the remaining items $(P \setminus s)$ to bins $g+1, ..., d$ such that all bins $g, ..., d$ are feasible.

If there is a path-dominance symmetry between B^d and some nogood N^g as defined above, B^d can be pruned. This follows from the definition nogoods and Proposition 1.

Our current implementation of path-dominance works as follows. We enumerate subsets of the current path items such that each such subset s is dominated by N^g and is maximal, i.e., there is no other item which can be packed into the N^g. For each such s, we test whether condition (2) of the path-dominance symmetry definition (Definition 3 is satisfied. If so, then a path-dominance has been detected, so the current node can be pruned. The test for condition (2) is the same as the corresponding test for path-symmetry in the previous section. Thus, the same four implementations of the check are possible: (a) full packing with BT, (b) full packing with FFD, (c) limited packing with BT, and (d) limited packing with FFD. In the worst case, this check is executed for each subset s that satisfies condition (1) of Definition 3, so checking for path-dominance can be quite expensive.

The following, highly restricted form of Path-Dominance was proposed by Fukunaga and Korf [6]. Given a bin assignment B^d for depth d, we can prune B^d if there is a nogood N^g with respect to B^d such that (1) N^g dominates B according to the MKP dominance criterion (Proposition 1), and (2) The items in B^d can be swapped with the current items in bin g, such that the resulting bin assignments are both feasible. In other words, this is a restricted Path-Dominance test where all bins are frozen except for the bin at depth d. We call this strategy *2-swap-path-dominance*.

3.3 Combining Symmetry Breaking Strategies

We have defined a spectrum of symmetry-breaking techniques above, ranging from the weakest, 2-swap-path-symmetry, to the strongest, full path-dominance with BT. Path-dominance, using the full packing with backtracking implementation, clearly subsumes all of the other criteria. For example, every node which can be pruned by path symmetry will also be pruned by path-dominance (but not vice versa). However, there is a trade-off between the amount of pruning enabled by a symmetry relation and the amount of overhead incurred at each node in order to detect the symmetry. To alleviate this trade-off, we combine the strategies by *chaining* a set of tests so that the cheapest, least powerful symmetry is applied first. If this prunes the node, then the cost of applying the more powerful (but costly) symmetries is not incurred. However, if the node is not pruned, then we apply another, more powerful symmetry, and so on.

A preliminary study presented at a workshop reported results on 11 different configurations of symmetry-checking tests [5]. While we have found that path-symmetry (including 2-swap-path-symmetry) and 2-swap-path-dominance are

relatively efficient and often offer a favorable trade-off between search reduction and increased cost per node, we have not yet found a way to reduce the cost of the more powerful variants (full/limited path dominance using either backtracking or FFD) sufficiently to justify their use. The specific configurations used this paper are described in Section 5.

3.4 Relationship to Previous Work on Symmetry Detection

Our MKP symmetry breaking mechanisms are domain-specific instances of the symmetry breaking via dominance detection (SBDD) approach [2; 3]. A significant difference is that in addition to detecting equivalences to previously explored subtrees (2-swap-path-symmetry and path-symmetry), our 2-swap-path-dominance and path-dominance algorithms also detect partial solutions which are dominated by previously explored subtrees (according to Proposition 1).

Our work is also similar to the pruning technique proposed by Focacci and Shaw [4] for constraint programming, which was applied to the TSP with time windows. Both methods attempt to prune the search by proving that the current node at depth j, which represents a partial j-variable (bin[3]) solution x, is dominated by some previously explored i-variable (bin) partial solution (nogood bin assignment) q, where $i < j$.

The main difference between our method and Focacci and Shaw's method is the approach used to test for dominance. Focacci and Shaw's method extends q to a j-variable partial solution q' which dominates x. They apply a local search procedure to find the extension q'. In contrast, our methods start with a partial, j-bin solution x and try to transform it to a partial solution x' such that $\bar{x}'_i$, the subset of x' including the first i bins, is dominated by the i-bin partial solution q. We do this by transforming (via item swaps) the contents of bins $i, i+1, ..., j$ in x to derive a feasible partial solution x' such that $\bar{x}'_i$ is dominated by q.

4 Bound and Bound

A powerful technique for solving the MKP is *bound-and-bound*, which was originally implemented in Martello and Toth's MTM solver for the MKP [10]. In standard branch-and-bound, an upper bound U is computed at each node in the search tree. If $U \leq L$, L, where L is a lower bound, e.g., the best (highest) objective function score found so far by branch-and-bound, then exploring the node further is futile, so the node can be pruned. On the other hand, if $U > L$, then standard branch-and-bound does not prune the node. Bound-and-bound extends this by applying some heuristic technique to attempt to *validate* the upper bound: When $U > L$, bound-and-bound attempts to prove that the upper bound U can be achieved somehow in the current subtree – if so, then we have found the value of the optimal subsolution under the current node and can backtrack.

The most powerful implementation of this idea is in Pisinger's Mulknap solver [12]. Mulknap is an item-oriented branch-and-bound algorithm. The items are

[3] Our analogues of CP variables and values are bins and bin assignments, respectively.

ordered according to non-increasing *efficiency* (ratio of profit to weight), so that the next item selected by the variable-ordering heuristic for the item-oriented branch-and-bound is the item with highest efficiency that was assigned to at least one container by a greedy bound-and-bound procedure (see below). The branches assign the selected item to each of the containers, in order of non-decreasing remaining capacity.

At each node, an upper bound is computed using a relaxation of the MKP called the *surrogate relaxed MKP* (SMKP), which is obtained by combining all of the remaining m containers in the MKP into a single container with aggregate capacity $C = \sum_{i=1}^{m} c_i$, resulting in the single-container, 0-1 knapsack problem: where the items are the remaining items and the knapsack has the capacity of the aggregate container. The SMKP, which is currently the most effective upper bound for the MKP [8], can be solved by applying any algorithm for optimally solving the 0-1 Knapsack problem.

At each node, Mulknap attempts to validate the SMKP upper bound by showing that there exists a partition of the SMKP 0-1 Knapsack solution into the remaining empty spaces in the m bins of the original MKP instance. This is done by solving a series of m subset-sum problems which allocate the items from the SMKP solution to each bin, minimizing the unused capacity in each bin (without exceeding capacity). If this partition is successful then the SMKP upper bound can be achieved by partitioning the SMKP solution into the remaining spaces in the bins, so we have validated the upper bound possible under the current branch-and-bound node (and thus, we can backtrack).

Bound-and-bound can be extremely powerful for solving the MKP. In fact, for many random benchmarks with a relatively large ratio of items to bins ($n/m > 5$), bound-and-bound can often validate the SMKP upper bound at the root node of the search tree, which means that the instance is solved at the root node *without requiring any branch-and-bound search.*

We implemented Pisinger's bound-and-bound mechanism into our bin-completion solver: at each node, we attempt to validate the SMKP upper bound by partitioning the SMKP solution into the remaining bins (recall that in bin-completion, at depth b, $m-b$ bins are empty). Our implementation of the SMKP bound is a straightforward, primal branch-and-bound. Our implementation of the splitting procedure uses a standard branch-and-bound procedure using the max-cardinality bound [8].

5 Experimental Results

We compared the following bin-completion based MKP solver configurations:

- **PureBC:** bin completion with no symmetry checking and no bound-and-bound.
- **2-Dom:** Apply 2-swap-path-symmetry first, and if the node is not pruned, then try applying 2-swap-path-dominance. This corresponds to the "Bin-completion with nogood dominance pruning" algorithm reported in [6].

- **PathSym:** First, try 2-swap-path-symmetry, then try 2-swap-path-dominance, and finally, apply path-symmetry, using the limited packing with FFD implementation described above.
- **2-Dom-BB:** Same as 2-Dom, with bound-and-bound.
- **PathSym-BB:** Same as PathSym, with bound-and-bound.

All of our algorithms were implemented in Common Lisp and compiled using the CMUCL compiler version 19d. In addition, we also compared our algorithms with Pisinger's Mulknap algorithm (using Pisinger's C implementation, compiled using `gcc` version 4.12 with the `-O3` option.

We evaluated the various solver configurations using the following four standard classes of problems from the MKP literature.

- *uncorrelated instances*, where the profits p_j and weights w_j are uniformly distributed in $[min, max]$.
- *weakly correlated instances*, where the w_j are uniformly distributed in [min,max] and the p_j are randomly distributed in $[w_j - (max - min)/10, w_j + (max - min)/10]$ such that $p_j \geq 1$,
- *strongly correlated instances*, where the w_j are uniformly distributed in [min,max] and $p_j = w_j + (max - min)/10$, and
- *multiple subset-sum instances*, where the w_j are uniformly distributed in $[min, max]$ and $p_j = w_j$.

In our experiments, $min = 1$, $max = 1000$. The first $m - 1$ bin capacities c_i were uniformly distributed in $[0.4 \sum_{j=1}^{n} w_j/m,\ 0.6 \sum_{j=1}^{n} w_j/m]$ for $1 \leq i < m$. The last capacity c_m is chosen as $c_m = 0.5 \sum_{j=1}^{n} w_j - \sum_{i=1}^{m-1} c_i$ to ensure that the sum of the capacities is half of the total weight sum. Degenerate instances were discarded as in Pisinger's experiments [12].

We used instances where the ratio of items to bins (n/m) ranged from 2 to 10. This is because for $n/m \geq 10$, Mulknap frequently finds a solution at the root node by succeeding in validating the SMKP upper bound with the subset-sum based bound-and-bound. For example, we generated 1000 instances each of the uncorrelated, weakly-correlated, strongly-correlated, and multiple subset-sum instances with 10 bins and 100 items, where $[min, max] = [1,1000]$. Mulknap solved all 4000 instances at the root node (i.e., without search) in less than 0.01 seconds per instance (see [12] for related results). On the other hand, for $n/m \leq 5$, the bound-and-bound at the root node usually fails, and Mulknap is forced to branch. It is therefore the instances with smaller n/m ratios that are in some sense the most difficult random MKP instances that can be generated using the model described above, so we focus on these problems.

The results are shown in Table 1. All experiments were run on a 2.4 GHz Intel Core2 Duo. Each experiment was run on 20 instances per (# bins, # items) pair (all configurations were run on the same instances), so a total of 480 instances were used. The *fail* column indicates the number of instances (out of 20) that were not solved within the time limit (300 seconds/instance). The *time* and *nodes* show average time spent and nodes searched on the successful runs, excluding

the failed runs. Thus, in the experiments where there were timeouts, the fail column is the most significant result.

There are several clear trends in the results. First, symmetry-based pruning is most effective for low n/m, and becomes less effective for high n/m. For $n/m < 5$, the variants that use some form of symmetry (2-Dom, 2-Dom-BB,PathSym, PathSym-BB) clearly search significantly less nodes than PureBC, and using less runtime. The only exception was for uncorrelated 12-bin, 48-item instances. For $n/m \geq 5$, the savings in nodes searched is insufficient to offset the cost of symmetry-based pruning. The % of nodes pruned due to symmetry techniques is highest for less correlated instances. This is because the dominance criterion is most powerful when item weights and profits are highly correlated, which means that most candidate bin assignments are pruned by the dominance criterion during `generate_dominated` (Fig 2, line 14), and are never considered.

Second, bound-and-bound becomes more effective as n/m increases, and the overhead associated with bound-and-bound *decreases* as n/m increases. For $n/m = 2$ (30-bin, 60-item instances), the overhead of bound-and-bound is sufficiently large enough that there is a significant performance degradation in 2-Dom-BB and PathSym-BB compared to 2-Dom and PathSym, respectively. However, for larger values of n/m, the relative overhead of bound-and-bound becomes less significant, and for $n/m \geq 5$, bound-and-bound is significantly enhancing the performance of the bin-completion variants.

The search behavior of Mulknap and bin-completion variants with bound and bound (2-Dom-BB and PathSym-BB) are similar when $n/m \geq 5$. In principle, when Mulknap can solve a problem at the root node without search, the bin-completion variants should also solve the same problem at the root node. Below the root node, the search behaviors of Mulknap and bin-completion with bound-and-bound can diverge, because Mulknap branches on individual items, using a variable ordering based on decreasing item efficiency (p/w ratio), while bin-completion is branching on undominated bin assignments, where the variable ordering is based on minimal cardinality, using profit as a tie-breaker.

The performance differences between Mulknap and our 2-Dom-BB/PathSym-BB variants on the strongly-correlated and multiple subset-sum instances for 10 bins/60 items, and 10 bins/100 items can be explained by a differences in the implementation of the 0-1 Knapsack solver used to compute the SMKP (lower bound) solution. There are cases where there exist multiple optimal solutions to the SMKP 0-1 Knapsack instance, all with the same total profit, but with differing total weight (such cases more common for multiple-subset sum instances and strongly correlated instances). Mulknap implements a specialized 0-1 Knapsack solver which is biased to find solutions with the smallest weight sum (which makes it more likely that the solution is splittable by the bound-and-bound subset sum solver) Our current 0-1 Knapsack solver did not implement this bias, and as a consequence, missed opportunities to successfully apply bound-and-bound. Thus, Mulknap performed significantly better than 2-Dom-BB and PathSym-BB for the strongly-correlated and multiple subset-sum instances for $n/m \geq 5$, even though in principle (with a better implementation of the SMKP 0-1 Knapsack

Table 1. Comparison on random instances for $2 \leq n/m \leq 10$. Item weights were in [1,1000]. The *fail* column indicates the number of instances (out of 20) that were not solved within the time limit (300 seconds/instance). The *time* and *nodes* show average runtimes and nodes searched on the successful runs, excluding the failed runs.

Uncorrelated Instances																		
	30 bins, 60 items			15 bins, 45 items			12 bins, 48 items			15 bins, 75 items			10 bins, 60 items			10 bins, 100 items		
	fail	time	nodes	fail	time	nodes	fail	time	nodes	fail	time	nodes	fail	time	nodes	fail	time	nodes
Mulknap	20	n/a	n/a	20	n/a	n/a	10	42.95	1202516	13	0.01	3	0	1.84	41271	**0**	< 0.01	**1**
PureBC	20	n/a	n/a	14	125.38	17593990	**3**	**38.69**	**4135959**	3	41.61	754025	0	21.30	415663	20	n/a	n/a
2-Dom	10	91.49	4180505	10	62.20	3725747	3	61.29	1930296	4	72.29	600277	0	53.34	414249	20	n/a	n/a
2-Dom-BB	12	79.987	2384741	10	74.36	3723605	3	62.84	1893539	2	6.65	38421	**0**	**1.04**	**7981**	**0**	< 0.01	**1**
PathSym	**3**	**40.30**	**752990**	**8**	**31.44**	**908677**	3	48.36	572037	4	78.07	556327	0	56.63	395254	20	n/a	n/a
PathSym-BB	4	37.16	549704	8	36.28	907531	3	48.89	555786	**2**	**6.50**	**35907**	0	1.11	7553	**0**	< 0.01	**1**
Weakly Correlated Instances																		
	30 bins, 60 items			15 bins, 45 items			12 bins, 48 items			15 bins, 75 items			10 bins, 60 items			10 bins, 100 items		
	fail	time	nodes	fail	time	nodes	fail	time	nodes	fail	time	nodes	fail	time	nodes	fail	time	nodes
Mulknap	20	n/a	n/a	20	n/a	n/a	20	n/a	n/a	20	n/a	n/a	11	116.11	2011539	**0**	< 0.01	**1**
PureBC	20	n/a	n/a	13	110.76	5008825	5	83.08	4596503	**19**	**278.44**	**10554776**	1	46.50	770159	20	n/a	n/a
2-Dom	6	70.26	1628331	9	65.27	2043285	4	79.95	2337020	20	n/a	n/a	2	59.96	650387	20	n/a	n/a
2-Dom-BB	7	72.14	1418034	9	71.29	2043123	4	83.14	2335930	20	n/a	n/a	1	37.08	546095	0	< 0.01	69
PathSym	**1**	**52.31**	**822188**	**6**	**57.67**	**1387753**	**4**	**43.36**	**799555**	19	297.59	2388087	2	56.86	367994	20	n/a	n/a
PathSym-BB	1	52.58	614972	6	62.53	1387625	4	45.12	798571	19	297.13	2349921	**1**	**33.38**	**276420**	0	< 0.01	69
Strongly Correlated Instances																		
	30 bins, 60 items			15 bins, 45 items			12 bins, 48 items			15 bins, 75 items			10 bins, 60 items			10 bins, 100 items		
	fail	time	nodes	fail	time	nodes	fail	time	nodes	fail	time	nodes	fail	time	nodes	fail	time	nodes
Mulknap	20	n/a	n/a	20	n/a	n/a	3	97.50	962223	0	14.28	137495	**0**	< 0.01	**1**	**0**	< 0.01	**1**
PureBC	17	129.07	2918108	5	62.61	1026434	2	31.39	609882	1	39.52	219995	1	45.65	251584	20	n/a	n/a
2-Dom	1	31.01	440718	3	36.93	676154	0	48.23	660830	1	59.52	219858	1	69.74	251495	20	n/a	n/a
2-Dom-BB	1	39.23	440718	3	41.29	676014	1	37.09	540183	0	4.61	17560	0	0.59	2850	3	< 0.01	1
PathSym	**0**	**11.65**	**143886**	**1**	**46.26**	**798791**	**0**	**29.23**	**321255**	1	62.41	218049	1	71.51	249179	20	n/a	n/a
PathSym-BB	0	15.20	143886	2	36.13	537265	0	30.37	318512	**0**	**4.68**	**17543**	0	0.61	2846	3	< 0.01	1
Multiple Subset-Sum Instances																		
	30 bins, 60 items			15 bins, 45 items			12 bins, 48 items			15 bins, 75 items			10 bins, 60 items			10 bins, 100 items		
	fail	time	nodes	fail	time	nodes	fail	time	nodes	fail	time	nodes	fail	time	nodes	fail	time	nodes
Mulknap	20	n/a	n/a	14	111.11	783653	2	25.52	168455	**0**	**0.01**	**2**	**0**	0.01	**1**	**0**	< 0.01	**1**
PureBC	16	122.46	2715571	4	34.54	989428	0	4.20	106243	0	7.51	18050	0	14.11	24830	15	0.54	10
2-Dom	1	9.40	225208	2	20.41	532674	0	3.13	64775	0	8.91	17993	0	16.60	24803	15	0.54	10
2-Dom-BB	1	11.48	225208	2	21.67	532671	0	3.13	64764	0	5.83	17993	0	6.19	24339	14	0.36	8
PathSym	**0**	**5.51**	**106551**	**1**	**27.92**	**565196**	**0**	**2.82**	**45671**	0	8.82	17902	0	16.81	24668	15	0.55	10
PathSym-BB	0	6.88	106551	1	29.91	565193	0	2.85	45660	0	5.76	17902	0	6.27	24213	14	0.36	8

solver), the performances should have been identical, as both algorithms could have solved all of these instances at the root node.

To highlight the performance differences between the various symmetry pruning techniques, we describe some experiments with smaller problem instances, where all bin-completion based configurations were more likely to find a solution within the time limit. For uncorrelated instances with 10 bins, 30 items, PureBC solves all instances in an average of 9.13 seconds and 1,662,504 nodes. In comparison, 2-Dom solves all instances in 0.57 seconds and 47,193 nodes, and PathSym solves all instances in 0.28 seconds and 9,432 nodes. Thus, 2-Dom and PathSym are searching 2 and 3 orders of magnitudes fewer nodes than PureBC, respectively. Finally, we consider another configuration, *PathDom*, which first applies the same sequence of symmetry tests as PathSym, and finally applies the full path-dominance test using backtracking – thus, PathDom applies our most powerful pruning criterion and searches the fewest number of nodes. PathDom solves all of these instances in 0.78 seconds and 5031 nodes. Thus, exploiting the most powerful dominance criterion can yield almost another factor of 2 reduction in nodes searched for these instances, but the additional cost per node results in an overall 3x slowdown. We have not found any configuration using path dominance (other than the highly restricted 2-Dom case) where the search reduction was sufficient to offset the additional cost per node.

Overall, PathSym significantly reduced the size of the branch-and-bound tree compared to 2-Dom, the previous state of the art [6] algorithm for MKP problems with low n/m ratios. The results in Table 1 show that exploiting symmetry is a very effective technique for hard MKP instances with low n/m ratio. Furthermore, integrating bound-and-bound was shown to significantly improve performance on instances with higher n/m ratios, while modestly penalizing performance on instances with lower n/m ratios. Thus, the PathSym-BB configuration, which successfully integrates bin-completion, symmetry-based pruning (a combination of 2-swap-path-symmetry, 2-swap-path-dominance and path-symmetry), and Pisinger's bound-and-bound technique, can be considered a new, state-of-the-art algorithm for instances for low n/m ratios.

6 Conclusions

This paper presented an algorithm for the multiple knapsack problem which integrates techniques from constraint programming (symmetry-based pruning), operations research (bound-and-bound, as well as the SMKP upper bound and other techniques borrowed from Mulknap and earlier MKP solvers from the OR literature), and the AI literature (the bin-completion search space [6]). We proposed two new, symmetry breaking mechanisms (path symmetry and path dominance) which are generalizations of previously studied strategies (2-swap-path-symmetry and 2-swap-path-dominance). We showed that integrating path-symmetry resulted in a new solver which significantly outperformed the previous state of the art, 2-swap dominance based bin-completion solver reported in [6].

We further showed that integrating bound-and-bound could significantly improve the performance on problems with higher n/m ratios.

There are several directions for future work. Although path-dominance is our most powerful symmetry relation, the current implementation is not competitive with path symmetry due to the large overhead incurred at each node. We are currently investigating improved implementations and approximate detection strategies to make path-dominance more viable. Likewise, our current implementation of bound-and-bound uses naive branch-and-bound algorithms for the SMKP upper bound and subset sum computation for bound validation. As discussed in Section 5, integration with more sophisticated algorithms is likely to result in significant performance improvements. Finally, the symmetry detection techniques described in this paper are not limited to the MKP. For example, it is straightforward to apply the symmetry techniques to improve the search efficiency of any of the bin-completion based solvers for bin packing, bin covering, and min-cost covering problems described in [6].

References

1. Eilon, S., Christofides, N.: The loading problem. Management Science 17(5), 259–268 (1971)
2. Fahle, T., Schamberger, S., Sellmann, M.: Symmetry breaking. In: Proceedings of the International Conference on Constraint Programming, pp. 93–107 (2001)
3. Focacci, F., Milano, M.: Global cut framework for removing symmetries. In: Proceedings of the International Conference on Constraint Programming, pp. 77–92 (2001)
4. Focacci, F., Shaw, P.: Pruning sub-optimal search branches using local search. In: Proc. CPAIOR, pp. 181–189 (2002)
5. Fukunaga, A.: Exploiting symmetry in multiple knapsack problems. In: Proc. Seventh International Workshop on symmetry and constraint satisfaction problems, pp. 39–46 (2007)
6. Fukunaga, A., Korf, R.: Bin-completion algorithms for multicontainer packing, knapsack, and covering problems. Journal of Artificial Intelligence Research 28, 393–429 (2007)
7. Kalagnanam, J.R., Davenport, A.J., Lee, H.S.: Computational aspects of clearing continuous call double auctions with assignment constraints and indivisible demand. Electronic Commerce Research 1, 221–238 (2001)
8. Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack Problems. Springer, Heidelberg (2004)
9. Labbé, M., Laporte, G., Martello, S.: Upper bounds and algorithms for the maximum cardinality bin packing problem. European Journal of Operational Research 149, 490–498 (2003)
10. Martello, S., Toth, P.: A bound and bound algorithm for the zero-one multiple knapsack problem. Discrete Applied Mathematics 3, 275–288 (1981)
11. Martello, S., Toth, P.: Knapsack problems: algorithms and computer implementations. John Wiley & Sons, Chichester (1990)
12. Pisinger, D.: An exact algorithm for large multiple knapsack problems. European Journal of Operational Research 114, 528–541 (1999)

Cost Propagation – Numerical Propagation for Optimization Problems

Birgit Grohe and Dag Wedelin*

Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden

Abstract. We investigate *cost propagation* for solving combinatorial optimization problems with finite domain variables, expressed as an additive component model. Cost propagation combines ideas from both constraint programming and integer programming into a single approach, where problems are iteratively solved by numerical propagation, with updates for single constraint terms in the component model.

We outline a theory of propagation in terms of equivalent problems with notions of consistency, local optimality, convergence and bounds. We define several different updates and analyze their properties.

Finally, we outline computational experiments on the simple assignment problem, the set partitioning problem, and a crossword puzzle. The experiments illustrate that even without a top level search, cost propagation can by itself solve non-trivial problems, and also be attractive compared to standard methods.

1 Introduction

Combinatorial optimization problems are often modelled as integer linear programming (ILP) problems and solved with standard ILP methods. This works well for many problems, but the model cannot easily handle non-linear cost functions and constraints that may require many variables and constraints. With constraint programming (CP), many kinds of constraints can be modelled in a constraint satisfaction problem (CSP), and the structure of the problem can be exploited with efficient propagation algorithms. On the other hand, it is non-trivial to model an objective function or costs in general.

Cost propagation can be described as a numerical optimization method with a CP-like structure for the model and the computations. Building on the previous work of Wedelin [14,15,17], the contribution of this paper consists of a formalization and extension of the theory (in particular non-conflicting updates are new), as well as some computational experiments. For more difficult problems, cost propagation may be combined with search, or can be used heuristically to ensure convergence and integer solutions (see [15]).

Our starting point is a *component model* (in [1] also called *nonserial unconstrained problems*)

$$\max_x f(x) = \sum_k g_k(x^k) \tag{1}$$

* Corresponding author.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 97–111, 2008.

where x is an array of finite domain variables and where the terms $g_k(x^k) \in \mathbb{R}$ are distinct arbitrary functions over subsets x^k of x. Let these terms be called the *components* of the objective function and let them be expressed e.g. as tables. We distiguish between *variable-components* with only one variable and *constraint–components* with two or more variables. We can use binary variables to efficiently represent discrete finite domain variables, so we can without loss of generality restrict the theory to this case. An upper bound can be obtained for any component model by simply adding the largest value of each component. We call this the *component bound*.

A CSP can be described by modelling constraints as constraint-components with the values 0 and $-\infty$ to express feasible and infeasible assignments, and we can say that a solution is *feasible* if it has an objective value greater than $-\infty$, and *infeasible* otherwise. A binary ILP can be translated into our model by additionally introducing a variable-component for each variable and its cost. For example, the problem

$$\max \{\, 2x_1 + 3x_2 + 2x_3 \;\mid\; x_1 + x_2 = 1,\; x_2 + x_3 = 1,\; x_j \text{ binary} \} \tag{2}$$

can be expressed as follows:

$$\max_{x \in \{0,1\}^3} g_1(x_1) + g_2(x_2) + g_3(x_3) + g_4(x_1, x_2) + g_5(x_2, x_3),$$

where the components have values as shown in figure 1.

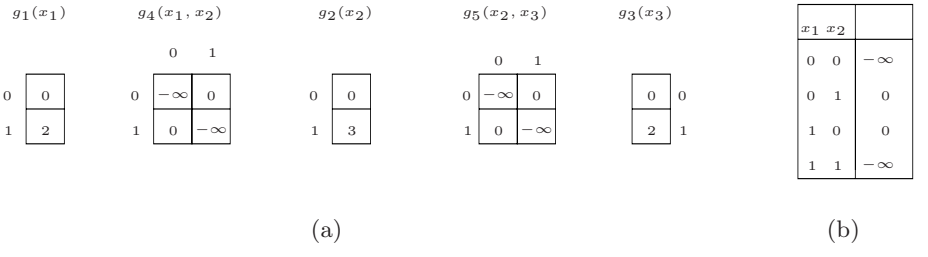


Fig. 1. (a) Component cost tables for the ILP example. (b) Alternative representation of $g_4(x_1, x_2)$.

A possible problem with this representation is that explicit tables grow exponentially with the number of variables in a constraint. However, for many common constraints the components can be represented implicitly, and as in CP it is possible to design efficient specialized update algorithms.

Any component model can in principle be translated into a binary ILP by introducing additional binary variables for each feasible entry in the component tables and linear constraints for consistency between all the variables. The resulting ILP is linear in the size of an explicit representation of the component model. While mostly of theoretical interest, such a translation can be used to link properties of this approach with well known results in optimization.

To illustrate the relationship between cost propagation and propagation in CP we will express the standard CP domain reduction operation (that we call the *CP-update*) numerically. Let $g_1(x_1), g_2(x_2), g_3(x_3)$ be variable–components representing the current domains for three finite domain variables, x_1, x_2, and x_3. Let $g_4(x_1, x_2, x_3)$ be a constraint-component representing a constraint over these variables. Now when T and F is represented by 0 and $-\infty$, the logical operator AND can be represented by addition and OR by maximization. The CP-update can then be described as follows:

1. Input to the update are the current domains represented by the components $g_1(x_1), g_2(x_2), g_3(x_3)$.
2. Combine the constraint and the current domains to form an intermediate table
$$h(x_1, x_2, x_3) = g_1(x_1) + g_2(x_2) + g_3(x_3) + g_4(x_1, x_2, x_3).$$
3. Calculate updated variable-components $g'_1(x_1), g'_2(x_2), g'_3(x_3)$ by maximizing the intermediate table over the other variables:
$$\begin{aligned} g'_1(x_1) &= \max_{x_2,x_3} h(x_1, x_2, x_3), \\ g'_2(x_2) &= \max_{x_1,x_3} h(x_1, x_2, x_3), \\ g'_3(x_3) &= \max_{x_1,x_2} h(x_1, x_2, x_3). \end{aligned}$$

The updated variable-components will show which variable values are feasible (component value 0) and infeasible (component value $-\infty$), just as for the CP-update.

So, from the perspective of CP, cost propagation can be seen as an exploration of a richer set of models and methods by using arbitrary numbers rather than just 0 and $-\infty$. There are within CP a number of approaches to using the objective function as a constraint, and different methods to use cost information and optimization in filtering algorithms, see e.g. [11,3,12,6,7]. It is our understanding however, that the propagation between constraints through the variable domains is still non-numerical and none of these approaches actually try to find the optimal solution to the problem with the propagation mechanism itself. Our approach is much closer to the kind of propagation used in AI for inference in graphical models or belief networks, although there the goal is not optimization but to calculate marginal probabilities, see e.g. [10,14]. Another source of inspiration is relaxation labeling, see e.g. [5].

From the perspective of ILP, we can in principle translate the entire component model into an ILP, where cost propagation can be seen as an iterative dual decomposition method. However, depending on the kind of update used, it may or may not be described in terms of maximization of the dual. Some characteristic properties of our approach are that we start from the component model (1), adopting a CP-style approach in defining the problem and the subproblems, we rely on constraint level update algorithms and propagation rather than on an LP-solver, and we treat the variable values 0 and 1 symmetrically.

Finally, the component model (1) is investigated by Bertelè and Brioschi [1] using nonserial dynamic programming. This kind of model is also commonly approached by dynamic programming algorithms such as the Viterbi algorithm [13] for Markov chains.

For space reasons, proofs and some derivations have been omitted, and the computational experiments are sketched. For a more complete account, we kindly refer the reader to [4].

2 Cost Propagation

Propagation will proceed as an iterative sequence of updates, considering one constraint-component at a time, with the aim to find an optimal solution, improve the bound and reduce domains. This is done by iteratively changing the values of the components, which are seen as variables in a computer program. Any changes to the components must preserve equivalence, i.e. the sum of all components in the problem must always be equal to the original objective function $f(x)$.

At any time, we have a current solution determined from the variable-components, defined as follows:

Definition 1. *The current solution is the solution to the trivial problem*

$$\max_x \sum_j g_j(x_j). \tag{3}$$

The normal case is that the current solution is unique, otherwise we consider it as undefined.

Note that this is the same as maximizing each variable-component separately.

We also need to define subproblems consisting of a single constraint-component and its associated variable-components:

Definition 2. *Let the subproblem with respect to the constraint–component $g_i(x^i)$ be the following maximization problem:*

$$\max_x g_i(x^i) + \sum_{j \in v(i)} g_j(x_j). \tag{4}$$

Here, $v(i)$ is the index set of variables involved in constraint–component $g_i(x^i)$, but it is a formal matter if the sum is extended to all variables. Figure 2 (a) shows an example of the components involved in a two-variable subproblem.

Components will be changed iteratively by *cost updates* defined as follows:

Definition 3. *A cost update for a subproblem is a transformation into an equivalent problem by changing only the components of the subproblem.*

For example, figure 2 (a) illustrates a subproblem with two variables. In figure 2 (b) the variable-components have been *moved in*, i.e. added to the constraint-component (the values of the variable-components are then 0 to preserve equivalence). The solution to the subproblem is found by identifying the maximum

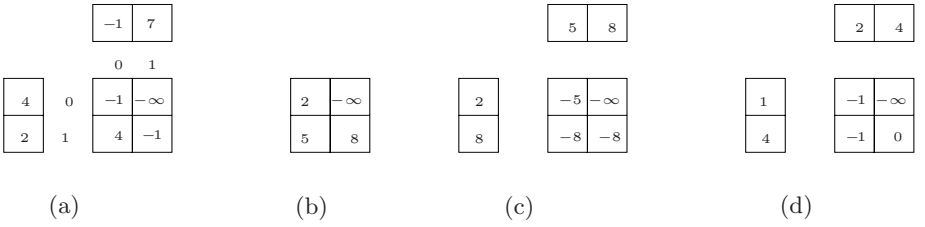


Fig. 2. (a) Original subproblem, (b) Intermediate table with the sum of all subproblem components, (c) After DP-update, (d) A non-conflicting update

value of this table. In figure 2 (c) and (d) new values for the variable-components have been *moved out*, i.e. subtracted from (b). Thus all cases (a)-(d) preserve equivalence.

Updates can be designed in different ways, e.g. to give the results in figure 2 (c) or (d). To characterize two different desirable properties of updates we will use the following definitions. The first is our version[1] of generalized arc-consistency in CP (see e.g. [2]):

Definition 4. *A variable-component is consistent with a subproblem if its feasible values can be extended to a feasible solution of the subproblem, i.e. with an objective value greater than* $-\infty$*. An update is consistent if it makes the variable-components of the subproblem consistent.*

Thus, a consistent update will place $-\infty$ in the variable-component if a value is determined to be infeasible. For optimization the following property is important:

Definition 5. *The current solution is locally optimal for a subproblem if it is an optimal solution to the subproblem. An update is locally optimal if it makes the current solution locally optimal. (If there are several solutions to the subproblem, this should be reflected by ties in the affected variable domains.)*

For example, in figure 2 (a) the current solution is not locally optimal, but in 2 (c) and (d) it is.

2.1 Cost Updates

We will now describe two different updates. The first one is a generalization of the CP-update which we call the *DP-update* because of its connection to dynamic programming, see Section 2.2.

Before giving a general definition we will illustrate the DP-update with the simple example of figure 2 (a)-(c). The first step of the DP-update is to move in the variable components into the constraint component, resulting in 2 (b). The new variable components are then computed as the maximum values of the rows and columns of 2 (b). Finally, the new variable components are moved

[1] We have changed the definition compared to [4].

out from the constraint component, resulting in 2 (c). After the update, the current solution will be both consistent and locally optimal with respect to this subproblem.

In the following general definition, let g denote the current components, and g' the updated components. The table $h(x^i)$ can be seen as an intermediate table holding the data after the variable components have been moved in.

Definition 6. *For a subproblem, the following operations define the DP-update:*

$$h(x^i) = g_i(x^i) + \sum_{j \in v(i)} g_j(x_j), \tag{5}$$

$$g'_j(x_j) = \max_{x \setminus x_j} h(x^i), \tag{6}$$

$$g'_i(x^i) = g_i(x^i) - \sum_{j \in v(i)} (g'_j(x_j) - g_j(x_j)). \tag{7}$$

The second update will be an example of an update for a component corresponding to the single linear constraint $ax = 1$ (we will call this a *set partitioning* constraint), where the subproblem can be described as the optimization problem

$$\max\{cx \mid ax = 1, x \text{ binary}\}, \tag{8}$$

and where the vector a is also binary. According to LP-theory, the vector c can be replaced by $\bar{c} = c - ya$ for any y without changing the optimal solution. If we want to find an y so that the current solution is feasible, y can be chosen in the interval $l \leq y \leq h$ where h and l are the largest and the second largest entries of the vector $\bar{c}$. By choosing $y = \frac{1}{2}(l + h)$, we receive the LP-update defined below. We here identify c with the *cost differences* $G_j = g_j(1) - g_j(0)$.

Definition 7. *For a subproblem, the following operations define the LP-update:*

$$g'_j(1) = g_j(1) - \tfrac{1}{2}(l + h), \quad g'_j(0) = g_j(0), \tag{9}$$

$$g'_i(x^i) = g_i(x^i) - \sum_{j \in v(i)} (g'_j(x_j) - g_j(x_j)), \tag{10}$$

where the values h and l are the largest and the second largest among the values $g_j(1)$ in the subproblem.

This update is consistent and locally optimal.

Efficient implementations of the DP- and LP-updates may perform actual calculations quite differently from the descriptions above. First, explicit use of large tables should be avoided, especially for constraints with many variables, where these tables may become exponentially large. Secondly, it is always sufficient

to use the cost differences G_j, although the full variable-components $g_j(x_j)$ are useful for the theory. To express the LP-update with the G_j only, we write (9) as

$$G'_j = G_j - \frac{1}{2}(l + h), \tag{11}$$

where h and l are the largest and the second largest among the values G_j. The values l and h can be obtained in linear time, and an explicit use of $g_i(x^i)$ is avoided. For a fully efficient update, the explicit change of the constraint-component in (10) needs to be avoided. Instead, we can implicitly remember the change of constraint-component i by keeping the differences $S^i_j = G'_j - G_j$ corresponding to the second part of equation (10).

To see how also the DP-update for the set partitioning constraint can be calulated using the cost differences, let m be the index of the variable with the cost h. The DP-update, i.e. equations (5) and (6) can then be written as

$$\begin{aligned} G'_m &= G_m - l, \\ G'_j &= G_j - h \quad \text{for } j \neq m. \end{aligned} \tag{12}$$

As in the case of the LP-update, explicit computation of (7) can be avoided by remembering the values S^i_j.

We conclude that for this particular constraint both the DP- and the LP-update can be performed with a fast and simple linear time algorithm.

2.2 Propagation with Cost Updates

We now consider propagation to find solutions to an entire problem. Here, the two updates presented in the previous section have very different properties. For the example (2), we show in table 1 the first few steps of propagation with the DP-update and the LP-update. First constraint 1 is updated, then constraint 2, and this is repeated.

We can see that for the DP-update the variable-components $g_j(x_j)$ converge after only two iterations (as do the constraint-components which are not shown

Table 1. Example of propagation using the DP-update and the LP-update

DP	$g_1(x_1)$		$g_2(x_2)$		$g_3(x_3)$	
	0	1	0	1	0	1
	0	2	0	3	0	2
constr 1	3	2	2	3	0	2
constr 2	3	2	4	3	3	4
constr 1	3	4	4	3	3	4
constr 2	3	4	4	3	3	4
constr 1	3	4	4	3	3	4
constr 2	3	4	4	3	3	4

LP	$g_1(x_1)$		$g_2(x_2)$		$g_3(x_3)$	
	0	1	0	1	0	1
	0	2	0	3	0	2
constr 1	0	−0.5	0	0.5	0	2
constr 2	0	−0.5	0	−0.75	0	0.75
constr 1	0	0.125	0	−0.125	0	0.75
constr 2	0	0.125	0	−0.4375	0	0.4375
constr 1	0	0.28125	0	−0.28125	0	0.4375
constr 2	0	0.28125	0	−0.359375	0	0.359375

in the table). In the LP-update the numbers are different and do not converge in the same way. However, after a few iterations, there are no more *sign changes* of the values $g_j(1)$, and consequently of the G_j, so the current solution has converged, and it is also optimal. We note that it is possible to consider the cost differences G_j instead of both values in the variable-components $g_j(x_j)$.

Depending on the kind of update used, the analysis of the iterative propagation process varies significantly. Note that while any locally optimal update gives the same current solution from the same numerical input, propagation with different updates will typically give different current solutions along the way (the example is an exception because it is so simple). We here restrict ourselves to some observations.

For locally optimal updates, the following theorem can be used to detect feasibility:

Theorem 1. *The current solution is feasible if it is unique, i.e. if all $G_j \neq 0$, and if every subproblem has been updated with a locally optimal update at least once without changing the current solution.*

With respect to convergence, it is common for locally optimal updates that if the situation in Theorem 1 has occurred, the current solution has converged and will not change during subsequent updates, although the numbers in the components continue to change. For more difficult problems a common situation is however that the current solution oscillates for some variables where numerically the differences G_j tend towards 0. For the LP-update, Theorem 1 in [15] provides a partial explanation for this kind of convergence problems:

Theorem 2. *A dual vector y such that the Lagrangian relaxation of the corresponding ILP problem gives a unique solution to the LP-relaxation exists if and only if the LP-relaxation has a unique integer solution.*

In other words, the existence of a unique integer solution is a necessary condition for finding a solution with the LP-update. An oscillating behavior may also be caused by infeasibility of the problem.

Turning to the question of optimality, one cannot generally expect that a solution that is locally optimal for every subproblem is also globally optimal. However, there are cases where optimality can be guaranteed. For example, the LP-update is designed to never give a suboptimal feasible solution, a fact that follows from LP-theory, see also section 3.

We finally discuss propagation with the DP-update, which can be interpreted as a dynamic programming algorithm if the *constraint graph* (hypergraph representing the sets x^i) of the problem is acyclic. From [15] we have the following result (in an adapted version):

Theorem 3. *If the constraint graph is acyclic, propagation with the DP-update converges to the optimal solution in d steps, where d is the diameter of the graph.*

All components have then converged numerically. For more details see [15].

If the constraint graph contains cycles, then the DP-update does not generally give the optimal solution, and convergence cannot be guaranteed. The reason is

that the numerical values of a variable-component may then influence themselves through a cycle. However, the behavior is dependent on the numerical values of the components and in some cases, feasible, good or even optimal results can be achieved despite of cycles, see [17] for a discussion. We note that the CP-update, as a special case of the DP-update, works independently of cycles in the constraint graph, since once a variable-component entry becomes $-\infty$, it will never change again.

3 Non-conflicting Updates

We will now investigate a particular kind of update that under certain conditions can guarantee globally optimal solutions.

Definition 8. *The current solution is non-conflicting if it corresponds to a largest entry in the constraint–component table. An update is non-conflicting if it makes the current solution non-conflicting. (If there are several solutions to the subproblem, this should be reflected by ties in the affected variable domains.)*

The definition is illustrated in figure 2 (d). For both the DP-update of 2 (c) and the non-conflicting update of 2 (d), the current solution is locally optimal for the subproblem. But additionally, also the constraint-component has the largest value for the optimal combination, which is not the case for the DP-update in 2 (c).

A non-conflicting solution is clearly always locally optimal, since the current solution will maximize the value of each term in (4).

The following theorem describes the relation between a non-conflicting solution and global optimality:

Theorem 4. *If the current solution is unique and non-conflicting with all subproblems, then the current solution is globally optimal.*

Global optimality can be detected during propagation using the theorem below.

Theorem 5. *The current solution is optimal if it is unique, and if each subproblem has been updated with a non-conflicting update at least once without changing the current solution.*

Theorems 4 and 5 do not imply that propagation with non-conflicting updates will find a solution, just that a solution, if found, will have certain properties.

3.1 Analysis of Non-conflicting Updates

We begin with the relationship between a non-conflicting update and the component bound:

Theorem 6. *A non-conflicting update minimizes the component bound for the subproblem, so that the bound becomes equal to the solution to the subproblem. In propagation, the component bound is therefore non-increasing in every step.*

For example, in figure 2 (a) the component bound for the subproblem is 13, in (c) it is 11, and in (d) it is 8, i.e. the same as the optimal solution to the subproblem. We note that this aspect of non-conflicting updates is easily explained in a LP interpretation, where the bound, i.e. the dual objective, is locally minimized.

From the definition and the examples of figure 2 (c) and (d) one can see that the non-conflicting property limits how much may be moved out from each constraint. A full analysis of this is complicated and we restrict ourselves to a few observations. It is here relevant to analyze differences directly, so let $D_j = g'_j(1) - g'_j(0)$ be the differences given by the DP-update in equation (6). For example, in figure 2 (b), $D_1 = 6$ and $D_2 = 3$. Assume without loss of generality that the D_j are all non-negative and in decreasing order, which can be achieved by permuting variables and their values appropriately. Further let T_j be the corresponding differences in $g'_j(x_j)$ for a non-conflicting update. Our first observation is that

$$T_j \leq D_j \,, \tag{13}$$

i.e. that each D_j is an upper bound for the respective T_j of the non-conflicting update, since moving out a difference greater than D_j would destroy the non-conflicting property. Thus, we can in a non-conflicting update always move out D_j to *one* of the the variables, but depending on $h(x^i)$ all other T_j may then have to be 0 to keep the update non-conflicting. For example in figure 2, setting $T_1 = D_1 = 6$ would force $T_2 = 0$. By reasoning along these lines, one can draw the conclusion that the following inequalities ensure that the non-conflicting property is satisfied (see [4] for details):

$$\begin{gathered} \sum_{k \geq j} T_k \leq D_j \,, \\ T_j \geq 0 \end{gathered} \tag{14}$$

These inequalites are weak, in the sense that for a particular constraint it may very well be possible to move out more, but it is the best that can be accomplished by using the D_j as the only source of information about $h(x^i)$.

3.2 The Fractional DP-Update

We are still left with a number of design decisions, since there are many ways to choose the T_j in a non-conflicting update. We wish to achieve $T_j > 0$, to ensure a unique current solution. Also, it is intuitively desirable to move out as much as possible from the constraint–components to the variable–components. Finally, the update should be easy to compute. One way of doing this is by letting T_j be proportional to D_j:

Definition 9. *The fractional DP-update is given by*

$$T_j = \alpha D_j, \quad \alpha > 0. \tag{15}$$

This fractional DP-update is not much more difficult to compute than the DP-update, and all $T_j > 0$ if all $D_j > 0$. The fractional DP-update will be non-conflicting for sufficiently low values of α, but not for higher. Based on (14) we can derive the following theorem:

Theorem 7. *A fractional DP-update with α according to*

$$\alpha \leq \min_j \{1, \frac{D_j}{\sum_{k \geq j} D_k}\} \tag{16}$$

is a non-conflicting update. Also, $\alpha = \frac{1}{n}$, where n is the number of variables in the constraint, is always a non-conflicting update.

We note that the fractional DP-update does not move out as 'much as possible'. However, it appears to be non-trivial to formulate such a notion in a useful way. We also note that it can be possible to move out more by iteratively applying the fractional DP-update to the same subproblem.

We end this section by mentioning that a stronger analysis can be carried out for special constraints. For example for the set partitioning constraint, it is possible to show that $\alpha = 1/2$ always gives a non-conflicting update. Finally, the LP-update, while not a fractional DP-update, is non-conflicting. This can be seen by noting that after this update, all feasible entries in the constraint-component have the same value.

4 Propagation Experiments

4.1 The Simple Assignment and Set Partitioning Problems

The formulation of the assignment problem in the component model is straightforward from the traditional binary ILP formulation. For our experiments we have used randomly generated assignment problems, and some from Beasley's OR-library [9]. For the propagation we used a linear-time implementation of the fractional DP-update ($\alpha = 0.5$). In summary, the result was that all tested problems with unique solutions coverged to optimality. The random problems were tight with sizes up to n=300, and were done within one minute on a Sun SunFire 280R with 900Mhz UltraSparc III+. When using CPLEX, which surely uses some specialized method, the instances can be solved in a few seconds each. Nevertheless, we were able to solve a number of instances of substantial size in reasonable time with propagation only. Finally, we note that CP's constraint propagation (domain reduction) would be unable even to approach a problem like this without a top level search. It should be noted that also from an LP-perspective this is not an entirely trivial result, since it is the symmetrical treatment of the values of 0 and 1 (rather than seeking complementary slackness), that avoids that the propagation gets stuck. We can therefore find a solution without any global control such as that required in the otherwise similar Hungarian algorithm.

The set partitioning problem is NP-hard and we therefore do not expect it to be solved by cost propagation only, and this was confirmed by running a couple of problems from Beasley's OR-library. In this case therefore, propagation needs to be combined with search. Alternatively, as done in Wedelin [15,16], the propagation can be augmented with a heuristic, for solving the set partitioning problem suboptimally.

4.2 A Crossword Puzzle

We now consider a crossword puzzle in a rectangle consisting of coded letters and black areas. The problem is to substitute the coded letters with uncoded letters in a one-to-one mapping so that English words are formed horizontally and vertically. To formally define the problem we let the set $\mathcal{A}$ be the letters in the alphabet and a special symbol , used to indicate the beginning and end of words, in total 27 symbols.

We now state our mathematical model for the crossword puzzle. We employ a Markov chain based language model where P_j denotes the probability of letter j and P_{jl} the probability of letter j followed by letter l, i.e. monogram and bigram statistics for ordinary text. Further let f_i be the frequency of the coded letter i in the crossword and let f_{ik} be the frequency of the coded bigram ik in the text. The problem can then be stated as finding a permutation ϕ of the letters and symbols in $\mathcal{A}$, maximizing the probability

$$P(\phi) = \prod_{i \in \mathcal{A}} P_{\phi(i)}^{f_i} \prod_{(i,k) \in \mathcal{A} \times \mathcal{A}} \left(\frac{P_{\phi(i)\phi(k)}}{P_{\phi(i)} P_{\phi(k)}}\right)^{f_{ik}} \tag{17}$$

The first product of (17) reflects the probability of a permutation based only on monogram statistics. If the crossword was just a linear sequence, the entire model (17) would be a Markov chain.

Translation to the component model is straightforward, and we outline the approach. We represent the permutation with a square of binary variables x_{ij}, $i, j \in \mathcal{A}$ where

$$x_{ij} = \begin{cases} 1 & \text{if the coded letter } i \text{ represents the uncoded letter } j, \\ 0 & \text{otherwise.} \end{cases} \tag{18}$$

The components are:

- **Components ensuring a feasible assignment**. A coded letter must represent exactly one uncoded letter and vice versa, this is in the square of variables expressed by horizontal and vertical set partitioning constraints just as in the simple assignment problem.
- **Components for single letters**. For every variable we introduce a variable-component $g(x_{ij} = 1) = f_i \cdot \log(P_j), g(x_{ij} = 0) = 0$. Together these components correspond to the first product in (17).
- **Components for adjacent letters**. For each pair of neighboring coded letters i, k in the crossword, we have a constraint-component corresponding to a factor in the second product in (17). The involved variables are those in rows i and k in the square of variables, ie in total 54 binary variables. Since only one variable in each row can be equal to 1, there can be no more than $27 * 27 = 729$ feasible solutions to the component, out of $2^5 4$ possible. When these variables are x_{ij} and x_{kl}, the value of the component is $\log \frac{P_{jl}}{P_j P_l}$. All other values are $-\infty$. The frequency f_{ik} is handled implicitly since we have one component for every bigram in the crossword, so if the same bigram

occurs several times, there will be several constraint-components. We chose to use a simplified constraint for the special case that $i = k$.

Note that this problem is easy to formulate as a component model but not as convenient to model directly in a CP or ILP framework. The example also illustrates how we can easily represent finite domain variables variables such as letters, with binary variables.

We have tested on four small (15 × 15), two medium size (15 × 30) and three large crossword problems (15 × 45). We have analyzed the problems, and the used Markov chain based model is sufficient for finding the correct solution for the large and medium size problems but not for all the small ones.

Table 2. Solution of crossword with cost propagation

name	Correct	CostP	UB	t
cw15.1	-340.38	-340.42 (17)	-282.2	*140*
cw15.2	-320.2	-319.2 (21)	-273.6	*143*
cw15.3	-334.2	-331.6 (19)	-274.5	*141*
cw15.4	-328.5	-350.4 (19)	-283.0	*143*
cw30.1	-674.5	-668.2 (23)	-601.7	*392*
cw30.2	-731.7	-731.7(26)	-663.0	*441*
cw45.1	-1013.9	-1020.9 (24)	-918.5	*582*
cw45.2	-1018.5	-1030.2 (24)	-912.1	*588*
cw45.3	-1165.3	-1202.5 (24)	-1040.5	*655*

Table 2 reports the results for cost propagation, using the fractional DP-update with $\alpha = 0.5$ for all constraints. The column $CostP$ shows the value of the best objective function value that the numerical propagation achieved within a fixed number of iterations. Since none of the crossword instances had unique integer solutions, propagation did not fully converge and was aborted after a maximum of 550 iterations reporting the best solution found. The number in parenthesis next to the value in column $CostP$ is the number of correct letters in the solution. Column UB shows the value of the component bound, after a fixed number of iterations. Column t shows the time in seconds.

In summary, the results show that the propagation finds correct or almost correct solutions to the large and medium size problem. To compare, we have also modelled the problem as an ILP in the straightforward way mentioned in Section 1 and used CPLEX to solve it. For the LP-relaxation, with highly fractional solutions, running times are at least 100 times longer, and ILP times are considerably longer than that. This is because the component model in a natural way captures the structure of the problem. In the components for adjacent letters the binary ILP model has to introduce a variable for every feasible entry, which we in contrast easily handle with a standard implementation of the DP-update. The difference would be even more significant if trigram statistics had been used.

5 Conclusions

Starting from the component model, we have formalized the use of cost propagation, and characterized several different updates and their properties. In summary, we think that cost propagation in a single approach captures some significant properties of both CP and ILP. We also hope that cost propagation can be useful in combination with search, providing in one mechanism both domain reduction, as well as direct optimization and bounds.

Our experiments illustrate that it is to a certain extent possible for propagation on its own to solve non-trivial problems, e.g. the simple assignment problem and the crossword puzzle. The simple assignment problem cannot be solved by constraint propagation only; a potentially costly top level search has to be employed. This is also the case for the crossword puzzle. On the other hand, word puzzles usually cannot be modelled efficiently by LP/ILP models because of the restriction to linear constraints, and the explosion of the number of variables in the models. In this case, cost propagation can be considered to give solutions of acceptable quality much faster.

It is natural that an investigation like this generates many new questions, of which we list some here:

- Properties of different updates can be analyzed further, in order to better understand their relation to special problem classes, convergence during propagation, and optimality. We also want to design as good updates as possible. We here note that while updates such as the DP-update do not in general give globally optimal solutions, they can on the other hand speed up convergence and also solve large integer problems by propagation only, see [15,16].
- Algorithm libraries with fast updates for different kinds of special constraints including so called global constraints can be developed as in CP.
- By combining cost propagation with search, one would have a complete method applicable to a much wider range of problems. It would also be easier to compare with corresponding approaches in ILP and CP. One obvious possibility is to use non-conflicting updates and a branch & bound search, another possibility is to explore branching in order to break cycles when using e.g. a DP-update.

Acknowledgements

We wish to thank Mattias Grönkvist for discussions regarding this work. We are also grateful to the reviewers for useful and detailed comments.

References

1. Bertelè, U., Brioschi, F.: Nonserial Dynamic Programming, Mathematics in Science and Engineering. Academic Press, London (1972)
2. Dechter, R.: Constraint Processing. Morgan Kaufmann, San Francisco (2003)

3. Focacci, F., Lodi, A., Milano, M.: Cost-Based Domain Filtering. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, Springer, Heidelberg (1999)
4. Grohe, B.: Cost Propagation - Numerical Propagation for Optimization Problems, Licenciate thesis, Chalmers University of Technology (2007)
5. Hummel, R.A., Zucker, S.W.: On the Foundations of Relaxation Labeling Processes. IEEE Trans. on Pattern Analysis and Machine Intelligence 3 (1983)
6. Khemmoudj, M.O.I., Bennaceur, H., Nagih, A.: Combining Arc-Consistency and Dual Lagrangean Relaxation for Filtering CSPs. In: Barták, R., Milano, M. (eds.) CPAIOR 2005. LNCS, vol. 3524, Springer, Heidelberg (2005)
7. Larrosa, J., Schiex, T.: Artificial Intelligence 159 (2004)
8. Milano, M., Ottosson, G., Refalo, P., Thorsteinsson, E.S.: Global constraints: When Constraint Programming meets Operations Research. INFORMS Journal on Computing, Special Issue on the Merging of Mathematical Programming and Constraint Programming (2001)
9. Beasley, J.E.: OR-library, http://people.brunel.ac.uk/mastjjb/jeb/orlib/assigninfo.html
10. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Network of Plausible Inference. Morgan Kaufmann, San Francisco (1988)
11. Régin, J.-C.: Cost based Arc Consistency for Global Cardinality Constraints. Constraints, an International Journal 7(3-4) (2002)
12. Sellmann, M.: Solving weighted CSP by maintaining are consistency. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 634–647. Springer, Heidelberg (2004)
13. Viterbi, A.J.: Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Transactions on Information Theory 13(2) (1967)
14. Wedelin, D.: Probabilistic Inference, Combinatorial Optimization and the Discovery of Causal Structure from Data, PhD thesis, Dept. of Computing Science, Chalmers University of Technology (1993)
15. Wedelin, D.: An algorithm for large scale 0-1 integer programming with application to airline crew scheduling. Annals of Operations Research 57 (1995)
16. Wedelin, D.: The design of a 0-1 integer optimizer and its application in the Carmen system. European Journal of Operations Research 87 (1995)
17. Wedelin, D.: Cost Propagation: A generalization of constraint programming for optimization problems. In: Proceedings of CPAIOR 2002 (2002)

Fitness-Distance Correlation and Solution-Guided Multi-point Constructive Search for CSPs

Ivan Heckman[1] and J. Christopher Beck[1,2]

[1] Department of Computer Science
[2] Department of Mechanical & Industrial Engineering
University of Toronto
{iheckman,jcb}@mie.utoronto.ca

Abstract. Solution-Guided Multi-Point Constructive Search (SGMPCS) is a complete, constructive search technique that has been shown to out-perform standard constructive search techniques on a number of constraint optimization and constraint satisfaction problems. In this paper, we perform a case study of the application of SGMPCS to a constraint satisfaction model of the multi-dimensional knapsack problem. We show that SGMPCS performs poorly. We then develop a descriptive model of its performance using fitness-distance analysis. It is demonstrated that SGMPCS search performance is partially dependent upon the correlation between the heuristic evaluation of the guiding solutions and their distance to the nearest satisfying solution. This is the first work to develop a descriptive model of SGMPCS search behavior. The descriptive model points to a clear direction in improving the performance of constructive search for constraint satisfaction problems: the development of heuristic evaluations for partial solutions.

1 Introduction

An important line of research over the past 15 years in combinatorial optimization has been the empirical study of average algorithm behavior: there has been significant study of phase transition phenomena [7,18], work on heavy-tailed distributions [8,13], and detailed models developed for tabu search for job shop scheduling [20]. In this paper, we build on this work to begin to develop an understanding of the search behavior of a recently proposed constructive search technique, Solution-Guided Multi-Point Constructive Search (SGMPCS) [2].

We examine the performance of SGMPCS on a set of benchmark instances of a constraint satisfaction version of the multi-dimensional knapsack problem. We show that both randomized restart and SGMPCS perform poorly on these instances. The core of the paper is the investigation of the conjecture that SGMPCS performance is partially affected by the quality of the heuristic that is used to select the guiding partial solutions. When we artificially control the quality of the heuristic evaluation, we observe substantial performance differences. We then investigate two new heuristics. The better heuristic results in significant gain in search performance and, more importantly, the observed performance differences among the three heuristics are consistent with the descriptive model. Approximately 44% of the variation in search performance can be accounted for by the quality of the heuristic.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 112–126, 2008.

This is the first work which demonstrates that SGMPCS exploits the heuristic evaluation of its guiding solutions. Given that standard constructive search techniques do not directly exploit this information, we believe that SGMPCS embodies a promising direction for improving constructive search performance.

In the next section, we present SGMPCS, briefly discuss the literature on empirical models of search behavior, and introduce the multi-dimensional knapsack problem. In Section 3, we present and discuss the initial empirical studies, demonstrating the poor performance of SGMPCS. Section 4 develops our descriptive model of SGMPCS performance. Section 4.4 proposes two new heuristic evaluation functions and empirically evaluates them. We discuss the implications and limitations of our study in Section 5.

2 Background

In this section, we present the Solution-Guided Multi-Point Constructive Search algorithm, previous work on building descriptive models for search performance, and introduce the multi-dimensional knapsack problem.

2.1 Solution-Guided Multi-point Constructive Search

Solution-Guided Multi-Point Constructive Search (SGMPCS) [2,11] is a constructive search technique originally proposed for optimization problems. For clarity, we present the basic approach in the optimization context before discussing the changes necessary for constraint satisfaction problems.

SGMPCS for Optimization. The primary novelty of SGMPCS is that it is guided by sub-optimal solutions that it has found earlier in the search. As with randomized restart techniques [8], the overall search consists of a series of tree searches limited by a computational resource bound. When the resource bound is reached, search restarts and may be guided by an *elite* solution. An elite solution is a high-quality, sub-optimal solution found earlier in the search.

Pseudocode for SGMPCS is shown in Algorithm 1. The algorithm initializes a set, e, of elite solutions and then enters a while-loop. In each iteration, with probability p, search is started from an empty solution (line 5) or from a randomly selected elite solution (line 10). In the former case, if the best solution found during the search, s, is better than the worst elite solution, s replaces the worst elite solution. In the latter case, s replaces the starting elite solution, r, if s is better than r. Each individual search is limited by a fail bound: a maximum number of fails that can be incurred. The entire process ends when the problem is solved, proved insoluble within one of the iterations, or when some overall bound on the computational resources (e.g., CPU time, number of fails) is reached.

Elite Solution Initialization The elite solutions can be initialized by any search technique. For each problem in our experiments, we use independent runs of standard chronological backtracking with a random variable and value ordering. The search effort is limited by a maximum number of fails for each run.

Algorithm 1. SGMPCS: Solution-Guided Multi-Point Constructive Search

```
   SGMPCS():
1  initialize elite solution set e
2  while not solved and termination criteria unmet do
3      if rand[0, 1) < p then
4          set fail bound, b
5          s := search(∅, b)
6          if s is better than worst(e) then
7              replace worst(e) with s
       else
8          r := randomly chosen element of e
9          set fail bound, b
10         s := search(r, b)
11         if s is better than r then
12             replace r with s
```

Bounding the Search Each individual search is bounded by an evolving fail bound: a single search (lines 5 and 10) will terminate, returning the best solution encountered, after it has failed the corresponding number of times.

Searching from an Empty Solution With some probability, p, search is started from an empty solution (line 5). Searching from an empty solution simply means using any standard constructive search with a randomized heuristic and a bound on the number of fails. In our experiments, the search from an empty solution uses the same search techniques used to initialize elite solutions.

Searching from an Elite Solution To search from an elite solution, we create a search tree using any variable ordering heuristic and specifying that the value assigned to a variable is the one in the elite solution, provided it is still in the domain of the variable. Otherwise, any other value ordering heuristic can be used to choose a value. Formally, given a constraint satisfaction problem (CSP) with n variables, a solution, s, is a set of variable assignments, $\{\langle V_1 = x_1\rangle, \langle V_2 = x_2\rangle, \ldots, \langle V_m = x_m\rangle\}, m \leq n$. When $m = n$, the solution is complete, but possibly infeasible; when $m < n$, s is a partial solution. A search tree is created by asserting a series of choice points of the form: $\langle V_i = x\rangle \vee \langle V_i \neq x\rangle$ where V_i is a variable and x the value that is assigned to V_i. The variable ordering heuristic has complete freedom to choose a variable, V_i, to be assigned. If $\langle V_i = x_i\rangle \in s$ and $x_i \in dom(V_i)$, the choice point is made with $x = x_i$. Otherwise any value ordering heuristic can be used to choose $x \in dom(V_i)$. The only difference between starting search from an empty solution and from an elite solution is that the latter uses the assignments of the elite solution as a value ordering heuristic.

Adapting SGMPCS for CSPs. To apply SGMPCS to constraint satisfaction problems, it is necessary to define what an elite solution is and how one is evaluated. The elite solutions are used as value ordering heuristics and the evaluation of a solution determines

whether it will be used to guide subsequent search. Therefore, the evaluation of a potential elite solution is a *heuristic evaluation*. Since the only purpose of an elite solution is to provide value-ordering guidance, we want our evaluation function to choose elite solutions likely to guide the search to a satisfying assignment.[1] We experiment with three different heuristic evaluation functions as described in Sections 3.1 and 4.4.

We define elite solutions as dead-ends: either an assignment to a proper subset of the variables that results in a domain wipe-out or a complete assignment for which one or more constraints is broken. The solver may visit a complete assignment of variables that is not a solution to the problem. Consider a situation where $n-2$ variables have been assigned in lexicographical order and v_{n-1} and v_n both have non-empty, non-singleton domains. The assignment of v_{n-1} may trigger the reduction of the domain of v_n to a singleton, followed by its immediate assignment, while there are still constraints in the propagation queue. If one of these other constraints is not satisfied by the now-complete assignment, we have a complete assignment that fails to satisfy all constraints. The rating of a dead-end is done, as noted above, with a heuristic evaluation function.

We identify the dead-ends that are candidates for the elite set by modifying the behavior of the solver to keep track of the highest rated dead-end found during a single search (i.e., during the calls at lines 5 and 10 of Algorithm 1). At the end of a single search that has not found a satisfying solution, the best dead-end is returned and is considered for insertion into the elite set.

An alternative approach is to adopt a soft constraint framework where each potential elite solution is a complete assignment that breaks one or more constraints and the evaluation is an aggregation of the cost of the broken constraints. This is an interesting area for future work, but we do not consider it here for a number of reasons.

- We are motivated by simplicity and the desire to modify the behavior of standard (crisp) constraint solvers as little as possible.
- A soft constraint approach cannot fully exploit the strong constraint propagation techniques that are one of the core reasons for the success of CP.
- It is unclear *a priori* which cost models for various global constraints [4,19] is appropriate for the purposes of providing a heuristic evaluation.

We return to the question of a soft constraint model in Section 5.

2.2 Descriptive Models of Algorithm Behavior

A descriptive model of algorithm behavior is a tool used to understand why an algorithm performs as it does on a particular class or instance of a problem. There has been considerable work over the past 15 years in developing models of problem hardness [7,18] as well as work that has focused more directly on modeling the behavior of specific algorithms or algorithm styles. The work on heavy-tailed phenomenon [8,13] models the dynamic behavior of constructive search algorithms while local search has been addressed in a number of models–see [11] for a detailed overview.

[1] This is true for optimization contexts as well. However, the existence of a cost function obscures the fact that guiding the search with sub-optimal solutions is only helpful for finding the optimal solution if such guidance is likely to lead the search to lower cost solutions.

In this paper, we develop a static cost model with the goal of correlating problem instance features to algorithm performance. Our primary interest is to understand why SGMPCS outperforms or, as we will see below, fails to outperform, other constructive search techniques. The approach we adopt is *fitness-distance analysis* [11], an *a posteriori* approach traditionally applied to local search algorithms. Local search algorithms move through the search space based on an evaluation of the quality of (suboptimal) "solutions" in the neighborhood of the current solution. Neighboring solutions are evaluated and, typically, the lowest cost solution is selected to be the next solution. In fitness-distance analysis, the quality of a solution (i.e., its *fitness*) is compared against its distance to the nearest optimal solution. Distance is measured as the minimum number of steps it would take to move from the solution in question to the nearest optimal solution. In problem instances where the search space and neighborhood function induce a high *fitness-distance correlation* (FDC), the standard behavior of moving to a solution with higher fitness will also tend to move the search closer to an optimal solution.

Standard constructive search techniques such as chronological backtracking, limited discrepancy search, and randomized restart do not exploit the fitness of sub-optimal solutions that are found during search. Even when there is a notion of sub-optimal solution, as in optimization problems, these techniques do not attempt to search in the "neighborhood" of high quality solutions. There are, however, some algorithms that are based on constructive search such as ant colony optimization [5] and adaptive probing [17] that have been shown to be sensitive to FDC on optimization problems [3].

We test the hypothesis that SGMPCS is sensitive to fitness-distance correlation and that, therefore, its search performance can be partially understood by the FDC of a problem instance. Note that the FDC is a function of both the heuristic used to evaluate states and a measure of distance in the search space.

2.3 The Multi-dimensional Knapsack Problem

Given n objects and a knapsack with m dimensions such that each dimension has capacity, $c_1, \ldots, c_m$, a *multi-dimensional knapsack problem* requires the selection of a subset of the n objects such that the profit, $P = \sum_{i=1}^{n} x_i p_i$, is maximized and the m dimension constraints, $\sum_{i=1}^{n} x_i r_{ij} \leq c_j$ for $j = 1, \ldots, m$, are respected. Each object, i, has a individual profit, p_i, a size for each dimension, r_{ij}, and a binary decision variable, x_i, specifying whether the object is included in the knapsack ($x_i = 1$) or not ($x_i = 0$).

There has been significant work on such problems in the operations research and artificial intelligence literature [14,6]. Our purpose is not to compete with these approaches but to develop an understanding of the behavior of SGMPCS. We selected the multi-dimensional knapsack problem because previous work has indicated that SGMPCS performs particularly poorly on such problems and we want to understand why [10].

Because we are solving a constraint satisfaction problem, we adopt the approach of [16] and pose the problem as a satisfaction problem by constraining P to be equal to the known optimal value, P^*. In addition to the constraints defined above, we therefore add $P = P^*$.

3 Initial Experiment

In this section, we present the details and results of our initial experiments.

3.1 Experimental Details

We compare three search techniques: chronological backtracking (*chron*), randomized restart (*restart*) [8], and SGMPCS. In all algorithms, the variable ordering is random. The value ordering for each algorithm, when not being guided by an elite solution, is also random. Any restart-based technique needs some randomization. The use of purely random variable and value ordering serves to simplify the experimental set-up.

Restart follows the same fail sequence as SGMPCS (see below) and initializes and maintains a set of elite solutions. However, it always searches from an empty solution (i.e., it is equivalent of SGMPCS with $p = 1$). Therefore, it has a small run time overhead to maintain the elite set as compared with standard randomized restart.

All algorithms were implemented in ILOG Solver 6.3 and run on a 2GHz Dual Core AMD Opteron 270 with 2GB RAM running Red Hat Enterprise Linux 4.

Parameter Values for SGMPCS Previous work has examined the impact of different parameter settings [2,1]. Here, we are interested in SGMPCS performance in general, and, therefore, adopt the following parameters for all experiments.

- Probability of searching from an empty solution: $p = 0.5$.
- Elite set size: $|e| = 8$.
- Backtrack method: chronological. For a single search, we have a choice as to how the tree search should be performed at lines 5 and 10.
- Fail sequence: Luby [15]. The fail sequence sets the number of fails allowed for each tree search. The Luby sequence corresponds to the optimal sequence when there is no knowledge about the solution distribution: 1,1,2,1,1,2,4,1,1,2,1,1,2,4,8, ... Following [12], we multiply each limit by a constant, in our case 32.
- Number of initial solutions: 20. At line 1, we generate 20 partial solutions and then choose the $|e|$ best to form the initial elite set.
- Initialization fail bound: 1. The effort spent in finding a good initial solution is controlled by the fail bound on the search for each initial solution. We simply stop at the first dead-end found.

These parameters were chosen based on previous work and some preliminary experiments that showed little performance variation for SGMPCS for different settings on multi-dimensional knapsack problems [10].

Problem Instances Two sets of six problems from the operations research library[2] are used. The instances range from 15 to 50 variables and 2 to 30 dimensions.

For each problem instance, results are averaged over 1000 independent runs with different random seeds and a limit of 10,000,000 fails per run. For each run of each problem instance, we search for a satisfying solution.

[2] http://people.brunel.ac.uk/~mastjjb/jeb/orlib/mknapinfo.html

Heuristic Evaluation for SGMPCS Following the idea of trying simple approaches before more complex ones, our initial heuristic evaluation is the number of unassigned variables. Recall that our elite solution candidates are dead-ends (Section 2) that either have one or more variables with an empty domain or break a constraint. When the solver encounters a dead-end, we simply count the number of unassigned variables and use that as the heuristic evaluation: the fewer unassigned variables, the better the dead-end. We make no attempt at a dead-end to assign any of the unassigned variables that have non-empty domains. We refer to this heuristic evaluation as H_1. This heuristic has shown strong performance on quasigroup-with-holes completion problems [1].

3.2 Results

Table 1 compares the performance of chronological backtracking, randomized restart, and SGMPCS as defined above. Both SGMPCS and randomized restart perform poorly when compared to chronological backtracking. There does not seem to be a large difference between the performance of SGMPCS and randomized restart.

Table 1. Comparison of multi-dimensional knapsack results for chronological backtracking (*chron*), randomized restart (*restart*) and SGMPCS using the H_1 heuristic evaluation function

	chron			restart			SGMPCS-H_1		
	%sol	fails	time	%sol	fails	time	%sol	fails	time
mknap1-0	100	**1**	0.0	100	2	0.0	100	3	0.0
mknap1-2	100	**26**	0.0	100	42	0.0	100	41	0.0
mknap1-3	100	**523**	0.0	100	1062	0.0	100	924	0.0
mknap1-4	100	**15123**	**0.4**	100	54635	1.5	100	44260	1.2
mknap1-5	100	**3271555**	**67.2**	54.5	6885489	167.2	70.8	5573573	137.0
mknap1-6	0.2	9990291	**279.9**	0.0	10000000	337.2	0.8	**9958245**	340.9
mknap2-PB1	100	**15223**	0.3	100	42651	0.8	100	28770	0.6
mknap2-PB2	100	**3088092**	**54.1**	80.3	4970049	102.0	88.1	3741187	77.8
mknap2-PB4	100	**10167**	0.1	100	38474	0.5	100	28406	0.4
mknap2-PB5	100	**7011**	0.1	100	16178	0.4	100	15077	0.3
mknap2-PB6	100	**16050**	**1.9**	100	28964	3.8	100	25954	3.4
mknap2-PB7	100	**1472499**	**138.7**	76.0	5374900	551.4	85.9	4113704	423.6

Previous results showed SGMPCS out-performing randomized restart and chronological backtracking on optimization problems [2] and quasigroup-with-holes constraint satisfaction problems [1].

4 Building a Descriptive Model

Beck [2] speculates that three, non-mutually exclusive, factors may have an impact on the performance of SGMPCS: the exploitation of heavy-tails, the impact of revisiting elite solutions, and the use of multiple elite solutions to diversify the search. Here we focus on developing a descriptive model based on the second factor. The intuition behind this factor is that each time a good solution is revisited with a different variable

ordering, a different set of potential solutions (i.e., a different "neighborhood") will be visited upon backtracking. If good solutions tend to be near other good solutions in the search space, revisiting a solution is likely to result in finding another good solution.

In this section, we develop a descriptive model of SGMPCS performance based on the fitness-distance correlation. We first define the measure of distance used and then present a deeper analysis of the SGMPCS results in the above table. We then build on the methodology of Beck & Watson [3], to create an artificial heuristic evaluation function that allows us to completely control the fitness-distance correlation of the problem instances. Experiments with this artificial heuristic demonstrate a strong interaction between FDC and search performance. Finally, we develop two new heuristic evaluation functions and examine their performance.

4.1 A Measure of Distance

A complete solution to a multi-dimensional knapsack problem can be represented by a binary vector $(x_1, ..., x_n)$ of the decision variables. The representation lends itself to using the Hamming distance as a measure of the distance between two (complete) assignments. This is the standard definition in fitness-distance analysis.[3]

Our elite solutions are dead-ends and so may not be complete assignments. Therefore, we must adapt the Hamming distance to account for unassigned variables. A given dead-end with m assigned variables, $m < n$, represents a set of 2^{n-m} points in the search space with varying distances from the nearest satisfying solution. If we assume a single satisfying solution to a problem instance (see below), then the distribution of distances for the sub-vector of unassigned variables follows a binomial distribution with a minimum sub-distance of 0 and maximum sub-distance of $n-m$. The mean of this distribution is $\frac{n-m}{2}$. We therefore calculate the distance from a dead-end to the satisfying solution as the mean distance of the points represented by the dead-end: the Hamming distance for the assigned variables plus one-half the number of unassigned variables. More formally, for a given elite solution candidate $S = (x_1, ..., x_m)$ and a satisfying solution $S^* = (x_1^*, ..., x_n^*), m \leq n$, the distance is calculated as follows:

$$D(S, S^*) = \sum_{1 \leq i \leq m} |x_i - x_i^*| + \frac{n-m}{2} \tag{1}$$

The normalized distance is $ND(S, S^*) = \frac{D(S,S^*)}{n}$.

4.2 Analysis of the Initial Experiments

Traces of SGMPCS-H_1 runs show that early in the search all the elite solutions have a heuristic evaluation of 0: all the variables are assigned but the solution does not satisfy all constraints. The uniformity of the heuristic evaluation suggests that our simple heuristic evaluation is too coarse to provide useful guidance.

[3] SGMPCS does not move in the search space with the freedom of local search as it is constrained by a search tree. A different definition of distance that takes into account the search tree may be more appropriate. We leave the investigation of such a distance function for future work.

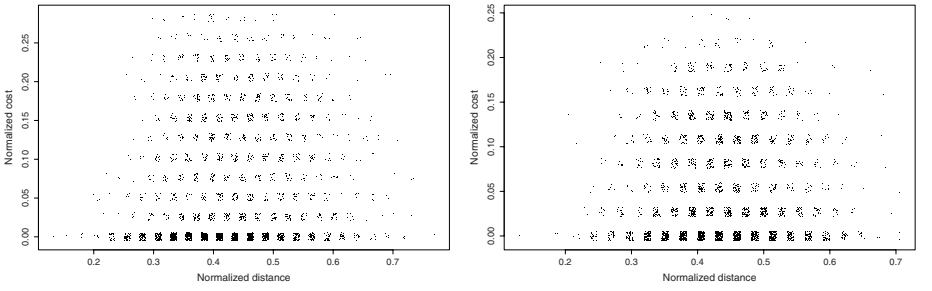


Fig. 1. Plots of the heuristic evaluation (fitness) of each elite solution vs. its normalized distance from the unique satisfying solution for two of the multi-dimensional knapsack problem instances: Left: mknap1-5; Right: mknap2-PB7. A small noise component is added to the fitness and distance for purposes of visibility in the plot–this noise is not present in the data.

To quantify this observation, we calculate the heuristic evaluation and distance of each elite solution encountered during the search. In order to do this, we must first find all satisfying solution to each instance. We did this using a small modification to the chronological backtracking algorithm. Each instance has a single satisfying solution, justifying our definition of D above.

Figure 1 presents plots of the distance vs. the fitness for two of the problem instances. The plots for the other problem instances are almost identical. It is clear, that the heuristic evaluation provides almost no real heuristic information. These data were gathered by instrumenting the SGMPCS solver to record the fitness and distance from the known satisfying solution of each new entry to the elite set.

4.3 Manipulating the Fitness-Distance Correlation

Figure 1 is consistent with our conjecture that fitness-distance correlation may have a role in a descriptive model of SGMPCS performance. It provides, however, rather weak support: the absence of an FDC accompanies poor performance. A stronger test of the conjecture is to directly manipulate the FDC and observe the performance of SGMPCS. To do this, we adopt the technique introduced in [3] to artificially set the heuristic evaluation based on knowledge of the distance to the satisfying solution.

Let $D(S, S^*)$ be defined as in Equation (1). We define the heuristic evaluation of the satisfying solution, S^*, to be $h(S^*) = 0$. We set the heuristic evaluation, $h_{FDC+}(S)$, of an elite solution S under perfect FDC equal to $D(S, S^*)$. Similarly, we set the heuristic evaluation $h_{FDC-}(S)$ of an elite solution with perfect negative FDC to be $(n - D(S, S^*))$. To generate instances with intermediate FDC, we interpolate between these two extremes as follows:

$$h(S) = \begin{cases} 0 \text{ if } S = S^* \\ \lceil \alpha \times h_{FDC+}(S) + (1-\alpha) \times RAND(S) \rceil \text{ if } S \neq S^* \wedge \beta = 0 \\ \lceil \alpha \times h_{FDC-}(S) + (1-\alpha) \times RAND(S) \rceil \text{ if } S \neq S^* \wedge \beta = 1 \end{cases} \quad (2)$$

where $\alpha \in [0,1]$, $\beta \in \{0,1\}$, and $RAND(S) \in [0,n]$; the latter value is uniformly generated from the interval, using the bit vector S as the random seed. The random component is added to achieve more realism in our model, while still manipulating the FDC. Clearly, when $\alpha = 0$, the heuristic evaluation is purely random. While α determines the strength of the FDC, β is a two-valued parameter governing its direction: $\beta = 0$ and $\beta = 1$ induce positive and negative FDC, respectively.

The only difference with our initial experiments is that the heuristic evaluation is changed to Equation (2). For a single instance and each pair of values for α and β, we solve the instance 1000 times with different random seeds. Following Watson [20] we compare FDC against the log of search cost, in our case, the log of the number of dead-ends to find a satisfying solution. Since our problems are of various sizes, the log of the mean number of fails of instance p with $\alpha = a, \beta = b$, $\bar{F}_{p,a,b}$, is normalized with the log of the search cost of *chron* on the same problem (C_p) as follows:

$$N_{p,a,b} = \frac{log(\bar{F}_{p,a,b}) - log(C_p)}{log(C_p)}$$

For each problem and setting of α and β, FDC values are measured by collecting every unique elite solution over the 1000 iterations and taking the correlation between the evaluation function for each entry and its distance to the one known satisfying solution as defined in Equation 1. Even though we are artificially defining the heuristic based on knowledge of the optimal solution, we are sampling the FDC as we would in a non-artificial setting.

Figure 2 shows that the manipulation of the FDC has a significant impact on the search performance of SGMPCS. The graph does not contain results for mknap1-0 and mknap1-2. As shown in Table 1, these are easily solved during the initialization phase of SGMPCS and so display no correlation with FDC. There is considerable noise for high negative values of FDC due to the fact that SGMPCS could not find a solution on a number of problem instances with high negative FDC, within the fail limit.

4.4 Toward Better Heuristic Evaluations

Figures 1 and 2 show that one possible explanation for the poor performance of SGMPCS-H_1 is the low fitness-distance correlation. The results of the experiment that manipulated the FDC demonstrated that the performance of SGMPCS is sensitive to the FDC, at least in an artificial setting. In this section, we develop two new heuristic evaluation functions. Our primary goal is to demonstrate that in a less artificial setting the FDC induced by the heuristic evaluation function has an impact on the search performance of SGMPCS.

The intuition behind both of the new heuristic evaluation functions is to include additional knowledge about the quality of the solution. In particular, we wish to create a finer heuristic evaluation that is able to better distinguish among the elite solutions (i.e., we would like fewer of the elite solutions to have a heuristic evaluation of zero than with the H_1 function). Our main goal in proposing these heuristics is to evaluate

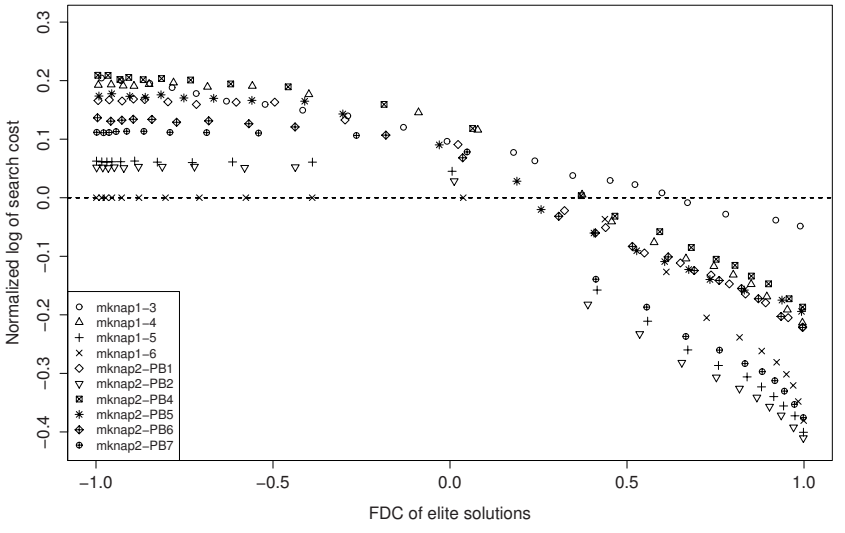


Fig. 2. A scatter-plot of the measured fitness-distance correlation versus the normalized log of search cost for the artificial heuristic evaluation in Equation (2). The low positive values of search cost for high negative FDC stem from problem instances (and settings of α and β) for which SGMPCS could not find a solution. The graph does not contain the results for problem instances mknap1-0 and mknap1-2 as they are trivially solved.

the relationship between FDC and search performance. We expect that these heuristics will have a different FDC and wish to test if this leads to a difference in performance.[4]

H_2. Recall (Section 2.3) that our CSP model of the multi-dimensional knapsack assumed that the value of the most profitable knapsack, P^*, is known. This knowledge is used in the constraint, $P = P^*$, but not otherwise exploited above. Here, we define $H_2 = |P^* - P|$.

H_3. Some preliminary experiments showed that even with H_2, the elite pool often stagnated on a set of elite solutions with a zero heuristic evaluation that break one or more constraints. Therefore, in order to further refine the heuristic evaluation, we choose to use the number of broken constraints as a tie-breaker: $H_3 = H_2 + |V|$ where $|V|$ is the number of constraints violated by the (partial) assignment.

It should be noted that the only difference among the H_1, H_2, and H_3 models is the heuristic evaluation function. In particular, the constraint model is identical in for all three heuristics. We now solve each of the problem instances 1000 times (with different random seeds) with each heuristic evaluation function. The other experimental details are the same as in Section 4.3.

[4] It does not seem likely that either of these heuristics will be useful, in general, for solving multi-dimensional knapsack problems because both make use of knowledge of the value of the most profitable knapsack.

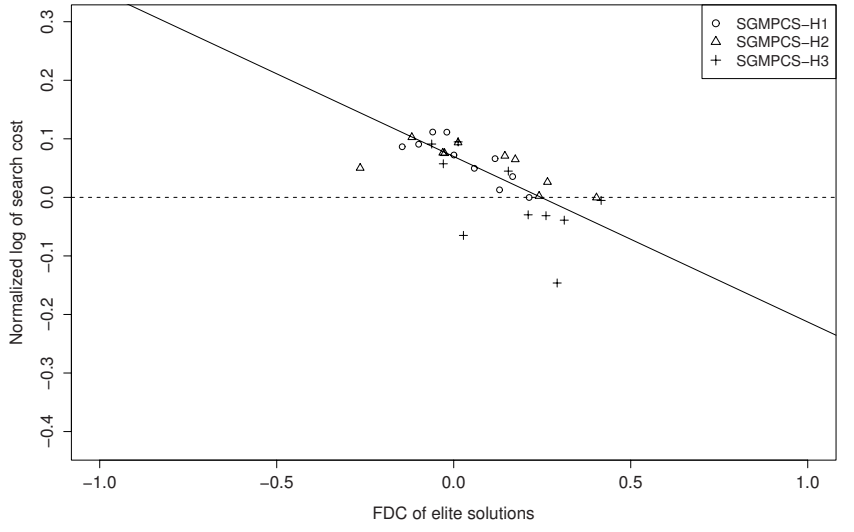


Fig. 3. A scatter-plot of the measured fitness-distance correlation versus the normalized log of search cost for three heuristic evaluation functions: H_1, H_2, H_3. The graph does not contain the results for problem instances mknap1-0 and mknap1-2 as they are trivially solved.

Results. Figure 3 displays the scatter plot of the normalized search performance versus FDC for all of our heuristic evaluation functions together with the minimum mean squared error line. As above, we do not include mknap1-0 and mknap1-2. Although limited by our small number of instances, the plot shows a trend of better search cost with higher FDC values. The r^2 value is 0.438 ($r = -0.662$).

Table 2 displays the performance of each SGMPCS variation. For completeness we repeat the results for chronological backtracking and SGMPCS-H_1 from Table 1. While not clearly superior, SGMPCS-H_3 is competitive with *chron* overall. For the harder instances (i.e., mknap1-5, mknap2-PB2, mknap2-PB6,and mknap2-PB7 where *chron* has a high number of fails) SGMPCS-H_3 is 1.5 to 8 times better than *chron* in terms of the number of fails.

Table 2. Comparison of multi-dimensional knapsack results for chronological backtracking (*chron*), and SGMPCS using the three heuristic evaluation functions H_1, H_2, H_3

	chron			SGMPCS-H_1			SGMPCS-H_2			SGMPCS-H_3		
	%sol	fails	time	%sol	fails	time	%sol	fails	time	%sol	fails	time
mknap1-4	100	15123	0.4	100	44260	1.2	100	29895	0.9	100	**11349**	0.5
mknap1-5	100	3271555	**67.2**	71	5573573	137.0	77	4839688	126.2	98	**1824457**	71.6
mknap2-PB1	100	**15223**	0.3	100	28770	0.6	100	28405	0.6	100	23445	0.7
mknap2-PB2	100	3088092	**54.1**	88	3741187	77.8	92	3191853	71.2	98	**1933160**	60.9
mknap2-PB4	100	**10167**	0.1	100	28406	0.4	100	24112	0.4	100	24370	0.5
mknap2-PB5	100	**7011**	0.1	100	15077	0.3	100	13747	0.3	100	11650	0.4
mknap2-PB6	100	16050	1.9	100	25954	3.4	100	26082	3.4	100	**8554**	1.8
mknap2-PB7	100	1472499	138.7	86	4113704	423.6	86	4287571	447.7	100	**184443**	**32.8**

5 Discussion

In this paper, we addressed the question of developing an understanding of the performance of SGMPCS on constraint satisfaction problems. We demonstrated that the correlation between the heuristic evaluation of an elite solution and its distance to the satisfying solution, is, itself, correlated with the search performance. Standard constructive search approaches such as chronological backtracking, randomized restart, and limited discrepancy search make no explicit use of such heuristic information.

There are a number of limitations to the study in this paper. First, it is a case study of 12 problem instances of one type of problem. While we believe these results are likely to be observed for other problems instances and types, a larger study is needed. Second, the poor performance of randomized restart on the multi-dimensional knapsack problems suggests that they do not exhibit heavy-tails. As a restart-based algorithm, SGMPCS does exploit heavy-tails in the same way as randomized restart [9]. Therefore, a full descriptive model of SGMPCS must address the impact of heavy-tailed distributions. In fact, one of the reasons that the multi-dimensional knapsack problem was chosen for this case study was precisely because we did not have to address the impact of heavy-tailed distributions. Third, multi-dimensional knapsack problems are strange CSPs since the underlying problem is an optimization problem and we exploit this in formulating the new heuristic evaluation functions in Section 4.4. Our original motivation for choosing to apply SGMPCS to a CSP version of multi-dimensional knapsack was simply because Refalo [16] did so and showed poor performance for randomized restart. Given the relationship between randomized restart and SGMPCS, this appeared to be a fertile choice. There remains some uncertainty regarding the application of the FDC-based descriptive model of SGMPCS performance on more "natural" CSPs. Nonetheless, our model makes clear, testable hypotheses that can be evaluated in future work. Finally, as a descriptive model, the work in this paper does not, on its own, produce a clear benefit for constraint solvers. We have not demonstrated any improvement on the state-of-the-art for any problem classes. That was not our aim in this paper. What we have done is provided a deeper understanding of the performance of SGMPCS and a potential new source of search guidance for CP search.

It was noted in Section 2.1 that an alternative way to apply SGMPCS to constraint satisfaction problems is to adopt a soft constraint framework. The work in this paper makes the prediction that the success of such an approach depends, at least partially, on achieving a high correlation between the "cost" of a solution that breaks some constraints and the distance of that solution from a satisfying (or optimal in the case of MAX-CSP) solution. Such work is an important test of the generality of the results presented here.

Another approach to the incorporation of soft constraints is to define the heuristic evaluation function to be based on a soft constraint model while the primary search is done within a crisp constraint model as above. That is, when the constructive search finds a potential elite solution, the evaluation of that solution could be done using a soft constraint model. The assignments of the elite solution could be extended to find a complete assignment that minimizes the cost of the broken constraints. That cost is then used as the heuristic evaluation of the corresponding elite solution. The results above suggest that the success of such an approach will be at least partially dependent

upon the correlation between the heuristic provided by the soft constraint model and the distance to the nearest satisfying solution.

6 Conclusion

In this paper, the first steps were taken in understanding the search behaviour of Solution-Guided Multi-Point Constructive Search (SGMPCS). Using a constraint satisfaction model of the multi-dimensional knapsack problem, a descriptive model of SGMPCS search behaviour was developed using fitness-distance analysis, a technique common in the metaheuristic literature [11]. Empirical results, both in an artificial context and using three different heuristic evaluation functions, demonstrated that the correlation between the heuristic evaluation of a state and its proximity to the satisfying solution has a strong impact on search performance of SGMPCS. This (partial) descriptive model is important for three main reasons:

1. It makes testable predictions about the behaviour of SGMPCS on other constraint satisfaction and optimization problems.
2. It provides a clear direction for improving SGMPCS search performance: the creation of, perhaps domain-dependent, heuristic evaluation functions for partial search states that are well-correlated with the distance to the nearest solution.
3. It re-introduces a heuristic search guidance concept to the constraint programming literature. Though guidance by heuristic evaluation of search states is common in metaheuristics, general AI search (e.g., A^* and game playing), and best-first search approaches, it does not appear to have been exploited in constructive, CP search. We believe this is an important direction for further investigation.

Acknowledgments

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada, the Canadian Foundation for Innovation, the Ontario Research Fund, Microway, Inc., and ILOG, S.A..

References

1. Beck, J.C.: Multi-point constructive search. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 737–741. Springer, Heidelberg (2005)
2. Beck, J.C.: Solution-guided multi-point constructive search for job shop scheduling. Journal of Artificial Intelligence Research 29, 49–77 (2007)
3. Beck, J.C., Watson, J.-P.: Adaptive search algorithms and fitness-distance correlation. In: Proceedings of the Fifth Metaheuristics International Conference (2003)
4. Beldiceanu, N., Petit, T.: Cost evaluation of soft global constraints. In: Régin, J.-C., Rueher, M. (eds.) CPAIOR 2004. LNCS, vol. 3011, pp. 80–95. Springer, Heidelberg (2004)
5. Dorigo, M., Di Caro, G., Gambardella, L.M.: Ant algorithms for discrete optimization. Artificial Life 5(2), 137–172 (1999)
6. Fekete, S.P., Schepers, J., van der Veen, J.C.: An exact algorithm for higher-dimensional orthogonal packing. Operations Research 55(3), 569–587 (2007)

7. Gent, I.P., MacIntyre, E., Prosser, P., Walsh, T.: The constrainedness of search. In: Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI 1996), vol. 1, pp. 246–252 (1996)
8. Gomes, C.P., Fernández, C., Selman, B., Bessière, C.: Statistical regimes across constrainedness regions. Constraints 10(4), 317–337 (2005)
9. Heckman, I.: Empirical Analysis of Solution Guided Multi-Point Constructive Search. PhD thesis, Department of Mechanical & Industrial Engineering, University of Toronto (2007)
10. Heckman, I., Beck, J.C.: An empirical study of multi-point constructive search for constraint satisfaction. In: Proceedings of the Third International Workshop on Local Search Techniques in Constraint Satisfaction (2006)
11. Hoos, H.H., Stüzle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann, San Francisco (2005)
12. Huang, J.: The effect of restarts on the efficiency of clause learning. In: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI 2007), pp. 2318–2323 (2007)
13. Hulubei, T., O'Sullivan, B.: The impact of search heuristics on heavy-tailed behaviour. Constraints 11(2-3), 159–178 (2006)
14. Korf, R.: Optimal rectangle packing: New results. In: Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS 2004), pp. 142–149 (2004)
15. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Information Processing Letters 47, 173–180 (1993)
16. Refalo, P.: Impact-based search strategies for constraint programming. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 557–571. Springer, Heidelberg (2004)
17. Ruml, W.: Adaptive tree search. PhD thesis, Dept. of Computer Science, Harvard University (2002)
18. Smith, B.M., Dyer, M.E.: Locating the phase transition in constraint satisfaction problems. Artificial Intelligence 81, 155–181 (1996)
19. van Hoeve, W.-J., Pesant, G., Rousseau, L.-M.: On global warming: Flow-based soft global constraints. Journal of Heuristics 12(4-5), 347–373 (2006)
20. Watson, J.-P.: Empirical Modeling and Analysis of Local Search Algorithms for the Job-Shop Scheduling Problem. PhD thesis, Dept. of Computer Science, Colorado State University (2003)

Leveraging Belief Propagation, Backtrack Search, and Statistics for Model Counting*

Lukas Kroc, Ashish Sabharwal, and Bart Selman

Department of Computer Science
Cornell University, Ithaca NY 14853-7501, U.S.A.
{kroc,sabhar,selman}@cs.cornell.edu

Abstract. We consider the problem of estimating the model count (number of solutions) of Boolean formulas, and present two techniques that compute estimates of these counts, as well as either lower or upper bounds with different trade-offs between efficiency, bound quality, and correctness guarantee. For lower bounds, we use a recent framework for probabilistic correctness guarantees, and exploit message passing techniques for marginal probability estimation, namely, variations of Belief Propagation (BP). Our results suggest that BP provides useful information even on structured loopy formulas. For upper bounds, we perform multiple runs of the `MiniSat` SAT solver with a minor modification, and obtain statistical bounds on the model count based on the observation that the distribution of a certain quantity of interest is often very close to the normal distribution. Our experiments demonstrate that our model counters based on these two ideas, `BPCount` and `MiniCount`, can provide very good bounds in time significantly less than alternative approaches.

1 Introduction

The model counting problem for Boolean satisfiability or SAT is the problem of computing the number of solutions or satisfying assignments for a given Boolean formula. Often written as #SAT, this problem is #P-complete [21] and is widely believed to be significantly harder than the NP-complete SAT problem, which seeks an answer to whether or not the formula in satisfiable. With the amazing advances in the effectiveness of SAT solvers since the early 90's, these solvers have come to be commonly used in combinatorial application areas like hardware and software verification, planning, and design automation. Efficient algorithms for #SAT will further open the doors to a whole new range of applications, most notably those involving probabilistic inference [1, 4, 12, 14, 17, 19].

A number of different techniques for model counting have been proposed over the last few years. For example, `Relsat` [2] extends systematic SAT solvers for model counting and uses component analysis for efficiency, `Cachet` [18] adds caching schemes to this approach, `c2d` [3] converts formulas to the d-DNNF form

* Research supported by IISI, Cornell University (AFOSR grant FA9550-04-1-0151), DARPA (REAL Grant FA8750-04-2-0216), and NSF (Grant 0514429).

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 127–141, 2008.

which yields the model count as a by-product, `ApproxCount` [23] and `SampleCount` [9] exploit sampling techniques for estimating the count, `MBound` [10] relies on the properties of random parity or XOR constraints to produce estimates with correctness guarantees, and the recently introduced `SampleMinisat` [8] uses sampling of the backtrack-free search space of systematic SAT solvers. While all of these approaches have their own advantages and strengths, there is still much room for improvement in the overall scalability and effectiveness of model counters.

We propose two new techniques for model counting that leverage the strength of message passing and systematic algorithms for SAT. The first of these yields probabilistic lower bounds on the model count, and for the second we introduce a statistical framework for obtaining upper bounds.

The first method, which we call `BPCount`, builds upon a successful approach for model counting using local search, called `ApproxCount`. The idea is to efficiently obtain a rough estimate of the "marginals" of each variable: what fraction of solutions have variable x set to TRUE and what fraction have x set to FALSE? If this information is computed accurately enough, it is sufficient to recursively count the number of solutions of only *one* of $F|_x$ and $F|_{\neg x}$, and scale the count up appropriately. This technique is extended in `SampleCount`, which adds randomization to this process and provides lower bounds on the model count with high probability correctness guarantees. For both `ApproxCount` and `SampleCount`, true variable marginals are estimated by obtaining several solution samples using local search techniques such as `SampleSat` [22] and computing marginals from the samples. In many cases, however, obtaining many near-uniform solution samples can be costly, and one naturally asks whether there are more efficient ways of estimating variable marginals.

Interestingly, the problem of computing variable marginals can be formulated as a key question in Bayesian inference, and the Belief Propagation or BP algorithm [15], at least in principle, provides us with exactly the tool we need. The BP method for SAT involves representing the problem as a factor graph and passing "messages" back-and-forth between variable and factor nodes until a fixed point is reached. This process is cast as a set of mutually recursive equations which are solved iteratively. From the fixed point, one can easily compute, in particular, variable marginals.

While this sounds encouraging, there are two immediate challenges in applying the BP framework to model counting: (1) quite often the iterative process for solving the BP equations does not converge to a fixed point, and (2) while BP provably computes exact variable marginals on formulas whose constraint graph has a tree-like structure (formally defined later), its marginals can sometimes be substantially off on formulas with a richer interaction structure. To address the first issue, we use a "message damping" form of BP which has better convergence properties (inspired by a damped version of BP due to [16]). For the second issue, we add "safety checks" to prevent the algorithm from running into a contradiction by accidentally eliminating all assignments.[1]

[1] A tangential approach for handling such fatal mistakes is incorporating BP as a heuristic within backtrack search, which our results suggest has clear potential.

Somewhat surprisingly, avoiding these rare but fatal mistakes turns out to be sufficient for obtaining very close estimates and lower bounds for solution counts, suggesting that BP does provide useful information even on highly structured loopy formulas. To exploit this information even further, we extend the framework borrowed from `SampleCount` with the use of biased coins during randomized value selection.

The model count can, in fact, also be estimated directly from just one fixed point run of the BP equations, by computing the value of so-called partition function [24]. In particular, this approach computes the exact model count on tree-like formulas, and appeared to work fairly well on random formulas. However, the count estimated this way is often highly inaccurate on structured loopy formulas. `BPCount`, as we will see, makes a much more robust use of the information provided by BP.

The second method, which we call `MiniCount`, exploits the power of modern DPLL [5, 6] based SAT solvers, which are extremely good at finding single solutions to Boolean formulas through backtrack search.[2] The problem of computing upper bounds on the model count has so far eluded solution because of an asymmetry which manifests itself in at least two inter-related forms: the set of solutions of interesting N variable formulas typically forms a minuscule fraction of the full space of 2^N variable assignments, and the application of Markov's inequality as in `SampleCount` does not yield interesting upper bounds. Note that systematic model counters like `Relsat` and `Cachet` can also be easily extended to provide an upper bound when they time out (2^N minus the number of non-solutions encountered), but these bounds are uninteresting because of the above asymmetry. To address this issue, we develop a statistical framework which lets us compute upper bounds under certain statistical assumptions, which are independently validated. To the best of our knowledge, this is the first effective and scalable method for obtaining good upper bounds on the model counts of formulas that are beyond the reach of exact model counters.

More specifically, we describe how the DPLL-based solver `MiniSat` [7], with two minor modifications, can be used to estimate the total number of solutions. The number d of branching decisions (not counting unit propagations and failed branches) made by `MiniSat` before reaching a solution, is the main quantity of interest: when the choice between setting a variable to TRUE or to FALSE is randomized,[3] the number d is provably not any lower, in expectation, than $\log_2$(model count). This provides a strategy for obtaining upper bounds on the model count, only if one could efficiently estimate the expected value, $\mathbb{E}[d]$, of the number of such branching decisions. A natural way to estimate $\mathbb{E}[d]$ is to perform multiple runs of the randomized solver, and compute the average of d over these runs. However, if the formula has many "easy" solutions (found with a low value of d) and many "hard" solutions, the limited number of runs one can perform in a reasonable amount of time may be insufficient to hit many of the

[2] [8] have recently independently proposed the use of DPLL solvers for model counting.
[3] `MiniSat` by default always sets variables to FALSE.

"hard" solutions, yielding too low of an estimate for $\mathbb{E}[d]$ and thus an incorrect upper bound on the model count.

Interestingly, we show that for many families of formulas, d has a distribution that is very close to the normal distribution. Under the assumption that d is normally distributed, when sampling various values of d through multiple runs of the solver, we need not necessarily encounter high values of d in order to correctly estimate $\mathbb{E}[d]$ for an upper bound. Instead, we can rely on statistical tests and conservative computations [20, 26] to obtain a statistical upper bound on $\mathbb{E}[d]$ within any specified confidence interval.

We evaluated our two approaches on challenging formulas from several domains. Our experiments with `BPCount` demonstrate a clear gain in efficiency, while providing much higher lower bound counts than exact counters (which often run out of time or memory) and competitive lower bound quality compared to `SampleCount`. For example, the runtime on several difficult instances from the FPGA routing family with over 10^{100} solutions is reduced from hours for both exact counters and `SampleCount` to just a few minutes with `BPCount`. Similarly, for random 3CNF instances with around 10^{20} solutions, we see a reduction in computation time from hours and minutes to seconds. With `MiniCount`, we are able to provide good upper bounds on the solution counts, often within seconds and fairly close to the true counts (if known) or lower bounds. These experimental results attest to the effectiveness of the two proposed approaches in significantly extending the reach of solution counters for hard combinatorial problems.

2 Notation

A Boolean variable x_i is one that assumes a value of either 1 or 0 (TRUE or FALSE, respectively). A truth assignment for a set of Boolean variables is a map that assigns each variable a value. A Boolean formula F over a set of n such variables is a logical expression over these variables, which represents a function $f : \{0,1\}^n \to \{0,1\}$ determined by whether or not F evaluates to TRUE under a truth assignment for the n variables. A special class of such formulas consists of those in the Conjunctive Normal Form or CNF: $F \equiv (l_{11} \vee \ldots \vee l_{1k_1}) \wedge \ldots \wedge (l_{m1} \vee \ldots \vee l_{mk_m})$, where each literal l_{lk} is one of the variables x_i or its negation $\neg x_i$. Each conjunct of such a formula is called a clause. We will be working with CNF formulas throughout this paper.

The constraint graph of a CNF formula F has variables of F as vertices and an edge between two vertices if both of the corresponding variables appear together in some clause of F. When this constraint graph has no cycles (i.e., it is a collection of disjoint trees), F is called a *tree-like* or *poly-tree* formula.

The problem of finding a truth assignment for which F evaluates to TRUE is known as the *propositional satisfiability* problem, or SAT, and is the canonical NP-complete problem. Such an assignment is called a *satisfying assignment* or a *solution* for F. In this paper we are concerned with the problem of counting the number of satisfying assignments for a given formula, known as the *propositional model counting* problem. This problem is #P-complete [21].

3 Lower Bounds Using BP Marginal Estimates

In this section, we develop a method for obtaining a lower bound on the solution count of a given formula, using the framework recently used in the SAT model counter `SampleCount` [9]. The key difference between our approach and `SampleCount` is that instead of relying on solution samples, we use a variant of belief propagation to obtain estimates of the fraction of solutions in which a variable appears positively. We call this algorithm `BPCount`. After describing the basic method, we will discuss two techniques that improve the tightness of `BPCount` bounds in practice, namely, *biased variable assignments* and *safety checks*.

3.1 Counting Using BP: `BPCount`

We begin by recapitulating the framework of `SampleCount` for obtaining lower bound model counts with probabilistic correctness guarantees. A variable u will be called *balanced* if it occurs equally often positively and negatively in all solutions of the given formula. In general, the *marginal probability* of u being TRUE in the set of satisfying assignments of a formula is the fraction of such assignments where $u =$ TRUE. Note that computing the marginals of each variable, and in particular identifying balanced or near-balanced variables, is quite non-trivial. The model counting approaches we describe attempt to estimate such marginals using indirect techniques such as solution sampling or iterative message passing.

Given a formula F and parameters $t, z \in \mathbb{Z}^+, \alpha > 0$, `SampleCount` performs t iterations, keeping track of the minimum count obtained over these iterations. In each iteration, it samples z solutions of (potentially simplified) F, identifies the most balanced variable u, uniformly randomly sets u to TRUE or FALSE, simplifies F by performing any possible unit propagations, and repeats the process. The repetition ends when F is reduced to a size small enough to be feasible for exact model counters like `Cachet`. At this point, let s denote the number of variables randomly set in this iteration before handing the formula to `Cachet`, and let M' be the model count of the residual formula returned by `Cachet`. The count for this iteration is computed to be $2^{s-\alpha} \times M'$ (where α is a "slack" factor pertaining to our probabilistic confidence in the bound). Here 2^s can be seen as scaling up the residual count by a factor of 2 for every uniform random decision we made when fixing variables. After the t iterations are over, the minimum of the counts over all iterations is reported as the lower bound for the model count of F, and the correctness confidence attached to this lower bound is $1 - 2^{-\alpha t}$. This means that the reported count is a correct lower bound with probability $1 - 2^{-\alpha t}$.

The performance of `SampleCount` is enhanced by also considering balanced variable pairs (v, w), where the balance is measured as the difference in the fractions of all solutions in which v and w appear with the same sign vs. with different signs. When a pair is more balanced than any single literal, the pair is used instead for simplifying the formula. In this case, we replace w with v or $\neg v$ uniformly at random. For ease of illustration, we will focus here only on identifying and randomly setting balanced or near-balanced variables.

The key observation in SampleCount is that when the formula is simplified by repeatedly assigning a positive or negative polarity to variables, the expected value of the count in each iteration, $2^s \times M'$ (ignoring the slack factor α), is exactly the true model count of F, from which lower bound guarantees follow. We refer the reader to Gomes et al. [9] for details. Informally, we can think of what happens when the first such balanced variable, say u, is set uniformly at random. Let $p \in [0, 1]$. Suppose F has M solutions, $F|_u$ has pM solutions, and $F|_{\neg u}$ has $(1 - p)M$ solutions. Of course, when setting u uniformly at random, we don't know the actual value of p. Nonetheless, with probability a half, we will recursively count the search space with pM solutions and scale it up by a factor of 2, giving a net count of $pM.2$. Similarly, with probability a half, we will recursively get a net count of $(1 - p)M.2$ solutions. On average, this gives $\frac{1}{2}.pM.2 + \frac{1}{2}.(1 - p)M.2 = M$ solutions.

Interestingly, the correctness guarantee of this process holds irrespective of how good or bad the samples are. However, when balanced variables are correctly identified, we have $p \approx \frac{1}{2}$ in the informal analysis above, so that for both coin flip outcomes we recursively search a space with roughly $M/2$ solutions. This reduces the variance tremendously, which is crucial to making the process effective in practice. Note that with high variance, the minimum count over t iterations is likely to be much smaller than the true count; thus high variance leads to poor quality lower bounds.

The idea of BPCount *is to "plug-in" belief propagation methods in place of solution sampling in the* SampleCount *framework,* in order to estimate "p" in the intuitive analysis above and, in particular, to help identify balanced variables. As it turns out, a solution to the BP equations [15] provides exactly what we need: an estimate of the marginals of each variable. This is an alternative to using sampling for this purpose, and is often orders of magnitude faster. One bottleneck, however, is that the basic belief propagation process is iterative and does not even converge on most formulas of interest. We therefore use a "message damping" variant of standard BP, very similar to the one introduced by [16]. This variant is parameterized by $\kappa \in [0, 1]$, and has the property that as κ decreases, the dynamics of the equations go from standard BP (for $\kappa = 1$) to a damped variant with assured convergence (for $\kappa = 0$). The equations are analogous to standard BP for SAT (see e.g. [13] Figure 4 with $\rho = 0$ for a full description), differing only in the added κ exponent in the iterative update equation as shown in Figure 1. We use its output as an estimate of the marginals of the variables in BPCount. Note that there are several variants of BP that assure convergence, such as by [25] and [11]; we chose the "κ" variant because of its good scaling behavior.

Given this process of obtaining marginal estimates from BP, BPCount works almost exactly like SampleCount and *provides the same lower bound guarantees.*

Using Biased Coins. We can improve the performance of BPCount (and also of SampleCount) by using biased variable assignments. The idea here is that when fixing variables repeatedly in each iteration, the values need not be chosen uniformly. The correctness guarantees still hold even if we use a biased coin

$$\eta_{a\to i} = \prod_{j\in V(a)\setminus i} \left[\frac{\left(\prod_{b\in C_a^s(i)}(1-\eta_{b\to i})\right)^{\kappa}}{\left(\prod_{b\in C_a^s(i)}(1-\eta_{b\to i})\right)^{\kappa} + \left(\prod_{b\in C_a^u(i)}(1-\eta_{b\to i})\right)^{\kappa}} \right]$$

Notation. $V(a)$: all variables in clause a. $C_a^u(i), i \in V(a)$: clauses where i appears with the *opposite* sign than it has in a. $C_a^s(i), i \in V(a)$: clauses where i appears with the *same* sign as it has in a (except for a).

Fig. 1. $BP(\kappa)$ update equation

and set the chosen variable u to TRUE with probability q and to FALSE with probability $1-q$, for any $q \in (0,1)$. Using earlier notation, this leads us to a solution space of size pM with probability q and to a solution space of size $(1-p)M$ with probability $1-q$. Now, instead of scaling up with a factor of 2 in both cases, we scale up based on the bias of the coin used. Specifically, with probability q, we go to one part of the solution space and scale it up by $1/q$, and similarly for $1-q$. The net result is that in expectation, we still get $q.pM/q + (1-q).(1-p)M/(1-q) = M$ solutions. Further, the variance is minimized when q is set to equal p; in `BPCount`, q is set to equal the estimate of p obtained using the BP equations. To see that the resulting variance is minimized this way, note that with probability q, we get a net count of pM/q, and with probability $(1-q)$, we get a net count of $(1-p)M/(1-q)$; these balance out to exactly M in either case when $q = p$. Hence, when we have confidence in the correctness of the estimates of variable marginals (i.e., p here), it provably reduces variance to use a biased coin that matches the marginal estimates of the variable to be fixed.

Safety Checks. One issue that arises when using BP techniques to estimate marginals is that the estimates, in some case, may be far off from the true marginals. In the worst case, a variable u identified by BP as the most balanced may in fact be a backbone variable for F, i.e., may only occur, say, positively in all solutions to F. Setting u to FALSE based on the outcome of the corresponding coin flip thus leads one to a part of the search space with no solutions at all, so that the count for this iteration is zero, making the minimum over t iterations zero as well. To remedy this situation, we use safety checks using an off-the-shelf SAT solver (`Minisat` or `Walksat` in our implementation) before fixing the value of any variable. The idea is to simply check that u can be set both ways *before* flipping the random coin and fixing u to TRUE or FALSE. If `Minisat` finds, e.g., that forcing u to be TRUE makes the formula unsatisfiable, we can immediately deduce $u =$ FALSE, simplify the formula, and look for a different balanced variable. This safety check prevents `BPCount` from reaching the undesirable state where there are no remaining solutions at all.

In fact, with the addition of safety checks, we found that the lower bounds on model counts obtained for some formulas were surprisingly good even when the marginal estimates were generated purely at random, i.e., without actually

running BP. This can perhaps be explained by the errors introduced at each step somehow canceling out when several variables are fixed. With the use of BP, the quality of the lower bounds was significantly improved, showing that BP does provide useful information about marginals even for loopy formulas. Lastly, we note that with `SampleCount`, the external safety check can be conservatively replaced by simply avoiding those variables that appear to be backbone variables from the obtained samples.

4 Upper Bound Estimation

We now describe an approach for estimating an upper bound on the solution count. We use the reasoning discussed for `BPCount`, and apply it to a DPLL style search procedure. There is an important distinction between the nature of the bound guarantees presented here and earlier: here we will derive *statistical* (as opposed to probabilistic) guarantees, and their quality may depend on the particular family of formulas in question. The applicability of the method will also be determined by a statistical test, which succeeded in most of our experiments.

4.1 Counting Using Backtrack Search: `MiniCount`

For `BPCount`, we used a backtrack-less branching search process with a random outcome that, in expectation, gives the exact number of solutions. The ability to randomly assign values to selected variables was crucial in this process. Here we extend the same line of reasoning to a search process *with* backtracking, and argue that the expected value of the outcome is an upper bound on the true count. We extend the `MiniSat` SAT solver [7] to compute the information needed for upper bound estimation. `MiniSat` is a very efficient SAT solver employing conflict clause learning and other state-of-the-art techniques, and has one important feature helpful for our purposes: whenever it chooses a variable to branch on, it is left unspecified which value should the variable assume first. One possibility is to assign values TRUE or FALSE randomly with equal probability. Since `MiniSat` does not use any information about the variable to determine the most promising polarity, this random assignment in principle does not lower `MiniSat`'s power.

Algorithm `MiniCount`**:** Given a formula F, run `MiniSat` with *no restarts*, choosing a value for a variable uniformly at random at each choice point (option `-polarity-mode=rnd`). When a solution is found, output 2^d where d is the number of choice points on the path to the solution (the final decision level), not counting those choice points where the other branch failed to find a solution.

The restriction that `MiniCount` cannot use restarts is the only change to the solver. This limits somewhat the range of problems `MiniCount` can be applied to compared to the original `MiniSat`, but is a crucial restriction for the guarantee of an upper bound (as explained below). We found that `MiniCount` is still efficient on a wide range of formulas. Since `MiniCount` is a probabilistic algorithm, its output, 2^d, on a given formula F is a random variable. We denote this random

variable by $\#F_{\text{MiniCount}}$, and use $\#F$ to denote the true number of solutions of F. The following proposition forms the basis of our upper bound estimation.

Proposition 1. $\mathbb{E}[\#F_{\text{MiniCount}}] \geq \#F$.

Proof. The proof follows a similar line of reasoning as for `BPCount`, and we give a sketch of it. Note that if no backtracking is allowed (i.e., the solver reports 0 solutions if it finds a contradiction), the result follows, with strict equality, from the proof that `BPCount` (or `SampleCount`) provides accurate counts in expectation. We will show that the addition of backtracking can only increase the value of $\mathbb{E}[\#F_{\text{MiniCount}}]$, by looking at its effect on any choice point. Let u be any choice point variable with at least one satisfiable branch in its subtree, and let M be the number of solutions in the subtree, with pM in the left branch (when $u =$FALSE) and $(1-p)M$ in the right branch (when $u =$TRUE). If both branches under u are satisfiable, then the expected number of solutions computed at u is $1/2.pM.2 + 1/2.(1-p)M.2 = M$, which is the correct value. However, if either branch is unsatisfiable, then two things might happen: with probability half the search process will discover this fact by exploring the contradictory branch first and u will not be counted as a choice point in the final solution (i.e., its multiplier will be 1), and with probability half this fact will go unnoticed and u will retain its multiplier of 2. Thus the expected number of reported solutions at u is $1/2.M.2 + 1/2.M = \frac{3}{2}M$, which is no smaller than M. This finishes the proof.

The reason restarts are not allowed in `MiniCount` is exactly Proposition 1. With restarts, only solutions reachable within the current setting of the restart threshold can be found. This biases the search towards "easier" solutions, since they are given more opportunities to be found. For formulas where easier solutions lie on paths with fewer choice points, `MiniCount` with restarts could undercount and thus not provide an upper bound in expectation.

With enough random sample outputs, $\#F_{\text{MiniCount}}$, obtained from `MiniCount`, their average value will eventually converge to $\mathbb{E}[\#F_{\text{MiniCount}}]$ by the Law of Large Numbers, thereby providing an upper bound on $\#F$ because of Proposition 1. Unfortunately, providing a useful correctness guarantee on such an upper bound in a manner similar to the lower bounds seen earlier turns out to be impractical, because the resulting guarantees, obtained using a reverse variant of the standard Markov's inequality, are too weak. Further, relying on the simple average of the obtained output samples might also be misleading, since the distribution of $\#F_{\text{MiniCount}}$ is often heavy tailed, and it might take very many samples for the sample mean to become as large as the true solution count.

4.2 Estimating the Upper Bound

In this section, we develop an approach based on statistical analysis of the sample outputs that allows one to estimate the expected value of $\#F_{\text{MiniCount}}$, and thus an upper bound with statistical guarantees, using relatively few samples.

Assuming the distribution of $\#F_{\text{MiniCount}}$ is known, the samples can be used to provide an unbiased estimate of the mean, along with confidence intervals

on this estimate. This distribution is of course not known and will vary from formula to formula, but it can again be inferred from the samples. We observed that for many formulas, the distribution of $\#F_{\text{MiniCount}}$ is well approximated by a log-normal distribution. Thus we develop the method under the assumption of log-normality, and include techniques to independently test this assumption. The method has three steps:

1. Generate n independent samples from $\#F_{\text{MiniCount}}$ by running `MiniCount` n times on the same formula.
2. Test whether the samples come from a log-normal distribution (or a distribution sufficiently similar).
3. Estimate the true expected value of $\#F_{\text{MiniCount}}$ from the samples, and calculate the $(1-\alpha)\%$ confidence interval for it, using the assumption that the underlying distribution is log-normal. We set the confidence level α to 0.01, and denote the upper bound of the resulting confidence interval by $c_{\max}$.

This process, some of whose details will be discussed shortly, yields an upper bound $c_{\max}$ along with a statistical guarantee that $c_{\max} \geq \mathbb{E}[\#F_{\text{MiniCount}}]$ and thus $c_{\max} \geq \#F$:

$$\Pr[c_{\max} \geq \#F] \geq 1 - \alpha$$

The caveat in this statement (and, in fact, the main difference from the similar statement for the lower bounds for `BPCount` given earlier) is that it is true only if our assumption of log-normality holds.

Testing for Log-Normality. By definition, a random variable X has a log-normal distribution if the random variable $Y = \log X$ has a normal distribution. Thus a test whether Y is normally distributed can be used, and we use the Shapiro-Wilk test [cf. 20] for this purpose. In our case, $Y = \log(\#F_{\text{MiniCount}})$ and if the computed p-value of the test is below the confidence level $\alpha = 0.05$, we conclude that our samples do *not* come from a log-normal distribution; otherwise we assume that they do. If the test fails, then there is sufficient evidence that the underlying distribution is not log-normal, and the confidence interval analysis to be described shortly will not provide any statistical guarantees. Note that non-failure of the test does not mean that the samples *are* actually log-normally distributed, but inspecting the Quantile-Quantile plots (QQ-plots) often supports the hypothesis that they are. QQ-plots compare sampled quantiles with theoretical quantiles of the desired distribution: the more the sample points align on a line, the more likely it is that the data comes from the distribution.

We found that a surprising number of formulas had $\log_2(\#F_{\text{MiniCount}})$ very close to being normally distributed. Figure 2 shows normalized QQ-plots for $d_{\text{MiniCount}} = \log_2(\#F_{\text{MiniCount}})$ obtained from 100 to 1000 runs of `MiniCount` on various families of formulas (discussed in the experimental section). The top-left QQ-plot shows the best fit of normalized $d_{\text{MiniCount}}$ (obtained by subtracting the average and dividing by the standard deviation) to the normal distribution: $(\text{normalized } d_{\text{MiniCount}} = d) \sim \frac{1}{\sqrt{2\pi}} e^{-d^2/2}$. The 'supernormal' and 'subnormal' lines show that the fit is much worse when the exponent of d is, for example,

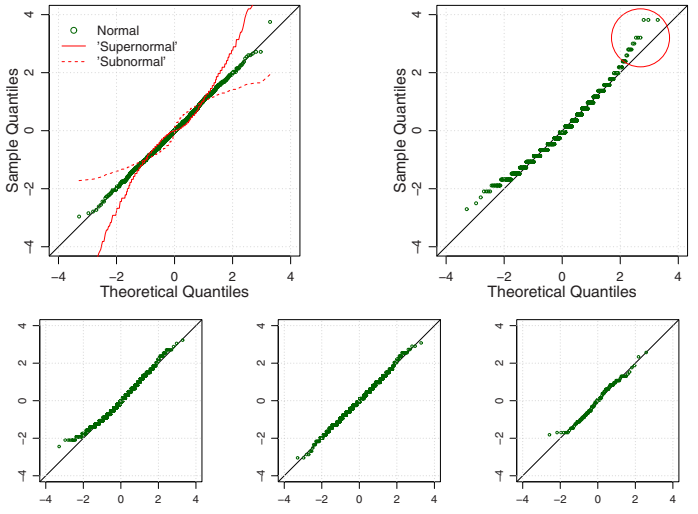


Fig. 2. Sampled and theoretical quantiles for formulas described in the experimental section (top: `alu2_gr_rcs_w8`, `lang19`; bottom: `2bitmax_6`, `wff-3-150-525`, `ls11-norm`)

1.5 or 2.5. The top-right plot shows that the corresponding domain (Langford problems) is somewhat on the border of being log-normally distributed, which is reflected in our experimental results to be described later.

Note that the nature of statistical tests is such that if the distribution of $\mathbb{E}[\#F_{\text{MiniCount}}]$ is not *exactly* log-normal, obtaining more and more samples will eventually lead to rejecting the log-normality hypothesis. For most practical purposes, being "close" to log-normally distributed suffices.

Confidence Interval Bound. Assuming the output samples from `MiniCount` $\{o_1, \ldots, o_n\}$ come from a log-normal distribution, we use them to compute the upper bound $c_{\max}$ of the confidence interval for the mean of $\#F_{\text{MiniCount}}$. The exact method for computing $c_{\max}$ for a log-normal distribution is complicated, and seldom used in practice. We use a conservative bound computation [26]: let $y_i = \log(o_i)$, $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$ denote the sample mean, and $s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(y_i - \bar{y})^2$ the sample variance. Then the conservative upper bound is constructed as

$$\tilde{c}_{\max} = \bar{y} + \frac{s^2}{2} + \left(\frac{n-1}{\chi^2_\alpha(n-1)} - 1\right)\sqrt{\frac{s^2}{2}\left(1 + \frac{s^2}{2}\right)}$$

where $\chi^2_\alpha(n-1)$ is the α-percentile of the chi-square distribution with $n-1$ degrees of freedom. Since $\tilde{c}_{\max} \geq c_{\max}$ we still have $\Pr[\tilde{c}_{\max} \geq \mathbb{E}[\#F_{\text{MiniCount}}]] \geq 1 - \alpha$.

The main assumption of the method described in this section is that the distribution of $\#F_{\text{MiniCount}}$ can be well approximated by a log-normal. This, of course, depends on the nature of the search space of `MiniCount` on a particular formula. As noted before, the assumption may sometimes be incorrect. In particular, one can construct a pathological search space where the reported upper

bound will be lower than the actual number of solutions. Consider a problem P that consists of two non-interacting subproblems P_1 and P_2, where it is sufficient to solve either one of them to solve P. Suppose P_1 is very easy to solve (e.g., requires few choice points that are easy to find) compared to P_2, and P_1 has very few solutions compared to P_2. In such a case, `MiniCount` will almost always solve P_1 (and thus estimate the number of solutions of P_1), which would leave an arbitrarily large number of solutions of P_2 unaccounted for. This situation violates the assumption that $\#F_{\text{MiniCount}}$ is log-normally distributed, but it may be left unnoticed. This possibility of a false upper bound is a consequence of the inability to prove from samples that a random variable is log-normally distributed (one may only disprove this assertion). Fortunately, as our experiments suggest, this situation is rare and does not arise in many real-world problems.

5 Experimental Results

We conducted experiments with `BPCount` as well as `MiniCount`, with the primary focus on comparing the results to exact counters and the recent `SampleCount` algorithm providing probabilistically guaranteed lower bounds. We used a cluster of 3.8 GHz Intel Xeon computers running Linux 2.6.9-22.ELsmp. The time limit was set to 12 hours and the memory limit to 2 GB.

We consider problems from five different domains, many of which have previously been used as benchmarks for evaluating model counting techniques: circuit synthesis, random k-CNF, Latin square construction, Langford problems, and FPGA routing instances from the SAT 2002 competition. The results are summarized in Table 1. The columns show the performance of `BPCount` and `MiniCount`, compared against the exact solution counters `Relsat`, `Cachet`, and `c2d` (we report the best of the three for each instance; for all but the first instance, `c2d` exceeded the memory limit) and `SampleCount`. The table shows the reported bounds on the model counts and the corresponding runtime in seconds.

For `BPCount`, the damping parameter setting (i.e., the κ value) we use for the damped BP marginal estimator is 0.8, 0.9, 0.9, 0.5, and either 0.1 or 0.2 for the five domains, respectively. This parameter is chosen (with a quick manual search) as high as possible so that BP converges in a few seconds or less. The exact counter `Cachet` is called when the formula is sufficiently simplified, which is when 50 to 500 variables remain, depending on the domain. The lower bounds on the model count are reported with 99% confidence. We see that a significant improvement in efficiency is achieved when the BP marginal estimation is used through `BPCount`, compared to solution sampling as in `SampleCount` (also run with 99% correctness confidence). For the smaller formulas considered, the lower bounds reported by `BPCount` border the true model counts. For the larger ones that could only be counted partially by exact counters in 12 hours, `BPCount` gave lower bound counts that are very competitive with those reported by `SampleCount`, while the running time of `BPCount` is, in general, an order of magnitude lower than that of `SampleCount`, often just a few seconds.

For `MiniCount`, we obtain $n = 100$ samples of the estimated count for each formula, and use these to estimate the upper bound statistically using the steps

Table 1. Performance of `BPCount` and `MiniCount`. [R] and [C] indicate partial counts obtained from `Cachet` and `Relsat`, respectively. `c2d` was slower for the first instance and exceeded the memory limit of 2 GB for the rest. Runtime is in seconds.

			`Cachet` / `Relsat` / `c2d` (exact counters)		`SampleCount` (99% confidence)		`BPCount` (99% confidence)		`MiniCount` S-W		(99% confidence)	
Instance	# of vars	True Count (if known)	Models	Time	LWR-bound	Time	LWR-bound	Time	Test	Average	UPR-bound	Time
CIRCUIT SYNTH.												
2bitmax_6	252	2.1×10^{29}	2.1×10^{29}	2 sec[C]	$\geq 2.4 \times 10^{28}$	29 sec	$\geq \mathbf{2.8 \times 10^{28}}$	**5 sec**	$\surd$	3.5×10^{30}	$\leq \mathbf{4.3 \times 10^{32}}$	**2 sec**
RANDOM k-CNF												
wff-3-3.5	150	1.4×10^{14}	1.4×10^{14}	7 min[C]	$\geq 1.6 \times 10^{13}$	4 min	$\geq \mathbf{1.6 \times 10^{11}}$	**3 sec**	$\surd$	4.3×10^{14}	$\leq \mathbf{6.7 \times 10^{15}}$	**2 sec**
wff-3-1.5	100	1.8×10^{21}	1.8×10^{21}	3 hrs[C]	$\geq 1.6 \times 10^{20}$	4 min	$\geq \mathbf{1.0 \times 10^{20}}$	**1 sec**	$\surd$	1.2×10^{21}	$\leq \mathbf{4.8 \times 10^{22}}$	**2 sec**
wff-4-5.0	100	—	$\geq 1.0 \times 10^{14}$	12 hrs[C]	$\geq 8.0 \times 10^{15}$	2 min	$\geq \mathbf{2.0 \times 10^{15}}$	**2 sec**	$\surd$	2.8×10^{16}	$\leq \mathbf{5.7 \times 10^{28}}$	**2 sec**
LATIN SQUARE												
ls8-norm	301	5.4×10^{11}	$\geq 1.7 \times 10^{8}$	12 hrs[R]	$\geq 3.1 \times 10^{10}$	19 min	$\geq \mathbf{1.9 \times 10^{10}}$	**12 sec**	$\surd$	6.4×10^{12}	$\leq \mathbf{1.8 \times 10^{14}}$	**2 sec**
ls9-norm	456	3.8×10^{17}	$\geq 7.0 \times 10^{7}$	12 hrs[R]	$\geq 1.4 \times 10^{15}$	32 min	$\geq \mathbf{1.0 \times 10^{16}}$	**11 sec**	$\surd$	6.9×10^{18}	$\leq \mathbf{2.1 \times 10^{21}}$	**3 sec**
ls10-norm	657	7.6×10^{24}	$\geq 6.1 \times 10^{7}$	12 hrs[R]	$\geq 2.7 \times 10^{21}$	49 min	$\geq \mathbf{1.0 \times 10^{23}}$	**22 sec**	$\surd$	4.3×10^{26}	$\leq \mathbf{7.0 \times 10^{30}}$	**7 sec**
ls11-norm	910	5.4×10^{33}	$\geq 4.7 \times 10^{7}$	12 hrs[R]	$\geq 1.2 \times 10^{30}$	69 min	$\geq \mathbf{6.4 \times 10^{30}}$	**1 min**	$\surd$	1.7×10^{34}	$\leq \mathbf{5.6 \times 10^{40}}$	**35 sec**
ls12-norm	1221	—	$\geq 4.6 \times 10^{7}$	12 hrs[R]	$\geq 6.9 \times 10^{37}$	50 min	$\geq \mathbf{2.0 \times 10^{41}}$	**70 sec**	$\surd$	9.1×10^{44}	$\leq \mathbf{3.6 \times 10^{52}}$	**4 min**
ls13-norm	1596	—	$\geq 2.1 \times 10^{7}$	12 hrs[R]	$\geq 3.0 \times 10^{49}$	67 min	$\geq \mathbf{4.0 \times 10^{54}}$	**6 min**	$\surd$	1.0×10^{54}	$\leq \mathbf{8.6 \times 10^{69}}$	**42 min**
ls14-norm	2041	—	$\geq 2.6 \times 10^{7}$	12 hrs[R]	$\geq 9.0 \times 10^{60}$	44 min	$\geq \mathbf{1.0 \times 10^{67}}$	**4 min**	$\surd$	3.2×10^{63}	$\leq \mathbf{1.3 \times 10^{86}}$	**7.5 hrs**
LANGFORD PROBS.												
lang-2-12	576	1.0×10^{5}	1.0×10^{5}	15 min[R]	$\geq 4.3 \times 10^{3}$	32 min	$\geq \mathbf{2.3 \times 10^{3}}$	**50 sec**	$\times$	5.2×10^{6}	$\leq 1.0 \times 10^{7}$	2.5 sec
lang-2-15	1024	3.0×10^{7}	$\geq 1.8 \times 10^{5}$	12 hrs[R]	$\geq 1.0 \times 10^{6}$	60 min	$\geq \mathbf{5.5 \times 10^{5}}$	**1 min**	$\surd$	1.0×10^{8}	$\leq \mathbf{9.0 \times 10^{8}}$	**8 sec**
lang-2-16	1024	3.2×10^{8}	$\geq 1.8 \times 10^{5}$	12 hrs[R]	$\geq 1.0 \times 10^{6}$	65 min	$\geq \mathbf{3.2 \times 10^{5}}$	**1 min**	$\times$	1.1×10^{10}	$\leq 1.1 \times 10^{10}$	7.3 sec
lang-2-19	1444	2.1×10^{11}	$\geq 2.4 \times 10^{5}$	12 hrs[R]	$\geq 3.3 \times 10^{9}$	62 min	$\geq \mathbf{4.7 \times 10^{7}}$	**26 min**	$\times$	1.4×10^{10}	$\leq 6.7 \times 10^{12}$	37 sec
lang-2-20	1600	2.6×10^{12}	$\geq 1.5 \times 10^{5}$	12 hrs[R]	$\geq 5.8 \times 10^{9}$	54 min	$\geq \mathbf{7.1 \times 10^{4}}$	**22 min**	$\surd$	1.4×10^{12}	$\leq \mathbf{9.4 \times 10^{12}}$	**3 min**
lang-2-23	2116	3.7×10^{15}	$\geq 1.2 \times 10^{5}$	12 hrs[R]	$\geq 1.6 \times 10^{11}$	85 min	$\geq \mathbf{1.5 \times 10^{5}}$	**15 min**	$\times$	3.5×10^{12}	$\leq 1.4 \times 10^{13}$	23 min
lang-2-24	2304	—	$\geq 4.1 \times 10^{5}$	12 hrs[R]	$\geq 4.1 \times 10^{13}$	80 min	$\geq \mathbf{8.9 \times 10^{7}}$	**18 min**	$\times$	2.7×10^{13}	$\leq 1.9 \times 10^{16}$	25 min
FPGA routing (SAT2002)												
apex7_*_w5	1983	—	$\geq 4.5 \times 10^{47}$	12 hrs[R]	$\geq 8.8 \times 10^{85}$	20 min	$\geq \mathbf{3.0 \times 10^{82}}$	**3 min**	$\surd$	7.3×10^{95}	$\leq \mathbf{5.9 \times 10^{105}}$	**2 min**
9symml_*_w6	2604	—	$\geq 5.0 \times 10^{30}$	12 hrs[R]	$\geq 2.6 \times 10^{47}$	6 hrs	$\geq \mathbf{1.8 \times 10^{46}}$	**6 min**	$\surd$	3.3×10^{58}	$\leq \mathbf{5.8 \times 10^{64}}$	**24 sec**
c880_*_w7	4592	—	$\geq 1.4 \times 10^{43}$	12 hrs[R]	$\geq 2.3 \times 10^{273}$	5 hrs	$\geq \mathbf{7.9 \times 10^{253}}$	**18 min**	$\surd$	1.0×10^{264}	$\leq \mathbf{6.3 \times 10^{326}}$	**26 sec**
alu2_*_w8	4080	—	$\geq 1.8 \times 10^{56}$	12 hrs[R]	$\geq 2.4 \times 10^{220}$	143 min	$\geq \mathbf{2.0 \times 10^{205}}$	**16 min**	$\surd$	1.4×10^{220}	$\leq \mathbf{7.2 \times 10^{258}}$	**16 sec**
vda_*_w9	6498	—	$\geq 1.4 \times 10^{88}$	12 hrs[R]	$\geq 1.4 \times 10^{326}$	11 hrs	$\geq \mathbf{3.8 \times 10^{289}}$	**56 min**	$\surd$	1.6×10^{305}	$\leq \mathbf{2.5 \times 10^{399}}$	**42 sec**

described earlier. The test for log-normality of the sample counts is done with a rejection level 0.05, that is, if the Shapiro-Wilk test reports p-value below 0.05, we conclude the samples do *not* come from a log-normal distribution, in which case no upper bound guarantees are provided (`MiniCount` is "unsuccessful"). When the test passed, the upper bound itself was computed with a confidence level of 99% using the computation of [26]. The results are summarized in the last set of columns in Table 1. We report whether the log-normality test passed, the average of the counts obtained over the 100 runs, the value of the statistical upper bound $c_{\max}$, and the total time for the 100 runs. We see that the upper bounds are often obtained within seconds or minutes, and are correct for all instances where the estimation method was successful (i.e., the log-normality test passed) and true counts or lower bounds are known. In fact, the upper bounds for these formulas (except `lang-2-23`) are correct w.r.t. the best known lower bounds and true counts even for those instances where the log-normality test failed and a statistical guarantee cannot be provided. The Langford problem family seems to be at the boundary of applicability of the `MiniCount` approach, as indicated by the alternating successes and failures of the test in this case. The approach is particularly successful on industrial problems (circuit synthesis, FPGA routing), where upper bounds are computed within seconds. Our results also demonstrate that a simple average of the 100 runs provides a very good approximation to the number of solutions. However, simple averaging can sometimes lead to an incorrect upper bound, as seen in `wff-3-1.5`, `ls13-norm`, `alu2_gr_rcs_w8`, and `vda_gr_rcs_w9`, where the simple average is below the true count or a lower bound obtained independently. This justifies our statistical framework, which as we see provides more robust upper bounds.

6 Conclusion

This work brings together techniques from message passing, DPLL-based SAT solvers, and statistical estimation in an attempt to solve the challenging model counting problem. We show how (a damped form of) BP can help significantly boost solution counters that produce lower bounds with probabilistic correctness guarantees. `BPCount` is able to provide good quality bounds in a fraction of the time compared to previous, sample-based methods. We also describe the first effective approach for obtaining good upper bounds on the solution count. Our framework is general and enables one to turn any state-of-the-art complete SAT/CSP solver into an upper bound counter, with very minimal modifications to the code. Our `MiniCount` algorithm provably converges to an upper bound, and is remarkably fast at providing good results in practice.

References

[1] Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and complexity results for #SAT and Bayesian inference. In: 44nd FOCS, pp. 340–351 (October 2003)
[2] Bayardo Jr., R.J., Pehoushek, J.D.: Counting models using connected components. In: 17th AAAI, Austin, TX, July 2000, pp. 157–162 (2000)

[3] Darwiche, A.: New advances in compiling CNF into decomposable negation normal form. In: 16th ECAI, Valencia, Spain, August 2004, pp. 328–332 (2004)
[4] Darwiche, A.: The quest for efficient probabilistic inference. In: IJCAI 2005 (July 2005) Invited Talk
[5] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. CACM 5, 394–397 (1962)
[6] Davis, M., Putnam, H.: A computing procedure for quantification theory. CACM 7, 201–215 (1960)
[7] Eén, N., Sörensson, N.: MiniSat: A SAT solver with conflict-clause minimization. In: 8th SAT, St. Andrews, U.K. (June 2005)
[8] Gogate, V., Dechter, R.: Approximate counting by sampling the backtrack-free search space. In: 22th AAAI, Vancouver, BC, July 2007, pp. 198–203 (2007)
[9] Gomes, C.P., Hoffmann, J., Sabharwal, A., Selman, B.: From sampling to model counting. In: 20th IJCAI, Hyderabad, India, January 2007, pp. 2293–2299 (2007)
[10] Gomes, C.P., Sabharwal, A., Selman, B.: Model counting: A new strategy for obtaining good bounds. In: 21th AAAI, Boston, MA, July 2006, pp. 54–61 (2006)
[11] Hsu, E.I., McIlraith, S.A.: Characterizing propagation methods for boolean satisfiabilty. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 325–338. Springer, Heidelberg (2006)
[12] Littman, M.L., Majercik, S.M., Pitassi, T.: Stochastic Boolean satisfiability. J. Auto. Reas. 27(3), 251–296 (2001)
[13] Maneva, E., Mossel, E., Wainwright, M.J.: A new look at survey propagation and its generalizations. J. Assoc. Comput. Mach. 54(4), 17 (2007)
[14] Park, J.D.: MAP complexity results and approximation methods. In: 18th UAI, Edmonton, Canada, August 2002, pp. 388–396 (2002)
[15] Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufmann, San Francisco (1988)
[16] Pretti, M.: A message-passing algorithm with damping. J. Stat. Mech. P11008 (2005)
[17] Roth, D.: On the hardness of approximate reasoning. AI J. 82(1-2), 273–302 (1996)
[18] Sang, T., Bacchus, F., Beame, P., Kautz, H.A., Pitassi, T.: Combining component caching and clause learning for effective model counting. In: 7th SAT (2004)
[19] Sang, T., Beame, P., Kautz, H.A.: Performing Bayesian inference by weighted model counting. In: 20th AAAI, Pittsburgh, PA, July 2005, pp. 475–482 (2005)
[20] Thode, H.C.: Testing for Normality. CRC Press, Boca Raton (2002)
[21] Valiant, L.G.: The complexity of computing the permanent. Theoretical Comput. Sci. 8, 189–201 (1979)
[22] Wei, W., Erenrich, J., Selman, B.: Towards efficient sampling: Exploiting random walk strategies. In: 19th AAAI, San Jose, CA, July 2004, pp. 670–676 (2004)
[23] Wei, W., Selman, B.: A new approach to model counting. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 324–339. Springer, Heidelberg (2005)
[24] Yedidia, J.S., Freeman, W.T., Weiss, Y.: Constructing free-energy approximations and generalized belief propagation algorithms. IEEE Trans. Inf. Theory 51(7), 2282–2312 (2005)
[25] Yuille, A.L.: CCCP algorithms to minimize the Bethe and Kikuchi free energies: Convergent alternatives to belief prop. Neural Comput. 14(7), 1691–1722 (2002)
[26] Zhou, X.-H., Sujuan, G.: Confidence intervals for the log-normal mean. Statistics In Medicine 16, 783–790 (1997)

The Accuracy of Search Heuristics: An Empirical Study on Knapsack Problems*

Daniel H. Leventhal and Meinolf Sellmann

Brown University
Department of Computer Science
115 Waterman Street, P.O. Box 1910
Providence, RI 02912
{dlev,sello}@cs.brown.edu

Abstract. Theoretical models for the evaluation of quickly improving search strategies, like limited discrepancy search, are based on specific assumptions regarding the probability that a value selection heuristic makes a correct prediction. We provide an extensive empirical evaluation of value selection heuristics for knapsack problems. We investigate how the accuracy of search heuristics varies as a function of depth in the search-tree, and how the accuracies of heuristic predictions are affected by the relative strength of inference methods like pruning and constraint propagation.

1 Motivation

The modern understanding [13] of systematic solvers for combinatorial satisfaction and optimization problems distinguishes between two fundamentally different principles. The first regards the intelligent reasoning about a given problem or one of its subproblems. In an effort to reduce the combinatorial complexity, solvers try to assess whether a part of the solution space can contain a feasible or improving solution at all. Moreover, when there is no conclusive evidence with respect to the global problem, solvers try to eliminate specific variable assignments that can be shown quickly will not lead to improving or feasible solutions. The principle of reasoning about a problem is called *inference* and is comprised of techniques like relaxation and pruning, variable fixing, bound strengthening, and, of course, constraint filtering and propagation. To strengthen inference, state-of-the-art solvers also incorporate methods like the computation of valid inequalities and, more generally, no-good learning and redundant constraint generation, as well as automatic model reformulation.

While inference can be strengthened to a point where it is capable of solving combinatorial problems all by itself (consider for instance Gomory's cutting plane algorithm for general integer problems [11] or the concept of k-consistency in binary constraint programming [8,3]), today's most competitive systematic solvers complement the reasoning about (sub-)problems with an active *search* for solutions. The search principle is in some sense the opposite of inference (see [13]): When reasoning about a problem we consider the solution space as a whole or even a relaxation thereof (for example by considering only one constraint at a time or by dropping integrality constraints). In

* This work was supported by the National Science Foundation through the Career: Cornflower Project (award number 0644113).

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 142–157, 2008.

search, on the other hand, we actually zoom into different parts of the space of potential solutions. Systematic solvers derive their name from the fashion in which this search is conducted. Namely, the space of potential solutions is partitioned and the different parts are searched systematically. This is in contrast to non-systematic solvers that are based on, for example, local search techniques. The main aspects of systematic solvers are how the solution space is partitioned, and in what order the different parts are to be considered. Both choices have an enormous impact on solution efficiency.

From an intellectual standpoint, we have every right to distinguish between search and inference in the way we described before. They are two different principles and constitute orthogonal concepts that each can be applied alone or in combination with the other. However, it has long been observed that (when applied in combination as practically all successful systematic solvers in constraint programming, satisfiability, and mathematical programming do) inference and search need to be harmonized to work together well (see, e.g. [2]).

Information provided by inference techniques can also be used to organize and guide the search process. For example, in integer linear programming fractional solution values are often used to determine whether a variable should be rounded up or down first. Such heuristics are commonly compared with each other based on some specific efficiency measure that is global to the specific approach, such as total time for finding and proving an optimal solution, or the best solution quality achieved after some hard time-limit.

However, to our knowledge very little research has been conducted which investigates how *accurate* information taken from inference algorithms actually is, or how this accuracy evolves during search. This is quite surprising given that many theoretical studies need to make certain assumptions about the relative correctness of search heuristics or the dependency of heuristic accuracy and, for instance, the depth in the search tree. The theoretical model that is considered in [20] to explain and study heavy-tailed runtime distributions, for example, is based on the assumption that the heuristic determining how the solution space is to be partitioned has a fixed probability of choosing a good branching variable. Moreover, Harvey and Ginsberg's limited discrepancy search (LDS) [12] is based on the assumption that the probability that the value selection heuristic returns a good value is constant (whereby, at the same time, they also assume that heuristics are marginally more likely to fail higher up in the tree than further below). Walsh's depth-bounded discrepancy search (DDS) [19], on the other hand, is implicitly based on the assumption that value selection heuristics are relatively uninformed at the top of the tree while the error probability (as a function of depth) converges very quickly to zero. Moreover, theoretical models for both LDS and DDS assume that there is a constant probability throughout search that the heuristic choice matters at all (which it does not when all subtrees contain (best) solutions that are equally good).

1.1 A Disturbing Case Study

Commonly, heuristics are evaluated on their overall performance in the context of a specific algorithm. In the following example, we show what can happen when heuristic accuracy is inferred from global efficiency.

Assume that we need to quickly compute high-quality solutions to knapsack problems. We have several heuristics and compare their performance when employed within

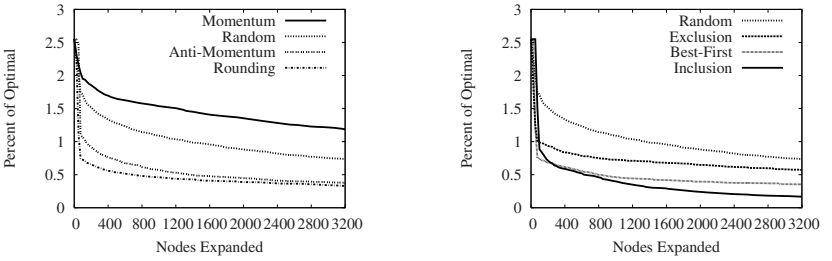


Fig. 1. Average optimality gap over number of search nodes when running PA with different heuristics on 500 almost strongly correlated instances with 50 items each. The curves are split into two graphs for better readability.

a depth-first search (DFS). Figure 1 shows how the average optimality gap, that is, the gap between current best solution and the optimal solution value, evolves when executing the algorithm with different heuristics on almost strongly correlated knapsack instances. (We will explain the heuristics and what strongly correlated knapsack instances are later - for now their exact definition is not important.) We see a clear separation: Heuristics inclusion, rounding, and best-first make very rapid improvements, while exclusion and momentum offer no significant gains over a random choice, or perform even worse.

In order to find a high-quality solution quickly, we clearly cannot afford a complete search. Therefore, we need to decide which parts of the search space we want to consider. LDS was developed exactly for this purpose. The strategy tries to guide the search to those parts of the search space that are most likely to contain feasible (in case of constraint satisfaction) or very good (in case of constraint optimization) solutions. Based on our findings in Figure 1, our recommendation would be to combine LDS with the inclusion heuristic.

The depressing result of this recommendation is shown in Figure 2: The choice of using the inclusion heuristic, whose predictions we so trusted, is the worst we could have made!

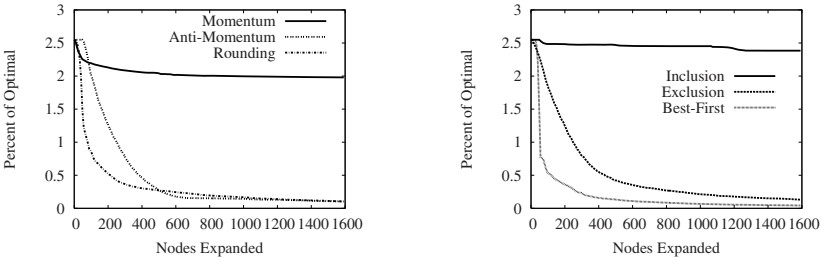


Fig. 2. Average optimality gap over number of search nodes when running PA with limited discrepancy search and different heuristics on 500 almost strongly correlated instances with 50 items each

1.2 A Case for the Direct Assessment of Heuristic Accuracy

Clearly, we lack some fundamental understanding of heuristic decision making. Obviously, it is not enough to design a heuristic that makes intuitive sense and to evaluate it within one specific algorithm. In this paper, we argue that *direct measurement* is needed on how accurate heuristic predictions are, how their accuracy differs with respect to the state of the search, and how it is affected by inference. One of the few examples of such research was presented in [10]: On the maximum satisfiability problem, Gomes et al. compare the accuracy of value selection heuristics based on a linear relaxation on one hand or a semi-definite relaxation on the other in correlation to the level of constrainedness of the given instances. Another example is given in [16] where the accuracy of relaxation-based search heuristics is found to depend on the sparsity of the linear program. We firmly believe that the first step towards the development of superior search algorithms must be a thorough empirical study how the accuracy of search heuristics depends on the current state of the search and how they interact with the inference mechanisms that are being used. With this paper, we wish to make a first step in that direction.

In the following section, we review the knapsack problem and describe a complete algorithm to tackle it. Before we present the results of our extensive experimentation that amounts to roughly 12,000 CPU hours on a dual-core 64-bit 2.8GHz Intel-Xeon processor, in Section 3 we introduce two performance measures for value selection heuristics. In Section 4, we investigate how the accuracies of different value selection heuristics behave as a function of depth in the search tree. Finally, we study how inference mechanisms like cutting plane generation and constraint propagation can influence the accuracy of those heuristics in Section 5.

2 Knapsack Problems

We begin our presentation by introducing the knapsack problem that will serve as the basis of our experimentation.

Definition 1 (Knapsack Problem). *Given a natural number $n \in \mathbb{N}$, we denote by $p_1, \ldots, p_n$ the* profits*, and by $w_1, \ldots, w_n$ the* weights *of n items. Furthermore, given a* knapsack capacity *$C \in \mathbb{N}$, the* knapsack problem *consists in finding a subset of items $S \subseteq \{1, \ldots, n\}$ such that the total weight of the selection does not exceed the knapsack capacity, i.e. $\sum_{i \in S} w_i \leq C$, and the total profit of the selection $\sum_{i \in S} p_i$ is maximized.*

We chose the knapsack problem as our target of investigation as it has some very desirable properties.

- First, the problem has a simple structure and is yet NP-hard [9].
- The knapsack problem is of great practical relevance as it can be viewed as a relaxation of very many real-world problems. Whenever we maximize some additive profit while a limited resource is exhausted linearly, knapsacks are somewhere hidden in the problem.
- Moreover, there exist many value selection heuristics for the problem so that we are in a position to base our study on more than just one special heuristic.

– Finally, for our experiments we need a large set of problem instances so that we can gather meaningful statistics. There exist knapsack generators that create instances in different classes that are well-documented and well-studied in the literature.

With respect to our last reason for considering knapsacks, we will exploit the following four benchmark classes of knapsack instances that are widely used in the knapsack literature [15]. In all cases, profits and weights are limited to the interval $[1, 10^6]$. When the procedure below results in an assignment of a value outside of this interval, we set the corresponding weight or profit to the closest interval bound.

– In **uncorrelated** instances, profits and weights of all items are drawn uniformly in $[1, 10^6]$ and independently from one another.
– **Weakly correlated** instances are generated by choosing profits uniformly in a limited interval around the corresponding weights. More precisely, after choosing a random weight for each item, we select the profit p_i uniformly from the interval $[w_i - 10^5, w_i + 10^5]$.
– Instances are **strongly correlated** when all profits equal their item's weight plus some constant. In our algorithm, we set $p_i = w_i + 10^5$.
– In **almost strongly correlated** instances, profits are chosen uniformly from a very small interval around the strongly correlated profit. Precisely, we choose p_i uniformly in $[w_i + 10^5 - 10^3, w_i + 10^5 + 10^3]$.

2.1 Branch and Bound for Knapsacks

Knapsacks with moderately large weights or profits are most successfully solved by dynamic programming approaches [14]. Once both profits and weights become too large, however, excessive memory requirements make it impossible to employ a dynamic program efficiently. Then, it becomes necessary to solve the problem by standard branch-and-bound techniques. As is common in integer linear programming, the bound that is used most often is the linear continuous relaxation of the problem. According to Dantzig, for knapsacks it can be computed very quickly [5]: First, we arrange the items in non-increasing order of efficiency, i.e., $p_1/w_1 \geq \cdots \geq p_n/w_n$. Then, we greedily select the most efficient item, until doing so would exceed the capacity of the knapsack, i.e., until we have reached the item s such that $\sum_{i=1}^{s-1} w_i \leq C$ and $\sum_{i=1}^{s} w_i > C$. We say that s is the *critical item* for our knapsack instance. We then select the maximum fraction of item s that can fit into the knapsack, i.e., $C - \sum_{j=1}^{s-1} w_j$ weight units of item s. The total profit of this relaxed solution is then $\tilde{P} := \sum_{j=1}^{s-1} p_j + \frac{p_s}{w_s}(C - \sum_{j=1}^{s-1} w_j)$.

Pruning the Search
Based on the relaxation, we can terminate the search early whenever we find that one of the following conditions holds [17]: *Optimality:* If no fractional part of the critical item can be added, then the optimal relaxed solution is already integral and therefore solves the given (sub-)problem optimally. *Insolubility:* If the capacity C of the (remaining) knapsack instance is negative, then the relaxation has no solution. *Dominance:* If the profit of the relaxed solution is not bigger than the value of any lower bound on the best integer solution, then the (sub-)tree under investigation cannot contain any improving solution. In all three cases, we may backtrack right away and thereby *prune* the search tree of branches that do not need explicit investigation.

For the sake of dominance detection, we need a lower bound, and the better (i.e. larger) that bound, the stronger our ability to prune the search. Obviously, every time we find an improving knapsack solution, we will update that lower bound. To speed up the search, however, we can do a little more: At the root-node, we compute an initial lower bound by taking all the profit of all non-fractional items of the initial relaxed solution. To strengthen this bound, we fill the remaining capacity with the remaining items (the ones after the critical item in the efficiency ordering), skipping subsequent items that fit only fractionally, until we either run out of capacity or items.

Organization of Search
We have outlined our approach in regard to inference. Now, with respect to search: At every search node, we choose to either include or exclude the critical item. This choice of the branching variable ensures that the upper bound that we compute has a chance of tightening in both children that are generated simultaneously. Note that this would not be the case if we would branch over any other item. Then, for at least one of the two child-nodes the upper bound would stay exactly the same as before. The value selection heuristics that we will study empirically are the following:

- **Random Selection Heuristic:** Choose at random the subtree that is investigated first. The reason why we may believe in this strategy may be that we feel that there are no good indicators for preferring one child over the other.
- **Inclusion Heuristic:** Always choose to insert the critical item first. A justification to consider this heuristic is that solutions with high total profits must include items.
- **Exclusion Heuristic:** Always choose to exclude the critical item first. This heuristics makes sense as it assures that no item with greater efficiency needs to be excluded to make room for the critical item.
- **Rounding Heuristic:** Consider the fractional value of the critical item: If it is at least 0.5, then include the critical item, otherwise try excluding it first. This heuristic makes intuitive sense when we trust the linear continuous relaxation as our guide.
- **Momentum Heuristic:** See whether the fractional value of the critical item is larger or lower than what its value was at the root-node. If the value has decreased, then exclude the item first, otherwise include it first. This heuristic is motivated by observing that the history of fractional values builds up a kind of momentum with respect to the "correct" value of the branching variable.
- **Anti-Momentum Heuristic:** The exact opposite of the momentum heuristic.
- **Best-First Heuristic:** Compute the relaxation value of both children and then consider the one with the higher value first. The intuition behind it is that, in a complete method, we will need to consider the better child anyway as there is no chance for the worse child to provide a lower bound that allows us to prune the better child.

2.2 Stronger Inference: Knapsack Cuts and Knapsack Constraints

The previous paragraphs sketch the baseline branch-and-bound algorithm that we will employ to study the different value selection heuristics. One question that we are interested in answering with this paper is how inference influences search heuristics. To this end, we consider two ways of strengthening inference for knapsacks.

The first is to compute a stronger upper bound. By adding valid inequalities known as "knapsack cuts," we can strengthen the linear continuous relaxation and thereby

obtain a tighter bound for pruning [4]. The redundant constraints are based on the following observation: When computing the upper bound, we find a subset of items $S := \{1, \ldots, s\}$ out of which it is infeasible to include more than $s-1$ in the knapsack, because this would exceed the limited capacity. As a matter of fact, out of the set of items $S' := S \cup \{i > s \mid w_i \geq w_j, \ \forall\ j \in S\}$, we can select at most $s-1$ items. Therefore, it is legitimate to add the inequality $\sum_{i \in S'} X_i \leq s-1$, where X_i is a binary variable which is 1 iff item i is included in the knapsack. We can add more such redundant constraints by adding one of them for each item skipped during the lower bound computation, because for each of these items we find a new set of items that, when added as a whole, would overload the knapsack.

The second way to strengthen inference for knapsacks is to add filtering to our baseline approach. That is, on top of the pruning based on upper and lower bound, we can also try to determine that certain items must be included in (or excluded from) any improving solution. For a given lower bound B, the corresponding constraint is true if and only if $\sum_i w_i X_i \leq C$ while simultaneously $\sum_i p_i X_i > B$. Achieving generalized arc-consistency for this global constraint is naturally NP-hard. However, we can achieve relaxed consistency with respect to Dantzig's bound in amortized linear time [6].

Now, adding solely the knapsack constraint that directly corresponds to the given knapsack instance is only of limited interest as constraint filtering draws its real strength from constraint propagation, i.e. the exchange of information between constraints that reflect different views of the problem. By exploiting the knapsack inequalities that we just discussed, we can add a couple of other knapsack constraints. At the root node, we compute the linear relaxation of the knapsack instance augmented by knapsack cuts. For the constraint $\sum_i w_i X_i \leq C$ we obtain a dual multiplier π_0. The same holds for the redundant constraints: For each cutting plane $\sum_{i \in I_r} X_i \leq s_r$, $1 \leq r \leq R$, we obtain a dual multiplier π_r. As suggested in [7], we use those multipliers to coalesce the different constraints into one new knapsack constraint

$$\sum_i p_i > B \qquad \text{and} \qquad \sum_i (\pi_0 w_i + \sum_{r \mid i \in I_r} \pi_r) X_i \leq \pi_0 C + \sum_r \pi_r s_r.$$

Finally, as was suggested in [18], we generate more redundant constraints by choosing a multiplier $\lambda \in \mathbb{N}$ and combining the different constraints: We multiply each constraint with a new multiplier that is a factor λ larger than the previous one (i.e., the first constraint is multiplied with 1, the next multiplier is λ, the next λ^2, and so on) and then we sum them all up. As this would quickly lead to knapsacks with extremely large item weights, we restrict ourselves to random selections of at most m out of the $R+1$ inequalities, whereby an inequality is added to the selection with a probability that is proportional to the optimal dual multiplier at the root-node. For our experiments, we chose $\lambda = 5$ and $m = 5$, and we generate ten constraints in this way.

So with the original knapsack constraint and the knapsack based on the optimal dual multipliers, we have a total of twelve constraints that we filter and propagate at every choice point until none of the constraints can exclude or include items anymore.

In summary, to evaluate how different inference techniques can influence the performance of search heuristics, we consider four different approaches in our experiments:

– **Pure Approach (PA):** Our baseline approach conducts branch and bound based on the simple linear continuous upper bound of the knapsack problem computed by

Dantzig's efficiency sorting procedure. The critical item determines the branching variable, the value selection heuristic is given as a parameter.

- **Valid Inequality Approach (VIA):** A variant of PA where we use the linear continuous relaxation augmented by knapsack cuts to determine upper bounds which is computed by the simplex algorithm. The branching variable is determined as the fractional variable that corresponds to the item with greatest efficiency (that is, profit over weight).
- **Constraint Programming Approach (CPA):** A variant of PA where, on top of pruning the search tree by means of the simple linear relaxation, we also perform knapsack constraint filtering and propagation at every search node.
- **Full Inference Approach (FIA):** The combination of VIA and CPA.

3 Two Performance Measures for Value Selection Heuristics

When evaluating value selection heuristics, the question arises how we should measure their quality. As mentioned in the introduction, for satisfiability problems theoretical models have been considered where assumptions were made regarding the probability with which a heuristic would guide us to a feasible solution. For optimization problems like knapsack, the mere existence of a feasible solution is not interesting. What actually matters is the quality of the best solution that can be found in the subtree that the heuristic tells us should be considered first.

This leads us directly to the first performance measure that we will study empirically in our experiments. The *probability measure* assesses the probability that the heuristic steers us in the direction of the subtree which contains the better solution, whenever that choice matters. That is to say that we consider only those cases where making a correct heuristic decision makes a difference. This is the case when at least one subtree contains an improving solution, and when the quality of the best solutions in both subtrees is different. To see how this relevance probability evolves in practice, in our experiments we keep track of how many search nodes are being considered at every depth level, and for how many of those choice points the heuristic decision to favor one child over the other makes a difference.

The second measure that we consider is the *accuracy measure* which takes a quantitative view of the heuristic choices by comparing the actual solution quality of the best solutions in both subtrees. Rather than measuring the actual difference in objective value (which would artificially increase the impact of instances with larger optimal solutions), we measure the difference relative to the current gap between upper and lower bound. Therefore, when we find that the average accuracy of a value selection heuristic at some depth is 40%, then this means that, by following the heuristic, we are guided to a subtree whose best solution closes on average 40% more of the current gap than the best solution in the subtree whose investigation was postponed.

4 Heuristic Accuracy as a Function of Depth

We specified the approaches that we use to solve different benchmark classes of knapsack instances, we introduced several value selection heuristics for the problem, and we described two different performance measures for these heuristics. In this section, we

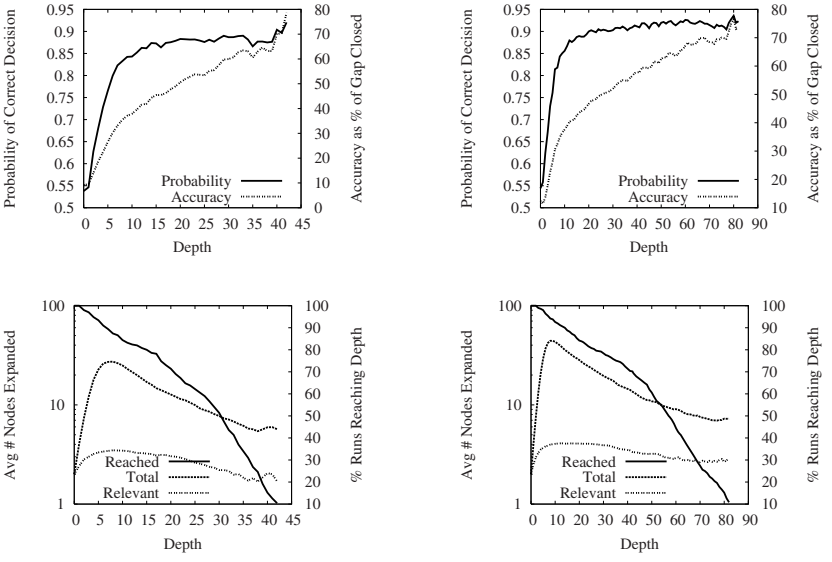


Fig. 3. Top: Average development of the probability and accuracy measure over depth for the rounding heuristic on 500 uncorrelated knapsack instances with 50 items (left) or 100 items (right). The solid line ('Probability') goes with the left vertical axis and depicts the probability that the heuristic makes the correct choice. The dashed line ('Accuracy') goes with the right vertical axis and depicts the average gap-closing percentile by which the heuristic choices outperform their opposites. Bottom: On the same benchmark sets, we show the percentage of instances that reach a certain depth (solid line, right vertical axis, 'Reached'). The dashed and dotted lines go with the left vertical axis (log-scale) and measure the average number of search nodes ('Total') on each depth level and the average number of nodes for which the heuristic choice is relevant ('Relevant'), respectively.

now investigate how the different heuristics perform in comparison. We implemented the algorithms presented earlier in Java 5, and we used Ilog Cplex 10.1 to compute linear continuous relaxations in algorithms VIA and FIA. Moreover, we implemented our own constraint propagation engine for algorithms CPA and FIA.

Note that our code is not tuned for speed as it has to gather and maintain all kinds of statistical information during search anyway. Fortunately, computational efficiency does not influence our performance measures for search heuristics or how that performance depends on the current state of the search or the inference mechanisms that are employed. Furthermore, within the branch-and-bound framework that we consider, the different knapsack heuristics all cause comparable overhead which is why a count of the number of search nodes gives us a very good estimate of the total time needed as well. However, note that this argument does not hold for the different inference methods that we will employ in Section 5.

To get a first insight into typical heuristic behavior, in Figure 3 we assess accuracy and probability of the rounding heuristic (that performs well with DFS and LDS, see Figures 1 and 2) when we run our baseline algorithm PA on uncorrelated instances. We see that the heuristic performs really well: Although it is rather uninformed at the

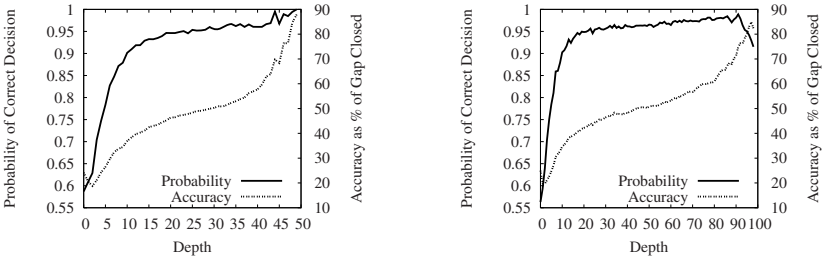


Fig. 4. The average probability and accuracy of the best-first heuristic on 500 weakly correlated knapsack instances with 50 (left) and 100 items (right)

beginning, it quickly becomes better and better. At depth 10, it already favors branches that close, on average, about 40% more of the current gap between upper and lower bound, and it steers us in the right direction in about 87% of the relevant cases.

Moreover, we observe a very distinct behavior with respect to the probability that the rounding heuristic makes the right choice when we ignore the extent to which the choice is better: The probability of favoring the subtree that contains the better solution is almost random (close to one half) at the very beginning of the search. Then, the probability of moving to the better subtree grows very quickly until it levels out at about 89% roughly at depth level 15. From then on, the probability stays pretty much constant. (We observe another brief increase to almost perfect prediction when we are very close to the maximum depth in the tree, but given the few runs that reach this depth and the few search-nodes on this depth this is not statistically significant.) This description holds for both the 50 and the 100 items benchmark, whereby for the latter the probability levels out only slightly later. Although we cannot show all our data here, we observe this *same* basic curve on the other classes of knapsack instances (including the one considered in Figures 1 and 2) and also for other "good" heuristics like best-first or exclusion. To give two more examples, in Figures 4 and 7, we depict the best-first and the exclusion heuristics, this time on weakly correlated knapsack instances.

To draw a first conclusion, our data suggests that a good value selection heuristic for knapsack has a much larger probability of misleading us higher up in the tree. Moreover, good heuristics become more accurate quickly and then keep their performance pretty much constant. Note that this behavior is addressed by neither LDS nor DDS: LDS has only a slight tendency to reconsider discrepancies higher up a little bit earlier than further down below, but only as far as leaves with the same number of discrepancies are concerned. Consequently, it wastes a lot of time reconsidering heuristic choices that are made deep down in the tree which are probably quite accurate, instead of putting more effort into the investigation of the more questionable decisions. DDS, on the other hand, does very well by quickly reconsidering choices that were made early in the search. However, it is not justified to assume that perfect predictions are already achieved at some rather shallow search-depth. From a certain depth-level on, simply minimizing the total number of discrepancies appears to be much better.

In the bottom part of Figure 3, we can see the number of choice points where the heuristic choice makes a difference is an order of magnitude lower than the total number of search nodes. This is caused by the fact that a good part of our search consists in a

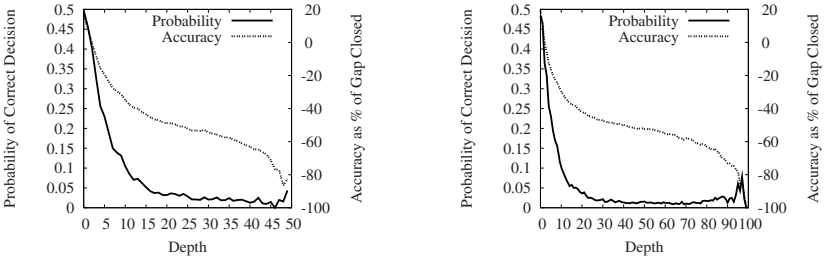


Fig. 5. The average probability and accuracy of the momentum heuristic on 500 strongly correlated knapsack instances with 50 (left) and 100 items (right)

proof of optimality. In this part of the search, the value selection heuristic is immaterial, just as it would be when considering an unsatisfiable instance.

It is interesting to note that, once the search has reached a sufficient depth, the probability that good value selection is relevant at some choice point is more or less independent of the depth level. We observe that the distance between the line depicting the average total number of choice points per level and the number of relevant choice points is almost constant. As we use a logarithmic scale, this indicates that the ratio of the number of relevant nodes over the total number of nodes does not change very much from some depth level on. In [12], the assumption is made that the relevance of value selection was constant. As with the accuracy of good heuristic value selection, this assumption appears only valid after we reached some critical depth in the tree. On very high levels, making good choices is much more of a gamble and at the same time much more likely to be important. Both are indications that a good search strategy ought to reconsider very early search decisions much more quickly than LDS does.

Thus far we have only studied heuristics that perform well. In Figure 5, we depict the performance of the momentum heuristic on strongly correlated instances. We see clearly that the heuristic performs poorly. On second thought, this is hardly surprising as it has a strong tendency to exclude high efficiency items and to include lower efficiency items. What our graph shows, however, is that making the general assumption that all heuristics ought to perform better as we dive deeper into the tree is quite wrong. As a matter of fact, for bad heuristics, performance may even decay.

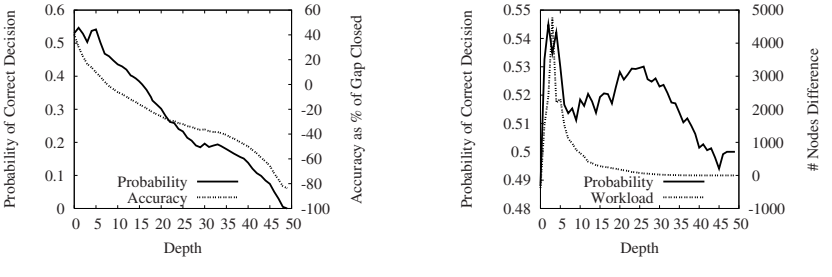


Fig. 6. Accuracy and probability measure for the inclusion heuristic on 500 almost strongly correlated knapsack instances with 50 items each

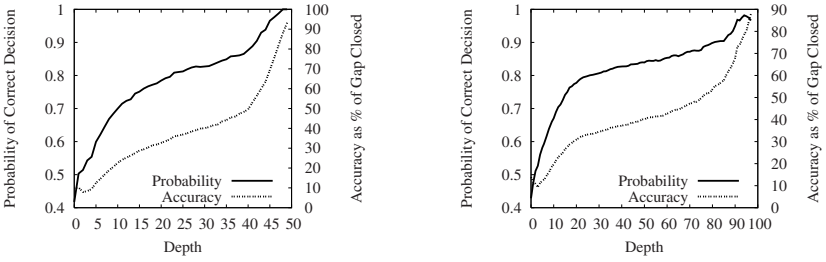


Fig. 7. The average probability and accuracy of the best-first (top) and the exclusion heuristic (bottom) on 1000 weakly correlated knapsack instances with 50 (left) and 100 items (right)

Now, what about the inclusion heuristic? We plot its performance in Figure 6. We see that the inclusion heuristic makes increasingly worse predictions where better solutions can be found as we dive deeper into the tree. This explains, of course, why inclusion is bound to perform poorly in combination with LDS as we saw in Figure 2. But why did it do so well in combination with DFS (see Figure 1)? The answer to this riddle is found in the right plot in Figure 6. Here, we assess the quality of inclusion while performing DFS by comparing the workload (that is, the total number of nodes) when exploring the subtree the heuristic chooses first. Again, we show the quantitative dependency of depth as the average number of nodes that the heuristic needs to investigate less (see the curve denoted "workload"), and a qualitative comparison which shows how often the heuristic investigates the subtrees in the right order (see the curve denoted "probability"). We see that, by including items first, the inclusion heuristic guides the search to more shallow subtrees first as the inclusion of items obviously has to be paid for by a reduction in remaining capacity. In those shallow subtrees, we quickly find a no-good (a new lower bound) which can be exploited when searching the second branch where better solutions can be found more quickly. When combining inclusion with LDS, however, we never get to searching those other branches, and while no-goods are helpful, we cannot afford to pay for them by excluding from our search the most interesting parts of the search space.

This example shows impressively that the accuracy of heuristic predictions can be abysmal even when a heuristic works well when combined with one specific search strategy (compare with Figure 1). In particular, even when a heuristic performs well with one search strategy, this is no indication that it is also suited *to guide* an incomplete search that is terminated early under hard time constraints. If one is to avoid that a heuristic like inclusion is combined with LDS to form an algorithm that is expected to provide good solutions quickly, then there is but one alternative: The performance of heuristics must be studied much more carefully than by a simple evaluation based on a global performance measure like best quality after some time-limit within a specific approach. Instead, a *direct assessment* of the accuracy of heuristic decisions as we provide it here is necessary. Otherwise, when designing algorithms, we are bound to a trial-and-error methodology that is far from the standards of scientific engineering.

Our direct assessment of the accuracy of heuristic decision making has allowed us to identify that there exists an interplay of heuristic decisions and no-goods that are learned during search – a matter which we have never seen discussed when heuristics for a problem are designed. What is more, we have shown that the belief that heuristics would generally become more accurate with increasing depth is wrong. This raises the

question why good heuristics *are* actually getting more accurate! An important insight here is that a heuristic like exclusion, which plainly sets the critical item to zero first, is of course not getting any "smarter" deeper down in the tree. Compare the accuracy at depth level 50 of the 100 item benchmark and the accuracy at level 0 in the 50 item benchmark in Figure 7. Actually, one should expect the accuracies to be the same. The reason why they differ is of course not that exclusion got more "experienced" while branching to depth-level 50 in the 100 item instances. The reason why the accuracy is higher than at the root-node level in the 50 item case can only lie in the different distribution of instances where the heuristic choices matter at depth-level 50 in the 100 item case. That is, exclusion (and the other heuristics for that matter) is getting more and more accurate because it is in some way self-enforcing: the decisions made higher up in the search tree result in problems for which the heuristic is more accurate. Inclusion, on the other hand, is not a good guide where good solutions will be found because it is self-weakening (which becomes clear when considering the repeated use of the inclusion heuristic which always trades the higher efficiency of an item to the left of the critical item for the efficiency of the latter).

We are not aware that self-enforcement has been noted before as an important aspect when designing value-selection heuristics. There exists, however, some work that tries to integrate different heuristics rather than sticking slavishly to the same one. For example, in [1], Balas et al. found that randomly combining different heuristics at each step of a greedy algorithm for set-covering actually produces better solutions than sticking to the single best heuristic for each instance throughout the algorithm.

5 How Inference Affects the Robustness of Search Heuristics

The final aspect that we are interested in is to what extent inference methods can influence the accuracy and the relevance of search heuristics. In this section, we therefore show how search heuristics perform when used with our different algorithm variants PA, VIA, CPA, and FIA (see Section 2).

Consider Figure 8, where we compare the accuracy of the rounding heuristic when used in combination with inference mechanisms of different strengths. On the left we consider algorithms PA and CPA that do not use knapsack cuts. The latter are used for the plots on the right that show the performance of rounding in combination with VIA and FIA. Analogously, at the top we show algorithms PA and VIA that do not use constraint filtering, whereas at the bottom we consider algorithms CPA and FIA that both propagate knapsack constraints. The results look very counter-intuitive: The more inference we perform, the worse the heuristic predictions become!

However, a look at the corresponding relevance data in Figure 9 reveals that, in reality, inference makes heuristic predictions far more robust: By adding just knapsack cuts, we already prevent searches from exploring deeper nodes. It is very impressive how constraint propagation (both in combination with knapsack cuts or alone) boosts this trend dramatically: Without constraint filtering, 50% of all searches reach a depth of 85, while with knapsack constraints, they only reach a depth of 20. On top of that, the maximum average number of relevant nodes per depth level is reduced to only 2.7![1]

[1] Note that this drastic reduction is also the reason why the curves in Figure 8 make such a ragged impression. There are simply far fewer sample points to average over.

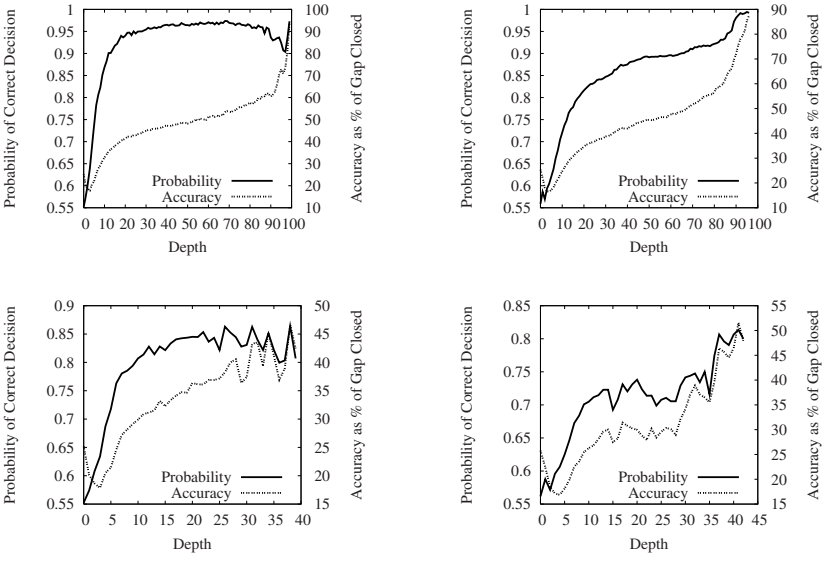


Fig. 8. Accuracy and probability of the rounding heuristic with algorithms PA (top left), VIA (top right), CPA (bottom left), and FIA (bottom right) on 1000 weakly correlated knapsack instances with 100 items. The 'Probability' of making the right decision goes with the left axis, the actual 'Accuracy' is again measured on the right axis as the relative gain in terms of percent of gap closed.

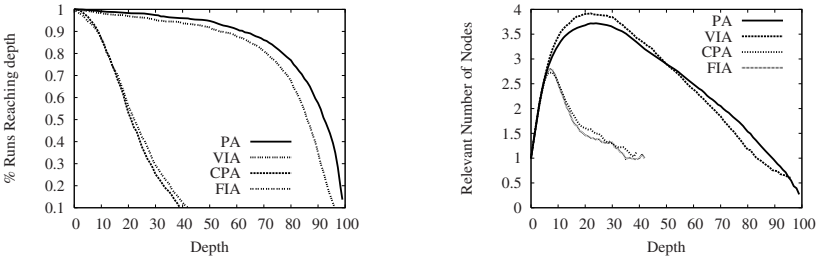


Fig. 9. Relevance of the rounding heuristic when employed within algorithms PA, VIA, CPA, and FIA on 1000 weakly correlated knapsack instances with 100 items. The left plot shows the percentage of instances that reach a certain depth, the right how many nodes are relevant, as an average over all runs that reach that depth.

Note that stronger inequalities like Mixed-Integer Gomory Cuts may have a similarly strong effect. However, while our statistics recording code does not allow us to provide a quantitative comparison, we would like to note that we found that constraint propagation could be conducted much faster than the computation of the strengthened linear relaxation bounds.

We conclude that stronger inference techniques can reduce the importance of good heuristic predictions. As a consequence, value selection heuristics are left with less room to make a dramatic difference and therefore appear less accurate on average.

6 Conclusions

We conducted an empirical study regarding the accuracy of value selection heuristics within incomplete systematic search. We found that global performance measures do not enable us to deduce how adequate heuristic predictions are. By measuring heuristic accuracy as a function of depth, we found that good value selection functions are most error prone when only few branching decisions have been made while the expected relevance of these decisions is greater. However, while bad heuristics may even decay with depth, good knapsack heuristics quickly improve their performance, which then stays practically constant for the remainder of the search.

To devise better value heuristics, we found two aspects to be essential: First, heuristics that quickly generate high-quality no-goods can actually work faster than more accurate heuristics, depending on the search strategy within which they are employed (recall the good performance of the inclusion heuristic within DFS). Second, improved heuristic accuracy is the result of self-enforcement: decisions higher-up in the tree drive the distribution of nodes where the heuristic choice matters in such a way that for these nodes the heuristic works better. In order to devise superior search heuristics, we believe that both aspects deserve to be studied in much more detail.

Finally, we found that inference methods render heuristic decisions far less relevant and thereby improve the robustness of search. Of course, whether invoking expensive inference techniques pays off or not depends largely on the accuracy of the heuristics that are available: After all, a good and robust value selection heuristic has the potential to guide us to feasible or optimal solutions, even when we choose branching variables poorly, or when inference is limited.

With respect to future work, we need to investigate whether the characteristic curve that describes the accuracy of many good heuristics for knapsack instances of various benchmark classes also applies to good heuristics of other combinatorial problems. Based on our findings, we then intend to devise new search strategies that actively incorporate the characteristic evolution of heuristic accuracy during search.

References

1. Balas, E., Carrera, M.: A dynamic subgradient-based branch-and-bound procedure for set covering. Operations Research 44, 875–890 (1996)
2. Beacham, A., Chen, X., Sillito, J., van Beek, P.: Constraint Programming Lessons Learned from Crossword Puzzles. In: Canadian Conference on AI, pp. 78–87 (2001)
3. Cooper, M.C.: An Optimal k-Consistency Algorithm. AI 41, 89–95 (1989)
4. Crowder, H., Johnson, E., Padberg, M.: Solving large scale zero-one linear programming problem. Operations Research 31, 803–834 (1983)
5. Dantzig, G.: Discrete variable extremum problems. Operations Research 5, 226–277 (1957)
6. Fahle, T., Sellman, M.: Cost-Based Filtering for the Constrained Knapsack Problem. AOR 115, 73–93 (2002)
7. Focacci, F., Lodi, A., Milano, M.: Cutting Planes in Constraint Programming. In: CP-AI-OR, pp. 45–51 (2000)
8. Freuder, E.: Backtrack-Free and Backtrack-Bounded Search. In: Search in Artificial Intelligence, pp. 343–369 (1988)
9. Garey, M.R., Johnson, D.S.: Computers and Intractability (1979)

10. Gomes, C., van Hoeve, W., Leahu, L.: The Power of Semidefinite Programming Relaxations for MAXSAT. In: Beck, J.C., Smith, B.M. (eds.) CPAIOR 2006. LNCS, vol. 3990, Springer, Heidelberg (2006)
11. Gomory, R.E.: Outline of an algorithm for integer solutions to linear programs. American Mathematical Society 64, 275–278 (1958)
12. Harvey, W.D., Ginsberg, M.L.: Limited discrepancy search. IJCAI, 607–613 (1997)
13. Hooker, J.N.: A search-infer-and-relax framework for integrating solution methods. In: CPAIOR, pp. 243–257 (2005)
14. Martello, S., Pisinger, D., Toth, P.: Dynamic programming and tight bounds for the 0-1 knapsack problem. Management Science 45, 414–424 (1999)
15. Pisinger, D.: Where are the hard knapsack problems? Computers and Operations Research 32(9), 2271–2284 (2005)
16. Sandholm, T., Suri, S., Gilpin, A., Levine, D.: CABOP: A Fast Optimal Algorithm for Winner Determination in Combinatorial Auctions. Management Science 51(3), 374–390 (2005)
17. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach (2002)
18. Trick, M.: A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. In: CP-AI-OR, pp. 113–124 (2001)
19. Walsh, T.: Depth-bounded discrepancy search. In: IJCAI, pp. 1388–1393 (1997)
20. Williams, R., Gomes, C., Selman, B.: On the Connections between Heavy-tails, Backdoors, and Restarts in Combinatorial search. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, Springer, Heidelberg (2004)

A Novel Approach For Detecting Symmetries in CSP Models

C. Mears[1], M. Garcia de la Banda[1], M. Wallace[1], and B. Demoen[2]

[1] Monash University, Australia
[2] Katholieke Universiteit Leuven, Belgium

Abstract. While several powerful methods exist for automatically detecting symmetries in *instances* of constraint satisfaction problems (CSPs), current methods for detecting symmetries in CSP *models* are limited to the kind of symmetries that can be inferred from the global constraints present in the model. Herein, a new approach for detecting symmetries in CSP models is presented. The approach is based on first applying powerful methods to a sequence of problem instances, and then reasoning on the resulting instance symmetries to infer symmetries of the model. Our results show that this approach deserves further exploration.

1 Introduction

A constraint satisfaction problem (CSP) consists of a set of variables, a set of domains (one per variable), and a set of constraints on the variables. CSPs can often be separated into two parts: the model and the data. The model is a parameterised version of the CSP that, while formally defining the type of variables, domains, and constraints, does not completely determine their number or their values. The data part provides concrete values to the parameters and, as a result, completely determines the number of variables, their domains and the constraints. Thus, while the model represents a *class* of CSPs, the model plus the data specifies an *instance* of that class (i.e., a particular CSP).

For example, the *Latin square problem* of size 3 involves a 3×3 square, where each of the 9 cells in the square takes a value from [1..3], in such a way that each value occurs exactly once in each row and once in each column. The associated CSP can be defined using 9 variables, each with finite domain 1..3, and 18 disequality constraints. Alternatively, it can be separated into a model that is parameterised on the board size N ($N \times N$ variables, each with domain $1..N$, and appropriate constraints), and the data part which simply indicates $N = 3$. Different instances (i.e., CSPs) of the class can be obtained with the same model simply by modifying the value of N in the data.

Solving a CSP can be made more efficient by exploiting the symmetries of the problem. This is because, during search, one can omit parts of the search space that are symmetric to others already explored. If these already explored parts led to a solution, the symmetric search space is known to contain only symmetric solutions (which can be automatically generated without search). If they led to failure, the symmetric search space is known not to contain solutions.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 158–172, 2008.

Considerable progress has been made in the automatic detection of symmetries of CSPs and their exploitation in speeding up the search (e.g., [9,1,11,17,12, 15,8,13,3,5,10,6,7]. Unfortunately, the most powerful methods ([11,1]) can only be applied to a CSP, rather than to its model. Therefore, the symmetries detected can only be used to accelerate the solving process for that CSP, and the cost of detecting these symmetries cannot be amortised over all CSPs in the class. Furthermore, the computation cost of these methods grows with the size of the CSP in such a way as to render them impractical for real-size CSPs.

While there are automatic symmetry detection methods for CSP models [14, 16], to our knowledge, they can only detect a relatively small set of "simple" symmetries (i.e., piecewise value and piecewise variable interchangeability), and only from the global constraints in the model. We propose a radically new approach that (1) uses symmetry detection on a series of small CSPs to elicit candidate symmetries, (2) parameterises these candidate symmetries to be defined over the model rather than over a particular CSP, and (3) determines whether these candidates are indeed symmetries of the model – herein referred to as the *parameterised CSP*. Our results show the approach has considerable potential. Furthermore, we believe the approach can be used to infer from the model many other kinds of information useful for optimisation.

2 Background and Definitions

A CSP is a tuple (X, D, C, dom) where X represents a set of variables, D a set of domains, C a set of constraints, and where dom is a function from X to D, so that $dom(x) \in D$ denotes the domain of variable $x \in X$. By an abuse of notation, when all variables have the same domain, D will simply denote this domain and dom will be omitted.

For a given CSP, a *literal* lit is of the form $x = d$ where $x \in X$ and $d \in dom(x)$. We will use $var(lit)$ to denote its variable x. We denote the set of all literals of a CSP P by $lit(P)$. An *assignment* A is a set of literals. An assignment *over a set of variables* $V \subseteq X$ has exactly one literal $x = d$ for each variable $x \in V$. An assignment over X is called a *complete* assignment.

A constraint c is defined over a set of variables, denoted by $vars(c)$, and specifies a set of *allowed* assignments over $vars(c)$. An assignment over $vars(c)$ that is not allowed by c is *disallowed* by c. An assignment A over $V \subseteq X$ *satisfies* constraint c if $vars(c) \subseteq V$ and the projection of A over $vars(c)$ (i.e., $\{lit \in A | var(lit) \in vars(c)\}$) is allowed by c. A *solution* is a complete assignment that satisfies every constraint in C.

Example 1. The CSP for the *Latin square problem* of size 3 introduced before can be defined as follows:

$$
\begin{aligned}
X &= \{x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}\} \\
D &= \{1, 2, 3\} \\
C &= \{x_{11} \neq x_{12}, x_{11} \neq x_{13}, x_{12} \neq x_{13}, x_{21} \neq x_{22}, x_{21} \neq x_{23}, x_{22} \neq x_{23}, \\
&\quad x_{31} \neq x_{32}, x_{31} \neq x_{33}, x_{32} \neq x_{33}, x_{11} \neq x_{21}, x_{11} \neq x_{31}, x_{21} \neq x_{31}, \\
&\quad x_{12} \neq x_{22}, x_{12} \neq x_{32}, x_{22} \neq x_{32}, x_{13} \neq x_{23}, x_{13} \neq x_{33}, x_{23} \neq x_{33}\}
\end{aligned}
$$

where variable x_{ij} represents the cell in row i, column j. The set $A = \{x_{21} = 1, x_{22} = 2, x_{23} = 2\}$ is an assignment containing 3 literals. While A satisfies constraint $x_{21} \neq x_{22}$ (since $\{x_{21} = 1, x_{22} = 2\}$ is allowed by it), A does not satisfy $x_{22} \neq x_{23}$ (since $\{x_{22} = 2, x_{23} = 2\}$ is disallowed by it).

A *solution symmetry* f of a CSP P is a permutation of $lit(P)$ that preserves the set of solutions [1], i.e., a bijection from literals to literals that maps solutions to solutions. Two important kinds of solution symmetries are induced by permuting either variables or values.

A permutation f of the set of variables X induces a permutation p_f of literals by defining $p_f(x = d)$ as $f(x) = d$. A *variable symmetry* is a permutation of the variables whose induced literal permutation is a solution symmetry [10]. Since the inverse of any such permutation is also a symmetry, we will use $\langle x_1, \ldots, x_n \rangle \leftrightarrow \langle x_{1'}, \ldots, x_{n'} \rangle$, where $x_1, \ldots, x_n, x_{1'}, \ldots, x_{n'} \in X$ to denote the symmetry that maps each x_i to $x_{i'}$ leaving the remaining variables in X unchanged.

A set of domain permutations $f_{dom(x)}$, one for each $x \in X$, induces a permutation p_f of literals by defining $p_f(x = v)$ as $x = f_{dom(x)}(v)$. A *value symmetry* is a set of domain permutations whose induced literal permutation is a solution symmetry [10]. We will use $\langle d_{i1}, \ldots, d_{in} \rangle \leftrightarrow \langle d_{i1'}, \ldots, d_{in'} \rangle$, where $\{d_{i1}, \ldots, d_{in}\} = dom(x_i) = \{d_{i1'}, \ldots, d_{in'}\}$, to denote a value symmetry for $x_i \in X$. A *variable-value* symmetry is any solution symmetry that is not a variable or a value symmetry. Note that it is not necessarily a composition of those variable and value symmetries that exist in the CSP.

Several methods [12,11,1] have been proposed to automatically detect the symmetries of a CSP by constructing its (hyper-)graph representation, and using graph automorphism techniques on it. Our approach uses the technique of Mears et al. [9] since it is more powerful than that of Puget [11] without being as computationally demanding as that of Cohen et al. [1]. However, any such method can be used. The general idea is to (a) represent every literal as a node, (b) represent every assignment disallowed by a constraint as a hyper-edge, and (c) add an edge between every two literals $x = d_1$ and $x = d_2$ where $d_1 \neq d_2$.

Example 2. The Latin square CSP of Example 1 has (a) variable symmetries that swap any columns: $\langle x_{11}, x_{21}, x_{31} \rangle \leftrightarrow \langle x_{12}, x_{22}, x_{32} \rangle$, $\langle x_{11}, x_{21}, x_{31} \rangle \leftrightarrow \langle x_{13}, x_{23}, x_{33} \rangle$, and $\langle x_{12}, x_{22}, x_{32} \rangle \leftrightarrow \langle x_{13}, x_{23}, x_{33} \rangle$, (b) similar variable symmetries that swap any rows, and (c) variable-value symmetries that transpose the rows, column and value dimensions, and correspond to flipping the 3×3 square using a diagonal. The associated graph (left hand side of Figure 1) has 9×3=27 nodes (labelled $^{[i,j]}_{k}$) representing the 27 literals $x_{i,j} = k$ where $i, j, k \in [1..3]$, and (18*3) + (9*3) edges representing the 3 assignments disallowed by each of the 18 constraints, and the 3 extra edges needed to disallow each pair of values of the 9 variables.

Given a hyper-graph $\langle V, E \rangle$, where V is a set of nodes, and E a set of unweighted and undirected hyper-edges, an *automorphism* f of graph $\langle V, E \rangle$ is a permutation of the nodes (i.e., a bijection among nodes) such that $\forall \{n_i, \cdots, n_j\} \in E : \{f(n_i), \cdots, f(n_j)\} \in E$. Since, for a given CSP P, the graph has a node

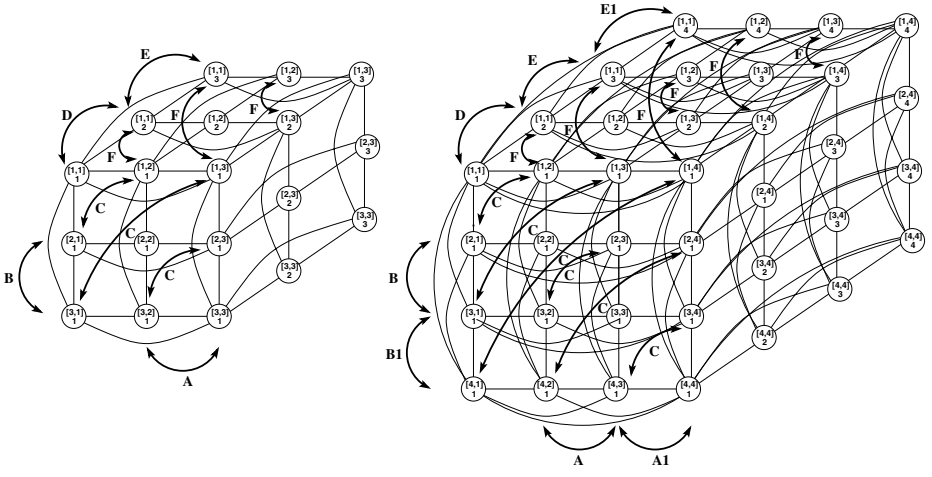


Fig. 1. Graphs and generators for LatinSquare[3] and LatinSquare[4]

for each literal in $lit(P)$, each graph automorphism has a direct interpretation as a permutation of the literals in $lit(P)$ and corresponds to a symmetry of P. Thus, in an abuse of terminology, we will sometimes use *symmetry of a graph* as a shorthand for *automorphism of the graph* associated with a CSP. Standard tools, such as Saucy [2], can compute the automorphisms of a graph and return its symmetry group (i.e., all possible symmetries) by means of a set of generators (a possibly minimal set of symmetries that can be used to generate all others).

Example 3. For the Latin square graph of size 3 given in Example 2, Saucy returns the following set of generators (illustrated in the left hand side of Figure 1):

A $\langle n_{121}, n_{122}, n_{123}, n_{221}, n_{222}, n_{223}, n_{321}, n_{322}, n_{323}\rangle \leftrightarrow$
$\langle n_{131}, n_{132}, n_{133}, n_{231}, n_{232}, n_{233}, n_{331}, n_{332}, n_{333}\rangle$

B $\langle n_{211}, n_{212}, n_{213}, n_{221}, n_{222}, n_{223}, n_{231}, n_{232}, n_{233}\rangle \leftrightarrow$
$\langle n_{311}, n_{312}, n_{313}, n_{321}, n_{322}, n_{323}, n_{331}, n_{332}, n_{333}\rangle$

C $\langle n_{121}, n_{122}, n_{123}, n_{131}, n_{132}, n_{133}, n_{231}, n_{232}, n_{233}\rangle \leftrightarrow$
$\langle n_{211}, n_{212}, n_{213}, n_{311}, n_{312}, n_{313}, n_{321}, n_{322}, n_{323}\rangle$

D $\langle n_{111}, n_{121}, n_{131}, n_{211}, n_{221}, n_{231}, n_{311}, n_{321}, n_{331}\rangle \leftrightarrow$
$\langle n_{112}, n_{122}, n_{132}, n_{212}, n_{222}, n_{232}, n_{312}, n_{322}, n_{332}\rangle$

E $\langle n_{112}, n_{122}, n_{132}, n_{212}, n_{222}, n_{232}, n_{312}, n_{322}, n_{333}\rangle \leftrightarrow$
$\langle n_{113}, n_{123}, n_{133}, n_{213}, n_{223}, n_{233}, n_{313}, n_{323}, n_{333}\rangle$

F $\langle n_{112}, n_{113}, n_{123}, n_{212}, n_{213}, n_{223}, n_{312}, n_{313}, n_{323}\rangle \leftrightarrow$
$\langle n_{121}, n_{131}, n_{132}, n_{221}, n_{231}, n_{232}, n_{321}, n_{331}, n_{332}\rangle$

where node n_{ijk} represents literal $x_{i,j} = k$. **A** states that columns 2 and 3 can be swapped, **B** that rows 2 and 3 can be swapped, **C** that the square can be reflected across the top-left/bottom-right diagonal, **D** that values 1 and 2 can be swapped, **E** that values 2 and 3 can be swapped, and **F** that the second dimension of the square can be swapped with the value dimension. Their combination

results in the symmetries given for Example 2 (e.g., to swap columns 1 and 2 ($\langle x_{11}, x_{21}, x_{31}\rangle \leftrightarrow \langle x_{12}, x_{22}, x_{32}\rangle$) apply first $\mathbf{F}$, then $\mathbf{D}$, and then $\mathbf{F}$).

3 From CSPs to Parameterised CSPs

There is no standard notation to distinguish between a CSP and its parameterised version. Herein, we denote a parameterised CSP as $CSP[Data]$, where $Data$ represents the parameters, and a particular CSP in that class as $CSP[d]$, where d is the value given to $Data$ to yield that CSP. While we will use mathematical notation to specify parameterised CSPs, any high-level modelling language can be used as long as it separates the model from the data, has multi-dimensional arrays of finite domain variables, and supports iteration over them.

Example 4. The parameterised LatinSquare[N] for the CSP of Example 1:

$$\begin{aligned} X[N] = &\ \{square_{ij} | i, j \in [1..N]\} \\ D[N] = &\ [1..N] \\ C[N] = &\ \{square_{ij} \neq square_{ik} | i, j \in [1..N], k \in [j+1..N]\} \cup \\ &\ \{square_{ji} \neq square_{ki} | i, j \in [1..N], k \in [j+1..N]\} \end{aligned}$$

defines $N \times N$ integer decision variables ($square_{ij}$) with values in $[1..N]$, and conjoins the inequality constraints for every row (i) and column (j).

Our aim is to determine the symmetries of every CSP in the class represented by $CSP[Data]$, i.e., the symmetries of $CSP[d]$, for every d possibly given to $Data$. To do so we define the *parameterised* graph $G[Data]$ of $CSP[Data]$ in such as way that, when instantiated by giving a value d to $Data$, $G[d]$ yields the graph of $CSP[d]$. Formally, $G[Data]$ is obtained from $CSP[Data] = (X[Data], D[Data], C[Data], dom[Data])$ as follows:

- $G[Data] = \langle V[Data], E_v[Data] \cup E_c[Data]\rangle$
- $V[Data] = \{x_i = d_i | x_i \in X[Data], d_i \in dom(x_i)[Data]\}$, i.e., $V[Data]$ contains a node for every literal in $CSP[Data]$.
- $E_v[Data] = \{\{x = d_i, x = d_j\} | x \in X[Data], d_i, d_j \in dom(x)[Data], d_i \neq d_j\}$, i.e., an edge exists for every two nodes that map a variable to different values.
- $E_c[Data] = \bigcup_{c \in C[Data]} \{A | vars(A) = vars(c),\ A \textit{ is an assignment disallowed by } c\}$, i.e., a hyper-edge exists for every disallowed assignment A of every constraint c, and connects the nodes associated with all literals in A.

Note that $G[Data]$ is simply a syntactic construct that represents a class of graphs, much as $CSP[Data]$ represents a class of CSPs.

Example 5. The parameterised graph $G[N]$ associated with LatinSquare[N] is as follows. $V[N]$ is defined as $\{n_{ijv} | i, j, v \in [1..N]\}$ where n_{ijv} denotes literal $square_{ij} = v$. $E_v[N]$ is defined as $\{\{n_{ijv_1}, n_{ijv_2}\} | i, j, v_1, v_2 \in [1..N], v_1 \neq v_2\}$, while $E_c[N]$ is obtained by transforming the two constraints in LatinSquare[N] into the set of assignments they disallow:

$$E_c[N] = \{\{n_{ijv}, n_{ikv}\} | i, j, v \in [1..N], k \in [j+1..N]\} \cup \\ \{\{n_{jiv}, n_{kiv}\} | i, j, v \in [1..N], k \in [j+1..N]\}$$

Note that the nodes in $G[N]$ maintain some of the knowledge about the structure of LatinSquare[N] thanks to the reuse of the i and j identifiers appearing in LatinSquare[N]. This is important to automate the construction of the edges in $G[N]$ and, as we will see later, to parameterise symmetries of a CSP.

We can now give a definition of a parameterised symmetry.

Definition 1. *Given a parameterised CSP*[*Data*] *and its parameterised graph* $G[Data]$*, a parameterised permutation* $f[Data]$ *is a bijection of the nodes of* $G[Data]$*. That is, for all values* d *given to Data,* $f[d]$ *permutes the nodes of* $G[d]$*. A parameterised symmetry of CSP*[*Data*] *is a parameterised permutation* $f[Data]$ *of the nodes in* $G[Data]$ *s.t. for all values* d *given to Data,* $f[d]$ *is a symmetry (i.e., an automorphism) of* $G[d]$*.*

We denote by $S[Data]$ the group of parameterised symmetries of $CSP[Data]$. Note that for all values d given to $Data$, $S[d]$ is a subset of the symmetries in $CSP[d]$. The subset is proper if some symmetry in $CSP[d]$ does not apply to all other instances of the CSP. In other words, parameterised symmetries must be determined by information explicitly represented in $CSP[Data]$, without requiring information only present in a particular d.

4 A Framework for Detecting Parameterised Symmetries

As the main concepts of parameterised CSPs and parameterised symmetries have been introduced, we can now turn to the problem of detecting parameterised symmetries for a class of CSPs. Our approach is based on a generic framework which, given a $CSP[Data]$, performs the following steps:

1. Detect symmetries of $CSP[d]$ for a number of values d given to $Data$,
2. Lift them to obtain parameterised permutations of the literals in $CSP[Data]$,
3. Filter the parameterised permutations to keep only those that are likely to be parameterised symmetries,
4. Prove that the selected parameterised permutations are indeed parameterised symmetries.

Note that while the parameterised CSP is a crucial element of our framework, the parameterised graph is currently used only as a means to define parameterised symmetries. However, as shown later, we plan to use the parameterised graph to obtain a better method than we currently have for step four.

4.1 Step One: Detecting Symmetries for Some $CSP[d]$

The first step of our generic framework can be realised in different ways by the choice of parameter values and of symmetry detection method. These choices are

somewhat mutually dependent. For example, using a powerful symmetry detection method will usually force the parameter values to be small. As mentioned before, our implementation uses the detection method of Mears et al. [9], which returns the group of symmetries in a $CSP[d]$ as a set of group generators[1]. Also, our implementation assumes that the parameter $Data$ is a tuple of k integers, $(p_1, p_2, \ldots, p_k)$ and chooses parameter values d by increasing each component of the tuple individually, starting from some user-defined base tuple (typically the smallest meaningful instance of the class).

Example 6. For LatinSquare[N], $Data$ has a single component: the board size N. If the user provides (3) as the base tuple, we increment the component twice obtaining three values for d: (3), (4), and (5). For the social golfers problem (see Section 5), $Data$ has three components: the number of weeks, groups per week and players per group. If $(2, 2, 2)$ is the base tuple, we increment twice each component to get nine values for d (seven of which are distinct): $(2, 2, 2), (3, 2, 2), (4, 2, 2), (2, 2, 2), (2, 3, 2), (2, 4, 2), (2, 2, 3), (2, 2, 2)$, and $(2, 2, 4)$.

4.2 Step Two: Lifting Symmetries to Parameterised Permutations

This step requires taking every symmetry g detected in step one for any of the $CSP[d]$ considered, and determining one or more parameterised permutation(s) $f[Data]$ for which $f[d] = g$. Since computing $f[Data]$ from g alone is quite a task, our implementation uses a much simpler, although incomplete, method: it first defines a set of "common" parameterised symmetries $Per = \{f_1[Data], \cdots, f_m[Data]\}$, and then checks every generator g against them.

The success of our implementation relies on the parameterised CSPs having literals that can be arranged into an n-dimensional matrix, and having parameterised symmetries that permute particular matrix elements, such as rows or columns. These are the kind of "common" symmetries that we will add to Per.

Consider a $CSP[Data]$ with an n-dimensional matrix-like structure $L[Data]$, whose elements correspond to the literals in $CSP[Data]$ (and, thus to the nodes in $G[Data]$). The exact number of elements in each of the n dimensions of $L[Data]$ depends on the value given to $Data$ and can be obtained by means of a function $Dims[Data] = (d_1, d_2, \ldots, d_n)$, where $d_i, i \in [1..n]$ indicates the exact number of elements in the i^{th} dimension.

Example 7. The parameterised LatinSquare[N] problem has a matrix like structure, since its literals can be arranged into a 3-dimensional matrix where each literal $square_{ij} = k$ is indexed as $L[N]_{i,j,k}$. This is clearly visible in Figure 1, where the only difference between $G[3]$ and $G[4]$ are the exact values of each dimension: $Dims[3] = (3, 3, 3)$ while $Dims[4] = (4, 4, 4)$.

Parameterised permutations can then be easily expressed as permutations on the elements of $L[Data]$ *without reference to any specific value d given to $Data$*. This

[1] Since the symmetry detection method chosen is incomplete (i.e., might miss some symmetries), our implementation of the generic framework is also incomplete.

allows us to express a parameterised permutation as a single entity, even though each specific instantiation might involve permuting different nodes. Some common parameterised permutations for a $CSP[Data]$ with n-dimensional matrix $L[Data]$ and $Dims[Data] = (d_1, d_2, \ldots, d_n)$ are:

- **Value swap:** interchanges values v and v' of the k^{th} dimension (e.g., symmetry represented by generator **D** in Figure 1) and is defined as:
  $L[Data]_{i_1,\ldots,i_{k-1},v,i_{k+1},\ldots,i_n} \leftrightarrow$
  $L[Data]_{i_1,\ldots,i_{k-1},v',i_{k+1},\ldots,i_n}$ $\forall i_j \in [1..d_j], j \in [1..n]$.
- **All values swap:** interchanges all values of the k^{th} dimension (e.g. symmetries represented by generators **D**, **E**, and their combinations in Figure 1) and is defined as: $L[Data]_{i_1,\ldots,i_{k-1},v,i_{k+1},\ldots,i_n} \leftrightarrow L[Data]_{i_1,\ldots,i_{k-1},v',i_{k+1},\ldots,i_n}$, $\forall v, v' \in [1..d_k], v \neq v'$, $i_j \in [1..d_j], j \in [1..n]$.
- **Dimension invert:** interchanges every value v of the k^{th} dimension with value $n - v + 1$ (e.g., symmetry represented by generator **A** and by generator **A1** in Figure 2) and is defined as: $L[Data]_{i_1,\ldots,i_{k-1},v,i_{k+1},\ldots,i_n} \leftrightarrow L[Data]_{i_1,\ldots,i_{k-1},n-v+1,i_{k+1},\ldots,i_n}$, $\forall v \in [1..d_k]$, $i_j \in [1..d_j], j \in [1..n]$.
- **Dimension swap:** swaps k^{th} and k'^{th} dimensions (e.g., symmetry represented by generator **B** in Figure 2) and is defined as:
  $L[Data]_{i_1,\ldots,i_{k-1},i_k,i_{k+1},\ldots,i_{k'-1},i_{k'},i_{k'+1},\ldots,i_n} \leftrightarrow$
  $L[Data]_{i_1,\ldots,i_{k-1},i_{k'},i_{k+1},\ldots,i_{k'-1},i_k,i_{k'+1},\ldots,i_n}$, $\forall i_j \in [1..d_j], j \in [1..n]$.

Example 8. The generators found for LatinSquare[3] in Example 3 can be automatically matched to the following parameterised permutations for $L[N]$:

A value swap with $k = 2, v = 2, v' = 3$: $L[N]_{i2l} \leftrightarrow L[N]_{i3l}, \forall i, l \in [1..N]$
B value swap with $k = 1, v = 2, v' = 3$: $L[N]_{2jl} \leftrightarrow L[N]_{3jl}, \forall j, l \in [1..N]$
C dimension swap with $k = 1, k' = 2$: $L[N]_{ijl} \leftrightarrow L[N]_{jil}, \forall i, j, l \in [1..N]$
D value swap with $k = 3, v = 1, v' = 2$: $L[N]_{ij1} \leftrightarrow L[N]_{ij2}, \forall i, j \in [1..N]$
E value swap with $k = 3, v = 2, v' = 3$: $L[N]_{ij2} \leftrightarrow L[N]_{ij3}, \forall i, j \in [1..N]$
F dimension swap with $k = 2, k' = 3$: $L[N]_{ijl} \leftrightarrow L[N]_{ikl}, \forall i, j, l \in [1..N]$

Consider the graph $G[4]$ associated with LatinSquare[4], shown in the right hand side of Figure 1. Saucy finds 9 generators for this graph. Six of them are simple extensions of those found for $G[3]$. For example, the extension of **A** is:

A $\langle n_{121}, n_{122}, n_{123}, n_{124}, n_{221}, n_{222}, \ldots, n_{321}, \ldots, n_{421}, \ldots\rangle \leftrightarrow$
$\langle n_{131}, n_{132}, n_{133}, n_{134}, n_{231}, n_{232}, \ldots, n_{331}, \ldots, n_{431}, \ldots\rangle$

and similarly for **B, C, D, E** and **F**. The other three generators found are:

A1 $\langle n_{131}, n_{132}, n_{133}, n_{134}, n_{231}, n_{232}, \ldots, n_{331}, \ldots, n_{431}, \ldots\rangle \leftrightarrow$
$\langle n_{141}, n_{142}, n_{143}, n_{144}, n_{241}, n_{242}, \ldots, n_{341}, \ldots, n_{441}, \ldots\rangle$
B1 $\langle n_{311}, n_{312}, n_{313}, n_{314}, n_{321}, n_{322}, \ldots, n_{331}, \ldots, n_{341}, \ldots\rangle \leftrightarrow$
$\langle n_{411}, n_{412}, n_{413}, n_{414}, n_{421}, n_{422}, \ldots, n_{431}, \ldots, n_{441}, \ldots\rangle$
E1 $\langle n_{113}, n_{123}, n_{133}, n_{143}, n_{213}, n_{223}, \ldots, n_{313}, \ldots, n_{413}, \ldots\rangle \leftrightarrow$
$\langle n_{114}, n_{124}, n_{134}, n_{144}, n_{214}, n_{224}, \ldots, n_{314}, \ldots, n_{414}, \ldots\rangle$

The generators **A, B, C, D, E** and **F** in $G[4]$ match the parameterised permutations used for $G[3]$, while **A1, B1** and **E1** match value swap with:

A1 $k = 2, v = 3, v' = 4$: $L[N]_{i3l} \leftrightarrow L[N]_{i4l}, \forall i, l \in [1..N]$
B1 $k = 1, v = 3, v' = 4$: $L[N]_{3jl} \leftrightarrow L[N]_{4jl}, \forall j, l \in [1..N]$
E1 $k = 3, v = 3, v' = 4$: $L[N]_{ij3} \leftrightarrow L[N]_{ij4}, \forall i, j \in [1..N]$

The generators found by Saucy for LatinSquare[5] are the simple extensions of **A, A1, B, B1, C, D, E, E1** and **F** (which can be parameterised as before), plus three more **A2, B2**, and **E2**, which can be parameterised as:

A2 $k = 2, v = 4, v' = 5$: $L[N]_{i3l} \leftrightarrow L[N]_{i4l}, \forall i, l \in [1..N]$
B2 $k = 1, v = 4, v' = 5$: $L[N]_{3jl} \leftrightarrow L[N]_{4jl}, \forall j, l \in [1..N]$
E2 $k = 3, v = 4, v' = 5$: $L[N]_{ij3} \leftrightarrow L[N]_{ij4}, \forall i, j \in [1..N]$

Considering a symmetry g in isolation is not always productive. This is because some parameterised permutation patterns, when instantiated, correspond to a group of symmetries rather than to a single symmetry. For example, the "all values swap" pattern (which interchanges all values in a dimension) is a combination of at least two generator symmetries. Thus, to detect such a parameterised pattern we cannot simply parameterise each symmetry on its own; we must consider groups of symmetries $\{g_1, \cdots, g_m\}$ such that $f[d] = \{g_1, \cdots, g_m\}$.

For the "all value swap" case, we group symmetries by keeping track of any pair of value-swap pattern symmetries which operate on the same dimension and whose interchanged values overlap. These are combined into a single symmetry stating that all values involved can be freely interchanged. Our implementation considers the "all values swap" pattern matched if, by applying this kind of combination until a fixpoint is reached, we obtain a symmetry that interchanges all $[1..d_k]$, where d_k is the value returned by $Dims[d]$ for dimension k.

Example 9. The generators **D** and **E** for LatinSquare[3] form an instance of the "all value swap" pattern $L[N]_{ijv} \leftrightarrow L[N]_{ijv'}, \forall v, v' \in [1..N], v \neq v'$, $i, j \in [1..N]$. The generators **D**, **E** and **E1** for LatinSquare[4] form the same pattern.

4.3 Step Three: Filtering Parameterised Permutations

Step two identifies our candidate parameterised symmetries. However, it is likely that some of these candidates apply only to a few instances, rather than to the entire class. We would like to eliminate unlikely permutations before performing the (possibly expensive) proof step. Our implementation uses a simple (and again incomplete) heuristic which selects as likely candidates the intersection of the parameterised permutations present in all tested instances.

Unfortunately, the success of such an intersection relies on Saucy returning the same (or equivalent) set of generators for each $CSP[d]$. This is because, as mentioned before, our implementation only attempts to parameterise the generators returned by Saucy (as opposed to every symmetry in the group), and a group can be obtained from many different sets of generators. We can solve this problem as follows. If a particular parameterised permutation is found in more than one instance but not in all, we check the group of symmetries of the other instances to see if the permutation is, in fact, present. This is done via the GAP system for computational group theory [4]. If the parameterised permutation is indeed found in all instances, it is marked as a candidate.

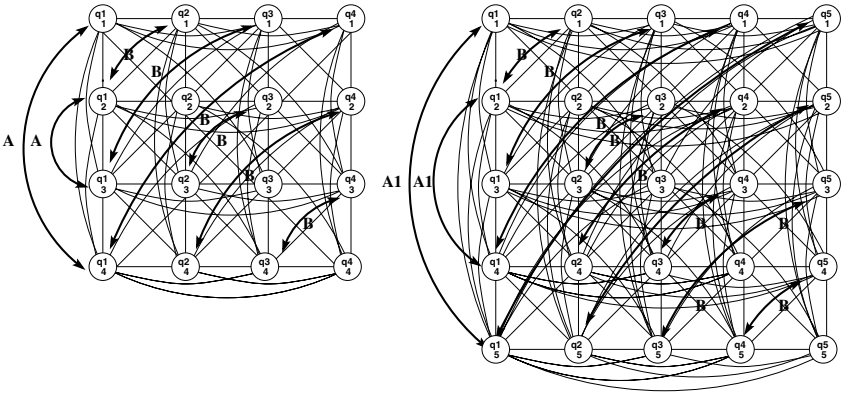


Fig. 2. Graph instances for Queens[4] and Queens[5]

Example 10. Consider the social golfers problem with values of d being $(2, 2, 2)$, $(3, 2, 2)$, $(4, 2, 2)$, $(2, 3, 2)$, $(2, 4, 2)$, $(2, 2, 3)$, $(2, 2, 4)$. Our implementation finds an instance of the "all value swap" pattern for the third dimension (golfers are interchangeable) for every value of d. However, the "all value swap" pattern for the first dimension (the weeks are interchangeable) is found for only 5 out of the 7 values of d, due to the particular generators given by Saucy. Searching explicitly for this pattern in the groups found for the other values of d shows that it is indeed present in all of them and can thus be considered a likely candidate.

4.4 Step Four: Proving Class Symmetries

This last step can be achieved, for example, by first representing both the parameterised CSP and the candidate parameterised permutation in the logic formalism described in [8], and then making use of theorem proving techniques. Of course, such a technique is in general undecidable. We are currently exploring an alternative approach that we hope will be more successful: to use graph techniques to prove that our likely candidate is an automorphism of the parameterised graph $G[Data]$. This is however not straightforward, since $G[Data]$ is not really a graph, but a syntactic construct that represents a class of graphs.

5 Detailed Examples

We have seen how our current implementation automatically detects as likely candidates all parameterised symmetries in LatinSquare[N]. This is, however, not always the case. Here we provide three other examples: Queens, for which it again detects all parameterised symmetries as likely candidates, Social golfers, for which it also detects all symmetries (after adding a new pattern), and Golomb ruler, for which it fails to detect any likely candidate.

Queens: aims at positioning N queens on an $N \times N$ chess board without attacking each other. The following parameterised CSP Queens[N] uses N integer variables (each being the row in which the queen appears) with domains in $[1..N]$.

$$
\begin{aligned}
X[N] = &\ \{q_i | i \in [1..N]\} \\
D[N] = &\ [1..N] \\
C[N] = &\ \{q_i \neq q_j | i \in [1..N], j \in [i+1..N]\} \cup \\
&\ \{q_i + i \neq q_j + j | i \in [1..N], j \in [j+1..N]\} \cup \\
&\ \{q_i - i \neq q_j - j | i \in [1..N], j \in [j+1..N]\}
\end{aligned}
$$

Its parameterised graph $G[N] = (V[N], E_c[N] \cup E_v[N])$ is:

$$
\begin{aligned}
V[N] = &\ \{q_{iv} | i, v \in [1..N]\} \\
E_c[N] = &\ \{\{q_{iv}, q_{jv}\} | i, v \in [1..N], j \in [i+1..N])\} \cup \\
&\ \{\{q_{iv_i}, q_{jv_j}\} | i, v_i, v_j \in [1..N], j \in [i+1..N], v_i + i = v_j + j)\} \cup \\
&\ \{\{q_{iv_i}, q_{jv_j}\} | i, v_i, v_j \in [1..N], j \in [i+1..N], v_i - i = v_j - j)\} \\
E_v[N] = &\ \{\{q_{iv_i}, q_{jv_j}\} | i, v_i, v_j \in [1..N], v_i \neq v_j\}
\end{aligned}
$$

where node q_{iv} represents literal $q_i = v$. Given the initial base tuple (4), our implementation generates $G[4]$, $G[5]$ and $G[6]$. Figure 2 shows the graph instances $G[4]$ and $G[5]$, together with the generators found by Saucy. For $G[4]$ it finds:

A $\langle q_{11}, q_{12}, q_{21}, q_{22}, q_{31}, q_{32}, q_{41}, q_{42} \rangle \leftrightarrow \langle q_{14}, q_{13}, q_{24}, q_{23}, q_{34}, q_{33}, q_{44}, q_{43} \rangle$
B $\langle q_{12}, q_{13}, q_{14}, q_{23}, q_{24}, q_{34} \rangle \leftrightarrow \langle q_{21}, q_{31}, q_{41}, q_{32}, q_{42}, q_{43} \rangle$

which can be parameterised to match:

A dimension invert with $k=2$: $L[N]_{iv} \leftrightarrow L[N]_{i(N-v+1)}, \forall v, i \in [1..N]$
B dimension swap with $k=1$ and $k'=2$: $L[N]_{ij} \leftrightarrow L[N]_{ji}, \forall i, j \in [1..N]$

and for $G[5]$ Saucy finds:

A1 $\langle q_{11}, q_{12}, q_{21}, q_{22}, q_{31}, q_{32}, q_{41}, q_{42}, q_{51}, q_{52} \rangle \leftrightarrow$
$\langle q_{15}, q_{14}, q_{25}, q_{24}, q_{35}, q_{34}, q_{45}, q_{44}, q_{55}, q_{54} \rangle$
B $\langle q_{12}, q_{13}, q_{14}, q_{15}, q_{23}, q_{24}, q_{25}, q_{34}, q_{35}, q_{54} \rangle \leftrightarrow$
$\langle q_{21}, q_{31}, q_{41}, q_{41}, q_{32}, q_{42}, q_{52}, q_{43}, q_{53}, q_{45} \rangle$

where **B** is an extension of the generator with the same name found for $G[4]$ (and matches the same dimension swap pattern), and **A1** is a new generator that matches the same dimension invert pattern as **A**. The generators found for $G[6]$ are, again, an extension of **B** that matches the dimension swap pattern, and a new generator **A2** that matches the same pattern as **A** and **A1**. The intersection of the patterns results in both being marked as likely candidates.

Social Golfers: aims at building a schedule of W weeks, with G equally-sized groups per week, and P golfers per group, such that each pair of golfers may play in the same group at most once. A parameterised CSP Golf$[W, G, P]$ is:

$$
\begin{aligned}
X[W,G,P] &= \{players_{wg} | w \in [1..W], g \in [1..G]\} \\
D[W,G,P] &= \wp(\{1..P * G\}) \\
C[W,G,P] &= \{|players_{wg}| = P | w \in [1..W], g \in [1..G]\} \cup \\
&\quad \{|players_{wg_1} \cap players_{wg_2}| = 0 | w \in [1..W], g_1, g_2 \in [1..G], g_1 < g_2\} \cup \\
&\quad \{|players_{w_1 g_1} \cap players_{w_2 g_2}| \leq 1 | w_1, w_2 \in [1..W], w_1 < w_2, g_1, g_2 \in [1..G], g_1 < g_2\}
\end{aligned}
$$

where $\wp$ is the powerset. The associated parameterised graph $G[W, G, P]$ is:

$$V[W,G,P] = \{n_{wgp} | w \in [1..W], g \in [1..G], p \in \wp([1..P*G])\}$$
$$E_c[W,G,P] = \{n_{wgp} | w \in [1..W], g \in [1..G], |p| \neq P\} \cup$$
$$\{\langle n_{wg_1p_1}, n_{wg_2p_2}\rangle | w \in [1..W], g_1, g_2 \in G, g_1 < g_2, p_1, p_2 \in \wp([1..P*G]), |p_1 \cap p_2| \neq 0)\} \cup$$
$$\{\langle n_{w_1g_1p_1}, n_{w_2g_2p_2}\rangle | w_1, w_2 \in [1..W], w_1 < w_2, g_1, g_2 \in G, g_1 < g_2, p_1, p_2 \in \wp([1..P*G]), |p_1 \cap p_2| > 1\}$$
$$E_v[W,G,P] = \{\langle n_{wgp,p}, n_{wgp_2}\rangle | w \in [1..W], g \in [1..G], p_1, p_2 \in \wp([1..P*G]), p_1 \neq p_2\}$$

where node n_{wgp} represents literal $players_{wg} = p$. The parameterised versions of the generators found for $G[2,2,2]$ are:

A $\{n_{ija} \leftrightarrow n_{ijb} | i \in [1..W], j \in [1..G], a, b \in \wp([1..P*G]), 1 \in a; b = (a \setminus \{1\}) \cup \{2\}\}$
B $\{n_{ija} \leftrightarrow n_{ijb} | i \in [1..W], j \in [1..G], a, b \in \wp([1..P*G]), 2 \in a; b = (a \setminus \{2\}) \cup \{3\}\}$
C $\{n_{ija} \leftrightarrow n_{ijb} | i \in [1..W], j \in [1..G], a, b \in \wp([1..P*G]), 3 \in a; b = (a \setminus \{3\}) \cup \{4\}\}$
D $\{n_{11v} \leftrightarrow n_{12v} | v \in \wp([1..P*G])\}$
E $\{n_{21v} \leftrightarrow n_{22v} | v \in \wp([1..P*G])\}$
F $\{n_{1jv} \leftrightarrow n_{2jv} | j \in [1..G], v \in \wp([1..P*G])\}$

Generators **A**, **B** and **C** represent symmetries that swap golfers 1 with 2, 2 with 3, and 3 with 4, respectively. Taken together, our implementation detects the combined all value swap permutation pattern that states that all golfers are interchangeable. Generator **F** represents the symmetry that swaps week 1 and week 2. This trivially matches the all value swap pattern that states that all weeks are interchangeable, and also the dimension invert pattern that reflects the weeks. Generators **D** and **E** represent symmetries that swap groups 1 and 2 within week 1, and within week 2, respectively. Our implementation did not consider parameterised patterns that perform a swap on only a subset of the literals and, thus, failed to detect such pattern as likely candidate. However, once we extended the set of patterns to include one that represents the interchanging of values within a particular row or column, this symmetry was captured.

The generators for $G[2,3,2]$, $G[2,2,3]$, $G[2,4,2]$ and $G[2,2,4]$ include the extended versions of generators **A** to **F** in $G[2,2,2]$, plus additional generators representing the interchangeability of the extra golfers, and of the extra groups. As before, our implementation detects the combined all value swap pattern that states that all golfers are interchangeable and that all weeks are interchangeable. With the inclusion of the pattern mentioned above, the interchangeability of groups within each week is also marked as likely candidate.

The generators for $G[3,2,2]$ include the extended versions of **A**, **B**, **C**, **D** and **E**. However, Saucy produces generators that do not have a simple parameterisation. The situation for $G[4,2,2]$ is similar. But since the weeks were found to be interchangeable in all of the other instances, the implementation consults GAP to check whether this holds for $[3,2,2]$ and $[4,2,2]$, even though the generators from Saucy don't directly correspond to it. GAP indicates that it does and, therefore, the symmetry is marked as a likely candidate.

Golomb ruler: is defined as a set of N integers (marks on the ruler) $a_1, \ldots, a_N$ such that the $\frac{N(N-1)}{2}$ differences $a_j - a_i, 1 \leq i < j \leq N$ are distinct. The problem involves finding a valid set of N marks. The following parameterised CSP Golomb[N] uses N integer variables (the marks) with domains in $[0..N^2]$, plus $\frac{N(N-1)}{2}$ integer variables (the differences) with domains $[1..N^2]$.

$$
\begin{aligned}
X[N] = &\quad \{mark_i | i \in [0..N]\} \cup \{diff_{ij} | i \in [1..N], j \in [i+1..N]\} \\
D[N] = &\quad \{[0..N^2], [1..N^2]\} \\
C[N] = &\quad \{mark_i - mark_j = diff_{ij} | i \in [1..N], j \in [i+1..N]\} \cup \\
&\quad \{diff_{ij} \neq diff_{ik} | i, j \in [1..N], k \in [j+1..N]\} \\
dom(m_i) = [0..N^2] \; &; \; dom(d_{ij}) = [1..N^2]
\end{aligned}
$$

The parameterised graph associated with Golomb[N] is:

$$
\begin{aligned}
V[N] = &\{m_{iv} | i \in [1..N], v \in [0..N^2]\} \cup \\
&\{d_{jiv} | i \in [1..N], j \in [(i+1)..N], v \in [1..N^2]\} \\
E_c[N] = &\{\{m_{iv_1}, m_{jv_2}, d_{ijv_3}\} | i \in [1..N], j \in [(i+1)..N], \; v_1, v_2, v_3 \in [1..N^2], v_1 - v_2 \neq v_3\} \cup \\
&\{\{d_{ijv}, d_{ijv}\} | i \in [1..N], j \in [(i+1)..N], v \in [1..N^2]\} \\
E_v[N] = &\{(m_{iv_1}, m_{iv_2}) | i \in [1..N], v_1, v_2 \in [1..N^2], v_1 \neq v_2\} \cup \\
&\{(d_{ijv_1}, d_{ijv_2}) | i \in [1..N], j \in [(i+1)..N], v_1, v_2 \in [1..N^2], v_1 \neq v_2\}
\end{aligned}
$$

where node m_{iv} represents literal $mark_i = v$ and node d_{ijv} literal $diffs_{ij} = v$. The generator found by Saucy for $G[3]$ is:

$$
\begin{aligned}
\mathbf{A} \; &\langle d_{121}, d_{122}, d_{123}, d_{124}, d_{125}, d_{126}, d_{127}, d_{128}, d_{129} \rangle \leftrightarrow \\
&\langle d_{231}, d_{232}, d_{233}, d_{234}, d_{235}, d_{236}, d_{237}, d_{238}, d_{239} \rangle \text{ plus} \\
&\langle m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, m_{15}, m_{16}, m_{17}, \ldots, m_{24} \rangle \leftrightarrow \\
&\langle m_{39}, m_{38}, m_{37}, m_{36}, m_{35}, m_{34}, m_{33}, m_{32}, \ldots, m_{25} \rangle
\end{aligned}
$$

which swaps the lengths of the spaces between the marks, i.e., turns the ruler back-to-front. This symmetry involves variables from two separate matrices, d_{ij} and m_i, and our simple implementation cannot yet handle this. Even if we only consider the search variables m_i, our implementation would need to obtain for $G[3]$, $G[4]$ and $G[5]$ the pattern $\{m_{iv} \leftrightarrow m_{jv'} | i, j \in [1..N], i = N - j + 1, v, v' \in [0..N^2], v = N^2 - v' + 1\}$. Since our implementation currently does not take this pattern into account, it cannot recognise the symmetry as likely candidate.

6 Results

Let us evaluate our simple implementation (which includes the patterns described in Section 4.2 plus the additional pattern described for Social Golfers) over a set of problems that include those discussed earlier, plus the following.

Balanced Incomplete Block Design: with parameters (v, b, k, r, λ), where the task is to arrange v objects into b blocks such that each block has exactly k objects, each object is in exactly r blocks, and every pair of objects occurs together in λ blocks. The objects are interchangeable and the blocks are interchangeable.

Graceful Graph: with parameters (m, n), where the edges (a, b) of the graph $K_m \times P_n$ are labeled by $|a - b|$, and there is no two edges with the same label. The corresponding vertices in each clique are simultaneously interchangeable, the order of the cliques is reversible, and the values are reversible.

N × N queens: where an $N \times N$ chessboard is coloured with N colours, so that a pair of queens in any two squares of the same colour do not attack each other. The symmetries are those of the chessboard, plus the colours are interchangeable.

Table 1. Symmetry detection results

Problem	Tuple	Amount	Symmetries	Time	Instance
BIBD	(2,2,2,2,2)	+3	objects blocks	19.0	20%
Social Golfers	(2,2,2)	+2	rows groups players	376.4	96%
Golomb Ruler	(3)	+3	flip X	6.7	99%
Graceful Graph	(2,2)	+3	intra-clique path-reverse value	9.0	44%
Latin Square	(3)	+3	dimensions value	13.7	10%
N × N queens	(4)	+3	chessboard colours	8.0	21%
Queens (int)	(8)	+3	chessboard	3.6	36%
Queens (bool)	(8)	+3	chessboard	5.4	64%
Steiner Triples	(3)	+3	triples value	16.8	32%

Queens (bool): which uses a Boolean matrix model for the Queens problem of Section 5. The symmetries are those of the chessboard.

Steiner Triples: where the task is to find $\frac{n(n-1)}{6}$ triples of distinct integers from 1 to n, such that any pair of triples has at most one element in common. The triples are interchangeable and the values are interchangeable.

Table 1 shows the results, where the columns indicate the problem name, the base tuple, the amount by which each component is increased, the known symmetries and whether they are found by our implementation, the total running time in seconds, and the percentage of that time spent in detection (as opposed to parametrisation). The experiments were run on an dual Intel Core 2 1.86GHz computer with 1GB of memory. No effort has been made to optimise detection time; the times are included simply to show the practicality of the approach.

7 Conclusions

The automatic detection of CSP symmetries is currently either restricted to problem instances, or limited to the class of symmetries that can be inferred from the global constraints present in the model. This paper provides a radically new framework that takes advantage of existing (and future) powerful detection methods defined for problem instances, by generalising their results to models without requiring them to use any particular syntax. We provide a very simple (and incomplete) implementation that requires the problem to have matrix-like structure and only considers a pre-determined number of model symmetries (those that correspond to permutations of the objects in the matrix). While this is a very limited implementation of the general framework, it is nonetheless capable of detecting symmetries that could previously only be detected for

instances. Of course, more complete implementations of the framework will be able to detect even more kinds of symmetries.

We now plan to integrate in our implementation techniques to validate or reject likely candidates. While theorem proving techniques are an obvious possibility, we are also investigating graph techniques that rely on the parameterised graph and which we think will be more efficient and complete.

References

1. Cohen, D., Jeavons, P., Jefferson, C., Petrie, K.E., Smith, B.M.: Symmetry Definitions for Constraint Satisfaction Problems. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 17–31. Springer, Heidelberg (2005)
2. Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting Structure in Symmetry Generation for CNF. In: 41st Design Automation Conference, pp. 530–534 (2004)
3. Frisch, A.M., Miguel, I., Walsh, T.: CGRASS: A System for Transforming Constraint Satisfaction Problems. In: O'Sullivan, B. (ed.) CologNet 2002. LNCS (LNAI), vol. 2627, pp. 15–30. Springer, Heidelberg (2003)
4. The GAP Group. GAP – Groups, Algorithms, and Programming, Version 4.4.9 (2006)
5. Gent, I.P., Harvey, W., Kelsey, T., Linton, S.: Generic SBDD using Computational Group Theory. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 333–347. Springer, Heidelberg (2003)
6. Gent, I.P., Smith, B.M.: Symmetry Breaking in Constraint Programming. In: ECAI 2000. 14th European Conference on Artificial Intelligence (2000)
7. Haselböck, A.: Exploiting Interchangeabilities in Constraint-Satisfaction Problems. In: IJCAI 1993, pp. 282–289 (1993)
8. Mancini, T., Cadoli, M.: Detecting and Breaking Symmetries by Reasoning on Problem Specifications. In: Zucker, J.-D., Saitta, L. (eds.) SARA 2005. LNCS (LNAI), vol. 3607, Springer, Heidelberg (2005)
9. Mears, C., de la Banda, M.G., Wallace, M.: On Implementing Symmetry Detection. In: SymCon 2006 (2006)
10. Puget, J.-F.: Symmetry Breaking Revisited. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 446–461. Springer, Heidelberg (2002)
11. Puget, J.-F.: Automatic Detection of Variable and Value Symmetries. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 475–489. Springer, Heidelberg (2005)
12. Romani, A., Markov, I.L.: Automatically Exploiting Symmetries in Constraint Programming. In: Danelutto, M., Vanneschi, M., Laforenza, D. (eds.) Euro-Par 2004. LNCS, vol. 3149, pp. 98–112. Springer, Heidelberg (2004)
13. Roney-Dougal, C.M., Gent, I.P., Kelsey, T., Linton, S.: Tractable Symmetry Breaking using Restricted Search Trees. In: ECAI 2004 (2004)
14. Roy, P., Pachet, F.: Using Symmetry of Global Constraints to Speed Up the Resolution of Constraint Satisfaction Problems. In: ECAI 1998 Workshop on Non-binary Constraints, pp. 27–33 (1998)
15. Sellmann, M., Van Hentenryck, P.: Structural Symmetry Breaking. In: IJCAI 2005 (2005)
16. Van Hentenryck, P., Flener, P., Pearson, J., Ågren, M.: Compositional Derivation of Symmetries for Constraint Satisfaction. In: Zucker, J.-D., Saitta, L. (eds.) SARA 2005. LNCS (LNAI), vol. 3607, Springer, Heidelberg (2005)
17. Walsh, T.: General Symmetry Breaking Constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, Springer, Heidelberg (2006)

Amsaa: A Multistep Anticipatory Algorithm for Online Stochastic Combinatorial Optimization

Luc Mercier and Pascal Van Hentenryck

Brown University, Box 1910, Providence, RI 02912, USA

Abstract. The one-step anticipatory algorithm (*1s-AA*) is an online algorithm making decisions under uncertainty by ignoring future non-anticipativity constraints. It makes near-optimal decisions on a variety of online stochastic combinatorial problems in dynamic fleet management, reservation systems, and more.

Here we consider applications in which *1s-AA* is not as close to the optimum and propose *Amsaa*, an anytime multi-step anticipatory algorithm. *Amsaa* combines techniques from three different fields to make decisions online. It uses the sampling average approximation method from stochastic programming to approximate the problem; solves the resulting problem using a search algorithm for Markov decision processes from artificial intelligence; and uses a discrete optimization algorithm for guiding the search.

Amsaa was evaluated on a stochastic project scheduling application from the pharmaceutical industry featuring endogenous observations of the uncertainty. The experimental results show that *Amsaa* significantly outperforms state-of-the-art algorithms on this application under various time constraints.

1 Introduction

In recent years, progress in telecommunication and in information technologies has generated a wealth of Online Stochastic Combinatorial Optimization (OSCO) problems. These applications require to make decisions under time constraints, given stochastic information about the future. Anticipatory algorithms have been proposed to address these applications [18]. We call an algorithm *anticipatory* if, at some point, it anticipates the future, meaning that it makes some use of the value of the clairvoyant. These anticipatory algorithms typically rely on two black-boxes: a conditional sampler to generate scenarios consistent with past observations and an offline solver for the deterministic version of the combinatorial optimization problem.

1s-AA is a simple one-step anticipatory algorithm. It works by transforming the multi-stage stochastic optimization problem into a 2-stage one by ignoring all non-anticipativity constraints but those of the current decision. This 2-stage problem is approximated by sampling, and the approximated problem is solved optimally by computing the offline optimal solutions for all pairs (scenario,decision). *1s-AA* was shown to be very effective on a variety of OSCO problems in dynamic fleet management [3,2], reservation systems [18], resource allocation [15], and jobshop scheduling [17]. Moreover, a quantity called the global anticipatory gap (GAG) was introduced by [14] to measure the stochasticity of the application and that paper showed that *1s-AA* returns high-quality solutions when the GAG is small.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 173–187, 2008.

Here we consider OSCO applications with a significant GAG and propose to address them with *Amsaa*, a multi-step anticipatory algorithm which provides an innovative integration of techniques from stochastic programming, artificial intelligence, and discrete optimization. Like *1s-AA*, *Amsaa* samples the distribution to generate scenarios of the future. Contrary to *1s-AA* however, *Amsaa* approximates and solves the multi-stage problem. The SAA problem is solved by an exact search algorithm [4] using anticipatory relaxations as a heuristic to guide the search.

Amsaa was evaluated on a stochastic resource-constrained project scheduling problem (S-RCPSP) proposed in [6] to model the design and testing of molecules in a pharmaceutical company. This problem is highly combinatorial because of precedence and cumulative resource constraints. It is also stochastic: the durations, costs, and results of the tasks are all uncertain. The S-RCPSP features what we call *endogenous observations*: the uncertainty about a task can only be observed by executing it. This contrasts with online stochastic combinatorial optimization (OSCO) problems studied earlier, in which the observations were exogenous, and leads to significant GAGs [8]. More generally, *Amsaa* applies to a class of problems that we call *Stoxuno problems* (STochastic Optimization with eXogenous Uncertainty and eNdogenous Observations). The experimental results indicate that *Amsaa* outperforms a wide variety of existing algorithms on this application.

The rest of the paper is organized as follows. Sections 2 and 3 describe the motivating problem and introduce Stoxuno problems. Section 4 presents the background in Markov Decision Processes and dynamic programming. Section 5 introduces the concept of Exogenous MDPs (X-MDPs) to model Stoxuno and exogenous problems. Section 6 describes *Amsaa*. Section 7 presents extensive experimental results. Section 8 compares *Amsaa* with a mathematical programming approach. Section 9 concludes the paper and discusses research opportunities.

2 A Stochastic Project Scheduling Problem

This section describes the *stochastic resource-constrained project scheduling problem* (S-RCPSP), a problem from the pharmaceutical industry [6]. A pharmaceutical company has a number of candidate molecules that can be commercialized if shown successful, and a number of laboratories to test them. Each molecule is associated to a project consisting of a sequence of tasks to be executed in order. A task is not preemptive and cannot be aborted once started. Its duration, cost, and result (failure, which ends the project, or success, which allows the project to continue) are uncertain. The realization of a task is a triplet (duration, cost, result). A project is successful if all its tasks are successful. A successful project generates a revenue which is a given decreasing function of its completion date. The goal is to schedule the tasks in the laboratories, satisfying the resource constraints (no more running tasks than the number of labs at any given time), to maximize the expected profit. The profit is the difference between the total revenues and the total cost. There is no obligation to schedule a task when a lab is available and there are tasks ready to start. Indeed, it is allowed to drop a project (never schedule a task ready to start), as well as to wait some time before starting a task. Waiting is sometimes optimal, like in dynamic fleet management [2].

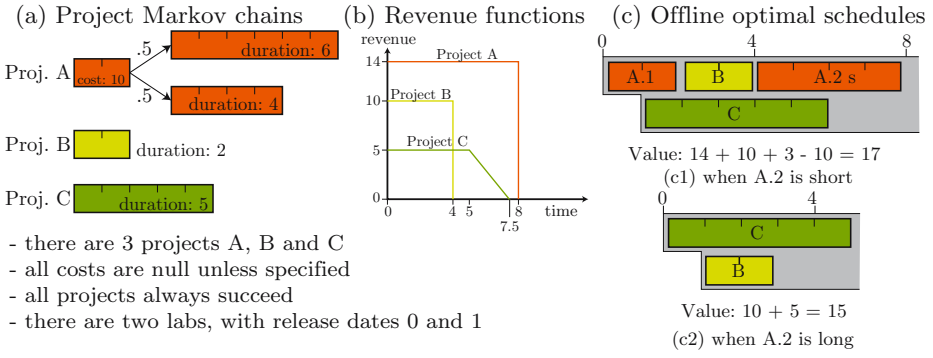


Fig. 1. An Instance of the Stochastic Project Scheduling Problem

Each project is modeled by its own finite heterogeneous first-order Markov chain. That is, for each task, the set of possible realizations is known. The Markov chain, which is given, provides the distribution of the realization of the first task, and the probability transition matrices that, for any realization of the i-th task, gives the distribution of the realization of the $(i+1)$-th task.

Figure 1 depicts a small instance to illustrate these concepts. In this instance, there are 3 projects and 4 tasks, and all the projects always succeed. In this instance, the offline optimal schedules for the two possible realizations, which are shown in Figure 1(c), differ at the first decision when the uncertainty is not yet resolved. Hence the optimal online policy is necessarily inferior to a perfect clairvoyant decision maker. The schedule in Figure 1(c2) is the optimal online solution.

3 Exogeneity and Endogeneity: Problem Classification

Traditionally, stochastic optimization problems were separated into two classes according to the exogenous or endogenous nature of their uncertainty. To delineate precisely the scope of *Amsaa*, we need to refine this classification.

Purely exogenous problems are those in which the uncertainty, and the way it is observed, is independent of the decisions. Customers and suppliers behavior is considered exogenous [18], as well as nature (e.g., water inflow in hydroelectric power scheduling), and prices in perfect markets. In this class, there is a natural concept of scenario (e.g., the sequence of customer requests) and, given two scenarios, it is possible to compute when they become distinguishable.

Purely endogenous problems are those for which there is no natural concept of scenarios. Most benchmark problems for Markov Decision Processes are of this nature. Problems in robotics where the uncertainty comes from the actuators are endogenous.

Stoxuno Problems (STochastic Optimization problems with eXogenous Uncertainty and eNdogenous Observations) are applications like the S-RCPSP, for which the underlying uncertainty is exogenous, but observations depend on the decisions. In these problems, the concept of scenario is natural. However, given two scenarios, it is not possible to decide when a decision maker will be able to distinguish them. Many scheduling

problems with uncertainty on tasks should belong to this category. The lot sizing problem in [10] also falls into that category.

Amsaa applies to both purely exogenous and Stoxuno problems.

4 Background in Stochastic Dynamic Programming

Stochastic Dynamic Programming is a field of research that aims at solving stochastic optimization problems modelled as Markov Decision Processes (MDPs). MDPs can model purely endogenous problems, purely exogenous, and Stoxuno problems. We only consider finite horizon MDPs with no reward discounting and no transition costs.

Markov Decision Processes. An MDP $(S, s_0, F, X, \perp, \mathscr{X}, f, \mathscr{P})$ consists of:

- a state space S, an initial state $s_0 \in S$, and a set of final states $F \subseteq S$.
- a decision space X containing a decision $\perp$ (denoting no action) and a function $\mathscr{X} : S \to X$ returning the set of feasible decisions in a given state such that $\forall s \in S, 0 < \#\mathscr{X}(s) < \infty$ and that $\forall s \in F, \mathscr{X}(s) = \{\perp\}$.
- a bounded reward function $f : F \to \mathbb{R}$.
- a transition function $\mathscr{P} : S \times X \to \text{prob}(S)$, where $\text{prob}(S)$ is the set of probability distributions over S, satisfying $\forall s \in F, \mathscr{P}(s, \perp)(\{s\}) = 1$.

For convenience, we write $\mathscr{P}(\cdot|s,x)$ instead of $\mathscr{P}(s,x)(\cdot)$. A run of an MDP $(S, s_0, F, X, \perp, \mathscr{X}, f, \mathscr{P})$ starts in the initial state s_0. In a given state s, the decision maker selects a decision $x \in \mathscr{X}(s)$ which initiates a transition to state $s' \in A \subseteq S$ with probability $\mathscr{P}(A|s,x)$. The resulting sequence of states and decisions, i.e. $s_0 \xrightarrow{x_0} s_1 \xrightarrow{x_1} \ldots \xrightarrow{x_{t-1}} s_t \xrightarrow{x_t} \ldots$, is called a trajectory. This is a Markovian process: conditionally on s_i and x_i, the distribution of s_{i+1} is independent of the past trajectory.

We assume horizon finiteness: there exists an integer T such that all trajectories starting in s_0 are such that s_T is final. As a corollary, the state space graph has to be acyclic. The objective of the decision maker is to maximize $\mathbb{E}[f(\boldsymbol{s}_T)]$.

Policies, Value functions, and Optimality. A (deterministic) Markovian policy $\pi : S \to X$ is a map from states to feasible decisions, i.e., that satisfies $\forall s \in S, \pi(s) \in \mathscr{X}(s)$. The value $v_\pi(s)$ of policy π in state s is the expected value obtained by running policy π from state s. A policy π is optimal if the value $v_\pi(s_0)$ is maximal among all policies.

A value function v is a map $S \to \mathbb{R}$. The Q-value function canonically associated to v is the mapping $S \times X \to \mathbb{R}$ defined by $Q(s,x) = \mathbb{E}_{\mathscr{P}}[v(\boldsymbol{s'})|s,x]$, which, in the case of finite state space, becomes $Q(s,x) = \sum_{s' \in S} v(s')\mathscr{P}(s'|s,x)$. Given a value function v and a state s, a decision $x \in \mathscr{X}(s)$ is *greedy* if $Q(s,x) = \max_{x' \in \mathscr{X}(s)} Q(s,x')$. We assume that there is a rule to break ties, so we can talk about "the" greedy decision even though it is not unique. The *greedy policy* π_v associated with a value function v is the policy defined by taking the greedy decision in every state. A value function is optimal if the associated greedy policy is optimal. A necessary and sufficient condition for v to be optimal is that, for all state s reachable under π_v, we have $v(s) = f(s)$ if s is final, and $Res_v(s) = 0$ otherwise, where $Res_v(s) = v(s) - \max Q(s,x)$ is called the *Bellman residual* of v at s. Under our assumptions, there is always an optimal value function $v^\star$.

5 Exogenous Markov Decision Processes

Section 3 discussed the *nature* of the uncertainty. MDPs can model problems of any nature, but *represents* the uncertainty endogenously. For exogenous problems, it is better to use a model that represents the uncertainty exogenously. Stochastic programs are an example of such models, but they cannot model Stoxuno problems. Therefore we introduce exogenous MDPs (X-MDPs) that allow the modeling of purely exogenous and of Stoxuno problems. They are neither more nor less expressive than traditional MDPs [18], but have computational advantages discussed at length in Section 6.

Model and Definitions. An exogenous Markov decision process (X-MDP) $(S, s_0, F, X, \perp, \mathscr{X}, f, \boldsymbol{\xi}, \mu_\xi, \tau)$ consists of:

- a state space S, an initial state $s_0 \in S$, and a set of final states $F \subseteq S$.
- a decision space X containing a decision $\perp$ (denoting no action) and a function $\mathscr{X} : S \to X$ returning the set of feasible decisions in a given state such that $\forall s \in S, 0 < \#\mathscr{X}(s) < \infty$ and that $\forall s \in F, \mathscr{X}(s) = \{\perp\}$.
- a bounded reward function $f : F \to \mathbb{R}$.
- a random variable $\boldsymbol{\xi}$, with values in a scenario space Ξ, and distribution μ_ξ.
- a (deterministic) transition function $\tau : S \times X \times \Xi \to S$ satisfying $\forall s \in S,\ \forall \xi \in \Xi,\ \tau(s, \perp, \xi) = s$.

Running an X-MDP consists of first sampling a realization ξ of the random variable $\boldsymbol{\xi}$. The decision maker doesn't know ξ, but it makes inferences by observing transition outcomes. Starting in s_0, it makes a decision, observes of the outcome of the transition, and repeats the process. For a state s and a decision x, the next state becomes $\tau(s, x, \xi)$. The alternation of decisions and state updates defines a trajectory $s_0 \xrightarrow[\xi]{x_0} s_1 \xrightarrow[\xi]{x_1} \dots \xrightarrow[\xi]{x_{t-1}} s_t$ satisfying (i) for all i, $x_i \in \mathscr{X}(s_i)$ and (ii) for all i, $s_{i+1} = \tau(s_t, x_t, \xi)$.

Like for MDPs, we assume finite horizon: there is a T such that s_T is final regardless of the decisions made and of $\boldsymbol{\xi}$. The objective also consists of maximizing $\mathbb{E}[f(\boldsymbol{s}_T)]$, which is always defined if f is bounded. We will also restrict attention to Markovian policies; in this order, we need to introduce a new concept before specifying the property that ensures their dominance.

In an X-MDP, scenario ξ is *compatible* with a trajectory $s_0 \xrightarrow{x_0} s_1 \xrightarrow{x_1} \dots \xrightarrow{x_{t-1}} s_t$ if $\tau(s_i, x_i, \xi) = s_{i+1}$ for all $i < t$. $\mathscr{C}\left(s_0 \xrightarrow{x_0} \dots \xrightarrow{x_{t-1}} s_t\right)$ is the set of such scenarios. A scenario is *compatible* with a state s if it is compatible with a trajectory from s_0 to s, and $\mathscr{C}(s)$ is the set of such scenarios.

The *Markov property for X-MDPs*, which ensures the dominance of Markovian policies, then reads:

$$\text{for all trajectory } s_0 \xrightarrow{x_0} \dots \xrightarrow{x_{t-1}} s_t, \quad \mathscr{C}\left(s_0 \xrightarrow{x_0} \dots \xrightarrow{x_{t-1}} s_t\right) = \mathscr{C}(s_t). \tag{1}$$

It will be easy to enforce this property in practice: simply include all past observations into the current state. An elementary but important corollary of this assumption is that

conditional probabilities on the past trajectory are identical to conditional probabilities on the current state, i.e.,

$$\forall A \subseteq \Xi, \quad \mathbb{P}\left(\boldsymbol{\xi} \in A \,\middle|\, \boldsymbol{\xi} \in \mathscr{C}\left(s_0 \xrightarrow{x_0} \dots \xrightarrow{x_{t-1}} s_t\right)\right) = \mathbb{P}\left(\boldsymbol{\xi} \in A \mid \boldsymbol{\xi} \in \mathscr{C}(s_t)\right),$$

Hence, sampling scenarios conditionally on the current state is equivalent to sampling scenarios conditionally on the past trajectory.

X-MDPs naturally exhibit an underlying *deterministic* and *offline* problem that has no counterpart in MDPs. The *offline value* of state s under scenario ξ, denoted by $\mathscr{O}(s,\xi)$, is the largest reward of a final state reachable from state s when $\boldsymbol{\xi} = \xi$. It is defined recursively by:

$$\mathscr{O}(s,\xi) = \begin{cases} f(s) & \text{if } s \text{ is final,} \\ \max_{x \in \mathscr{X}(s)} \mathscr{O}(\tau(s,x,\xi),\xi) & \text{otherwise.} \end{cases}$$

Consider the instance shown in Figure 1. If ξ_s and ξ_l denote the scenarios in which A.2 is short and long respectively, then $\mathscr{O}(s_0,\xi_s) = 17$ and $\mathscr{O}(s_0,\xi_l) = 15$.

Policies and Optimality for X-MDPs. Like for MDPs, it is possible to define the value of a policy for an X-MDP. Let A be an X-MDP and $\pi : S \to X$ be a policy for A. Consider a past trajectory $s_0 \xrightarrow{x_0} \dots \xrightarrow{x_{t-1}} s_t$, not necessarily generated by π. Remember that for any sequence of decisions s_T is final. Therefore the expected value obtained by following π after this past trajectory is well defined and is denoted by $v_\pi\left(s_0 \xrightarrow{x_0} \dots \xrightarrow{x_{t-1}} s_t\right)$. By the Markov property, this quantity only depends on s_t, so we denote it $\pi(s_t)$. A policy π is optimal if the value $v_\pi(s_0)$ is maximal among all policies.

Modelling the Stochastic RCPSP as an X-MDP. It is easy to model the S-RCPSP as an X-MDP. A state contains: (1) the current time, (2) the set of currently running tasks with their start times (but without lab assignment), and (3) the set of all past observed task realizations. Thanks to (3) the Markov property for X-MDPs is satisfied.

6 Amsaa: An Algorithm for Decision Making in X-MDPs

Overview of *Amsaa*. This section presents a high-level overview of *Amsaa*, the Anytime Multi-Step Anticipatory Algorithm, which aims at producing high-quality decisions for X-MDPs. Its pseudo-code follows.

Because we want an *anytime* algorithm, that is, one that can be stopped and return something at any time, there is an outer loop for which the condition can be anything. In an operational setting, it will most likely be a time constraint (e.g., "make a decision within a minute"), and in a prospective setting, it could be a stopping criteria based on some accuracy measure (for example, the contamination method [9]).

Amsaa's first step is to approximate the X-MDP to make it more tractable. It then converts it to an MDP in order to apply standard search algorithm for MDPs. This search is guided by an upper bound that exploits the existence of offline problems due to the exogenous nature of the uncertainty. For efficiency, lines 3–4 are incremental, so

Function. `Amsaa` (*X-MDP A*)

1 **while** *some condition* **do**
2 Approximate the X-MDP A by replacing $\boldsymbol{\xi}$ with a random variable $\boldsymbol{\xi}'$ whose support is smaller, or refine the current approximation.
3 Convert the resulting X-MDP to a standard MDP.
4 Solve the resulting MDP with a search algorithm for MDPs, using the *offline upper bound* $h_{\mathbb{E},\max}(s) = \mathbb{E}\left[\mathscr{O}(s,\boldsymbol{\xi}') \,\middle|\, \boldsymbol{\xi}' \in \mathscr{C}(s)\right]$.
5 **return** the greedy decision at the root node of the MDP.

that when the approximation is refined (line 1), the amount of work to be done is small if the refinement does not change the approximated problem too much.

We will now present the details of the approximation, of the convertion to an MDP, of the MDP solving, and, finally, of the incrementality.

Approximating the X-MDP by Sampling. The first step of *Amsaa* is to approximate the original X-MDP by replacing the distribution of the scenarios by one with a finite and reasonably small support. The simplest way of doing so is by sampling. For stochastic programs, this idea is called the Sample Average Approximation (SAA) method [16], and it can be extended to X-MDPs. Suppose we want a distribution whose support has cardinality at most n: just sample $\boldsymbol{\xi}$ n times, independently or not, to obtain $\boldsymbol{\xi}^1,\ldots,\boldsymbol{\xi}^n$ and define $\hat{\boldsymbol{\mu}}_n$ as the empirical distribution induced by this sample, that is, the distribution that assigns probability $1/n$ to each of the sampled scenarios. Some results of the SAA theory translate to X-MDPs. In particular, if Ξ is finite and the sampling iid, then the SAA technique produces almost surely optimal decisions with enough scenarios.

Benefits of Exterior Sampling for X-MDPs. Sampling can be used either to compute an optimal policy for an approximated problem (The SAA method, used in *Amsaa*); or to compute an approximately optimal policy for the original problem, like in [12], who proposed an algorithm to solve approximately an MDP by sampling a number of outcomes at each visited state (interior sampling). Their algorithm was presented for discounted rewards but generalizes to finite horizon MDPs. We argue that the SAA method is superior because sampling internally does not exploit a fundamental advantage of problems with exogenous uncertainty: *positive correlations*.

Indeed, in a state s, the optimal decision maximizes $Q^\star(s,x)$, where $Q^\star$ is the Q-value function associated to the optimal value function $v^\star$. However, estimating this value precisely is not important. What really matters is to estimate the sign of the difference $Q^\star(s,x_1) - Q^\star(s,x_2)$ for each pair of decisions $x_1,x_2 \in \mathscr{X}(s)$. Now, consider two functions g and h mapping scenarios to reals, for example the optimal policy value obtained from a state s after making a first decision. That is, $g(\xi) = v^\star(\tau(s_0,x_1,\xi))$ and $h(\xi) = v^\star(\tau(s_0,x_2,\xi))$ for two decisions $x_1,x_2 \in \mathscr{X}(s_0)$. If $\boldsymbol{\xi}^1$ and $\boldsymbol{\xi}^2$ are iid scenarios:

$$\operatorname{var}\left(g(\boldsymbol{\xi}^1) - h(\boldsymbol{\xi}^2)\right) = \operatorname{var}\left(g(\boldsymbol{\xi}^1)\right) + \operatorname{var}\left(h(\boldsymbol{\xi}^2)\right),$$
$$\operatorname{var}\left(g(\boldsymbol{\xi}^1) - h(\boldsymbol{\xi}^1)\right) = \operatorname{var}\left(g(\boldsymbol{\xi}^1)\right) + \operatorname{var}\left(h(\boldsymbol{\xi}^1)\right) - 2\operatorname{cov}\left(g(\boldsymbol{\xi}^1), h(\boldsymbol{\xi}^1)\right),$$

and therefore $\operatorname{var}\left(g(\boldsymbol{\xi}^1) - h(\boldsymbol{\xi}^1)\right) = \left(1 - \operatorname{acorr}\left(g(\boldsymbol{\xi}^1), h(\boldsymbol{\xi}^1)\right)\right) \cdot \operatorname{var}\left(g(\boldsymbol{\xi}^1) - h(\boldsymbol{\xi}^1)\right)$, where $\operatorname{acorr}(X,Y) = \frac{\operatorname{cov}(X,Y)}{1/2(\operatorname{var}(X)+\operatorname{var}(Y))}$ is a quantity we call *arithmetic correlation*. Note

that $\text{acorr}(X,Y) \approx \text{corr}(X,Y)$ when $\text{var}(X)$ and $\text{var}(Y)$ are close. Now consider an infinite iid sample $\boldsymbol{\xi}^1, \boldsymbol{\xi}^{1'}, \boldsymbol{\xi}^2, \boldsymbol{\xi}^{2'}, \ldots$, and a large integer n. By the central limit theorem, the distributions of $\frac{1}{n}\sum_{i=1}^{n} g(\boldsymbol{\xi}^i) - h(\boldsymbol{\xi}^i)$ and of $\frac{1}{n\gamma}\sum_{i=1}^{n\gamma} g(\boldsymbol{\xi}^i) - h(\boldsymbol{\xi}^{i'})$ are almost the same when $1/\gamma = 1 - \text{acorr}\left(g(\boldsymbol{\xi}^1), h(\boldsymbol{\xi}^1)\right)$. Therefore, for some specified accuracy, the number of required scenarios to estimate the expected difference between $g(\boldsymbol{\xi})$ and $h(\boldsymbol{\xi})$ is reduced by this factor γ when the same scenarios (exterior sampling) are used instead of independent scenarios (interior sampling).

This argument is not new, and can be found for example in [16]. However, no empirical evidence of high correlations were given, which we now report. Consider an SAA problem approximating the standard instance of the S-RCPSP application with 200 scenarios generated by iid sampling, and consider the optimal policy values in the initial state for the 6 possible initial decisions (the first is to schedule nothing, the others are to schedule the first task of each project). Associating a column with each decision, the values for the first five scenarios are:

$$OptPolicyValue = 1e4 \times \begin{pmatrix} 0 & 2.110 & 2.038 & 1.910 & 1.893 & 2.170 \\ 0 & -0.265 & -0.275 & -0.275 & -0.275 & -0.225 \\ 0 & -0.205 & -0.230 & -0.230 & -0.170 & -0.170 \\ 0 & 1.375 & 1.405 & 1.279 & 1.345 & 1.365 \\ 0 & 1.045 & 1.070 & 1.015 & 1.105 & 1.160 \end{pmatrix}$$

The correlation is evident. Excluding the first decision (which is uncorrelated to the others), the arithmetic correlations range from 94% to 99%, computed on the 200 scenarios. Moreover, the minimal correlation is 98.7% among the second, third, and fourth decisions, which are the three good candidates for being selected as the initial decision.

It remains to see whether these correlations are a characteristic of the problem or even of the instance. In most OSCO problems, some scenarios are more favorable than others regardless of the decisions, causing these correlations: in the S-RCPSP, scenarios with many successful projects bring more money than scenarios with many failures, and this is very visible on the matrix above. As a result we conjecture that, for most OSCO problems, exterior sampling converges with far fewer scenarios than interior sampling.

Converting the X-MDP into an MDP. This is the second step of *Amsaa*.

Definition 1. *Given an X-MDP A with state-space S and final states set F, the* trimmed *X-MDP B induced by A is the X-MDP that is in all equal to A, except:*

1. *its state space is* $S' = \{s \in S | \mathscr{C}(s) \neq \varnothing\}$;
2. *its set of final states is* $F' = F \cup \{s \in S' | \#\mathscr{C}(s) = 1\}$, *and the function* $\mathscr{X}$ *is modified accordingly;*
3. *its reward function* f' *is defined, for states* $s \in F' \setminus F$, *by* $f(s) = \mathscr{O}(s,\xi)$, *where* ξ *is the unique scenario compatible with s.*

A trimmed X-MDP is equivalent to the original one, in the sense that an optimal policy in A induces an optimal policy in B and vice versa.

Definition 2. *Let* $B = (S, s_0, F, X, \perp, \mathscr{X}, f, \boldsymbol{\xi}, \mu_\xi, \tau)$ *be the trimmed X-MDP induced by an X-MDP A. Define* $\mathscr{P}$ *from* $S \times X$ *to the set of probability distributions on S by:*

$$\forall s \in S, x \in X, U \subseteq S, \quad \mathscr{P}(U|s,x) = \mathbb{P}\left(\tau(s,x,\boldsymbol{\xi}) \in U \mid \boldsymbol{\xi} \in \mathscr{C}(s)\right).$$

Then $C = (S, s_0, F, X, \bot, \mathscr{X}, f, \mathscr{P})$ *is the MDP* induced *by X-MDP A.*

The induced MDP is equivalent to the original problem in the following sense.

Theorem 1. *Let A be an X-MDP, C the induced MDP, and* π *be a policy that is optimal in A for states in* $F' \setminus F$*. Then, for all states* $s \in S'$, $v^A_\pi(s) = v^C_\pi(s)$.

This theorem is a consequence of the Markov property for X-MDPs, which implies that, following π in B or C, for all t the distribution of $\boldsymbol{s}_t$ is the same in B and in C.

Solving MDPs. Once the approximated X-MDP is converted into an MDP, it is possible to apply existing algorithms for solving the MDP exactly. We use a*heuristic search algorithm*, which, despite its name, is an exact algorithms.

Heuristic Search Algorithms for MDPs. Heuristic search algorithms for MDPs perform a partial exploration of the state space, using a — possibly monotone — upper bound to guide the search. A value function $h : S \to \mathbb{R}$ is an *upper bound* if $\forall s \in S$, $h(s) \geq v^\star(s)$, and is a *monotone upper bound* if, in addition, $Res_h(s) \geq 0$ for all state s. A monotone upper bound is an optimistic evaluation of a state that cannot become more optimistic if a Bellman update is performed.

Function. `findRevise(MDP A)`

precondition: h is a upper bound for A, $h(s) = f(s)$ if s is final
foreach $s \in S$ **do** $v(s) \leftarrow h(s)$
repeat
  Pick a state s reachable from s_0 and π_v with $|Res_v(s)| > 0$
  $v(s) \leftarrow \max_{x \in \mathscr{X}(s)} Q(s,x)$
until *no such state is found*
return v

Function `findAndRevise`, introduced by [4], captures the general schema of heuristic search algorithm for MDPs and returns an optimal value function upon termination. At each step, the algorithm selects a state reachable with the current policy π_v whose Bellman residual is non-zero and performs a Bellman update. When h is monotone, only strictly positive (instead of non-zero) Bellman residuals must be considered. Different instantiations of this generic schema differ in the choice of the state to reconsider. They include, among others, HDP [4], Learning Depth-First Search (LDFS) [5], Real-Time Dynamic Programming (RTDP) [1], Bounded RTDP [13], and LAO* [11]. These algorithms only manipulate partial value functions defined only on the states visited so far, performing the initialization $v(s) \leftarrow h(s)$ on demand. We chose to use the acyclic version of Learning Depth-First Search (a-LDFS). It applies to acyclic problems (ours are), and requires a monotone upper bound, which we have.

The Upper Bound $h_{\mathbb{E},\max}$. The performance of heuristic search algorithms strongly depends on the heuristic function h. For MDPs induced by X-MDPs, a good heuristic function can be derived from the deterministic offline problems. More precisely, for a state s, the heuristic consists of solving the deterministic offline problems for the scenarios compatible with s in the original X-MDP and taking the resulting expected offline value, i.e., $h_{\mathbb{E},\max}(s) = \mathbb{E}_\mu \left[\mathscr{O}(s, \boldsymbol{\xi}) \mid \boldsymbol{\xi} \in \mathscr{C}(s) \right]$, where μ is $\boldsymbol{\xi}$'s distribution. Function

$h_{\mathbb{E},\max}$ is a monotone upper bound. It is attractive for guiding the search because it leverages the combinatorial structure of the application (black-box offline solver) and can be computed efficiently because the sets $\mathscr{C}(s)$ are finite and small. $h_{\mathbb{E},\max}$ provides a significant computational advantage to X-MDPs over MDPs.

Incrementality and Anytime Decision Making. Incrementality is the ability to resolve the MDP quickly after a small change in the approximated problem. Incrementality enables fine-grained refinement, providing for efficient anytime decision making and openning the door to sequential sampling [7]. It is based on the following theorem.

Theorem 2. *Let $\mathscr{A}$, $\mathscr{B}$ and $\mathscr{C}$ be three X-MDPs that differ only by their respective distributions μ, ν, and ρ and let $\rho = \lambda\mu + (1-\lambda)\nu$ for some $0 < \lambda < 1$. Let h_μ and h_ν be monotone upper bounds for $\mathscr{A}$ and $\mathscr{B}$ respectively. Define $h : S \to \mathbb{R}$ by $h(s) = -\infty$ if $\rho(\mathscr{C}(s)) = 0$, and otherwise by*

$$h(s) = \frac{1}{\rho(\mathscr{C}(s))}\Big(\ \lambda\mu(\mathscr{C}(s))\ h_\mu(s)\ \ +\ \ (1-\lambda)\nu(\mathscr{C}(s))h_\nu(s)\ \Big).$$

Then h is a monotone upper bound for the induced MDP of $\mathscr{C}$.

This theorem is used in the following setting. μ is the old sample distribution, and we have solved $\mathscr{A}$ optimally with `findAndRevise()`. The optimal value function it returned is the monotone upper bound $h_\mu(s)$. ν is the distribution of the newly added scenarios, and h_ν is the $h^\nu_{\mathbb{E},\max}$, the offline upper bound for $\mathscr{B}$. ρ is the new sample distribution, and includes the old sample and the newly added scenarios. λ is the weight of the old sample in the new sample. Our experiments showed adding the scenarios one-by-one instead of all at once produced only a 20% slowdown on 500-scenario problems.

7 Experimental Results on Anytime Decision Making

Experimental Setting. The benchmarks are based on the collection of 12 instances for the S-RCPSP from [8]. The reference instance, Reg, is similar to the one in [6]. It has 2 laboratories, 5 projects, and a total of 17 tasks. The number of realizations for each tasks range from 3 to 7, giving a total of 10^9 possible scenarios. The 11 other instances are variant of Reg, scaling the costs, scaling the time axis of the revenue functions, or chaning the structure of the Markov chains for each molecule.

For each instance, we generated 1,000 realizations of the uncertainty. A *run* of an algorithm on one of these realizations consists of simulating one trajectory in the X-MDP. At each encountered state, the online algorithm takes a decision with hard time constraints. If the online algorithm has not enough time to decide, a default decision, closing the labs, is applied. The algorithms were tested on all the realizations and various time limits. With 4 tested algorithms and time limits of 31 ms, 125 ms, 500 ms, 2s, 8s, 32s, this gives a total of 288,000 runs taking more than 8,000 hours of cpu time.

The Four Compared Algorithms. Amsaa was used with iid sampling and sample sizes growing by increments of 10%. Its performance relies on a fast offline solver. We used the branch and bound algorithm from [8] whose upper bound relaxes the resource constraints for the remaining tasks. This branch and bound is very fast thanks to a good

preprocessing step: it takes on average less than 1ms for the reference instance. *1s-AA* is the one-step anticipatory algorithm with iid sampling. It uses the same offline solver than *Amsaa*. BRTDP is the Bounded Real Time Dynamic Programming algorithm [13]. The lower bound $h^-(s)$ correspond to not scheduling anything after state s. The upper bound is $h^+(s)$ is a very slight relaxation of $h_{\max,\max}$, using the offline solver on an hypothetical best scenario. Like in RTDP, and as opposed to B-RTDP, decisions are taken greedily with respect to the upper bound value function v^+: Indeed experimental results showed that making decisions with respect to v^- provides very poor decisions. HC-DP is the Heuristically Confined Dynamic Programming algorithm from [6] enhanced into an anytime algorithm. The offline learning phase is removed and performed within the given time limits. A full Bellman update is performed at increasing larger intervals, so that the decision can be updated. Less than half the computation time is spent doing updates, the rest being spent exploring the state-space. Its results outperform those of the original HC-DP algorithm in [6].

The Performance of *Amsaa*. Figure 2 summarizes the results for anytime decision making. It contains a table for each of the 12 instances. The first line of this table contains the empirical mean value obtained by running *Amsaa*. The three lines below report the relative gap between the expected value of the considered algorithm and *Amsaa* with the same time constraint. In addition, the background color carries information about the statistical significance of the results, at the 5% level, as indicated by the legend of the figure. It indicates whether the considered algorithm is better than *Amsaa*-32s (no occurrence here); not worse than *Amsaa*-32s (e.g., *Amsaa*-500ms on Cost2); significantly worse than *Amsaa*-32s, but better than *Amsaa*-31ms (e.g., *1s-AA*-31 ms on P3); worse than *Amsaa*-32s, but not than *Amsaa*-31ms (e.g., B-RTDP-2s on Agr); or worse than *Amsaa*-31ms (e.g., HC-DP-32s on Reg).

Overall *Amsaa* exhibits excellent performance. The solution quality of *Amsaa*-32s is often higher by at least 10% than *1s-AA*-32s, HC-DP-32s, and B-RTDP-32s and *Amsaa* is robust across all instances. With 32s, *Amsaa* is significantly better than all other algorithms on 11 instances and as good as any other algorithm on Cost5. Moreover, the remaining three algorithms lacks robustness with respect to the instances: They all rank second and last at least once. Note that, on Cost5, the optimal policy is not to schedule anything. HC-DP is able to realize that quickly, with only 125 ms, because it uses very fast heuristics. *Amsaa*-32s and HC-DP with at least 125ms are optimal on this problem.

Amsaa is also robust with respect to the available computation time. On most instances, the rankings of the four algorithms do not vary much with respect to the computation times. One might think that with very strong time constraints, *1s-AA* is preferable to *Amsaa*, because *1s-AA* can use more scenarios in the same amount of time. Yet, there are only two instances on which *1s-AA*-31ms beats *Amsaa*-31ms (Agr and P3) and 3 on which they exhibit similar results. Note that B-RTDP-31ms has a zero score on many instances due to the fact that even a single B-RTDP trial has to go deep in the state space and compute the bounds h^+ and h^- for many states. Under such strict time constraints, B-RTDP cannot even perform one trial before the deadline.

Empirical Complexity of *Amsaa*. Figure 3(a) shows how the sample size grows with the available runtime on instance Reg, measured on the making of the initial decision. Because *Amsaa* is exponential in the worst case, one might fear that the number of

scenarios grows logarithmically with the runtime. Yet, a power model for the expected sample size $\mathbb{E}[n]$ as a function of the computation time t fits almost perfectly the empirical data. The fitted model is $\mathbb{E}[n] = 105 \times t^{0.61}$, which indicates that *Amsaa*'s execution time grows subquadratically in the number of scenarios ($1/0.61 = 1.64 < 2$)

However, one may argue that this behavior may be a consequence of iid sampling and is not a convincing evidence that *Amsaa* performs well. Indeed, in the case of a continuous distribution of the uncertainty, all the scenarios would almost surely be dispatched to different states after the first observation and *Amsaa* with iid sampling would have a linear complexity. The stochastic RCPSP has finite distributions but a similar behavior, i.e., a fast divergence of the scenarios, could explain its good performance.

To test whether this is the case, we measured the number of states in the trimmed approximated X-MDP that are reachable by an optimal policy, as depicted on figure 3(b). With a continuous distribution, the number of reachable states would almost surely be $n+1$ for n scenarios: the root node and n leaves. If observations were Bernoulli random variables with parameter $1/2$, the solution state space would be a roughly balanced binary tree with $2n-1$ nodes. These two extreme cases suggest to fit a linear model of the form $(nb\ reachable\ states) = a + bn$. Such a model fits perfectly the experimental results with a slope of 1.96, making it much closer to a Bernoulli case than a continuous distribution. This provides evidence that scenarios do not diverge too quickly with iid sampling and that the SAA problems become harder with the number of scenarios.

Comparison with Gap Reduction Techniques. The following table reports the relative gap (in %) between [8]'s best algorithm, called $\mathscr{A}_{TEPR}$, based on gap reduction techniques, and *Amsaa*-32s. The background color provides significance information: on Cost2 and R.6, $\mathscr{A}_{TEPR}$ beats *Amsaa*-32s at the 5% significance level. On Reg, Cost5, and R1.5, none is better than the other. On D.6 gap reduction is worse than *Amsaa*-31ms, and on the others gap reduction is worse than *Amsaa*-32s but better than *Amsaa*-31ms.

Reg	Agr	Cost2	Cost5	D.6	D1.5	P1	P2	P3	P4	R.6	R1.5
-0.24	-1.11	+9.96	0.00	-16.8	-0.43	-1.98	-2.80	-0.57	-0.62	+5.40	+0.39

Gap reduction techniques are an attractive alternative to *Amsaa*. Nethertheless, *Amsaa* outperforms them on most instances here, sometimes with a large gap (17% on D.6), and converges to the optimal decisions (gap reduction techniques do not).

8 Comparison with Mathematical Programming

Stochastic programming traditionally focuses on purely exogenous problems. However, [10] proposed an integer programming (IP) formulation for SAA problems of a Stoxuno lot-sizing problem. We investigated a similar approach for the solving of SAA problems for S-RCPSP using an IP closely following model $(P2)$ in [10]. In this model, the number of binary variables is quadratic in the number of scenarios and linear in the time horizon. A 20-scenario problem generated by iid sampling had, after CPLEX's presolve, $47 \cdot 10^3$ binary variable and $20 \cdot 10^6$ non-zeros. On this problem, CPLEX 10.1 runs out of memory before finding the first integer solution, while *Amsaa* solves it in 0.2s, and solves 1,000-scenario problems within minutes. With 1,000 scenarios, the IP model would have about 10^8 binary variables ($(10^3)^2 \times 100$: there are about 100 time steps), which is outside the scope of today's IP solvers.

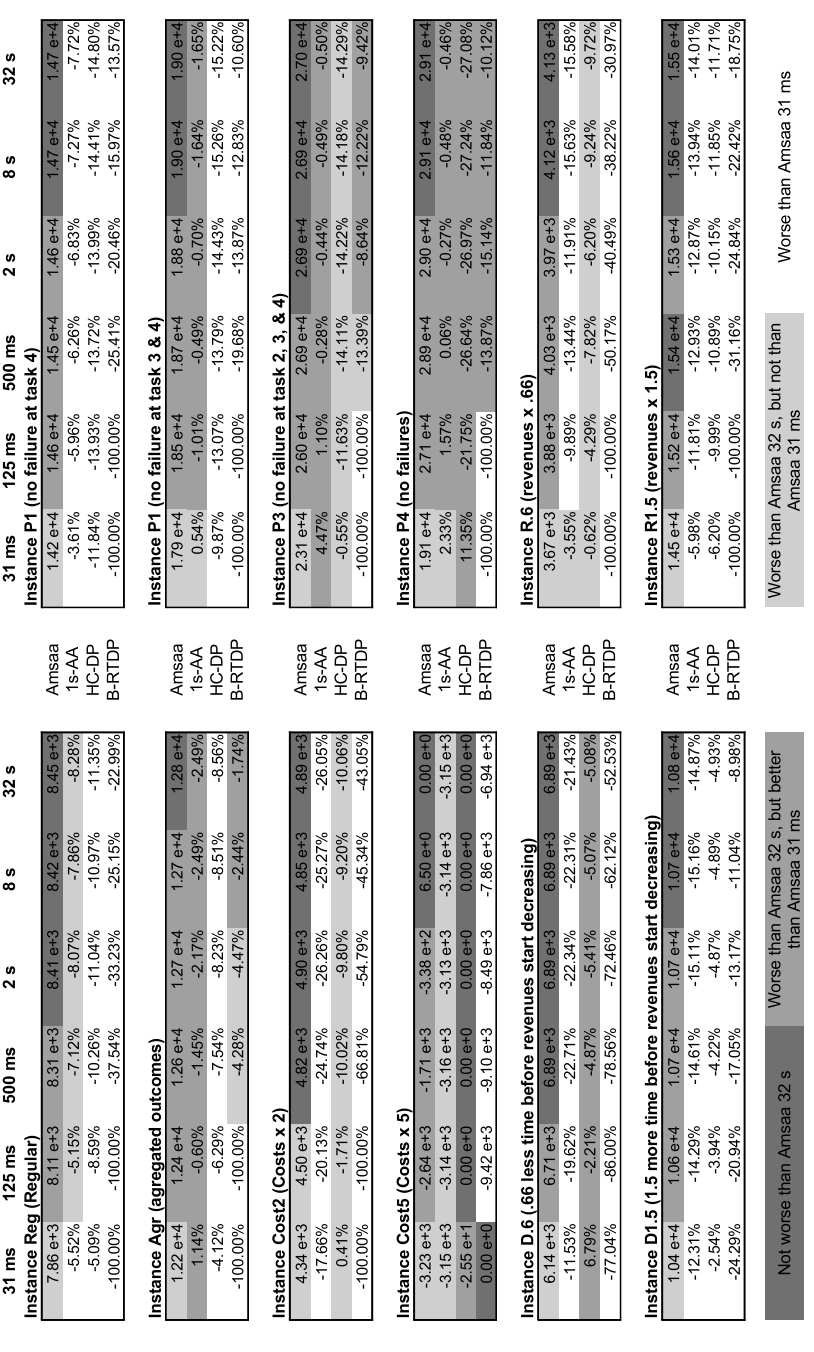
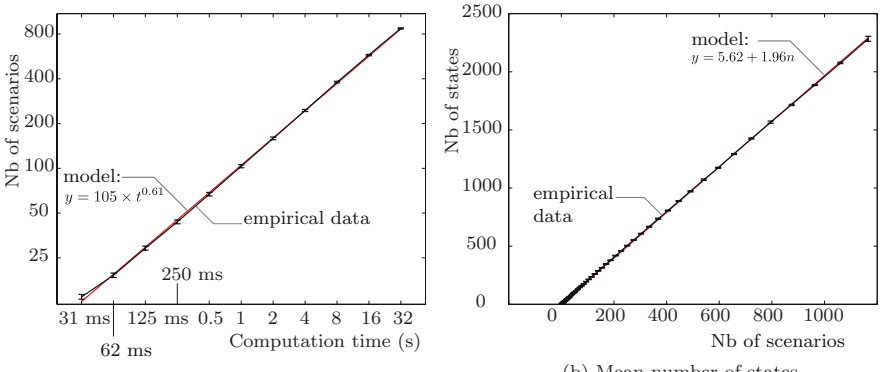


Fig. 2. Experimental Results for Anytime Decision Making on the S-RCPSP



(a) Average sample size

(b) Mean number of states reachable by the optimal policy

Fig. 3. Empirical complexity of *Amsaa*

[10] proposed to solve this IP using a branch-and-bound algorithm based on a Lagrangian relaxations of the non-anticipativity constraints. Yet, with 1,000 scenarios, their algorithm would relax 10^9 constraints (10 non-anticipatory constraints for each binary variable), so there would be a billion Lagrange multipliers to optimize at each node of the tree, which is not reasonable either.

Why is *Amsaa* so much more scalable on this problem? The main difference is the way non-anticipativity constraints are handled in the two approaches. In Grossman's approach, these are *relaxed by Lagragian duality* whereas, in *Amsaa*, they are enforced *lazily*. The lazy approach has two major advantages. First, the presence of Lagrangian multipliers alter the structure of the problem, precluding the use of a highly optimized ad-hoc solver like in *Amsaa*. Second, it makes *Amsaa* able to exploit the discrete nature of the decisions, using states and transitions instead of discretizing time.

9 Conclusion and Research Opportunities

We proposed *Amsaa*, the Anytime Multi-Step Anticipatory Algorithm, designed to address the limitations of the one-step anticipatory algorithm on very stochastic applications. *Amsaa* applies to online combinatorial stochastic optimization problems with exogenous uncertainty and exogenous or endogenous observations. Experimental results on stochastic resource-constraint project scheduling indicate that *Amsaa* significantly outperforms existing algorithms under a variety of time constraints and of instances.

The essence of *Amsaa* lies in the integration of three ideas from different fields: the SAA method from stochastic optimization to exploit positive correlations between decisions, search algorithms from AI to solve MDPs exactly without time discretization, and the use of black-box offline solvers from online stochastic combinatorial optimization to compute good upper bounds quickly.

There are many research avenues to improve *Amsaa*. They include the use of lower bounds like in B-RTDP (recall that we are maximizing) and of weaker but faster upper bounds. Other research questions concern the generation of the approximated problems. The stochastic programming literature include a few techniques to produce better

sample than by iid sampling [9]. It is not yet clear which of these techniques could be applied to Stoxuno problems.

Acknowlegments. Many thanks to Grégoire Dooms for his help. This research is partially supported by NSF awards DMI-0600384 and ONR award N000140610607.

References

1. Barto, A.G., Bradtke, S.J., Singh, S.P.: Learning to act using real-time dynamic programming. Artificial Intelligence 72(1), 81–138 (1995)
2. Bent, R., Van Hentenryck, P.: Waiting and Relocation Strategies in Online Stochastic Vehicle Routing. In: Proceedings of the 20th Int. Joint Conf. on A.I. (IJCAI 2007) (January 2007)
3. Bent, R., Van Hentenryck, P.: Scenario-Based Planning for Partially Dynamic Vehicle Routing Problems with Stochastic Customers. Operations Research 52(6) (2004)
4. Bonet, B., Geffner, H.: Faster heuristic search algorithms for planning with uncertainty and full feedback. In: IJCAI, pp. 1233–1238 (2003)
5. Bonet, B., Geffner, H.: Learning depth-first search: A unified approach to heuristic search in deterministic and non-deterministic settings, and its application to mdps. In: ICAPS (2006)
6. Choi, J., Realff, M.J., Lee, J.H.: Dynamic prog. in a heuristically confined state space: A stochastic resource-constrained project scheduling appplication. Computers and Chemical Engineering (2004)
7. Dempster, M.A.H.: Sequential Importance Sampling Algorithms for Dynamic Stochastic Programming. Journal of Mathematical Sciences 133, 1422–1444 (2006)
8. Dooms, G., Van Hentenryck, P.: Gap Reduction Techniques for Online Stochastic Project Scheduling. In: CPAIOR 2008 (2008)
9. Dupacova, J., Consigli, G., Wallace, S.W.: Scenarios for multistage stochastic programs. Annals of Operations Research (2000)
10. Goel, V., Grossmann, I.E.: A class of stochastic programs with decision dependent uncertainty. Math. Program 108(2-3), 355–394 (2006)
11. Hansen, E.A., Zilberstein, S.: LAO: A heuristic-search algorithm that finds solutions with loops. Artificial Intelligence 129(1-2), 35–62 (2001)
12. Kearns, M., Mansour, Y., Ng, A.: A Sparse Sampling Algorithm for Near-Optimal Planning in Large Markov Decision Processes. In: IJCAI 1999, pp. 1231–1324 (1999)
13. McMahan, H.B., Likhachev, M., Gordon, G.J.: Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In: ICML, pp. 569–576 (2005)
14. Mercier, L., Van Hentenryck, P.: Performance Analysis of Online Anticipatory Algorithms for Large Multistage Stochastic Programs. In: Proceedings of the 20th Int. Joint Conf. on AI (IJCAI) (2007)
15. Parkes, D., Duong, A.: An Ironing-Based Approach to Adaptive Online Mechanism Design in Single-Valued Domains. In: AAAI 2007, Vancouver, Canada, pp. 94–101 (2007)
16. Ruszczynski, A., Shapiro, A. (eds.): Stochastic Programming. Hanbooks in Operations Research and Management Series, vol. 10. Elsevier, Amsterdam (2003)
17. Thomas, M., Szczerbicka, H.: Evaluating Online Scheduling Techniques in Uncertain Environments. In: Proceedings of the 3rd Multidisciplinary International Scheduling Conference (MISTA 2007) (2007)
18. Van Hentenryck, P., Bent, R.: Online Stochastic Combinatorial Optimization. The MIT Press, Cambridge (2006)

Optimal Deployment of Eventually-Serializable Data Services

L. Michel[2], A. Shvartsman[2], E. Sonderegger[2], and P. Van Hentenryck[1]

[1] Brown University, Box 1910, Providence, RI 02912
[2] University of Connecticut, Storrs, CT 06269-2155

Abstract. Replication is a fundamental technique for increasing throughput and achieving fault tolerance in distributed data services. However, its implementation may induce significant communication costs to maintain consistency between the replicas. Eventually-Serializable Data Service (ESDS) has been proposed to reduce these costs and enable fast operations on data, while still providing guarantees that the replicated data will eventually be consistent. This paper reconsiders the deployment phase of ESDS, in which a particular implementation of communicating software components must be mapped onto a physical architecture. This deployment aims at minimizing the overall communication costs, while satisfying the constraints imposed by the protocol. Both MIP and CP models are presented and applied to realistic ESDS instances. The experimental results indicate that both models can find optimal solutions and prove optimality. The CP model, however, provides orders of magnitude improvements in efficiency. The limitations of the MIP model and the critical aspects of the CP model are discussed. Symmetry breaking and parallel computing are also shown to bring significant benefits.

1 Introduction

Data replication is a fundamental technique in distributed systems: it improves availability, increases throughput, and eliminates single points of failure. Data replication however induces a communication cost to maintain consistency among replicas. This cost can be reduced by the use of Eventually-Serializable Data Services (ESDS) [6], which allow the users to selectively relax the consistency requirements in exchange for improved performance. Given a definition of an arbitrary serial data type, ESDS guarantees that the replicated data will eventually be consistent (i.e., presenting a single-copy centralized view of the data to the users), although it may not be at a particular point during the execution.

The design, analysis, and implementation of ESDS is not an easy task however, and dedicated specification languages have been developed to express these algorithms and protocols formally. See, for instance, the framework of (timed) I/O automata [12,9] and their associated tools [13] which allow theorem provers (e.g., PVS [15]) and model checkers (e.g., UPPAAL [10,4]) to reason about their correctness. The ESDS algorithm is in fact formally specified with I/O automata

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 188–202, 2008.

and proved correct [6]. Once a specification is deemed correct, it must be implemented and deployed. The implementation typically consists of communicating software modules whose collective behaviors cannot deviate from the set of acceptable behaviors of the specification; see [5] for a methodic implementation of the algorithm and a study of its performance. The deployment then focuses on mapping the software modules on a distributed computer system to maximize performance or, more precisely, to minimize communication costs between the software components.

This research focuses on the last step of this process: the deployment of the implementation on a specific architecture. The deployment problem can be viewed as a resource allocation problem in which the objective is to minimize the network traffic while satisfying the constraints imposed by the distributed algorithms. These constraints include, in particular, the requirements that replicas cannot be allocated to the same computer since this would weaken fault tolerance. The ESDS Deployment Problem (ESDSDP) was considered by [3] and was modeled as a MIP. Unfortunately, the experimental results were not satisfactory at the time as even small instances could not be solved optimally [3].

This paper reconsiders the ESDSDP and studies both MIP and CP formulations. It demonstrates that MIP solvers can now solve practical instances in reasonable times, although the problems remain surprisingly challenging. It also presents a constraint-programming approach which dramatically improves the performance of the MIP model. The CP model is a natural encoding of the ESDSDP together with a simple search heuristic focusing on the objective function. The paper also evaluates empirically the strength of the filtering algorithms, the use of symmetry breaking, and the benefits of parallel computing.

Surveying the current results in more detail, the CP model brings orders of magnitude improvements in performance over the MIP model; for the examples considered here, it returns optimal solutions and proves optimality within a couple of minutes in the worst-case and within 15 seconds in general. The CP model enforces arc consistency on all different and multi-dimensional element constraints, which is critical for good performance. Symmetry breaking brings significant speedups (up to a factor 13), while parallel computing produces linear speedups on a 4 processor SMP machine.

The results in this paper also open new horizons for on-line optimization of distributed deployment. Given the observed improvements in obtaining optimal solutions, it becomes feasible next to consider optimizing deployment of components in reconfigurable consistent data services for dynamic systems, such as RAMBO [11,7]. Here configurations (quorum systems) of processors maintaining data replicas can be changed dynamically. Any server maintaining a replica can propose a new configuration and choosing suitable configurations is crucial for the performance of the service. There exists a trade-off between fast reconfiguration and the choice of suitable configurations. Enabling the servers to propose optimized configuration based on their local observations and decisions will substantially benefit such services. It is of note that realistic instance sizes (cf. [1,16]) are now within the current ability to compute optimal deployment.

The rest of this paper is organized as follows. Section 2 presents an overview of ESDS and illustrates the deployment problem on a basic instance. Section 3 introduces the high-level deployment model, while Sections 4 and 5 present the MIP and CP models. Section 6 reports the experimental results and analyzes the behavior of the models in detail. Section 7 concludes the paper.

2 Deployment of Eventually-Serializable Data Services

An Eventually-Serializable Data Service (ESDS) consists of three types of components: *clients*, *front-ends*, and *replicas*. Clients issue requests for operations on shared data and receive responses returning the results of those operations. Clients do not communicate directly with the replicas; instead they communicate with front ends which keep track of pending requests and handle the communication with the replicas. Each replica maintains a complete copy of the shared data and "gossips" with other replica to stay informed about operations that have been received and processed by them. Since multiple clients may issue requests concurrently, the responses are not uniquely defined. The service only guarantees that the responses are consistent with an eventual total order on the operations. Each replica maintains a set of the requested operations and a partial ordering on those operations consistent with the responses. Clients may specify constraints on how the requested operations are ordered. If no constraints are specified by the clients, the operations may be reordered after a response has been returned. A request may include a list of previously requested operations that must be performed before the currently requested operation. Lastly, a request also may be "strict", which means that the response must be consistent with the eventual total order. For any sequences of requests issued by the clients, the service guarantees eventual consistency of the replicated data [6].

ESDS is well-suited for implementing applications such as a distributed directory service, cf. Internet's Domain Name System [8], which needs redundancy for fault-tolerance and good response time for name lookup but does not require immediate consistency of naming updates. Indeed, the access patterns of such applications of ESDS are dominated by queries, with infrequent update requests.

Optimizing the deployment of an ESDS application can be challenging due to non-uniform communication costs induced by the actual network interconnect, as well as the various types of software components and their communication patterns. In addition, for fault tolerance, no more than one replica should reside on any given node. Finally, there is a tradeoff between the desire to place front-ends near the clients with whom they communicate the most and the desire to place the front-ends near replicas. Note also that the client locations may be further constrained by exogenous factors. Deployment instances typically involve a handful of front-ends to mitigate between clients and servers, a few replicas, and a few clients. Instances may not be particularly large as the (potentially numerous) actual users are *external* to the system and simply forward their demands to the *internal* clients modeled within the ESDS.

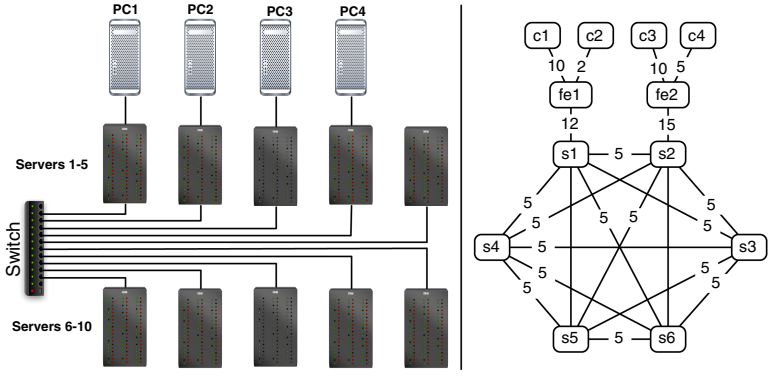


Fig. 1. A Simple ESDS Deployment Problem

Figure 1 depicts a simple ESDP Deployment Problem (ESDSDP). The left part of the figure shows the hardware architecture, which consists of 10 heavy-duty servers connected via a switch (full interconnect) and 4 "light" servers connected via direct links to the first four heavy-duty servers. For simplicity, the cost of sending a message from one machine to another is the number of network hops. For instance, a message from PC_1 to PC_2 requires 3 hops, since a server-to-server message through the switch requires one hop only. The right part of Figure 1 depicts the software implementation of the ESDS. The ESDS software modules fall in three categories: (1) client modules that issue queries $(c_1, \cdots, c_4)$; (2) front-end modules (fe_1, fe_2) that mediate between clients and servers and are responsible for tracking the sequence of pending queries; and (3) replicas $(s_1, \cdots, s_6)$. Each software module communicates with one or several modules and the right side of the figure specifies the volume of messages that must flow between the software components in order to implement the service. The problem constraints in this problem are as follows: the first 3 client modules must be hosted on the light servers $(PC_1, \cdots, PC_4)$ while the remaining components $(c_4, fe_1, fe_2, s_1, \cdots, s_6)$ must run on the heavy-duty servers. Additionally, the replicas s_1 through s_6 must execute on distinct servers to achieve the fault tolerance promised by the ESDS. The deployment problem consists of finding an assignment of software components to servers that satisfies the constraints stated above and minimizes the overall network traffic expressed as the volume of messages sent given the host assignments.

3 Modeling Optimal EDSD Deployments

We now present the deployment model originally developed in [2,3]. The input data consists of

- The set of software modules C;
- The set of hosts N;

- The subset of hosts to which a component can be assigned is denoted by booleans $s_{c,n}$ equal to *true* when component c can be assigned to host n;
- The network cost is directly derived from its topology and expressed with a matrix h where $h_{i,j}$ is the minimum number of hops required to send a message from host i to host j. Note that $h_{i,i} = 0$ (local messages are free);
- The message volumes. In the following, $f_{a,b}$ denotes the average frequency of messages sent from component a to component b;
- The separation set *Sep* which specifies that the components in each $S \in Sep$ must be hosted on a different servers;
- The co-location set *Col* which specifies that the components in each $S \in Col$ must be hosted on the same servers;

The decision variables x_c are associated with each module $c \in C$ and $x_c = n$ if component c is deployed on host n. An optimal deployment minimizes

$$\sum_{a \in C} \sum_{b \in C} f_{a,b} \cdot h_{x_a, x_b}$$

subject to the following. Components may only be assigned to supporting hosts

$$\forall c \in C \ : \ x_c \in \{i \in N \mid s_{c,i} = 1\}.$$

For each separation constraint $S \in Sep$, we impose

$$\forall i, j \in S \ : \ i \neq j \Rightarrow x_i \neq x_j.$$

Finally, for each co-location constraint $S \in Col$, we impose

$$\forall i, j \in S \ : \ x_i = x_j.$$

4 The MIP Model

We now present a MIP model for the deployment problem. It is interesting to observe that the ESDSDP is a generalization of the Quadratic Assignment Problem (QAP). Indeed, in a QAP, $C = N$, the variables x_i are required to form a permutation on N, and there is no separation and co-location constraint. A QAP is also obtained when the co-location constraints are absent and the model contains a single separation constraint over the set of components C.

Basic Model. The MIP model uses a four-dimensional matrix y of 0/1-variables such that $y_{a,i,b,j} = 1$ if $x_a = i \wedge x_b = j$. It also uses a two-dimensional matrix z of 0/1-variables satisfying $z_{a,i} = 1 \Leftrightarrow x_a = i$. The ESDSDP can then be specified as the minimization of

$$\sum_{a \in C} \sum_{i \in N} \sum_{b \in C} \sum_{j \in N} f_{a,b} \cdot h_{i,j} \cdot y_{a,i,b,j}$$

subject to

$$z_{a,i} \leq s_{a,i} \quad \forall a \in C, \forall i \in N \tag{1}$$

$$\sum_{i \in N} z_{a,i} = 1 \quad \forall a \in C \tag{2}$$

$$y_{a,i,b,j} \leq z_{a,i} \quad \forall a, b \in C, \forall i, j \in N \tag{3}$$

$$y_{a,i,b,j} \leq z_{b,j} \quad \forall a, b \in C, \forall i, j \in N \tag{4}$$

$$y_{a,i,b,j} \geq z_{a,i} + z_{b,j} - 1 \quad \forall a, b \in C, \forall i, j \in N \tag{5}$$

$$\sum_{a \in S} z_{a,i} \leq 1 \quad \forall S \in Sep\ , \forall i \in N \tag{6}$$

$$z_{a,i} = z_{b,i} \quad \forall i \in N\ , \forall S \in Col\ , \forall a, b \in S \tag{7}$$

Constraints (1) require the components to be hosted on supporting hosts and the constraints (2) that each component be deployed on exactly one host. The constraints (3,4,5) enforce the semantic definition of the z variables. The constraints (6) encode the separation constraints and (7) the co-location constraints.

Improving the formulation. In the above formulation, the conjunction $z_{a,i} = 1 \wedge z_{b,j} = 1$ is represented twice: once with $y_{a,i,b,j}$ and once with $y_{b,j,a,i}$. It is thus possible to use only half the variables in y. In addition, for all components $a \in C$ and nodes $i, j \in N$, $i \neq j \Rightarrow y_{a,i,a,j} = 0$, since component a cannot be deployed on nodes i and j simultaneously. Moreover, $h_{i,i} = 0$ and therefore all the terms on the diagonal can be removed from the objective function. The objective function thus only needs to feature variables $y_{a,i,b,j}$ such that $a \prec b$, where $\prec$ is a total ordering relation on C.

5 The CP Model

We now review a COMET program for the ESDSDP shown in Figure 2.

The Model. The model is depicted in lines 1–25 in Figure 2. The data declarations are specified in lines 2–8 and should be self-explanatory. The decision variables are declared in line 9 and are the same as in the model presented in Section 3: variable $x[c]$ specifies the host of component c and its domain is computed from the support matrix s.

The objective function is specified in line 10 and eliminates the diagonal elements (since $h_{i,i} = 0$ for every $i \in N$). The CP formulation features a two-dimensional *element* constraint since the matrix h is indexed by variables. Lines 12-15 state the co-location constraints: for each set S (line 12), an element $c_1 \in S$ is selected (randomly) and the model imposes the constraint $x_{c_1} = x_{c_2}$ for each other elements c_2 in S. Lines 16-17 state the separation constraints for every set in *Sep* using alldifferent constraints. The `onDomains` annotations indicate that arc-consistency must be enforced on the equations and alldifferent constraints.

Consider the pruning performed by the objective function when an upper bound is available. In COMET, a multi-dimensional *element* constraint is implemented in terms of a table T containing all the tuples $\langle a, b, h_{a,b} \rangle$ for $(a, b \in C)$.

```
1  Solver<CP> cp();
2  range C = ...; // The Components
3  range N = ...; // The Hosts
4  int[,] s = ...; // The supports matrix
5  int[,] f = ...; // The frequency matrix
6  int[,] h = ...; // The hops matrix
7  set{set{int}} Sep = ...; // The separation sets
8  set{set{int}} Col = ...; // The co−location sets
9  var<CP>{N} x[c in C](cp,setof(n in N) (s[c,n] == 1));
10 minimize<cp> sum(a in C,b in C: a != b) f[a,b] * h[x[a],x[b]]
11 subject to {
12    forall(S in Col)
13       select(c1 in S)
14          forall (c2 in S: c1 != c2)
15             cp.post(x[c1] == x[c2],onDomains);
16    forall(S in Sep)
17       cp.post(alldifferent(all(c in S) x[c]),onDomains);
18 } using {
19   while (!bound(x)) {
20      selectMax(i in C: !x[i].bound(), j in C)(f[i,j])
21         tryall<cp>(n in N) by (min(l in N: x[j].memberOf(l)) h[n,l])
22            cp.post(x[i] == n);
23         onFailure cp.post(x[i] != n);
24   }
25 }
```

Fig. 2. The Constraint-Programming Model in COMET

COMET creates a variable $\sigma_{a,b}$ for each term h_{x_a,x_b} in the objective and imposes

$$(x_a, x_b, \sigma_{a,b}) \in T$$

on which it achieves arc consistency. The objective then becomes

$$\sum_{a \in C} \sum_{b \in C} f_{a,b} \cdot \sigma_{a,b}.$$

The Search Procedure. The search procedure is depicted in lines 19–24. It is a variable labeling with dynamic variable and value orderings. Lines 20–23 are iterated until all variables are bound (line 19) and each iteration nondeterministically assigns a variable `x[i]` to a host n (lines 21–23). The variable and value orderings are motivated by the structure of the objective function

$$\sum_{i \in C} \sum_{j \in C} f_{i,j} \cdot h_{x_i,x_j}.$$

In the objective, the largest contributions are induced by assignments of components i and j that are communicating heavily and are placed on distant hosts. As a result, the variable and value ordering are based on two ideas:

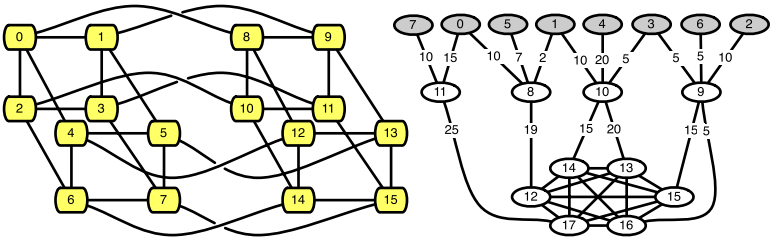


Fig. 3. Instance `HYPER16`: Deploying 18 Components on a 16-Node Hypercube

1. Assign first a component i whose communication frequency $f[i, j]$ with a component j is maximal (line 20);
2. Try the hosts for component i in increasing number of hops required to communicate with component j (line 21).

The variable selection thus selects first components with the heaviest (single) communications, while the value selection tries to minimize the number of hops.

6 Experimental Results

The experimental results on the ESDSDP are reported for both the MIP and CP model. We first describe the benchmarks and then present the results.

6.1 The Benchmarks

The models are evaluated on a collection of synthetic benchmarks that are representative of realistic proprietary instances [1]. The benchmarks cover instances with different configurations of software components and different hardware architectures. All instances, which are available upon request, have from 12 to 18 software modules, and the hardware platforms range from 14 to 16 machines with 2 to 4 front-ends. In particular, Figure 3 depicts instance `HYPER16` which deploys 18 components on an hypercube, while Figure 4 depicts instance `SCSS2SNUFE`.

Table 1 gives a more detailed description of the instances. For each benchmark, it gives the number of hosts and components, the separation and co-location constraints, the size of the search space and the hardware and software configurations. The specification `3:6S:3FE:4C` indicates that there are three separation sets, one for the six replicas, one for the 3 front-ends, and one for the 4 clients. The hardware setups named H_1 through H_5 are specified as follows.

H_1 is the hardware platform depicted in Figure 1.
H_2 is a simple extension of H_1 with a fifth client connected to the fifth server.
H_3 is a network with 10 servers on two subnets and 1 server (machine 8) acting as a gateway between the subnets. Each subnet is arbitrated by a switch. The four clients are connected to the first 4 servers on the first subnet.
H_4 is based on an 8-node hypercube with 7 extra machines connected by a directed link to a single vertex of the hypercube (so the degrees of 7 vertices of the hypercube are 4 and the last vertex has a degree of 3).
H_5 is the 16-node hypercube depicted in Figure 3.

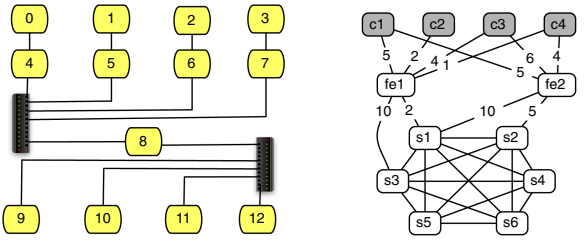


Fig. 4. Instance `SCSS2SNUFE`: A Deployment on Two Subnets

The software configurations named S_1 through S_3 are specified as follows.

S_1 is the architecture depicted in Figure 1.
S_2 is identical to S_1. However, clients can communicate with more than one front-end and front-ends communicate with several replicas as well.
S_3 is a scaled-up version of S_2 and it is shown in Figure 3.

Table 1. High-Level Descriptions of the Benchmarks

Bench	$\#N$	$\#C$	S	T	$SearchSpace$	Hardware	Software
SIMPLE2	14	12	1:6S	0	$4^3 \cdot 10^7$	H_1	S_1
SIMPLE1	14	12	1:6S	0	$4^3 \cdot 10^8$	H_1	S_1
SIMPLE0	14	12	1:6S	0	$4^3 \cdot 10^9$	H_1	S_1
fe3c5pc	14	14	2:6S:3FE	0	$4^4 \cdot 10^{10}$	H_1	S_1
fe3c5pc5	15	14	2:6S:3FE	0	$4^4 \cdot 10^{10}$	H_2	S_1
fe3c5sun	14	14	2:6S:3FE	0	$4^3 \cdot 10^{11}$	H_1	S_1
fe3c6pc5	15	15	2:6S:3FE	0	$5^5 \cdot 10^{10}$	H_2	S_1
fe3c7pc5	15	16	2:6S:3FE	0	$5^5 \cdot 10^{11}$	H_2	S_1
fe3c7pc5CS	15	16	3:6S:3FE:4C	0	$5^5 \cdot 10^{11}$	H_2	S_1
fe3c7pc5CST	15	16	3:6S:3FE:4C	1:2C	$5^5 \cdot 10^{11}$	H_2	S_1
fe3dist	14	13	2:6S:3FE	0	$4^3 \cdot 10^{10}$	H_1	S_1
SCSS1SNUFE	14	12	2:6S:4FE	0	$4^3 \cdot 12^3 \cdot 10^6$	H_1	S_2
SCSS2SNUFE	14	12	2:6S:4FE	0	$4^3 \cdot 12^3 \cdot 10^6$	H_3	S_2
SCSS2SNCFE	14	12	2:6S:4FE	0	$4^3 \cdot 10^9$	H_3	S_2
HYPER8	15	18	2:6S:4FE	0	$5^6 \cdot 10^{12}$	H_4	S_3
HYPER16	16	18	2:6S:4FE	0	$5^6 \cdot 10^{12}$	H_5	S_3

6.2 The MIP Model

We ran the MIP model using CPLEX version 11 on an AMD Athlon64 at 2Ghz with 2 gigabytes of RAM. Various attempts were made to find the best settings for CPLEX. Changing the variable branching heuristic to use the coefficients of the objective function (`set mip ordertype 1` in the interactive solver) seems to deliver the best performance. CPLEX then finds the optimal solution early on, in general, contrary to the default settings. All times are reported in seconds.

Table 2. Experimental Results for the MIP Model

Bench	T_{end} $v10.1$	T_{end} $v11.0$	T_{opt}(s)	$\#Nodes$	MTS	$\#R$	$\#C$	Gap
SIMPLE2	6	4	2	68		8834	3070	61.34%
SIMPLE1	30	1420	247	4238	2.58	11295	3900	25.77%
SIMPLE0	4466	18	12	502		14056	4830	38.49%
fe3c5pc	30576	5091	1717	22741	25.04	18442	6312	39.79%
fe3c5pc5	6813	5407	960	26360	30.59	19804	6770	37.85%
fe3c5sun	17904	7877	210	29216	33.94	20458	6990	88.50%
fe3c6pc5	85832	10556	507	35901	41.54	21605	7375	78.21%
fe3c7pc5	186731	14712	312	26818	34.38	25356	8635	70.50%
fe3c7pc5CS	43788	44677	387	81673	116.00	25356	8635	67.82%
fe3c7pc5CST	51953	6154	480	51223	66.44	25290	8610	64.50%
fe3dist	4584	4105	3100	33889	34.62	17097	5860	15.44%
SCSS1SNUFE	41865	844	135	16350	14.73	17492	5994	70.08%
SCSS2SNUFE	7750	1624	90	32846	29.82	17492	5994	83.33%
SCSS2SNCFE	2583	3026	1551	72711	55.20	14048	4830	26.12%
HYPER8	79150	53906	51381	124918	190.95	31583	10725	5.82%

Table 2 reports the results for the MIP model. Column T_{end} reports the running times (in seconds) to find the optimal solution and prove optimality. Column T_{opt} reports the times to find the optimal solutions. Column $\#Nodes$ reports the number of nodes in the branch & bound tree, column MTS gives the peak size of the branch & bound tree (in megabytes), columns $\#R$ and $\#C$ give the number of rows and columns after presolve, and column Gap returns the optimality gap as a percentage of the optimal solution when it is is found. All the values are for $v11$ (the very first column reports the runtime of $v10.1$).

Overall, the results indicate that optimal solutions can be found in reasonable time (except when the network topology is an hypercube) and that optimality proofs require significant computational resources. More precisely, the computational results can be summarized as follows.

1. CPLEX *v11.0* delivers solutions faster than *v10.1* and the improvement is noticeable (a few instances are worse though). The changes are dependent on the instances and sometime lead to a 200-fold reduction in the number of explored nodes. For most instances this reduction translates into improvements for the runtime by up to one order of magnitude (some instances do not benefit at all or get a little worse).
2. On the `FExCySz` instances, the solver (*v10.1*) finds an optimum relatively quickly (from 20 to 705 seconds) but the proofs of optimality are very costly.
3. On the `SCSSx` instances, the MIP solver also finds optimal solutions reasonably quickly. The proof of optimality takes significant time when only one subnet is used. It becomes significantly faster (by about a factor 6) when two subnets are used, probably because the linear-programming relaxation recognizes the higher communication costs for hosts on different subnets.
4. On the `HYPERx` instances, the MIP solver has great difficulties and only solves the smallest instance. It took slightly more than $7\frac{1}{2}$ hours to obtain the optimal solution on `HYPER8` and almost 22 hours to prove its optimality.

Table 3. Experimental Results for the CP Model

Bench	T_{end}	$\#CHPT$	T_{opt}
SIMPLE2	0.23	2510	.19
SIMPLE1	1.38	15408	.19
SIMPLE0	7.75	87491	.19
fe3c5pc	2.76	14597	.19
fe3c5pc5	4.64	24130	.22
fe3c5sun	6.29	30621	.20
fe3c6pc5	3.54	18547	.20
fe3c7pc5	7.83	35726	.20

Bench	T_{end}	$\#CHPT$	T_{opt}
fe3c7pc5CS	7.77	35312	.20
fe3c7pc5CST	13.68	70495	.20
fe3dist	4.16	29750	.19
SCSS1SNUFE	43.34	392628	.19
SCSS2SNUFE	66.43	380117	49.82
SCSS2SNCFE	50.83	322472	36.04
HYPER8	65.07	123213	8.06
HYPER16	309.46	513051	254.4

5. The MIP solver explores a large number of nodes on these instances, takes significant memory, and exhibits large optimality gaps. The linear relaxation is not particularly strong, which explains the large search tree.

Although they highlight the computational progress in MIP solvers, these results are quite sobering. Indeed, the MIP solver took 4,466 seconds for proving the optimality of the instance discussed in Section 2 after obtaining the optimal solution in about 20 seconds. This instance does not seem particularly difficult, since its software communication patterns are rather simple and well-structured.

6.3 The CP Model

Table 3 reports the results for the CP model with COMET 0.07 (executing on an Intel Core at 2Ghz with 2 Gigabytes of RAM). Column T_{end} gives the time in seconds to find the optimum and prove optimality, column $\#Chpt$ reports the number of choice points and column T_{opt} reports the time in seconds to find the optimum. Several observations should be conveyed about these results.

1. The CP model dramatically outperforms the MIP model with speedup factors ranging from 121 on `SIMPLE2` to 1,209 on `HYPER8`. The number of search nodes is orders of magnitude smaller for CP, showing the significant pruning obtained by the CP solver compared to the MIP solver.
2. On the `FExCySz` instances, the CP solver finds the optimal solutions and proves optimality in less than 14 seconds.
3. On the `SCSSx` instances, the CP solver also finds the optimal solutions and proves optimality in less than 67 seconds. Interestingly, having several subnets does not seem to help the CP solver as far as computation times are concerned, although the number of choice points decreases.
4. On the `HYPERx` instances, the CP solver finds the optimum and proves optimality in less than 4 minutes. These instances are more challenging (as they were for the MIP solver) but the computation times remain reasonable.
5. The instances `HYPERx` and `SCSSx` are hard on two counts: to find the optimum and to prove optimality. `SCSS1x` is the only exception which indicates that the network topology significantly impacts the search.

Table 4. The Value of Arc Consistency for the CP Model

Algo	CP-BC		CP-AC	
Bench	T_{end}	$\#Chpt$	T_{end}	$\#Chpt$
SIMPLE2	1.2	7582	0.2	2510
SIMPLE1	6.1	46874	1.4	15408
SIMPLE0	37.2	307365	7.7	87491
fe3c5pc	94.8	748118	2.8	14597
fe3c5pc5	639.9	4705378	4.6	24130
fe3c5sun	166.3	1336353	6.2	30621
fe3c6pc5	1039.2	7238665	3.5	18547
fe3c7pc5	2107.1	14M	7.8	35726

Algo	CP-BC		CP-AC	
Bench	T_{end}	$\#Chpt$	T_{end}	$\#Chpt$
fe3c7pc5CS	1916.5	12M	7.8	35312
fe3c7pc5CST	1286.4	8.5M	13.7	70495
fe3dist	93.6	839781	4.2	29750
SCSS1SNUFE	62.8	482601	43.3	392628
SCSS2SNUFE	60.4	442373	66.4	380117
SCSS2SNCFE	30.4	246228	50.8	322472
HYPER8	7653.0	34M	65.0	123213
HYPER16	34570.9	157M	237.5	513051

Note that the simple instance discussed in Section 2 now requires less than 8 seconds for finding the optimal and proving its optimality (instead of 4,466 seconds for the MIP solver). This remains sobering given the simplicity of that particular instance but it is perhaps reassuring that more complex `FExCySz` instances (from a visual standpoint) require roughly the same time.

The Value of Arc-Consistency. The CP model presented in this paper is quite elegant since it enforces arc consistency on all constraints and the objective function.[1] One may wonder whether arc consistency is critical in ESDSDPs or whether a weaker form of consistency is sufficient. Table 4 depicts the results when only bound reasoning is performed on the objective. The second and the third column report the results of the CP solver when bound consistency is enforced on the objective, while the fourth and the fifth columns reproduce those in Table 3. The experimental results show a dramatic loss in performance when arc consistency is not used. On some benchmarks, the CP model with bound consistency on the objective becomes about 300 times slower than the model with arc consistency on the objective. The all-different constraints are less critical and reverting to a weaker consistency there does not induce significant losses.

Exploiting Value Symmetries. ESDSDPs may feature a variety of symmetries, which could be removed to improve performance. Consider the instance presented in Section 2 and the 10 heavy-duty servers in particular. Four of these servers are connected through dedicated links to the "light" servers that must host some of the clients. This creates two *classes* of heavy-duty servers: those connected to the light servers and those that are not. Moreover, the heavy-duty servers $6-10$ are fully interchangeable in that any permutation of these servers in a solution would also produce a solution. Techniques for removing these symmetries during search are well-known (see, for instance, [17]).

Figure 5 illustrates how to enhance the search procedure presented earlier with symmetry breaking for all instances except the hypercube networks (the inter-node distances –hops– reduce the potential symmetries and are more difficult

[1] As traditional, the solver dynamically adds new constraints whenever a new solution is produced, forcing the objective function to improve upon the best known solution.

```
1 while (!bound(x)) {
2   selectMax(i in C: !x[i].bound(), j in C)(f[i,j]) {
3     int L = max(firstSym,1 + max(s in C: x[s].bound()) x[s].getMin());
4     tryall<cp>(n in N: n <= L) by (min(l in N: x[j].memberOf(l)) h[n,l])
5       cp.post(x[i] == n);
6     onFailure cp.post(x[i] != n);
7   }
8 }
```

Fig. 5. The Search Procedure with Value Symmetry Breaking

Table 5. The Impact of Value Symmetry Breaking

Algo	CP+SYM		CP	
Bench	T_{end}	$\#C$	T_{end}	$\#C$
fe3c7pc5	2.21	8628	7.83	35726
fe3c7pc5CS	2.07	7949	7.77	35312
fe3c7pc5CST	3.03	13654	13.68	70495
fe3dist	0.31	1708	4.16	29750

Algo	CP+SYM		CP	
Bench	T_{end}	$\#C$	T_{end}	$\#C$
SCSS1SNUFE	3.24	20805	43.34	392628
SCSS2SNUFE	14.74	71692	66.43	380117
SCSS2SNCFE	8.48	48740	50.83	322472

to exploit). For simplicity, we take the convention that the servers are divided into two classes: 1..*firstSym-1* (non-interchangeable servers) and *firstSym*..$|C|$ (interchangeable servers) (It is easy to generalize these conventions by manipulating sets when there are more classes of interchangeable servers.) In the simple instance, *firstSym* = 6. Line 3 determines the possible hosts for component i: these are the already used hosts (the max expression) and at most one new server in *firstSym*..$|C|$. The outer `max` ensures that all the servers up to *firstSym* are considered, since these are not interchangeable.

Table 5 depicts the results on the larger instances in which there are interchangeable heavy servers. The second and the third column report the results of the CP solver with symmetry breaking while the fourth and the fifth columns reproduce the standard results from Table 3. The experimental results show nice improvements, including speedups close to 13.5 on some instances. The ability to break symmetries during search is important here to avoid interfering with the search heuristics. All these instances (which do not include the hypercube instances) are now solved optimally in less than 15 seconds.

Note that there are potentially many other forms of symmetries in ESDSDPs which have not been exploited. For instance, in the simple ESDSDP presented earlier, the software components $s_3, \ldots, s_6$ are symmetric, revealing some variable symmetries. Similarly, the hardware pairs (PC-i,Heavy-Servers-i) are symmetric. We did not make any effort to remove these symmetries since they are instance-dependent and were not necessary to solve the CP model effectively.

Parallel Computing. Since COMET supports transparent parallelism [14], we evaluated the performance of the CP model with a 4 processors SMP machine on the hypercube instances. The results in Table 6 exhibit linear speedups.

Table 6. The Impact of Parallelism (each value is an average over 5 runs)

Bench	1	2	3	4
HYPER8	50.54	24.46	16.58	12.79

Bench	1	2	3	4
HYPER16	198.43	100.71	65.53	48.28

7 Conclusion

In distributed systems, Eventually-Serializable Data Services (ESDS) support data replication and reduce the communication costs to maintain consistency between the replicas by allowing the users of the service to take advantage of the semantics of the sequential data types to relax consistency for certain operations while ensuring eventual consistency. Once ESDS protocols and algorithms have been designed and formally verified, they are deployed on specific architectures. More precisely, in ESDS deployments, a collection of communicating software components must be mapped onto a physical architecture to minimize the communication costs while preserving the safety of the ESDS service. The ESDS deployment problem was considered in the late 1990s, but practical instances could not be solved optimally at the time by state-of-the-art MIP solvers.

This paper reconsidered the deployment problem in ESDS, presented the traditional MIP model, and proposed a CP model. The models were evaluated experimentally on synthetic instances capturing the sizes and properties of actual ESDS. The experimental results indicate that MIP solvers are now capable of solving ESDS deployment problems, although these applications remain extremely challenging. In particular, the optimality gap is substantial for the MIP model, which gives rise to very large search trees. The CP model brings orders of magnitude improvements in performance over the MIP model; it returns optimal solutions and proves optimality within a couple of minutes in the worst-case and within 15 seconds in general. The CP model enforces arc consistency on all different and multi-dimensional element constraints, which is critical for good performance. The benefits of symmetry breaking and parallelism were also studied. Symmetry breaking brings significant speedups (up to a factor 13), while parallel computing produces linear speedups.

Future work will exploit the structure of ESDS deployments more finely, as these problems often exhibit some regularities either in the underlying hardware or in the software components. Given the improvements in obtaining optimal solutions, it makes sense to explore a portfolio of algorithms that includes a CP model for the on-line deployment optimization of dynamic reconfigurable distributed data services such as RAMBO.

Acknowledgements

This work was partially supported through the following NSF awards: DMI-0600384, IIS-0642906 and CCF-0702670 as well as an ONR award N000140610607.

References

1. Aguilera, M.: Hewlett-Packard, Personal communication (2007)
2. Bastarrica, M., Demurjian, S., Shvartsman, A.: Software architectural specification for optimal object distribution. In: SCCC 1998: Proc. of the XVIII Intl. Conf. of the Chilean Computer Science Society, Washington, DC, USA (1998)
3. Bastarrica, M.C.: Architectural specification and optimal deployment of distributed systems. PhD thesis, University of Connecticut (2000)
4. Behrmann, G., David, A., Larsen, K., Möller, O., Pettersson, P., Yi, W.: Uppaal - present and future. In: Proceedings of the 40th IEEE Conference on Decision and Control (CDC 2001), pp. 2881–2886 (2001)
5. Cheiner, O., Shvartsman, A.: Implementing an eventually-serializable data service as a distributed system building block. Networks in Distributed Computing 45, 43–71 (1999)
6. Fekete, A., Gupta, D., Luchangco, V., Lynch, N., Shvartsman, A.: Eventually-serializable data services. In: PODC 1996: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, pp. 300–309 (1996)
7. Gilbert, S., Lynch, N.A., Shvartsman, A.A.: RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In: DSN, pp. 259–268. IEEE Computer Society Press, Los Alamitos (2003)
8. IETF. Domain name system, rfc 1034 and rfc 1035 (1990)
9. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: The Theory of Timed I/O Automata, Synthesis Lectures in Computer Science. Morgan & Claypool Publishers, San Francisco (2006)
10. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
11. Lynch, N., Shvartsman, A.: RAMBO: A reconfigurable atomic memory service for dynamic networks. In: Proceedings of the 16th International Symposium on Distributed Computing, pp. 173–190 (2002)
12. Lynch, N., Tuttle, M.: An introduction to Input/Output Automata. CWI-Quarterly 2(3), 219–246 (1989)
13. Lynch, N.A., Garland, S., Kaynar, D., Michel, L., Shvartsman, A.: The Tempo Language User Guide and Reference Manual. VeroModo Inc. (December 2007), http://www.veromodo.com
14. Michel, L., See, A., Van Hentenryck, P.: Parallelizing constraint programs transparently. In: Proceedings of the 13th International Conference on the Principles and Practice of Constraint Programming (CP 2007), Providence, RI (2007)
15. Owre, S., Rajan, S., Rushby, J.M., Shankar, N., Srivas, M.K.: PVS: Combining specification, proof checking, and model checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 411–414. Springer, Heidelberg (1996)
16. Saito, Y., Frølund, S., Veitch, A.C., Merchant, A., Spence, S.: Fab: building distributed enterprise disk arrays from commodity components. In: Mukherjee, S., McKinley, K.S. (eds.) ASPLOS, pp. 48–58. ACM Press, New York (2004)
17. Van Hentenryck, P., Flener, P., Pearson, J., Ågren, M.: Tractable symmetry breaking for CSPs with interchangeable values. In: International Joint Conference on Artificial Intelligence (IJCAI 2003) (2003)

Counting Solutions of Knapsack Constraints

Gilles Pesant and Claude-Guy Quimper

Département de génie informatique et génie logiciel
École Polytechnique de Montréal
C.P. 6079, succ. Centre-ville
Montreal, Canada H3C 3A7
pesant@crt.umontreal.ca, claude-guy.quimper@polymtl.ca

Abstract. This paper furthers the recent investigation of search heuristics based on solution counting information, by proposing and evaluating algorithms to compute solution densities of variable-value pairs in knapsack constraints. Given a domain consistent constraint, our first algorithm is inspired from what was proposed for regular language membership constraints. Given a bounds consistent constraint, our second algorithm draws from discrete uniform distributions. Experiments on several problems reveal that simple search heuristics built from the information obtained by these algorithms can be very effective.

1 Introduction

Recent work on search heuristics using information about the number of solutions of constraints has shown encouraging results to solve constraint satisfaction problems [4,8]. Working at the level of individual constraints, it asks not only whether there exists a solution in which variable x takes value d, which corresponds to the familiar concept of consistency, but also *how many* of the solutions feature that particular assignment. Such an approach requires efficient ways to answer that question for each type of constraint commonly found in constraint programs. This paper examines knapsack constraints, present in many problems.

The $\texttt{knapsack}(\mathbf{x}, \mathbf{c}, \ell, u)$ constraint holds if

$$\ell \leq \mathbf{cx} \leq u$$

where $\mathbf{c} = (c_1, c_2, \ldots, c_n)$ is an integer row vector, $\mathbf{x}$ is a column vector of finite domain variables $(x_1, x_2, \ldots, x_n)^T$ with $x_i \in D_i$, and ℓ and u are integers. To be interpreted as a knapsack, the integer values involved (including those in the finite domains) are non negative. We will come back to this restriction in Section 5. We assume that l and u are finite as they can always be set to the smallest and largest value that $\mathbf{cx}$ can take.

To prepare us to manipulate information on the number of solutions of knapsack constraints, we recall some definitions and notation from [4,8].

Definition 1 (solution count). *Given a constraint $\gamma(x_1, \ldots, x_k)$ and respective finite domains D_i $1 \leq i \leq k$, let $\#\gamma(x_1, \ldots, x_k)$ denote the number of solutions of constraint γ, called its* solution count.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 203–217, 2008.

Definition 2 (solution density). *Given a constraint $\gamma(x_1, \ldots, x_k)$, respective finite domains D_i $1 \leq i \leq k$, a variable x_i in the scope of γ, and a value $d \in D_i$, we will call*

$$\sigma(x_i, d, \gamma) = \frac{\#\gamma(x_1, \ldots, x_{i-1}, d, x_{i+1}, \ldots, x_k)}{\#\gamma(x_1, \ldots, x_k)}$$

the solution density *of pair (x_i, d) in γ. It measures how often a certain assignment is part of a solution.*

In the rest of the paper, Section 2 presents the counting algorithm for knapsack constraints on which domain consistency is enforced, Section 3 presents another counting algorithm for bounds consistent knapsack constraints, and Section 4 reports several experiments. Final comments are given in Section 5.

2 Counting with Domain Consistent Knapsacks

In [6], Trick proposes a filtering algorithm for knapsack constraints that relies on a graph whose structure is very similar to that of the `regular` constraint [3]. He notes that every path in that graph corresponds to a solution and that counting the number of solutions is easily obtained through a recursion without the need to enumerate these solutions. Not surprisingly then, the computation of solution counts and solution densities for knapsack constraints follows quite directly from the work on the `regular` constraint.

We start from the reduced graph described in [6], which is a layered directed graph $G(V, A)$ with a vertex $v_{i,b} \in V$ for $1 \leq i \leq n$ and $0 \leq b \leq u$ whenever

$$\forall\, j \in [1, i],\ \exists\, d_j \in D_j \text{ such that } \sum_{j=1}^{i} c_j d_j = b$$

and

$$\forall\, j \in (i, n],\ \exists\, d_j \in D_j \text{ such that } \ell - b \leq \sum_{j=i+1}^{n} c_j d_j \leq u - b,$$

and an arc $(v_{i,b}, v_{i+1,b'}) \in A$ whenever

$$\exists\, d \in D_{i+1} \text{ such that } c_{i+1} d = b' - b.$$

We define the following two recursions to represent the number of incoming and outgoing paths at each node.

For every vertex $v_{i,b} \in V$, let $\#ip(i, b)$ denote the number of paths from vertex $v_{0,0}$ to $v_{i,b}$:

$$\begin{aligned} \#ip(0, 0) &= 1 \\ \#ip(i+1, b') &= \sum_{(v_{i,b}, v_{i+1,b'}) \in A} \#ip(i, b), \quad 0 \leq i < n \end{aligned}$$

Let $\#op(i,b)$ denote the number of paths from vertex $v_{i,b}$ to a vertex $v_{n,b'}$ with $\ell \leq b' \leq u$.

$$\begin{aligned} \#op(n,b) &= 1 \\ \#op(i,b) &= \sum_{(v_{i,b},v_{i+1,b'})\in A} \#op(i+1,b'), \quad 0 \leq i < n \end{aligned}$$

The total number of paths (i.e. the *solution count*) is given by

$$\#\texttt{knapsack}(\mathbf{x},\mathbf{c},\ell,u) = \#op(0,0)$$

in time linear in the size of the graph even though there may be exponentially many of them. The *solution density* of a variable-value pair (x_i,d) is given by

$$\sigma(x_i,d,\texttt{knapsack}) = \frac{\sum_{(v_{i-1,b},v_{i,b+c_i d})\in A} \#ip(i-1,b)\cdot\#op(i,b+c_i d)}{\#op(0,0)}.$$

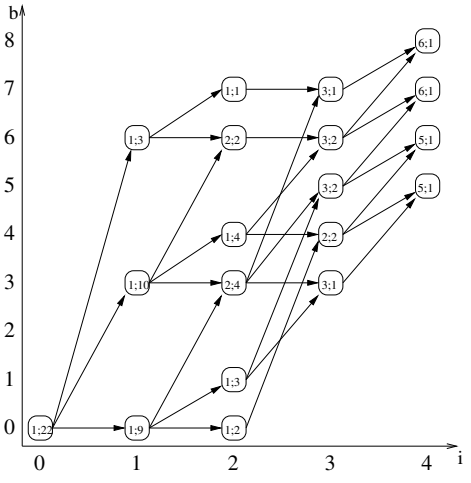


Fig. 1. Reduced graph for knapsack constraint $5 \leq 3x_1 + x_2 + 2x_3 + x_4 \leq 8$ with $D_1 = \{0,1,2\}, D_2 = \{0,1,3\}, D_3 = \{0,1,2\}, D_4 = \{1,2\}$. Vertex labels represent the number of incoming and outgoing paths.

In Figure 1, the left and right labels inside each vertex give the number of incoming and outgoing paths for that vertex, respectively. Table 1 reports the solution densities for every variable-value pair.

The time required to compute these recursions is related to the number of arcs, which is in $\mathcal{O}(nu \max_{1\leq i\leq n}\{|D_i|\})$. Then each solution density computes a summation over a subset of the arcs but each arc of the graph is involved in at most one such summation, so the overall time complexity of computing every solution density is $\mathcal{O}(nu \max_{1\leq i\leq n}\{|D_i|\})$ as well.

Table 1. Solution densities for the example of Fig. 1

	variable			
value	x_1	x_2	x_3	x_4
0	9/22	8/22	9/22	–
1	10/22	8/22	7/22	11/22
2	3/22	–	6/22	11/22
3	–	6/22	–	–

3 Counting with Bounds Consistent Knapsacks

Knapsack constraints, indeed most arithmetic constraints, have traditionally been handled by enforcing bounds consistency, a much cheaper form of inference. In some situations, we may not afford to enforce domain consistency in order to get the solution counting information we need to guide our search heuristic. Can we still retrieve such information, perhaps not as accurately, from the weaker bounds consistency?

Consider the variable x with domain $D = [a, b]$. Each value in D is equiprobable. We associate to x the discrete random variable X which follows a discrete uniform distribution with probability mass function $f(v)$, mean $\mu = E[X]$, and variance $\sigma^2 = Var[X]$.

$$f(v) = \begin{cases} \frac{1}{b-a+1} & \text{if } a \leq v \leq b \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

$$\mu = \frac{a+b}{2} \tag{2}$$

$$\sigma^2 = \frac{(b-a+1)^2 - 1}{12} \tag{3}$$

To find the distribution of a variable subject to a knapsack constraint, one needs to find the distribution of a linear combination of uniformly distributed random variables. Lyapunov's central limit theorem allows us to approximate the distribution of such a linear combination.

Theorem 1 (Lyapunov's central limit theorem). *Consider the independent random variables $X_1, \ldots, X_n$. Let μ_i be the mean of X_i, σ_i^2 be its variance, and $r_i^3 = E[|X_i - \mu_i|^3]$ be its third central moment. If*

$$\lim_{n\to\infty} \frac{(\sum_{i=1}^n r_i^3)^{\frac{1}{3}}}{(\sum_{i=1}^n \sigma_i^2)^{\frac{1}{2}}} = 0,$$

then the random variable $S = \sum_{i=1}^n X_i$ follows a normal distribution with mean $\mu_S = \sum_{i=1}^n \mu_i$ and variance $\sigma_S^2 = \sum_{i=1}^n \sigma_i^2$.

The probability mass function of the normal distribution with mean μ and variance σ^2 is the Gaussian function:

$$\varphi(x) = \frac{e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{\sigma\sqrt{2\pi}} \tag{4}$$

Note that Lyapunov's central limit theorem does not assume that the variables are taken from identical distributions. This is necessary since variables with different domains have different distributions.

Lemma 1 defines an upper bound on the third central moment of the expression kX where k is a positive coefficient and X is a uniformly distributed random variable.

Lemma 1. *Let Y be a discrete random variable equal to kX such that k is a positive coefficient and X is a discrete random variable uniformly distributed over the interval $[a, b]$. The third central moment $r^3 = E[|Y - E[Y]|^3]$ is no greater than $k^3(b-a)^3$.*

Proof. The case where $a = b$ is trivial. We prove for $b-a > 0$. The proof involves simple algebraic manipulations from the definition of the expectation.

$$r^3 = \sum_{i=ka}^{kb} |i - E[Y]|^3 f(i) \tag{5}$$

$$= \sum_{j=a}^{b} |kj - kE[X]|^3 f(j) \tag{6}$$

$$= k^3 \sum_{j=a}^{b} \left| j - \frac{a+b}{2} \right|^3 \frac{1}{b-a+1} \text{ since } k > 0 \tag{7}$$

$$= \frac{k^3}{b-a+1} \left(\sum_{j=a}^{\frac{a+b}{2}} \left(\frac{a+b}{2} - j \right)^3 + \sum_{j=\frac{a+b}{2}}^{b} \left(j - \frac{a+b}{2} \right)^3 \right) \tag{8}$$

$$= \frac{k^3}{b-a+1} \left(\sum_{j=0}^{\frac{b-a}{2}} j^3 + \sum_{j=0}^{\frac{b-a}{2}} j^3 \right) \tag{9}$$

$$\leq \frac{2k^3}{b-a} \sum_{j=0}^{\frac{b-a}{2}} j^3 \text{ since } b-a > 0 \tag{10}$$

Let $m = \frac{b-a}{2}$.

$$r^3 \leq \frac{k^3}{m} \sum_{j=0}^{m} j^3 \tag{11}$$

$$\leq \frac{k^3}{m} \left(\frac{1}{4}(m+1)^4 - \frac{1}{2}(m+1)^3 + \frac{1}{4}(m+1)^2 \right) \tag{12}$$

$$\leq \frac{k^3}{m} \left(\frac{m^4}{4} + \frac{m^3}{2} + \frac{m^2}{4} \right) \tag{13}$$

$$\leq \frac{k^3}{m} \left(\frac{m^4}{4} + m^4 + m^4 \right) \text{ since } m \geq \tfrac{1}{2} \tag{14}$$

$$\leq \frac{9}{4} k^3 m^3 \tag{15}$$

Which confirms that $r^3 \leq \frac{9}{32}k^3(b-a)^3 \leq k^3(b-a)^3$. □

Lemma 2 defines the distribution of a linear combination of uniformly distributed random variables.

Lemma 2. *Let $Y = \sum_{i=1}^{n} c_i X_i$ be a random variable where X_i is a discrete random variable uniformly chosen from the interval $[a_i, b_i]$ and c_i is a non-negative coefficient. When n tends to infinity, the distribution of Y tends to a normal distribution with mean $\sum_{i=1}^{n} c_i \frac{a_i+b_i}{2}$ and variance $\sum_{i=1}^{n} c_i^2 \frac{(b_i-a_i+1)^2-1}{12}$.*

Proof. Let $Y_i = c_i X_i$ be a random variable. We want to characterize the distribution of $\sum_{i=1}^{n} Y_i$. Let $m_i = \frac{b_i-a_i}{2}$. The variance of the uniform distribution over the interval $[a_i, b_i]$ is $\sigma_i^2 = \frac{(b_i-a_i+1)^2-1}{12} = \frac{(m_i+\frac{1}{2})^2}{3} - \frac{1}{12}$. We have $Var[Y_i] = c_i^2 Var[X_i] = c_i^2\sigma_i^2$. Let r_i^3 be the third central moment of Y_i. By Lemma 1, we have $r_i^3 \leq c_i^3(b_i-a_i)^3$. Let L be the term mentioned in the condition of Lyapunov's central limit theorem:

$$L = \lim_{n\to\infty} \frac{\left(\sum_{i=1}^{n} r_i^3\right)^{\frac{1}{3}}}{\left(\sum_{i=1}^{n} c_i^2\sigma_i^2\right)^{\frac{1}{2}}} \tag{16}$$

Note that the numerator and the denominator of the fraction are non-negative. This implies that L itself is non-negative. We prove that $L \leq 0$ as n tends to infinity.

$$L \leq \lim_{n\to\infty} \frac{\left(\sum_{i=1}^{n} 8c_i^3 m_i^3\right)^{\frac{1}{3}}}{\left(\sum_{i=1}^{n} c_i^2 \left(\frac{(m_i+\frac{1}{2})^2}{3} - \frac{1}{12}\right)\right)^{\frac{1}{2}}} \tag{17}$$

$$\leq \lim_{n\to\infty} \frac{\left(8\sum_{i=1}^{n} c_i^3 m_i^3\right)^{\frac{1}{3}}}{\left(\frac{1}{3}\sum_{i=1}^{n} c_i^2 m_i^2\right)^{\frac{1}{2}}} \tag{18}$$

$$\leq \lim_{n\to\infty} 2\sqrt{3}\sqrt[6]{\frac{\left(\sum_{i=1}^{n} c_i^3 m_i^3\right)^2}{\left(\sum_{i=1}^{n} c_i^2 m_i^2\right)^3}} \tag{19}$$

$$\leq \lim_{n\to\infty} 2\sqrt{3}\sqrt[6]{\frac{\sum_{i=1}^{n}\sum_{j=1}^{n}(c_i c_j m_i m_j)^3}{\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}(c_i c_j c_k m_i m_j m_k)^2}} \tag{20}$$

Note that in the last inequality, the terms $(c_i c_j m_i m_j)^3$ and $(c_i c_j c_k m_i m_j m_k)^2$ are of the same order. However, there are n times more terms in the denominator than the numerator. Therefore, when n tends to infinity, the fraction tends to zero which proves that $L = 0$ as n tends to zero. By Lyapunov's central limit theorem, as n tends to infinity, the expression $Y = \sum_{i=1}^{n} Y_i$ tends to a normal distribution with mean $E[Y] = \sum_{i=1}^{n} c_i E[X_i] = \sum_{i=1}^{n} c_i \frac{a_i+b_i}{2}$ and variance $Var[Y] = \sum_{i=1}^{n} c_i^2 Var[X_i] = \sum_{i=1}^{n} c_i^2 \frac{(b_i-a_i+1)^2-1}{12}$. □

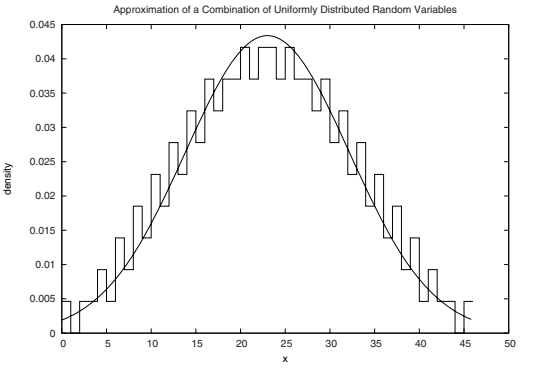


Fig. 2. The histogram is the actual distribution of the expression $3x + 4y + 2z$ for $x, y, z \in [0, 5]$. The curve is the approximation given by the Gaussian curve with mean $\mu = 22.5$ and variance $\sigma^2 = 84.583$.

Consider the knapsack constraint $\ell \leq \sum_{i=1}^{n} c_i x_i \leq u$. Let x_{n+1} be a variable with domain $D_{n+1} = [\ell, u]$. We obtain $x_j = \frac{1}{c_j}(x_{n+1} - \sum_{i=1}^{j-1} c_i x_i - \sum_{i=j+1}^{n} c_i x_i)$. Some coefficients in this expression might be negative. They can be made positive by setting $c'_i = -c_i$ and $D'_i = [-\max(D_i), -\min(D_i)]$. When n grows to infinity, the distribution of x_j tends to a normal distribution as stated in Lemma 2. In practice, the normal distribution is a good estimation even for small values of n. Figure 2 shows the actual distribution of the expression $3x + 4y + 2z$ for $x, y, z \in [0, 5]$ and its approximation by a normal distribution.

Given a variable x_i subject to a knapsack constraint, Algorithm 1 returns the assignment $x_i = v$ with the highest solution density. The *for loop* computes the average mean μ_i and the variance σ_i^2 of the uniform distribution associated to each variable x_i. Lines 4 to 5 compute the mean and the variance of the distribution of $x_{n+1} - \sum_{j=1}^{n} c_j x_j$ while Lines 6 and 7 compute the mean and the variance of $x_i = \frac{1}{c_i}(x_{n+1} - \sum_{j=1}^{i-1} c_j x_j - \sum_{j=i+1}^{n} c_j x_j)$. Since this normal distribution is symmetric and unimodal, the most likely value k_i in the domain D_i is the one closest to the mean μ_i. The algorithm finds and returns this value as well as its density d_i. The density d_i is computed using the normal distribution. Since the variable x_i must be assigned to a value in its domain, the algorithm normalizes on Line 9 the distribution over the values in the interval $[\min(D_i), \max(D_i)]$.

Lines 1 through 5 take $O(n)$ time to execute. Line 8 depends on the data structure used by the solver to encode a domain. We assume that the line takes $O(\log |D_i|)$ time to execute. The summation on Line 9 can be computed in constant time by approximating the summation with $\Phi_{m,v}(\max(D_i) + \frac{1}{2}) - \Phi_{m,v}(\min(D_i) + \frac{1}{2})$ where $\Phi_{m,v}$ is the normal cumulative distribution function with average m and variance v. The constant $\frac{1}{2}$ is added for the continuity correction. Other lines have a constant running time. The total complexity of Algorithm 1 is therefore $O(n + \log |D_i|)$. Note that Line 1 to Line 5 do not depend on the value of i. Their computation can therefore be cached for subsequent calls

1 **for** $j \in [1,n]$ **do**
2 $\quad \mu_j \leftarrow \frac{\min(D_j)+\max(D_j)}{2}$;
3 $\quad \sigma_j^2 \leftarrow \frac{(\max(D_j)-\min(D_j)+1)^2-1}{12}$;
4 $E \leftarrow \frac{l+u}{2} - \sum_{j=1}^{n} c_j \mu_j$;
5 $V \leftarrow \frac{(u-l+1)^2-1}{12} + \sum_{j=1}^{n} c_j^2 \sigma_j^2$;
6 $e \leftarrow \frac{E+c_i\mu_i}{c_i}$;
7 $v \leftarrow \frac{V-c_i^2\sigma_i^2}{c_i^2}$;
8 $k_i \leftarrow \operatorname{argmin}_{k \in D_i} |k-e|$;
9 $d_i \leftarrow \frac{1}{v\sqrt{2\pi}} e^{-\frac{(k_i-e)^2}{2v^2}}$;
10 **return** $\langle x_i = k_i, d_i \rangle$

Algorithm 1. FindDensity($[X_1, \ldots X_n]$, i) returns the assignment $x_i = k$ with the highest density as well as its density.

to the function over the same knapsack constraint. Using this technique, finding the variable $x_i \in \{x_1, \ldots, x_n\}$ which has an assignment $x_i = k$ of maximum density takes $O(\sum_{i=1}^{n} \log |D_i|)$ time.

A source of alteration of the distribution are values in the interval which are absent from the actual domain. Bounds consistency approximates the domain of a variable with its smallest covering interval. In order to reduce the error introduced by this approximation, one can compute the actual mean and actual variance of a domain D_i on Lines 2 and 3 instead of using the mean and the variance of the covering interval, at a revised overall cost of $O(\sum_{i=1}^{n} |D_i|)$.

4 Experiments

We evaluated the usefulness of solution counting information from knapsack constraints on four types of problems. The first two are benchmarks from the literature and feature 0-1 knapsack constraints. The third one is a magic square completion problem, featuring integer knapsack constraints. The last one is inspired from the area of rostering and features integer knapsack constraints as well. These problems were chosen because they are modeled using (almost) only knapsack constraints, in order to avoid the separate issue of combining heuristic information from different types of constraints.

A word on the notation used for the search heuristics investigated. The prefix "`Lexico`" refers to variable selection according to lexicographic order, whereas "`Dom`" refers to variable selection in increasing order of domain size. Keyword "`Random`" refers to random value selection. Keyword "`MaxSD`" refers to variable/value selection in decreasing order of solution density, or solely as a value selection heuristic if a prefix indicates a particular variable selection heuristic. Note that densities are considered separately from every constraint. In the case

of bounds consistency, keyword "MaxSD+" indicates that the actual mean and variance were computed from the domains in Algorithm 1.

The experiments were performed with ILOG Solver 5.1 on a Sun Fire 4800 (1.2 GHz CPU, 16 Gb RAM, 43 Gb swap) running under SunOS 5.10. Measures reported for heuristics involving random choices are an average over five runs.

4.1 Market Split Problem

The Market Split Problem was introduced by [2] as a challenge to LP-based branch-and-bound approaches. An optimization version of the problem exists but it was originally introduced as a satisfaction problem. An instance consists of m 0-1 equality knapsack constraints on the same $10(m-1)$ variables. Even small instances ($4 \leq m \leq 6$) are surprisingly hard to solve by standard means. We used the generator from [7], whose resulting instances have the same characteristics as those used in [6] and [1]. Table 2 reports the performance of three search heuristics with two levels of consistency on ten instances with $m = 4$. Note that heuristics based on domain size do not apply here.

Table 2. Results of a few search heuristics on ten 4-30 Market Split instances generated from [7]

consistency	heuristic	backtracks			time (sec.)		
		mean	min	max	mean	min	max
domain	LexicoRandom	245234.2	22770	689261	180.7	17.9	443.9
	LexicoMaxSD	93212.6	4634	248324	159.8	8.2	433.6
	MaxSD	59870.6	7015	175596	257.3	34.5	630.4
bounds	LexicoRandom	5848346.0	20671	19403712	121.3	0.4	396.9
	LexicoMaxSD	2463102.0	91989	8647160	116.8	4.4	400.1
	MaxSD	12543500.0	2701376	22010947	957.6	209.7	1664.7

With domain consistency. We observe that the `MaxSD` heuristic, based on the solution density of variable-value pairs, achieves about a four-fold reduction in the average number of backtracks compared to a random heuristic (variable selection is lexicographic but the coefficients of the constraints were randomly generated), but at the expense of slightly higher runtimes. The number of backtracks of `LexicoRandom` is consistent with what was reported in [6] with the same domain consistency algorithm for knapsack constraints. The `LexicoMaxSD` heuristic uses solution density information only to guide value selection. Since fewer solution densities are examined (the choice of variable is fixed), the search heuristic will be faster but probably less accurate as well. Despite a noticeable increase in the average number of backtracks with respect to `MaxSD`, the average runtime improves enough to beat `LexicoRandom`. The random restart strategy was tried in combination with the heuristics but it significantly deteriorated their performance: several instances could not be solved within the one-hour time limit.

Table 3. Number of backtracks of a few search heuristics on six multidimensional knapsack instances. Instances are labeled by their size: "number of variables; number of constraints". A cutoff time of one hour was used.

consistency	heuristic	instance (#vars;#constraints)					
		6;11	15;11	20;11	28;11	39;6	50;6
domain	LexicoRandom	0	10	157	3119	85275	–
	LexicoMaxSD	0	2	56	448	39305	–
	MaxSD	0	2	40	18	1438	–
bounds	LexicoRandom	1	66	729	22052	176615	–
	LexicoMaxSD	0	35	376	16937	98993	21532762
	MaxSD	0	0	3676	260952	–	–

Table 4. Runtime in seconds of a few search heuristics on six multidimensional knapsack instances. Instances are labeled by their size: "number of variables; number of constraints". A cutoff time of one hour was used.

consistency	heuristic	instance (#vars;#constraints)					
		6;11	15;11	20;11	28;11	39;6	50;6
domain	LexicoRandom	0.0	0.2	2.2	79.8	912.7	–
	LexicoMaxSD	0.0	0.2	1.9	22.8	795.0	–
	MaxSD	0.0	0.2	1.4	2.1	57.3	–
bounds	LexicoRandom	0.0	0.0	0.0	0.9	5.9	–
	LexicoMaxSD	0.0	0.0	0.1	3.1	13.4	3047.7
	MaxSD	0.0	0.0	0.9	72.5	–	–

With bounds consistency. As expected with this weaker level of consistency, the number of backtracks is significantly larger than with domain consistency but most runtimes are reduced as well. The `LexicoMaxSD` heuristic is the fastest overall. `MaxSD` does not perform well here.

4.2 Multidimensional Knapsack Problem

This set corresponds to the six `mknap` instances used in [5]. The `mknap1` set from the OR-Library, which are optimization problems, are transformed into satisfaction problems by fixing the objective function to its optimal value, thereby introducing a 0-1 equality knapsack constraint. The other constraints are upper bounded knapsack constraints on the same variables. The instances are of increasing size.

Tables 3 and 4 report the performance of search heuristics on the six instances. Among the heuristics tested, only `LexicoMaxSD` with bounds consistency is able to solve the last instance within an hour. With domain consistency, we note a correlation between increased use of exact solution densities and decreased backtracks and runtimes. Note however that these results are not statistically very significant because there are only three instances of reasonable size – the instances were used because they previously appeared in [5]. It is difficult to

Table 5. Results of a few search heuristics on twenty 9×9 Magic Square completion instances with about 90% holes. A cutoff time of one hour was used.

consistency	heuristic	backtracks			time (sec.)			# unsolved
		mean	min	max	mean	min	max	instances
domain	DomRandom	9362.4	3	354346	27.1	7.2	661.3	1.8
	DomMaxSD	694.4	2	3642	23.5	20.6	28.6	–
	MaxSD	3095.1	2	50750	27.3	21.4	36.4	–
bounds	DomRandom	133931.6	43	3864972	4.4	0.0	123.1	0.4
	DomMaxSD	266358.0	34	3159163	12.7	0.1	145.0	1.0
	DomMaxSD+	1258030.0	281	11816592	94.8	0.3	838.6	–
	MaxSD	200574.0	34	3148707	27.0	0.1	421.2	1.0
	MaxSD+	161512.0	280	757915	47.8	0.2	224.3	3.0

Table 6. Results of a few search heuristics on twenty 9×9 Magic Square completion instances with about 50% holes. A cutoff time of one hour was used.

consistency	heuristic	backtracks			time (sec.)			# unsolved
		mean	min	max	mean	min	max	instances
domain	DomRandom	15679.0	18	341084	72.8	2.2	2020.0	–
	DomMaxSD	2442.9	6	16014	14.6	2.3	96.6	–
	MaxSD	3102.6	7	17969	19.6	2.6	137.4	–
bounds	DomRandom	856060.1	183	17754444	36.3	0.0	744.9	0.2
	DomMaxSD	1503090.0	1232	26527968	88.7	0.1	1565.0	–
	DomMaxSD+	236459.0	1023	1257436	20.1	0.1	111.1	–
	MaxSD	284006.0	1098	2741124	62.3	0.2	597.1	1.0
	MaxSD+	200094.0	647	1219949	84.9	0.3	402.5	1.0

compare our results to those because in that paper only bounds consistency was enforced on knapsack constraints. With that same level of consistency, our heuristics do not perform as well.

4.3 Magic Square Completion Problem

This very old puzzle is built on a square $n \times n$ grid and asks that we place the first n^2 integers in the grid so that each row, column and main diagonal sums up to the same value. A partially filled Magic Square Problem asks for a solution, if one exists. It can be made harder to solve than the traditional version starting from a blank grid. This time we have two types of constraints, $2n+2$ integer knapsack constraints on n variables and one alldifferent constraint on n^2 variables. Note that the knapsack constraints have unit coefficients. Each variable ranges over n^2 values.

We first generated twenty 9×9 instances with about 10% of the squares already filled in. Table 5 reports our results. Our heuristics exploiting domain consistency perform well but with a noticeable computational fixed cost probably due to the size of the underlying graph, which impacts the solution density

computation. Despite the fact that these instances seem to have many solutions (or maybe because of it) — a `DomRandom` heuristic with bounds consistency solves almost every instance, often very fast — our heuristics based on discrete uniform distributions perform worse than `DomRandom`.

We then generated twenty 9×9 instances with about half of the squares already filled in. Table 6 reports our results. With domain consistency, all three heuristics solve every instance but our heuristics perform better both in terms of number of backtracks and runtime. Our heuristics with bounds consistency require about two orders of magnitude more backtracks but runtimes that are less than an order of magnitude longer.

4.4 Cost-Constrained Rostering Problem

This set was constructed for this paper and is inspired from a rostering context. Here a 25-day schedule is planned for four employees, who each day either work a two-, three-, five-hour shift, or not at all. Every day, exactly one of each type of shift must be covered. There is an hourly cost for making someone work, which varies both across employees and days. For each employee, the total cost must lie within a certain range. Finally, some employees are unavailable for certain shifts on certain days.

An employee-centered model for this problem has 100 variables in all (one per employee and per day), each with domain $\{0, 2, 3, 5\}$. There are 25 alldifferent constraints on four variables each (one for each day). There are four knapsack constraints on 25 variables each (one for each employee): the integer coefficients corresponding to the hourly costs are drawn uniformly at random from $[0, 9]$. Ten unavailabilities exclude some values from the domains. We consider two variants.

Upper-bounded costs. In this variant, the total cost for each employee is bounded above by an integer drawn uniformly at random from $[240, 260]$. This translates into upper bounded integer knapsack constraints.

Table 7. Results of a few search heuristics on ten rostering instances with upper bounded costs. A cutoff time of one hour was used.

consistency	heuristic	backtracks			time (sec.)			# unsolved
		mean	min	max	mean	min	max	instances
domain	DomRandom	152607.6	0	2653226	197.3	0.3	3280.5	3.0
	DomMaxSD	0	0	0	0.6	0.5	0.7	–
	MaxSD	0	0	0	0.9	0.8	1.0	–
bounds	DomRandom	644552.3	0	10692243	50.8	0.0	862.4	2.0
	DomMaxSD	0	0	0	0.0	0.0	0.0	–
	DomMaxSD+	0	0	0	0.0	0.0	0.0	–
	MaxSD	0	0	0	0.0	0.0	0.0	–
	MaxSD+	0	0	0	0.1	0.1	0.1	–

Table 7 reports the performance of search heuristics on ten feasible instances. These instances are very easy for our heuristics based on solution densities, either exact of approximate: every instance is backtrack-free. The `DomRandom` heuristics leave a few instances unsolved. Refining variable selection by considering domain size over dynamic degree did not help `DomRandom` but adding a random restarts strategy made it possible to solve every instance, with an average runtime under a second in the case of `DomRandom` with bounds consistency.

Exact Costs. In this variant, the total cost for each employee must equal an integer drawn uniformly at random from $[220, 240]$. This translates into equality integer knapsack constraints. Table 8 reports the performance of the search heuristics on ten feasible instances.

Table 8. Results of a few search heuristics on ten rostering instances with exact costs. A cutoff time of one hour was used.

consistency	heuristic	backtracks			time (sec.)			# unsolved
		mean	min	max	mean	min	max	instances
domain	DomRandom	–	–	–	–	–	–	5.8
	DomMaxSD	7.1	0	49	0.3	0.3	0.3	1.0
	MaxSD	5.2	0	40	0.4	0.4	0.5	–
bounds	DomRandom	–	–	–	–	–	–	5.3
	DomMaxSD	232.0	0	1976	0.0	0.0	0.0	1.0
	DomMaxSD+	29.0	0	161	0.0	0.0	0.0	1.0
	MaxSD	508.1	0	4930	0.1	0.0	0.5	–
	MaxSD+	89.0	2	560	0.1	0.1	0.2	–

Here heuristic behavior is similar for domain and bounds consistency. Even with smallest-domain variable selection and domain (or bounds) consistency enforced on every constraint of the problem, the behavior of the `DomRandom` heuristic is very erratic, failing to solve more than half of the instances on average and exhibiting backtrack numbers ranging from zero to almost three million. As before, dynamic degree did not help. The addition of a random restarts strategy on top of `DomRandom` helps to solve a few more instances but the overall behavior remains the same. In contrast, our heuristics using solution densities for value selection perform extremely well and show more robustness: low average and maximum number of backtracks. Note however that one instance out of the ten could not be solved by the `DomMaxSD` heuristic given one hour of computing time — using solution densities for variable selection as well appears to be more robust for this problem. A more extensive experiment with 100 similar instances still gave very few backtracks for the `MaxSD` heuristic. For the "bounds consistency" heuristics, there is a noticeable decrease of the number of backtracks when exact means and variances are computed.

5 Discussion

We showed how to evaluate the solution density of a set of variables subject to a knapsack constraint. The first method based on domain consistency computes the exact solution density. The second method approximates variable domains with intervals as it is done with bounds consistency. The experiments generally showed a significant advantage of search heuristics based on such information both in the number of backtracks and the computation time. The fact that the increased accuracy of the solution density information is almost always accompanied by a decreased number of backtracks indicates that this is relevant heuristic information.

However the experimental results so far do not clearly indicate which of the two algorithms should be used or even when one dominates the other. It is also unclear yet whether computing the exact mean and variance of a discrete domain generally helps. We plan to clarify those points in a further investigation. For the moment, we at least measured the relative error made by Algorithm 1 when computing solution densities for the Magic Square Completion Problem. On the instances with 90% holes, we observed a 5% error with exact mean and variance and a 9% error with approximated mean and variance. On the instances with 50% holes, the error was 30% in the first case and 35% in the other.

We haven't attempted here any aggregation of the solution density information from different constraints beyond simply taking the maximum. A true assessment of the potential of such an approach to heuristic search must consider more ways to aggregate. For example, taking the average solution density of a variable-value pair over the constraints in whose scope it is and choosing the pair maximizing that average was tried on the multidimensional knapsack instances and this outperformed the approach of [5].

Note that both solution density algorithms proposed can be easily adapted to lift the restriction of non-negative coefficients and domain values, at the expense of a larger graph in the case of the first algorithm. This means that the scope of this work can be broadened to general linear constraints.

Acknowledgments

We wish to thank the anonymous referees for their excellent comments and suggestions. Financial support for this research was provided by the Natural Sciences and Engineering Research Council of Canada.

References

1. Aardal, K., Hurkens, C.A.J., Lenstra, A.K.: Solving a System of Linear Diophantine Equations with Lower and Upper Bounds on the Variables. Math. Op. Res. 25, 427–442 (2000)
2. Cornuéjols, G., Dawande, M.: A Class of Hard Small 0-1 Programs. INFORMS J. Computing 11, 205–210 (1999)

3. Pesant, G.: A Regular Language Membership Constraint for Finite Sequences of Variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 482–495. Springer, Heidelberg (2004)
4. Pesant, G.: Counting Solutions of CSPs: A Structural Approach. In: Proc. IJCAI 2005, pp. 260–265. Professional Book Center (2005)
5. Refalo, P.: Impact-Based Search Strategies for Constraint Programming. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 557–571. Springer, Heidelberg (2004)
6. Trick, M.A.: A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. Annals of Operations Research 118, 73–84 (2003)
7. Wassermann, A.: The Feasibility Version of the Market Split Problem (last consulted on November 10, 2007), http://did.mat.uni-bayreuth.de/~alfred/marketsplit.html
8. Zanarini, A., Pesant, G.: Solution Counting Algorithms for Constraint-Centered Search Heuristics. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 743–757. Springer, Heidelberg (2007)

From High-Level Model to Branch-and-Price Solution in G12

Jakob Puchinger[1], Peter J. Stuckey[1], Mark Wallace[2], and Sebastian Brand[1]

[1] NICTA Victoria Research Laboratory
Department of Computer Science & Software Engineering
University of Melbourne, Australia
{jakobp,pjs,sbrand}@csse.unimelb.edu.au
[2] School of Computer Science and Software Engineering
Monash University, Melbourne, Australia
mgw@mail.csse.monash.edu.au

Abstract. The G12 project is developing a software environment for stating and solving combinatorial problems by mapping a high-level model of the problem to an efficient combination of solving methods. Model annotations are used to control this process. In this paper we explain the mapping to branch-and-price solving. G12 supports the selection of specialised sub-problem solvers, the aggregation of identical sub-problems, automatic disaggregation when required by search, and the use of specialised branching rules. We demonstrate the benefits of the G12 framework on three examples: a trucking problem, cutting stock, and two-dimensional bin packing.

1 Introduction

Combinatorial optimisation problems are easy to state, but hard to solve, and they arise in a huge variety of applications. Branch-and-price is one of many powerful methods for solving them. This paper describes how Dantzig-Wolfe decomposition, column generation and branch-and-price are integrated into the hybrid optimisation platform G12 [27]. The G12 project is developing a software environment for stating and solving combinatorial problems by mapping a high-level model of the problem to an efficient combination of solving methods. We call such a combination of methods a *hybrid algorithm*. Because there is no method for choosing the best way to solve a given problem, we believe the (human) problem solver must be able to experiment with different hybrid algorithms. To meet this purpose the G12 project is developing user-controlled mappings from a high level model to different solving methods. These mappings must satisfy three conflicting objectives. They must be

- efficient, enabling the human problem solver to tightly control the behaviour of the algorithm if necessary for performance;
- flexible, allowing plug-and-play between different sub-algorithms;
- easy-to-use and easy-to-change for efficient experimentation with alternative hybrid algorithms.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 218–232, 2008.

The mapping to branch-and-price presented in this paper is designed to meet all three objectives (in reverse order):

- The user can select branch-and-price and control its behaviour by annotating a high-level model of the problem.
- The generated algorithm can use a separate solver for the subproblem. The user can control the decomposition and select the subproblem solver by further annotations.
- Inefficiencies arising as a result of identical subproblems are avoided by aggregating them, but the user is still enabled to express search control in terms of variables in the original model. The system also supports specialised branching rules, allowing fine-grained control of search where necessary.

The G12 platform consists of three major components, the modelling language ZINC [12], the model transformation language CADMIUM [10], and several internal and external solvers written and/or interfaced using the general-purpose programming language MERCURY [26]. All solvers and solver instances are specified in terms of their specific capabilities, i.e. the type of problems they can solve, the type of information they can return, and how they solve a problem.

On the MERCURY level these specifications are described using type classes. Basic solvers such as a Finite Domain Constraint Programming (FD) solver or a Linear Programming (LP) solver can be used as building blocks for other solvers. The column generation and branch-and-price modules use and implement such solver type classes. This system of pluggable components allows us to quickly design new hybrid algorithms and to combine existing solvers in innovative ways.

Trucking problem. Consider the following trucking problem, inspired by [5]. We are given `N` trucks; each truck has a cost and an amount of material it can load. We are further given `T` time periods; in each time period a given demand of material has to be shipped. Each truck also has constraints on usage: in each consecutive k time periods it must be used at least l and at most u times. The ZINC model of the problem follows:

Trucking.zinc

```
int: P;                                          type Periods = 1..P;
int: T;                                          type Trucks = 1..T;
array[Periods] of int: Demand;                   array[Trucks] of int: Cost;
array[Trucks] of int: Load;                      array[Trucks] of int: K;
array[Trucks] of int: L;                         array[Trucks] of int: U;
array[Periods] of var set of Trucks: x;

constraint forall(p in Periods)(sum_set(x[p], Load) >= Demand[p]);

constraint forall(t in Trucks)(
        sequence([bool2int(t in x[p]) | p in Periods], L[t], U[t], K[t]));

solve minimize sum(p in Periods)(sum_set(x[p], Cost));
```

At each time period we need to choose which trucks to use in order to ship enough material and satisfy the usage limits. The `sum_set`(s, f) function returns $\Sigma_{e \in s} f(e)$, while the `sequence`$([y_1, \ldots, y_n], l, u, k)$ constrains the sum of each

subsequence of length k, $y_i + \cdots + y_{i-k-1}, 1 \leq i \leq n-k+1$ to be between l and u inclusive. As it stands this model is directly executable in an FD solver that supports set variables. There exist specialised propagators for `sum_set` and `sequence`. In ZINC we can control the search by adding an *annotation* on the solve item, for example:

```
solve :: set_search(x, "first_fail", "indomain", "complete")
      minimize sum(p in Periods)(sum_set(x[p], Loads) >= Demand[p]);
```

which indicates we label the set 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.

2 Dantzig-Wolfe Decomposition and Column Generation

Dantzig-Wolfe decomposition is a standard way to decompose an integer programming model into a master problem and one or several subproblems [8,9]. The bound on the objective resulting from the LP relaxation of the decomposed model is usually stronger than that of the original formulation (if the subproblem does not have the integrality property). This can result in a smaller search space in LP-based branch-and-bound algorithms.

The *Original Problem* has the form

$$\begin{aligned} \mathsf{OP}: \quad & \text{minimise} && \sum\nolimits_{k\in K} c^k x^k && \\ & \text{subject to} && \sum\nolimits_{k\in K} A_j^k x^k \geq b_j && \forall j = 1 \ldots M \\ & && x^k \in \mathcal{D}^k && k \in K. \end{aligned}$$

The $\mathcal{D}^k$ are finite sets of vectors in $\mathbb{Z}_+^{N_k}$ implicitly defined by additional constraints. We view the elements of $\mathcal{D}^k$ to be indexed using an index set P^k; that is, we have $\mathcal{D}^k = \{d_p^k \mid p \in P^k\}$. We can then alternatively write

$$\mathcal{D}^k = \{e^k \in \mathbb{R}^{N_k} \mid e^k = \sum\nolimits_{p\in P^k} d_p^k \lambda_p^k,\ \sum\nolimits_{p\in P^k} \lambda_p^k = 1;\ \lambda_p^k \in \{0,1\}\ \forall p \in P^k\}.$$

Substituting the x^k by the λ_p^k in OP, we obtain the *Master Problem*:

$$\begin{aligned} \mathsf{MP}: \quad & \text{minimise} && \sum\nolimits_{k\in K} \sum\nolimits_{p\in P^k} c^k d_p^k \lambda_p^k && && \\ & \text{subject to} && \sum\nolimits_{k\in K} \sum\nolimits_{p\in P^k} A_j^k d_p^k \lambda_p^k \geq b_j && \forall j = 1 \ldots M && (1) \\ & && \sum\nolimits_{p\in P^k} \lambda_p^k = 1 && k \in K && (2) \\ & && \lambda_p^k \in \{0,1\} && \forall p \in P^k,\ k \in K. && \end{aligned}$$

Dantzig-Wolfe decomposition typically results in a Master Problem with many variables. To deal with a possibly exponential number of variables, delayed column generation [9] is used. Starting from a restricted LP-relaxation of the

original problem, the Restricted Master Problem (RMP), variables (columns) are lazily included in order to find an optimal solution.

The simplex algorithm for solving linear programs proceeds from one basic feasible solution to the next one, always in direction of a potential improvement of the objective function. This is achieved by adding a variable with profitable reduced cost to the basis and by removing some other variable from it. Reduced costs can be seen as an optimistic estimate of the amount of improvement achieved by a unit increase of their corresponding variable. This is the crucial property of the simplex algorithm exploited in column generation. For every $\mathcal{D}^k$, a subproblem is solved to determine such variables. In case of a minimisation problem, the objective is to find feasible columns d^k with negative reduced cost:

$$(c^k - \pi A^k)d^k - \mu^k$$

where π are the dual variable values corresponding to the constraints (1) and μ^k is the dual value of the kth convexity constraint (2). We do not need to find a column with maximum profit; adding a "good" column is sufficient.

3 Solving with G12

The G12 system allows one to take a model written in Zinc, transform it to various underlying solvers using Cadmium, and then execute it. We can use standard or user-defined Cadmium transformations. Mappings from Zinc to FD or LP models are available [6]. To control these transformations the user can annotate the model. The trucking problem example illustrates the use of an annotation to define search for an FD solver.

At the solver programming language (Mercury) level, G12 defines interfaces to solvers such as an FD solver, a continuous interval constraint solver, and linear programming solvers using type classes. Various implementations of these interfaces are provided, e.g. for LP/MIP solvers such as CPLEX, COIN-OR/OSI, and others. The Dantzig-Wolfe decomposition column generation, default branch-and-bound, and branch-and-price solvers heavily rely on the LP solver interfaces. These interfaces provide standard predicates for variable creation, constraint posting, setting an objective function, and LP and MIP optimisation.

The advantage of this architecture is that we can easily plug different LP solvers into modules such as column generation and branch-and-bound.

3.1 Dantzig-Wolfe Decomposition and Column Generation in Zinc

In order to use Dantzig-Wolfe decomposition and column generation on a high-level model in G12, we need to annotate the model to explain: what parts define the sub-problems, which solver is to be used for each subproblem, and which solver is to be used for the master problem.

For instance, the trucking problem example can be annotated as follows:

```
array[Periods] of var set of Trucks: x :: colgen_var;

constraint forall(p in Periods)(
    sum_set(x[p], Load) >= Demand[p] :: colgen_subproblem_constraint(p, "mip"));

solve :: colgen_solver("lp") :: lp_bb(x, most_frac, std_split)
    minimize sum(p in Periods)(sum_set(x[p], Cost));
```

which exposes which variables `x` will be used in column generation. For each `Period` a subproblem is defined in terms of its constraints and solver. Note that we could have used a more specialised solver here since the subproblem is a knapsack problem. Finally, the solver for the master problem and the search specification, branch-and-bound selecting the most fractional variable first and performing a standard split, are attached to the `solve` item.

We then perform a Dantzig-Wolfe decomposition on the model, separating original, master, and subproblem variables, as well as adding constraints linking those variables:

Trucking.zinc (changes)

```
array[Periods] of var set of Trucks: mx :: colgen_master_var;
array[Periods] of var set of Trucks: sx :: colgen_subproblem_var;

constraint forall(p in Periods)(
    colgen_link_constraint([x[p]], mx[p], sx[p]);

constraint forall(p in Periods)(
    sum_set(sx[p], Load) >= Demand[p] :: colgen_subproblem_constraint(p, "mip"));

constraint forall(t in Trucks)(
        sequence([bool2int(t in mx[p]) | p in Periods], L[t], U[t], K[t]));

solve :: colgen_solver("lp") :: lp_bb(x, most_frac, std_split)
    minimize sum(p in Periods)(sum_set(mx[p], Cost));
```

The so called master variables are place-holders representing the implicit sums of the λ variables $\sum_{p \in P^k} d_p^k \lambda_p^k$ as introduced in MP. Note that the search is still expressed in terms of the original problem variables.

Since column generation is to be used, the transformation must linearise the master constraints and objective function. The subproblem solver could use the original set representation of the variables, but for this example it too requires linearisation of the sub-problem constraints.

We can linearise the master and subproblem constraints giving linear definitions for the `sum_set` and `sequence` globals. This can be done in Zinc as:

```
function var int:sum_set(var set of $T:s, array[$T] of int: cost) =
    sum(e in index_set(cost))(cost[e] * bool2int(e in s));

predicate sequence(array[int] of var int:y, int:l, int:u, int:k) =
    forall(i in min(index_set(y)) .. max(index_set(y)) - k + 1)(
       let { var int: s = sum(j in i .. i + k - 1)( y[j] ) } in
           s >= l /\ s <= u);
```

where `index_set` returns the set of indices of its array argument, and `bool2int` coerces a Boolean to `0..1`. Finally, we transform the array of set variables `x` to a two-dimensional array of `0..1` variables such that `x[p,t] = 1` if `t in x[p]`.

3.2 Implementation

Column generation works by first transforming the original model to the form demanded by the solvers. Then it builds the subproblems and attaches them to the requested solvers. These solvers must support optimisation with a linear objective function, and preferably support it in an incremental way.

Then the restricted master problem is defined and attached to a solver that supports delayed column generation: currently LP solvers, although we are working on adding a hybrid volume algorithm/LP solver [3,2].

The G12 Dantzig-Wolfe decomposition and column generation solver interface implements most of the standard functionality of the G12 LP solver interface. From the outside it looks mostly like a standard LP solver set up with the original problem using the original (linearised) variables. The mapping between the original variables and the master problem variables is straight-forward; we simply set

$$x^k = \sum_{p \in P^k} d_p^k \lambda_p^k.$$

The main difference lies in the initialisation of the column generation module. First the subproblem solver instances have to be added, then the variables to be decomposed are created, and finally the master problem constraints are posted.

Similarly to the simplex algorithm, column generation requires an initial feasible solution. If it is not provided by the user, we introduce artificial variables in order to determine it automatically. At the end of this first phase the artificial variables are removed from the problem [29].

Since the column generation algorithm alone only solves the LP-relaxed version of the problem, we have to branch in order to guarantee integrality of the variables. The default branch-and-bound module is a simple, standard linear programming based branch-and-bound algorithm branching on the original model variables, which does not affect the subproblem structure [30].

The additional branching constraints could of course render the RMP infeasible. But, since we are usually not dealing with the complete master problem, additional columns could restore feasibility of the RMP. Such columns are obtained by solving a problem very similar to the pricing problem [15].

The availability of the original variables in the column generation solver is the key to being able to use this solver in further hybrids. We can use it with an arbitrary search strategy on the original variables, or for example in combination with an FD solver, by communicating bounds on the original variables.

3.3 The Trucking Problem

We solved several different instances of our trucking example showing the advantages of using DW-decomposition. Table 1 shows results on five different instances displaying the number of search nodes and the time required for solving the model using an FD solver, using a linearised model with branch-and-bound (LP-BB), and using DW-decomposition and column generation (DW). For the trucking example the DW-decomposition is so strong that it yielded the optimal

Table 1. The trucking example: finite domain model versus linearised branch-and-bound versus DW-decomposition

Instance		FD		LP-BB			DW		
Trucks	Periods	Nodes	Time	Nodes	LP opt.	Time	Columns	LP/IP opt.	Time
4	6	4655	0.80s	3282	177.0	0.55s	19	220.0	0.18s
4	6	5860	0.85s	1992	177.0	0.47s	12	210.0	0.16s
4	6	4607	0.77s	3102	177.0	0.55s	20	224.0	0.18s
4	8	39848	5.04s	25646	267.0	2.64s	24	324.0	0.18s
6	7	2361926	215.90s	194000	244.8	18.75s	18	287.0	0.18s

integral solution in the root node without a need to branch; so instead of nodes we show the number of columns generated for the DW-decomposed problem. We also display the value of the LP-relaxation at the root node for the linear models. For this problem we used our own branch-and-bound module using CPLEX as LP solver as well as IP subproblem solver. In general, any kind of LP solver (with G12 interfaces) can be used as master solver, and also any kind of subproblem solver is supported.

4 Identical Subproblems

Dantzig-Wolfe decomposition often results in highly symmetrical models because of structurally identical subproblems, i.e. the objective coefficients, the master problem constraints and the subproblem constraints are identical. A typical example for such a model is the cutting stock problem [18,14].

4.1 Aggregating Identical Subproblems

Solving problems with identical subproblems by the pure Dantzig-Wolfe approach can be quite inefficient. This issue is usually overcome by aggregating the identical subproblems. The set K of subproblem indices is partitioned into sets K^s by grouping the indices of identical subproblems; s ranges over some S. We turn

$$\sum\nolimits_{k\in K^s}\sum\nolimits_{p\in P^k} d_p^k\lambda_p^k \qquad \text{into} \qquad \sum\nolimits_{p\in P^s} d_p^s\lambda_p^s$$

where λ_p^s are integer variables satisfying $0 \le \lambda_p^s \le |K^s|$ and $\sum_{p\in P^s}\lambda_p^s = |K^s|$.

The Master Problem MP becomes the *Aggregated Master Problem*:

$$\begin{aligned}
\text{AMP:}\quad \text{minimise}\quad & \sum\nolimits_{s\in S}\sum\nolimits_{p\in P^s} c^s d_p^s\lambda_p^s & \\
\text{subject to}\quad & \sum\nolimits_{s\in S}\sum\nolimits_{p\in P^s} A_j^s d_p^s\lambda_p^s \ge b_j & \forall j = 1\dots M\\
& \sum\nolimits_{p\in P^s}\lambda_p^s = |K^s| & s\in S\\
& \lambda_p^s \le |K^s|,\ \lambda_p^s \in \mathbb{Z}_+ & \forall p\in P^s,\ s\in S.
\end{aligned}$$

4.2 Automatic Disaggregation When Branching on Original Variables

The direct mapping between the original variables and the newly introduced variables is not obvious anymore. In the aggregated case we have

$$x^k = \sum\nolimits_{p \in P^s} \lambda_p^s d_p^s / |K^s|.$$

Unfortunately, this usually leads to highly fractional values for the original variables, even if the λ_p^s variables take integer values. We therefore first decompose the λ_p^s values into (non-aggregated) λ_p^k values preserving integrality as much as possible, and then we use the mapping for the non-aggregated case.

In order to allow branching on the original variables we have to disaggregate the problem as required by the branching. The column generation module allows one to post any kind of linear constraint on the original problem variables without affecting the subproblem structure. Each aggregated subproblem appearing in these constraints is automatically disaggregated and considered by the column generation iterations in the subsequent nodes. Given K identical subproblems, if a constraint is posted involving an original variable belonging to the kth subproblem, this subproblem becomes different to the others and is disaggregated (while the remaining $K-1$ subproblems are kept aggregated). In order to implement this complex behaviour, the column generation module maintains a mapping between the original variables and their associated subproblems. It also tracks the aggregation status of all the subproblems by keeping a list of active subproblems. The disaggregations are rolled back upon backtracking.

4.3 The Cutting Stock Problem

In the cutting stock problem, we are given `N` items with associated lengths and demands. We are further given stock pieces with length `L` and an upper bound `K` on the number of required stock pieces for satisfying the demand (a trivial upper bound is the sum over all the demands). The following ZINC model corresponds to the formulation by Kantorovich [18]:

CuttingStock.zinc

```
int: K;                                 type Pieces = 1..K :: colgen_symmetric;
int: N;                                 type Items  = 1..N;
int: L;
array[Items] of int: i_length;          array[Items] of int: i_demand;

array[Pieces]        of var 0..1: pieces :: colgen_var;
array[Pieces, Items] of var int:  items  :: colgen_var;

solve :: lp_bb([pieces, items], most_frac, std_split)
      :: colgen_ph(100, 10) :: colgen_solver("lp")
    minimize sum([ pieces[k] | k in 1..K]);

constraint forall(i in 1..N)(sum([items[k, i] | k in 1..K]) >= i_demand[i]);

constraint forall( k in 1..K)(
               sum(i in 1..N)(items[k,i] * i_length[i]) <= pieces[k] * L
                   :: colgen_subproblem_constraint(k, "knapsack"));
```

The original model is a linear program. The annotations for the column generation variables and subproblems are as before. But this time we introduce a new annotation `colgen_symmetric` which annotates a type. This indicates that the model is symmetric in this dimension and the resulting column generation should aggregate in this dimension. A CADMIUM transformation can then be used to create an aggregated version of the variables and constraints as follows:

CuttingStockAgg.zinc (changes)

```
var 0..1: s_pieces                  :: colgen_subproblem_var;
array[Items] of var int: s_items    :: colgen_subproblem_var;
var int: m_pieces                   :: colgen_master_var;
array[Items] of var int: m_items    :: colgen_master_var;

solve :: lp_bb([pieces, items], most_frac, std_split)
      :: colgen_ph("mip", 100, 10) :: colgen_solver("lp")
    minimize m_pieces;

constraint colgen_link(pieces, m_pieces, s_pieces);

constraint forall(i in Items) (
        colgen_link( [ items[k,i] | k in Pieces], m_items[i], s_items[i])
    );

constraint forall(i in 1..N)(m_items[i] >= i_demand[i]);

constraint sum(i in 1..N)(s_items[i] * i_length[i]) <= s_pieces * L
               :: colgen_subproblem_constraint(1, "knapsack");
```

The `colgen_link` constraints associate the aggregated master and subproblem variables with the original problem variables. The `m_pieces` and `m_items` variables are place-holders representing the implicit sums of aggregated λ variables $\sum_{p \in P^s} d_p^s \lambda_p^s$ as introduced in the AMP. The `s_pieces` and `s_items` variables are the actual subproblem variables. This model is similar to the well-known column generation formulation first described by Gilmore and Gomory [14], although that does not retain the original variables. Note that it is conceivable to use CADMIUM to *detect* symmetries and automatically add `colgen_symmetric` annotations.

In the following experiment we evaluate possible differences when using the aggregated and the non-aggregated DW-decomposition. The results are shown in Table 2. We display in percent how often an optimal solution, a feasible solution, or no solution was found. We further give average objective values and number of explored nodes where at least a feasible solution was found. Average run-times over all the instances are also shown. The maximum run-time per instance was 5 minutes. We used CPLEX as LP solver and a specialised dynamic programming algorithm implemented in MERCURY for solving the knapsack subproblems. The CPLEX MIP solver was used as primal heuristic to solve the restricted master problem to integrality at every 100th node with a time limit of 10 seconds, as specified using the `colgen_ph` annotation. The instances used are randomly generated using CUTGEN1 [13]. Instances of Classes 1–12 have stock length $L = 1000$; each class consists of 10 instances.

For almost all classes, aggregating identical subproblems presents an advantage in the number of solved instances, solution quality and solving time.

Table 2. Results for cutting-stock with a maximum run-time of 5 min.

Class	Items	No Aggregation						Aggregation					
		Opt %	Feas %	No %	Obj	Nodes	Time [s]	Opt %	Feas %	No%	Obj	Nodes	Time[s]
Class1	10	30	70	0	12.70	3325.80	210.40	30	70	0	12.60	3596.60	209.95
Class2	10	70	10	20	118.75	125	100.89	90	10	0	112.90	283.80	59.36
Class3	20	30	0	70	23.33	766.67	242.52	20	80	0	24.50	823.10	250.05
Class4	20	0	0	100	n.a.	n.a.	298.63	10	30	60	222.50	400	268.17
Class5	10	100	0	0	49.50	75.20	6.07	100	0	0	49.50	0	0.32
Class6	10	80	10	10	518.56	38.89	68.39	100	0	0	494.90	143.40	21.84
Class7	20	70	20	10	90.22	212.22	105.18	90	10	0	90	225.90	50.00
Class8	20	60	0	40	947.83	16.67	184.24	90	10	0	893.50	40.60	30.51
Class9	10	100	0	0	64	20	2.04	100	0	0	64	50	1.79
Class10	10	80	10	10	657.67	43.78	70.08	90	10	0	639.70	169	39.27
Class11	20	70	10	20	117.75	104.75	95.10	80	20	0	115.50	253.60	60.15
Class12	20	70	10	20	1182.25	10.25	154.79	80	20	0	1146.90	120.60	50.06
Average		63.33	11.67	25	330.74	457.60	128.19	73.33	21.67	5	327.46	514.61	86.79

5 Specialised Branching Rules

In order to overcome symmetry issues, specialised branching rules for specific problem types were developed; see e.g. [4]. They usually require changes to the subproblems during the branch-and-bound process. G12 enables users to implement such specialised branching rules, changing the structure of the subproblems, but preserving aggregations.

The column generation module allows one to ask for fractional columns of the DW-decomposed model. It returns their values as well as their entries in the constraint matrix of the master problem. Using this information the user can define specialised branching rules by introducing constraint branches on subproblem variables. In the master problem these constraint branches can be enforced by setting forbidden columns to zero in their respective branch. The column generation module provides a predicate by which the user can specify a list of column patterns that have to be set to zero. In our current system the specialised branching rules are implemented in Mercury. We are working on extensions to Zinc so that users will be able to specify such rules at the modelling level.

The Two-Dimensional Bin Packing Problem

In order to demonstrate the effectiveness of specialised branching rules we implemented a simple, well-known rule for the two-dimensional bin packing problem. It is similar to the one described in [22], which is based on a well known branching rule for set partitioning [25]. The solution space is divided by branching on whether two different items are in the same bin. We always choose the two highest items appearing in a pattern whose corresponding column generation master variable λ has an LP solution value closest to 0.5.

In the two-dimensional bin packing problem (2DBPP), we are given `N` rectangular items of given height and width. These items have to be placed on (or

cut out) of bins of height H and width W, of which there are at most K. The variant we consider here does not allow items to be rotated; only level packings are allowed. Each bin can be divided into several levels, and each level contains the items beside each other [19]. For ease of modelling, we assume that the items are sorted by non-increasing heights. The formulation in Zinc is as follows:

2DBinPacking.zinc

```
int: K;                 type Bins  = 1..K :: colgen_symmetric;
int: N;                 type Items = 1..N;

int: W;                 array[Items] of int: ItemWidth;
int: H;                 array[Items] of int: ItemHeight;

array[Bins]        of var 0..1: bin  :: colgen_var;
array[Bins, Items] of var 0..1: item :: colgen_var;

solve :: bp([bin, item], most_frac_master, special_split)
      :: colgen_ph("mip", 100, 10) :: colgen_solver("lp")
    minimize sum(k in Bins)(bin[k]);

constraint forall(j in Items)(sum(k in Bins)(item[k, j]) >= 1);

constraint forall(k in Bins)(
               is_feasible_packing(bin[k], [item[k, j] | j in Items])
                   :: colgen_subproblem_constraint(k, "mip"));

set of tuple(Items, Items): Idx = {(i, j) | i, j in Items where j >= i};

predicate
    is_feasible_packing(var 0..1: l_bin, array[Items] of var 0..1: l_item) =
        let { array[Idx] of var 0..1: x } in
            forall (i in Items)(
                sum(j in i..N)(ItemWidth[j] * x[i, j]) <= W * x[i, i])
            /\
            sum(i in Items)(ItemHeight[i] * x[i, i]) <= l_bin * H
            /\
            forall(j in Items)(l_item[j] = sum(i in 1..j)(x[i, j]));
```

The bp annotation to the solve item tells the system to use the branch-and-price algorithm choosing the most fractional master variable and using the specialised branching rule.

Table 3 displays the results of applying standard branching on the original variables or using the specialised branching rule. We tested these approaches on the set of 500 instances described in [19]. They are divided in 10 classes of 50 instances each, with item numbers ranging from 20 to 100 in each class. While many instances could be solved to optimality in the root node, our specialised branching rules did reach optimal solutions more often in the given limited run-time.

6 Related Work and Conclusion

The practical usefulness of column generation and branch-and-price has been well-established over the last 20 years [9,4]. More recently it has emerged that column generation provides an ideal method for combining approaches, such as constraint programming, local search, and integer/linear programming. Columns can be generated by constraint programming or application-specific algorithms, while the master problem is handled using branch-and-price [17,31,24,22].

Table 3. Results for two-dimensional bin packing with a maximum run-time of 5 min.

Class	Std. Branching						Sp. Branching					
	Opt %	Feas %	No %	Obj	Nodes	Time [s]	Opt %	Feas %	No%	Obj	Nodes	Time[s]
Class1	68	22	10	19.49	45.87	109.90	90	8	2	39.90	41.14	53.54
Class2	26	0	74	1.31	0	223.24	30	2	68	64.19	6.19	203.08
Class3	70	10	20	13.05	10	116.37	84	8	8	13.85	11.87	82.90
Class4	26	0	74	1.31	0	228.76	26	0	74	1.31	0	228.74
Class5	84	6	10	17.40	8.89	69.65	90	2	8	17.61	3.93	53.13
Class6	24	0	76	1.08	0	228.03	24	0	76	1.08	0	227.97
Class7	76	16	8	16.30	33.70	80.52	88	10	2	16.78	57.18	57.52
Class8	78	10	12	15.73	14.77	89.48	84	6	10	15.98	13.60	77.04
Class9	96	4	0	42.62	6.72	13.94	100	0	0	42.60	0.32	2.17
Class10	48	4	48	7.46	18.15	155.95	52	0	48	7.46	7.54	149.39
Average	59.6	7.2	33.2	17.95	17.58	131.58	66.8	3.6	29.6	23.65	18.38	113.55

For systems such as G12 that support hybrid algorithms, Dantzig-Wolfe decomposition, column generation and branch-and-price provide an elegant way for the different solving techniques to be combined. However, the specification of this form of hybrid is quite complex, as it requires adaptation of simplex-based approaches to support the lazy generation of columns. Thus systems such as ABACUS [16], MINTO [20], OPL script [28], MAESTRO [7], COIN/BCP [23], and SCIP [1] offer facilities to support the implementation of branch-and-price on top of generic integer/linear programming packages. However, these systems still require the user to understand the technical details of branch-and-price: the purpose was to support algorithm implementation rather than problem modelling.

Certainly column generation is technical, but for people trying to solve combinatorial problems the most important requirement is to be able to try out an algorithm, or more generally a hybrid algorithm, quickly and easily without rewriting the problem specification. The first attempt to provide a column generation library was in ECL^{i}PS^e [11]. This system introduced the idea of an aggregate variable appearing in the master problem to represent a set of values returned as columns from multiple solutions to identical subproblems. However this library assumes a fixed set of variables in each subproblem, and precludes search choices which break some of the subproblem symmetries. In order to achieve tight control over branch-and-price, sophisticated ECL^{i}PS^e users have required special adaptations of the column generation library in order to be able to work directly with low level primitives [21].

The facility to annotate the same Zinc model in two different ways, as in the examples above, and thus have the problem solved by the FD solver, or by column generation according to the annotation, is completely novel. Moreover the facility to perform search on user variables and have any symmetries which are dynamically broken during search still correctly and efficiently handled automatically by the column generation solver is also new. Thirdly the facility to define specialised search still using the mapping provided by the library provides the full flexibility needed by the expert user.

The G12 scheme is to add annotations to a conceptual problem model, and thus turn it into a design model that maps to a specific algorithm. Annotating a constraint, occurring in the conceptual model, with the (name of the) solver that will handle it, is a simple example of this scheme.

Column generation is an interesting challenge because it does not naturally fit into the above scheme. Certainly we view the column generation module as a solver in the normal way (as discussed in Section 3). However annotating a constraint with the column generation solver is not enough: the solver needs to know which subproblem the constraint belongs to, the master problem or the subproblem. Moreover there is not one column generation solver: the master problem might be sent to one underlying solver and the subproblem to another. Finally branch-and-price search is closely connected with the column generation solver, and annotations to control the search can be crucial to the performance of the algorithm.

Each requirement has been satisfied in Zinc by having a sufficiently expressive annotation language. For example an annotation with a compound term (`colgen_subproblem_constraint(p, "mip")`) was used to specify the subproblem solver in Section 3.1, and the search was specified by multiple annotations.

The next particular challenge of column generation is that the variables (and constraints) used in the conceptual model of the problem are quite different from those needed in the design model. Our column generation module automates this mapping using G12's Cadmium mapping language. To ensure the annotations are still meaningful with respect to the new variables, the annotations have to be transformed by Cadmium in the same way. Moreover the search control as illustrated in Section 4 must be mapped to search steps expressed in terms of the design model variables.

The greatest design and implementation challenge was to have these still work, fully automatically, when handling symmetry by generating aggregated variables (used when solving the subproblem) and dynamically disaggregating some of them during search. Indeed, each symmetry-breaking search step causes the design model to be updated so as to operate on a new set of variables.

The design model is expressed in terms of a simplified version of Zinc, illustrated in Section 4.3. The specification of our language for expressing design models is still fluid, and so currently the translation to the Mercury code – which is very similar, but uses different syntax – is by hand.

One interesting challenge arising out of this work is how to automatically detect identical subproblems. This is a completely novel form of automated symmetry detection, which is of significant practical value, as the results in Table 2 reveal.

We plan to implement the search annotation transformations necessary to enable specialised branching schemes to be expressed in Zinc. We also plan to build in an implementation of the generic branching scheme described in [29]. We further intend to address issues related to adding multiple columns and column pool management.

Finally we envisage to explore the use of the column generation module for solving a subproblem within a larger problem – thus supporting, for example, a combination of row and column generation.

Acknowledgements

We would like to thank the members of the G12 team at NICTA VRL for helpful discussions and implementation work.

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

References

1. Achterberg, T.: SCIP - a framework to integrate constraint and mixed integer programming. Technical Report 04-19, Zuse Institute Berlin, (2004), http://www.zib.de/Publications/abstracts/ZR-04-19/
2. Anbil, R., Forrest, J., Pulleyblank, W.: Column generation and the airline crew pairing problem. In: Documenta Mathematica, Extra Volume ICM (1998)
3. Barahona, F., Anbil, R.: The volume algorithm: producing primal solutions with a subgradient method. Mathematical Programming 87(3), 385–399 (2000)
4. Barnhart, C., Johnson, E.L., Nemhauser, G.L., Savelsbergh, M.W.P., Vance, P.H.: Branch-and-price: Column generation for solving huge integer programs. Operations Research 46(3), 316–329 (1998)
5. Boland, N., Surendonk, T.: A column generation approach to delivery planning over time with inhomogeneous service providers and service interval constraints. Annals of Operations Research 108, 143–156 (2001)
6. Brand, S., Duck, G.J., Puchinger, J., Stuckey, P.J.: Flexible, rule-based constraint model linearisation. In: Hudak, P., Warren, D. (eds.) Practical Aspects of Declarative Languages (PADL 2008). LNCS, vol. 4902, pp. 68–83. Springer, Heidelberg (2008)
7. Chabrier, A.: Génération de Colonnes et de Coupes utilisant des sous-problèmes de plus court chemin. PhD thesis, Université d'Angers, France (2002)
8. Dantzig, G.B., Wolfe, P.: Decomposition principle for linear programs. Operations Research 8(1), 101–111 (1960)
9. Desaulniers, G., Desrosiers, J., Solomon, M. (eds.): Column Generation. GERAD 25th Anniversary Series. Springer, Heidelberg (2005)
10. Duck, G.J., Stuckey, P.J., Brand, S.: ACD term rewriting. In: Etalle, S., Truszczyński, M. (eds.) ICLP 2006. LNCS, vol. 4079, pp. 117–131. Springer, Heidelberg (2006)
11. Eremin, A.: Using Dual Values to Integrate Row and Column Generation into Constraint Logic Programming. PhD thesis, Imperial College London (2003)
12. Garcia de la Banda, M., Marriott, K., Rafeh, R., Wallace, M.: The modelling language Zinc. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 700–705. Springer, Heidelberg (2006)
13. Gau, T., Wäscher, G.: CUTGEN1: a problem generator for the standard one-dimensional cutting stock problem. European Journal of Operational Research 84(3), 572–579 (1995)

14. Gilmore, P.C., Gomory, R.E.: A linear programming approach to the cutting-stock problem (part I). Operations Research 9, 849–859 (1961)
15. Gunluk, O., Ladanyi, L., Vries, S.D.: A branch-and-price algorithm and new test problems for spectrum auctions. Management Science 51(3), 391–406 (2005)
16. Jünger, M., Thienel, S.: The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization. Software: Practice and Experience 30(11), 1325–1352 (2000)
17. Junker, U., Karisch, S.E., Kohl, N., Vaaben, B., Fahle, T., Sellmann, M.: A framework for constraint programming based column generation. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 261–274. Springer, Heidelberg (1999)
18. Kantorovich, L.V.: Mathematical methods of organizing and planning production. Management Science 6(4), 366–422 (1960)
19. Lodi, A., Martello, S., Vigo, D.: Models and bounds for two-dimensional level packing problems. Journal of Combinatorial Optimization 8(3), 363–379 (2004)
20. Nemhauser, G.L., Savelsbergh, M.W.P., Sigismondi, G.C.: MINTO, a Mixed INTeger Optimizer. Operations Research Letters 15, 47–58 (1994)
21. Papadakos, N.: Integrated airline scheduling. Computers and Operations Research, available online (August 27, 2007) (to appear, 2007)
22. Puchinger, J., Raidl, G.R.: Models and algorithms for three-stage two-dimensional bin packing. European Journal of Operational Research 183(3), 1304–1327 (2007)
23. Ralphs, T., Ladanyi, L.: COIN/BCP users manual (2001)
24. Rousseau, L.-M., Gendreau, M., Pesant, G., Focacci, F.: Solving VRPTWs with constraint programming based column generation. Annals of Operations Research 130(1), 199–216 (2004)
25. Ryan, D.M., Foster, B.: An integer programming approach to scheduling. In: Wren, A. (ed.) Computer scheduling of public transport urban passenger vehicle and crew scheduling, pp. 269–280. North Holland, Amsterdam (1981)
26. Somogyi, Z., Henderson, F., Conway, T.: The execution algorithm of Mercury, an efficient purely declarative logic programming language. Journal of Logic Programming 29(1-3), 17–64 (1996)
27. Stuckey, P.J., Garcia de la Banda, M., Maher, M.J., Marriott, K., Slaney, J.K., Somogyi, Z., Wallace, M., Walsh, T.: The G12 project: Mapping solver independent models to efficient solutions. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 13–16. Springer, Heidelberg (2005)
28. Van Hentenryck, P., Michel, L.: OPL Script: Composing and controlling models. In: Apt, K.R., Kakas, A.C., Monfroy, E., Rossi, F. (eds.) Compulog Net WS 1999. LNCS (LNAI), vol. 1865, pp. 75–90. Springer, Heidelberg (2000)
29. Vanderbeck, F.: Branching in branch-and-price: a generic scheme. Technical Report U-05.14, Applied Mathematics, University Bordeaux 1, France (2005)
30. Villeneuve, D., Desrosiers, J., Lübbecke, M.E., Soumis, F.: On compact formulations for integer programs solved by column generation. Annals of Operations Research 139(1), 375–388 (2005)
31. Yunes, T.H., Moura, A.V., de Souza, C.C.: A hybrid approach for solving large scale crew scheduling problems. In: Pontelli, E., Santos Costa, V. (eds.) PADL 2000. LNCS, vol. 1753, pp. 207–293. Springer, Heidelberg (2000)

Simpler and Incremental Consistency Checking and Arc Consistency Filtering Algorithms for the Weighted Spanning Tree Constraint

Jean-Charles Régin

ILOG Sophia Antipolis
Les Taissounières HB2,
1681 route des Dolines,
06560 Valbonne, France
regin@ilog.fr

Abstract. The weighted spanning tree contraint is defined from a set of variables X and a value K. The variables X represent the nodes of a graph and the domain of a variable $x \in X$ the neighbors of the node in the graph. In addition each pair $(variable, value)$ is associated with a cost. This constraint states that the graph defined from the variables and the domains of the variables admits a spanning tree whose cost is less than K. Efficient algorithms to compute a minimum spanning tree or to establish arc consistency of this constraint have been proposed. However, these algorithms are based on complex procedures that are rather difficult to understand and to implement. In this paper, we propose and detail simpler algorithms for checking the consistency of the constraint and for establishing arc consistency. In addition, we propose for the first time incremental algorithms for this constraint, that is algorithms that have been designed in order to be efficiently maintained during the search for solution.

1 Introduction

In this paper, we consider the weighted spanning tree constraint (wst constraint).

Several filtering algorithms for constraints based on graph theory and particularly on trees have been proposed. For instance, the robust spanning tree problem[1] with interval data has been addressed in [2]; the "tree" constraint has been studied in [3] (this constraint enforces the partionning of a digraph into a set of vertex-disjoint anti-arborescences), and recently, the "Not-Too-Heavy Spanning Tree" constraint has been introduced in [7]. 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.

[1] From [2]:the robust spanning tree problem, given an undirected graph with interval edge costs, amounts to finding a tree whose cost is as close as possible of that minimum spanning tree under any possible assignment of costs.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 233–247, 2008.

In order to define it without introducing set variables or graph variables, we recall first the definition of a spanning tree and then we present the neighbor variables representation of a graph in CP.

A tree is a connected and acyclic graph. A tree $T = (X', E')$ is a spanning tree of $G = (X, E)$ if $X' = X$ and $E' \subseteq E$. In addition, if each edge of G is associated with a cost then the cost of a spanning tree of G is the sum of the costs of the edges of the tree.

The neighbor variables representation of a graph G consists of a variable set X corresponding to the nodes of G (i.e. x_i is associated with the node i in G and conversely) such that the domain of a variable x_i is equivalent to the neighbors of i in G (i.e. $j \in D(x_i) \Leftrightarrow j \in N(i)$ of G). Then, there is an equivalence between the cost of an edge in G and the cost of a value of a variable (i.e. $cost(i, j) = cost(x_i, j)$).

The weighted spanning tree constraint (wst constraint) is a constraint defined on the neighbor representation of 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 kind of constraint is not often present directly in real world applications, but it is used frequently as a lower bound of a more complex problems like hamiltonian path or node covering problems. For instance the minimum spanning tree is a well known bound of the traveling 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 [7]. These two problems were solved for a long time. The search for a minimum spanning tree can be solved by several methods and we will consider here the Kruskal's algorithm. The second problem is close of another problem called "Sensitivity Analysis of Minimum Spanning Trees" [14]. The best algorithms solve this problem in linear time. Unfortunately they are quite complex to understand and to implement (see [6] or [11] for instance).

Therefore, in this paper, we propose a simpler and easy to implement consistency checking and filtering algorithms for the wst constraint, because, currently, there is no CP Solver which contains such a constraint. This algorithm is based on the creation of a new tree while running Kruskal's algorithm for computing an mst. 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.

In addition, we will consider an important aspects of the algorithms which is usually ignored: the incremental aspect. This aspect is quite important in CP as shown for instance in[12] because the consistency checking algorithms and the filtering algorithms are systematically called during the search for solution. Thus, it is worthwhile to design algorithms that are able to exploit the previous computations in order to solve more quickly the problems they consider. In this

case and because the algorithms are called very often with only few modifications between two calls, any real saving is beneficial in practice.

The paper is organized as follows: First, we recall some concepts of graph theory and constraint programming. Then, we formally define the propositions on which the consistency and the arc consistency of the weighted spanning tree constraint are based. Next, we introduce a new data structure named tree of connected components which will lead us to propose a simple algorithm to establish arc consistency. Afterwards, we modify this algorithm in order to maintain it efficiently during the search for solution when some modifications happen or when a backtrack occurs. At last, we conclude.

2 Preliminaries

2.1 Graph Theory

A **tree** is a connected and acyclic graph. A tree $T = (X', E')$ is a spanning tree of $G = (X, E)$ if $X' = X$ and $E' \subseteq E$. The edges of E' are the **tree edges** of T and the edges of $E - E'$ are the **nontree edges** of T. A **forest** is a disjoint union of trees.

There are different methods to traverse all the nodes of a tree, we recall the one we will use in this paper: **the inorder traversal**. To traverse a non-empty binary tree in inorder, perform the following operations: 1. Traverse the left subtree in inorder. 2. Visit the root. 3. Traverse the right subtree in inorder.

2.2 Constraint Programming

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 introduce the particular notation $\mathcal{D}_0 = \{D_0(x_1), \ldots, D_0(x_n)\}$ to represent the set of initial domains of $\mathcal{N}$. on which constraint definitions were stated.

A **constraint** C on the ordered set of variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ is a subset $T(C)$ of the Cartesian product $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ that specifies the **allowed** combinations of values for the variables $x_{i_1}, \ldots, x_{i_r}$. An element of $D_0(x_{i_1}) \times \cdots \times D_0(x_{i_r})$ is called a **tuple on** $X(C)$. A value a for a variable x is often denoted by (x, a). Let C be a constraint. A tuple τ on $X(C)$ is **valid** if $\forall (x, a) \in \tau, a \in D(x)$. C is **consistent** iff there exists a tuple τ of $T(C)$ which is valid. A value $a \in D(x)$ is **consistent with** C iff $x \notin X(C)$ or there exists a valid tuple τ of $T(C)$ with $(x, a) \in \tau$. A constraint is **arc consistent** iff $\forall x_i \in X(C), D(x_i) \neq \emptyset$ and $\forall a \in D(x_i)$, a is consistent with C.

Definition 1. *A* **weighted spanning tree constraint** *is a constraint C defined on X the neighbor variable representation of a graph G, and associated with* `cost` *a cost function on the edge of G, and an integer K such that*
$T(C) = \{\ \tau$ *such that* τ *is a tuple on* $X(C)$
and the graph defined by τ *is a tree whose cost is less than* $K\}$
It is denoted by $wst(X, cost, K)$.

3 Consistency Checking

Proposition 1. *The constraint* $wst(X, cost, K)$ *is consistent if and only if the graph* G *defined by* X *has a minimum spanning tree* T^* *with* $cost(T^*) \leq K$.

We propose to use Kruskal's algorithm for searching for a mst. Kruskal's algorithm is a greedy algorithm that finds a minimum spanning tree for a connected weighted graph. The algorithm starts with a forest where each node in the graph is a separate tree. Then, it adds edges which join two nodes belonging to different trees of the forest and merges the two trees into one. The particularity of the algorithm is that the edges are selected in regards to their costs. For each step the edge which connects two distinct trees and whose cost is minimum is selected. Thus, Kruskal's algorithm can be easily implemented by traversing the edges in nondecreasing order of their costs and by adding edges connecting two disjoint trees, until all nodes of the graph are in the same connected component. The main issue to obtain an efficient implementation is to detect whether two nodes belong to the same tree or not. This operation can be efficiently performed by using the well known union-find data structure of Tarjan [15]. By combining the path compression and the union by rank heuristics, m operations on the union-find performed on a set of n elements run in $O(m\alpha(m, n))$ time [15], where $\alpha(m, n)$ is a functional inverse of Ackerman's function. Thus, we have:

Property 1. *If the list of edges ordered by non decreasing cost is available then Kruskal's algorithm can be implemented in* $O(m\alpha(m, n))$.

Algorithm 1. Kruskal's algorithm for computing a minimum spanning tree

```
getCCRoot(i): return find(i)
mergeCC(r_i, r_j): union(r_i, r_j)
initCC(n): for i = 1 to n do makeSet(i)
addEdge(ccT, T, r_i, r_j, {i, j})
    mergeCC(r_i, r_j)
1   updateCCTree(ccT, r_i, r_j, {i, j})
    add {i, j} to T

MinimumSpanningTree(nonDecrEC): (mst,ccTree)
    initCC(n)
2   initCCTree(ccT, n)
    T ← ∅
    for each {i, j} ∈ nonDecrEC while |T| < n − 1 do
        r_i ← getCCRoot(i); r_j ← getCCRoot(j)
        if r_i ≠ r_j then addEdge(ccT, T, r_i, r_j, {i, j})
    return (T, ccT)
```

Algorithm 1 is a possible implementation of Kruskal's algorithm using the union-find data structure. The algorithm returns the largest forest that can be built. At this point, we recommend to ignore lines 2 and 1 and parameter ccT.

Functions MAKESET, FIND and UNION are the classical union-find functions:
MAKESET(x) :{ $p[x] \leftarrow x$; $rank[x] \leftarrow 0$ }
FIND(x): { if $p[x] \neq x$ then $p[x] \leftarrow$ FIND($p[x]$) endif; return $p[x]$ }
LINK(x, y) :{ if $rank[x] > rank[y]$ then $p[y] = x$ else $p[x] = y$ endif
. if $rank[x] = rank[y]$ then $rank[y] \leftarrow rank[y] + 1$ endif }
UNION(x, y): {LINK(FIND(x),FIND(y)) }

4 Arc Consistency Filtering Algorithm

For each nontree edge $\{i, j\}$, we have to find the cost of a minimum spanning tree subject to the condition that the tree must contain the edge $\{i, j\}$. First, we recall the Optimality Conditions of a mst:

Theorem 1

- **[Path Optimality Condition].** *A spanning tree T^* is a minimum spanning tree if and only if it satisfies the following path optimality conditions: for every nontree edge $\{i, j\}$ of G, $cost(i, j) \geq cost(u, v)$ for every edge $\{u, v\}$ contained in the path in T^* connecting nodes i and j.*
- **[Cut Optimality Condition].** *A spanning tree T^* is a minimum spanning tree if and only if it satisfies the following cut optimality conditions: for every tree edge $\{i, j\}$ of G, $cost(i, j) \leq cost(u, v)$ for every edge $\{u, v\}$ contained in the cut formed by deleting edge $\{i, j\}$ from T^*.*

We will call $\{i, j\}$-tree, a tree which must contain the edge $\{i, j\}$. Then:

Property 2. *Let $G = (X, E)$ be a graph, $\{i, j\} \in E$ be an edge of G, and v be the minimum of the edge costs minus 1. Then, a minimum spanning $\{i, j\}$-tree of G is the mst of G when the cost of $\{i, j\}$ is equal to v. The cost of the minimum spanning $\{i, j\}$-tree is then equal to the cost of the mst plus $cost(i, j) - v$.*

Proof. Since $\{i, j\}$ is the edge with the minimum cost when its cost is equal to v, then it will necessary be a tree edge of any mst. ⊙
For the sake of clarity we will consider that T^* is a minimum spanning tree of G. The filtering algorithm is based on the following Proposition [7]:

Proposition 2. *Let $\{i, j\}$ be a nontree edge of G, and $\{u, v\}$ be the edge with the maximum cost contained in the path in T^* connecting nodes i and j.*
The tree T corresponding to the tree T^* in which the edge $\{u, v\}$ has been replaced by the edge $\{i, j\}$ is a minimum spanning $\{i, j\}$-tree of G.

Proof. If the edge $\{i, j\}$ is added to the tree then a cycle is created and the Path Optimality Condition implies that the edge of the cycle having the largest cost must be removed. Since an $\{i, j\}$-tree is wanted and from Property 2, we consider that $\{i, j\}$ has the smallest cost. So the edge that must be removed is $\{u, v\}$ because it has the largest cost. Thus a tree T is obtained. This tree satisfies the Path Optimality Condition for all the nontree edges because T^* does

and $\{i,j\}$ is considered as having the smallest cost. T also satisifies the path optimality condition for $\{u,v\}$. $\odot$

Let $minEC(T)$ and $maxEC(T)$ be the cost of the edge of T having respectively the minimum and the maximum cost. We deduce two corollaries:

Corollary 1. *All the nontree edges $\{i,j\}$ such that*
(i) $cost(i,j) > K - cost(T^*) + maxEdgeCost(T^*)$ are not consistent with C
(ii) $cost(i,j) \leq K - cost(T^*) + minEdgeCost(T^*)$ are consistent with C

So, we can immediately delete all the edges satisfying Corollary 1 (i) and avoid studying the edges satisfying Corollary 1 (ii). For the other edges, we have:

Definition 2. *Let $\{i,j\}$ be a nontree edge of G which does not satisfy Corollary 1, and $\{u,v\}$ be the edge with the maximum cost contained in the path in T^* connecting nodes i and j. Then, $\{u,v\}$ is called a* **support** *of $\{i,j\}$, and $S(u,v)$ is the list of nontree edges that are supported by $\{u,v\}$.*

Proposition 3. *Let $\{i,j\}$ be an nontree edge of T^* which does not satisfy Corollary 1, $\{u,v\}$ be the support of $\{i,j\}$.*
$\{i,j\}$ is consistent with C if and only if $cost(i,j) \leq K - cost(T^*) + cost(u,v)$.

We propose to efficiently compute the supports by introducing a new tree while running Kruskal's algorithm,

4.1 Tree of Connected Components Merges

Kruskal's algorithm proceeds by merging disjoint trees. Each time an arc is added to the spanning tree, two trees are merged together. We propose to explicitly represent these operations by creating a specific tree called: **connected component tree** or **ccTree**. Every merge is represented by a node in the ccTree.

A bottom-up creation of this tree is used. The leaves correspond to the nodes of the graph, because, in Kruskal's algorithm, initially each node defines a tree. Each time an edge is added to the mst by Kruskal's algorithm, a new ccTree node is created. This ccTree node has two children: one for each tree (so the ccTree is binary) that have been merged. Each tree created in Kruskal's algorithm has a pointer to the ccTree node which represents it. The ccTree contains at most $2n-1$ nodes. Figure 1 gives a minimum spanning tree of a graph and Figure 2 shows a tree of connected components obtained after running Kruskal's algorithm on this graph. The ccTree involves the following data:

- $ccT.size$: the current number of nodes of the tree
- $ccT.p[r]$: the ccTree leaf corresp. to the canonical element of node r of G
- $ccT.left[k]$ and $ccT.right[k]$: the left and the right child of the ccTree node k
- $ccT.parent[k]$: the parent of the ccTree node k
- $ccT.Gedge[k]$: the edge of G which lead to the creation of the ccTree node k
- $ccT.inorder[i]$: the i^{th} ccTree node visited by the inorder traversal
- $ccT.pos[k]$: the inorder index of the ccTree node k
- $ccT.height[k]$: the height (distance from the root) of ccTree node k

```
0
633 0
257 390 0
 91 661 228 0
412 227 169 383 0
150 488 112 120 267 0
 80 572 196  77 351  63 0
134 530 154 105 309  34 29  0
259 555 372 175 338 264 232 249 0
505 289 262 476 196 360 444 402 495 0
353 282 110 324  61 208 292 250 352 154 0
324 638 437 240 421 329 297 314 95  578 435 0
 70 567 191  27 346  83  47  68 189 439 287 254 0
211 466  74 182 243 105 150 108 326 336 184 391 145 0
268 420  53 239 199 123 207 165 383 240 140 448 202  57 0
246 745 472 237 528 364 332 349 202 685 542 157 289 426 483 0
121 518 142  84 297  35  29  36 236 390 238 301  55  96 153 336 0
```

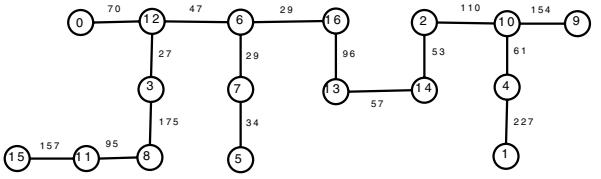
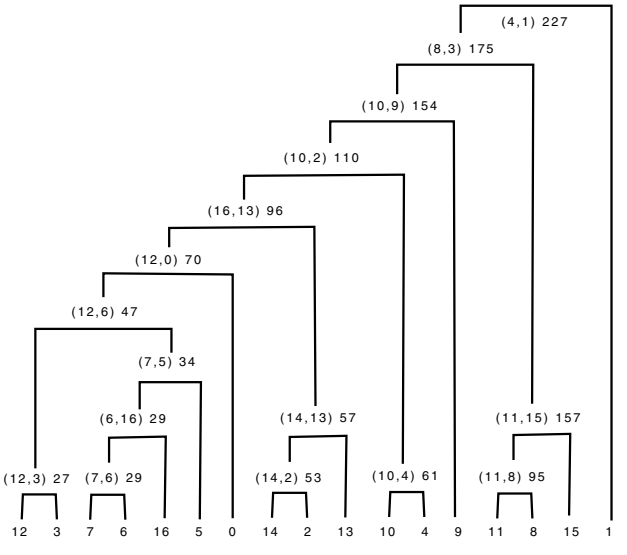


Fig. 1. The lower triangular matrix of problem gr17 of the TSPLIB and a Minimum Spanning Tree of this Graph



indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	(12,3)	(7,6)	(6,16)	(7,5)	(12,6)	(14,2)

indice	23	24	25	26	27	28	29	30	31	32
value	(14,13)	(10,4)	(12,0)	(11,8)	(16,13)	(10,2)	(10,9)	(11,15)	(8,3)	(4,1)

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
H	10	9	10	6	10	9	10	8	10	7	10	5	10	4	10	9	10	8	10	3	10	9	10	2	10	1	10	9	10	8	10	0	10
inorder	12	17	3	21	7	18	6	19	16	20	5	25	0	27	14	22	2	23	13	28	10	24	4	29	9	31	11	26	8	30	15	32	1

Pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
value	12	32	16	2	22	10	6	4	28	24	20	26	0	18	14	30	8	1	5	7	9	3	15	17	21	11	27	13	19	23	29	25	31

Fig. 2. Tree of Connected Components Merges of mst of problem gr17. The nodes that are not leaves contain the edge of G and its cost. Array of indices, array H, inorder and Pos are also represented.

The creation of the ccTree can be easily done while running Kruskal's algorithm as shown by Algorithm 1 (See Line 1.). Function UPDATECCTREE (ccT: tree, r_i , r_j, $\{i,j\}$) creates a new node in the ccTree whose children are $p[r_i]$ and $p[r_j]$ and with $Gedge = \{i,j\}$, Function INITCCTREE creates n leaves corresponding to the node of G, and Function INORDERTREETRAVERSAL performs an inorder tree traversal of the ccTree.

Once the ccTree is built, the support of any edge $\{i,j\}$ is the Gedge associated with the ccTree node created when the tree containing i and the tree containing j have been merged together. This node is the least common ancestor of the ccTree node i and the ccTree node j.

Definition 3. Lowest Common Ancestor *(LCA)*
For nodes u and v of tree T , query $LCA_T(u,v)$ returns the lowest common ancestor of u and v in T , that is, it returns the node farthest from the root that is an ancestor of both u and v.

Proposition 4. *Let i and j be two nodes of G, and ccT be the connected component tree built while running Kruskal's algorithm on G.*
Then, $ccT.Gedge[LCA_{ccT}(i,j)]$ is the edge that merged the tree of i and the tree of j while running Kruskal's algorithm on G.

Proof. First, note that a node i of G corresponds to the node i of ccT which is leaf. From the definition of ccT each node which is not a leaf corresponds to the merge of two trees. Therefore, the lowest common ancestor of two leaves of ccT corresponds to the merge of two disjoint trees of G that are identified by the extremities of the edge merging them. The *Gedge* data associated with this ccTree node contains it. ⊙

Corollary 2. *Let T^* be a mst of G, ccT be the connected component tree built while running Kruskal's algorithm and $\{i,j\}$ be a nontree edge of T^*. Then $\{u,v\} = ccT.Gedge[LCA_{ccT}(i,j)]$ is the support of $\{i,j\}$.*

Proof. By definition of the ccTree and the LCA, the LCA corresponds to the arc with the greatest cost in the path from i to j in T^*.⊙ The are several methods to solve directly the LCA problem, starting with [1] and improved by [10] and [13]. Unfortunately these methods are complex especially when the binary tree is not well balanced, which happens in our case. Another approach to solve the LCA problem in a non direct way has been introduced in [9]: the LCA problem is linearly equivalent to the Range Minimum Query Problem. Thus, by efficiently solving the RMQ problem, we obtain an efficient solution of the LCA problem.

Definition 4. Range Minimum Query *(RMQ)*
Let A be a length n array of numbers. For indices i and j between 1 and n, query $RMQ_A(i,j)$ returns the index of the smallest element in the subarray $A[i,...,j]$.

We will use the simple and nice transformation of LCA to RMQ proposed by [8]: "Let T be a rooted binary tree with n nodes.

- First perform an inorder tree walk in T and store it in an array $inorder[1, n]$.
- Store the heights of each node: $H[i]$ is the height of node $inorder[i]$ in T.
- Let Pos be the inverse array of $inorder$, i.e., $inorder[Pos[i]] = i$. It is easy to see that $LCA_T(v, w) = inorder[RMQ_H(Pos[v], Pos[w])]$: the elements in $inorder$ between $Pos[v]$ and $Pos[w]$ are exactly the nodes encountered between v to w during an inorder tree walk in T, so the RMQ returns the position k in H of the shallowest such nodes. As the LCA of v and w must be encountered between v and w during the inorder tree walk, $LCA(v, w) = inorder[k]$".

Figure 2 gives an example of ccTree, inorder traversal, H and Pos arrays. For instance, $LCA_T(10, 6) = inorder[RMQ_H(Pos[10], Pos[6])]$ which is equal to $inorder[RMQ_H(20, 6)] = inorder[19] = 28$ that is the index of the edge $\{10, 2\}$.

Now, the goal is to solve some RMQ requests as fast as possible. Harel and Tarjan [10] have shown that if several requests will be made then it is worthwhile to spend some time on preprocessing the tree in order to answer future queries faster. In [8] an $O(n)$ preprocessing is given, and with it any RMQ problem request for two values can be answered in $O(1)$. Unfortunately, this algorithm is quite complex and the authors doubt about its advantages in practice. Thus, we will use the much simpler algorithm proposed by [4]. It has a simple preprocessing step which is in $O(n \log(n))$ and solve each problem $RMQ(i, j)$ in $O(1)$ with only computing the minimum of two values. It is based on the fact that the RMQ problem for two values i and j can be easily solved if we have previously solved the RMQ problems for four values i, u, v, j such that $i \leq v \leq u \leq j$:

Property 3. *[4] Given i, j, u, v four integers such that $i \leq v \leq u \leq j$, $r_{iu} =$RMQ(i, u) and $r_{vj} =$RMQ(v, j). Then, If $A[r_{iu}] \leq A[r_{vj}]$ then* RMQ$(i, j) = r_{iu}$ *else* RMQ$(i, j) = r_{vj}$

Then, the nice idea is to work only with intervals whose length is a power of two, because any interval $[i, j]$ can be splitted into two such intervals:

Corollary 3. *[4] Given i, j two integers such that $i \leq j$, and $k = \lfloor \log_2(i) \rfloor$, $r_1 =$RMQ$(i, i + 2^k - 1)$ and $r_2 =$RMQ$(j - 2^k + 1, j)$. Then, If $A[r_1] \leq A[r_2]$ then* RMQ$(i, j) = r_1$ *else* RMQ$(i, j) = r_2$

If all the intervals whose length is a power of two are precomputed, then:

Corollary 4. *[4] Let A be an array of n values, and $M[i][k] =$RMQ$(i, i+2^k-1)$ with $i = 1..n$ and $k = 0..\lfloor \log_2(n) \rfloor$. Then, each* RMQ$(i, j)$*, with $1 \leq i < j \leq n$ can be computed in $O(1)$.*

The number of intervals $[i, p]$ with $p \leq n$ and whose length is a power of 2 is in $O(\log(n))$. Since there are n starting values, the overall complexity is in $O(n \log(n))$. Algorithm 2 is a possible implementation of the RMQ Problem. Note that this algorithm uses the arrays $Log2Array$ and $Pow2Array$ which contain respectively for a value k the result of mathematical operations: $\lfloor \log(k) \rfloor$ and 2^k. The values of these arrays can be computed in $O(n + \log(n))$ and this can be done once for all when the constraint is defined.

Algorithm 2. AC Filtering Algorihtm based on LCA Problem

```
precomputeRMQ(Rmq, n)
    for i = 1 to n do M[i][0] ← i
    for j = 1 to Log2Array[n] do
        for i = 1 to n − Pow2Array[j + 1] do
            minL ← Rmq.M[i][j − 1]
            minR ← Rmq.M[i + Pow2Array[j − 1]][j − 1]
            if Rmq.A[minL] ≤ Rmq.A[minR] then Rmq.M[i][j] ← minL
            else Rmq.M[i][j] ← minR
RangeMinimumQuery(Rmq, i, j): Integer
    logWidth ← Log2Array[j − i + 1]
    minL ← Rmq.M[i][logWidth]
    minR ← Rmq.M[j − Pow2Array[logWidth] + 1][logWidth]
    if Rmq.A[minL] ≤ Rmq.A[minR] then return minL
    else return minR
LowestCommonAncestor(ccT, Rmq, i, j): Integer
    p_i ← ccT.pos[i]; p_j ← ccT.pos[j]
    return ccT.inorder[RangeMinimumQuery(Rmq, p_i, p_j)]
computeAllSupports(ccT, SE)
    reduceCCTree(ccT, nonIncrEC)
    inorderTreeTraversal(ccT)
    Rmq.A ← ccT.height
    precomputeRMQ(Rmq, ccT.num)
    for each {i, j} ∈ SE do
        lca ← LowestCommonAncestor(ccT, Rmq, i, j)
        {u, v} ← ccT.Gedges[lca]
        append {i, j} to S(u, v); support(i, j) ← {u, v}
computePendingEdges(nonIncrEC, ccT): return ∅
computeEnteringEdges(nonIncrEC, T_1, T_2): Edge Set
    return {{i, j} ∈ nonIncrEC s.t. cost(i, j) > K − cost(T_2) + minEC(T_2)}
ACFilter(nonIncrEC, oldT, oldccT, R, T, ccT)
    for each {i, j} ∈ nonIncrEC while cost(i, j) > K − cost(T) + maxEC(T)
    do deleteEdge({i, j}, nonIncrEC)
    SE ← computeEnteringEdges(nonIncrEC, oldT, T)
    SE ← SE ∪ computePendingEdges(R, oldccT)
    computeAllSupports(ccT, SE)
    for each {u, v} ∈ T do
        for each {i, j} ∈ S(u, v) while cost(i, j) > K − cost(T) + cost(u, v) do
            deleteEdge({i, j}, nonIncrEC)
```

The preprocessing step is in $\theta(n \log(n))$ because the computation needs to be systematically done. However, it can be transformed into a maximum complexity because we can consider less than n nodes. The nodes that are not an extremity of an edge for which we need to compute a support are not needed in the ccTree,

so we can remove them. In order to maintain a binary tree, after a removal each node having only one child is contracted that is the node is deleted and its child becomes the child of its father. These operations have an amortized cost of $O(1)$ per removal. Thus, the number of nodes of the *ccTree* is less than or equal to $2n$ and so the complexity of the preprocessing step of the RMQ Problem is in $O(n \log n)$. Function REDUCECCTREE implements this idea.

The main function for implementing an AC filtering algorithm are given in Algorithm 2. The first call of a weighted spanning tree constraint can be implemented as follows (we consider that the set of edges has been sorted first):

$(T, ccT) \leftarrow$ MINIMUMSPANNINGTREE($nonDecrEC$)
if $|T| < n - 1$ **or** $cost(T) > K$ then trigger a failure
ACFILTER($nonIncrEC - T, \emptyset, \emptyset, \emptyset, T, ccT$)

Proposition 5. *Arc consistency of the weighted spanning tree constraint can be established in* $O(n + m + n \log(n))$

5 Maintenance During the Search

First, we consider the incremental aspects of the problem, that is we study the computation of the consistency of the constraint or the establishement of arc consistency when some modifications happen. Then, we will consider the problem of the restoration of the data structures when a backtrack occurs.

Note that the list of ordered edges is easy to maintain because we have just to manage the deletion of elements. So if any edge knows its previous and its next element in the ordered list then it can be removed from that list in $O(1)$.

There are two possible events: either a nontree edge is removed or a tree edge is removed. In the first case, the minimum spanning tree remains a minimum spanning tree and the condition of consistency or arc consistency remain satisified (See Propositions 1 and 2).So, there is nothing to do. This case may happen frequently because there are m edges and only $n - 1$ tree edges. The latter case is more complex and deserves a careful study, because a new spanning tree must be computed, so the ccTree may change and the lists of supported values also. This is the purpose of the next section.

5.1 Consistency Checking

If we accept an $O(n)$ complexity when some modifications happen, there is no need to maintain the union find and the ccTree data structures. In fact, each involves at most $2n$ elements. The new minimum spanning tree can be built from the current one by using its tree edges, and some computations can be saved if we rerun Kruskal's algorithm:

Proposition 6. *Let* $T^* = (X, A)$ *be a mst of* G *and* $\{i, j\}$ *a tree edge. There exists a mst of* $G - \{i, j\}$ *containing the set of edges* $A - \{i, j\}$.

Proof. Let be $\{u, v\}$ be the edge with the minimum cost contained in cut forming by deleting $\{i, j\}$ from T^*. Let T be the tree corresponding to T^* where $\{i, j\}$

has been replaced by $\{u, v\}$ then T satisifies the Cut Optimality Condition of $G - \{i, j\}$ and so is a minimum spanning tree of $G - \{i, j\}$ and T contains the edges $A - \{i, j\}$. ⊙

Proposition 7. *Let $T^* = (X, A)$ be a mst of G and $R = \{r_1, ..., r_k\}$ be a subset of the tree edge set. There exists a mst of $G - R$ containing the set of edges $A - R$ and a set $S = \{s_1, ..., s_k\}$ of edges such that for each $i = 1..k$ $r_i \leq s_i$.*

Proof. by induction on the number of element of R. From Prop. 6, this is true for 1 that is for $R = r_1$, because the cost of the mst of $G - r_1$ is greater than the cost of T^* so $s_1 \geq r_1$. Suppose it is true until i, that is for $R = r_1, ...r_i$. This means that we can build a tree T containing the edges of $A - \{r_1, ...r_i\}$. Now from Prop.6 if the edge r_{i+1} is removed then we can build another tree that wil contain the edges of T minus r_{i+1}. This tree will also contain an arc s_{i+1} such that $cost(s_{i+1}) \geq cost(r_{i+1})$ because T is a mst of $G - \{r_1, ..., r_i\}$. Therefore this is true for $i + 1$ and the proposition holds. ⊙

Consider that the sets A and R of edges are ordered w.r.t. the cost of the edges. While traversing the edges of E to build the new mst T, we can add the edges of $A - R$ and avoid considering some edges of E. Suppose that we search for an edge s_i replacing the edge r_i and that we have found replacement edges for all the edges of R smaller than r_i. If s_i is smaller than r_{i+1} then we can immediatly add to T all the edges of $A - R$ between r_i and r_{i+1} and we can search for a replacement of r_{i+1} from that position in E (See Algorithm 3).

5.2 AC Filtering Algorithm

The computation of a new mst changes the boundaries of Corollary 1. Thus, some edges can be immediately deleted and some supports must be computed for the first time for some other edges, named **entering** edges. In addition, the ccTree has been rebuilt when checking the consistency, so some support lists may be no longer correct. Consider ccT^* the ccTree associated with the old mst T^* and ccT the ccTree associated with the new mst T. We need first to run again the preprocessing of the RMQ problems for ccT. Then, we need to identify the edges for which their support is no longer valid or for which the validity must be verified. These edges are called **pending** edges, These are the edges belonging to any support list $S(u, v)$ where the node of ccT^* associated with the edge $\{u, v\}$ or a descendant of this node in ccT^* is associated with an edge of G which has been removed. Once these lists have been identified, it is necessary to compute the supports for all the edges contained in these lists and then to recompute new lists of supports. Then, all the lists of supports can be checked. This is required because the cost of the mst changed and so some edges that were consistant may become inconsistant. These checks of consistency of edges within support lists can be greatly improved if the elements are sorted, due to the structure of Proposition 3, so we need to sort the elements contained in the union of all the unvalid lists of support. Fortunately, it is possible to achieve such a sort in a very efficient way:

Proposition 8. *Let $G = (X, E)$ be a graph where E is ordered, and OE be the array of ordered indices of E (i.e. $OE[e] = k$ means that the edge e in in the k^{th}*

Algorithm 3. Recomputation of a mst after modifications

```
RecomputeMST(T, R, nonDecrEC): (mst, ccTree)
    initCC(n)
    initCCTree(ccT, n)
    A is the edge set of T; newT is empty
    {u, v} ← first(A)
    while {u, v} ≤ last(A) do
        ne ← next(A, {u, v})
        if {u, v} ∈ (A − R) then
            r_i ← getCCRoot(i); r_j ← getCCRoot(j)
            addEdge(ccT, newT, r_i, r_j, {i, j})
        else
            cpt ← cpt + 1
            for each {i, j} ∈ nonDecrEC from {u, v} while cpt > 0 do
                r_i ← getCCRoot(i); r_j ← getCCRoot(j)
                if {i, j} ≥ ne then
                    if {i, j} = ne then addEdge(ccT, newT, r_i, r_j, {i, j})
                    ne ← next(A, ne)
                else
                    if r_i ≠ r_j then
                        addEdge(ccT, newT, r_i, r_j, {i, j})
                        cpt ← cpt − 1
        {u, v} ← ne
    return (newT, ccT)
```

position in E). Let F be a subset of E and $n = |X|, m = |E|, m' = |F|$. Then, we can sort the elements of F with the same order as for E in $O(n + m')$.

Proof. Consider a Least Significant Digit Radix Sort and b a base (or radix) used to represent numbers. Such a sort is able to sort an array of numbers ranging from 0 to $\Delta - 1$ in $\log_b(\Delta)$ calls to a stable sort [5]. A stable sort like counting sort [5] is able to sort num numbers ranging from 0 to $b - 1$ in $O(num + b)$. Therefore the time complexity of a radix sort can be expressed as: $\log_b(\Delta) \times O(num + b)$. The edge set E is already sorted, and we can access for each edge to its position in E, so instead of considering the value associated with each element, it is equivalent to consider the position of the element in E. There are m possible positions, so to order F we need to order elements taking their value in $[0..m - 1]$. With a Radix Sort combined with a counting sort we can sort F in $\log_b(m) \times O(m' + b)$, because $\Delta = m$ and $num = m'$ in our case. If we use n as base b then we have $\log_n(m) \times O(m' + n)$. We have $m \leq n^2$ so $\log_n(m) \leq \log_n(n^2) = 2\log_n(n) = 2$. Therefore $\log_n(m) \times O(m' + n)$ is equivalent to $2 \times O(m' + n)$, that is $O(m' + n)$. ⊙

When used during the search for a solution the consistency checking and the arc consistency filtering of a wst constraint can be implemnted as follows (See also Algorithm 4). Let R be the set of edges of T that are deleted:

Algorithm 4. Incremental AC Filtering Algorithm

```
computeEnteringEdges(nonIncrEC, T1, T2): Edge Set
    if cost(T2) ≥ cost(T1) then
        return {{i, j} ∈ nonIncrEC s.t.
        K − cost(T2) + minEC(T2) < cost(i, j) ≤ K − cost(T1) + minEC(T1)}
    else
        return {{i, j} ∈ nonIncrEC s.t.
        K − cost(T2) + maxEC(T2) < cost(i, j) ≤ K − cost(T1) + maxEC(T1)}
computePendingEdges(R, ccT): Edge Set
    SE ← ∅
    add to UN the nodes of ccT associated with edges of R
    for each x ∈ UN do
        SE ← SE ∪ S(ccT.Gedge[x])
        S(ccT.Gedge[x]) ← ∅
        if ccT.parent[x] ∉ UN then add(ccT.parent[x], UN)
    sort(SE)
    return SE
```

$(newT, newccT) \leftarrow$ RecomputeMST$(T, R, nonDecrEC)$
if $|newT| < n - 1$ **or** $cost(newT) > K$ then trigger a failure
ACFilter$(nonIncrEC - newT, T, ccT, R, newT, newccT)$
$T \leftarrow newT$; $ccT \leftarrow newccT$

5.3 Restoration

There are two possible ways to deal with backtracks: either the state is exactly restored or an equivalent state is defined [12]. With a boundary based constraint the optimal solution at a node n may not be an optimal solution for the ancestors of n, therefore it is needed to restore the same state when a backtrack occurs. For the wst constraint, it means that we need to save all the modifications affecting the current minimum spanning tree. Then, we can easily restore the previous spanning tree because we know the set P of edges that have been deleted at a given search node and the set R of edges that have been added to the mst for this node (See Algorithm 5). All the restored edges must also be added to the pending edges. Here is a possible procedure to restore the previous state:

Algorithm 5. Restoration of a mst

```
RestoreMST(T, P, R, nonDecrEC): (mst, ccTree)
    initCC(n)
    initCCTree(ccT, n)
    A is the edge set of T; newT is empty
    for each {i, j} ∈ (A − R) ∪ P do
        r_i ← getCCRoot(i); r_j ← getCCRoot(j)
        addEdge(ccT, newT, r_i, r_j, {i, j})
    return (newT, ccT)
```

$(newT, newccT) \leftarrow$ RestoreMST$(T, P, R, nonDecrEC)$
if $|newT| < n - 1$ **or** $cost(newT) > K$ then trigger a failure
ACFilter$(nonIncrEC - newT, T, ccT, R, newT, newccT)$
$T \leftarrow newT$; $ccT \leftarrow newccT$

6 Conclusion

In this paper we have presented simpler algorithms for checking the consistency and for establishing arc consistency of the weighted spanning tree constraint. We have detailed, by giving the pseudo-code, several versions of these algotihms that are able to exploit the modifications that happen during the search for a solution in order to save some computations . The complexity of all the proposed filtering algorithms neither exceeds $O(m + n \log(n))$ which is quite good.

References

1. Aho, A., Hopcroft, J., Ullman, J.: On finding lowest common ancestors in trees. SIAM J. Comput. 5(1), 115–132 (1976)
2. Aron, I., Van Hentenryck, P.: A constraint satisfaction approach to the robust spanning tree problem with interval data. In: Proc. of UAI, pp. 18–25 (2002)
3. Beldiceanu, N., Flener, P., Lorca, X.: The tree constraint. In: Barták, R., Milano, M. (eds.) CPAIOR 2005. LNCS, vol. 3524, pp. 64–78. Springer, Heidelberg (2005)
4. Bender, M., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. Journal of Algorithms 57, 75–94 (2005)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction to Algorithms. MIT Press, Cambridge (1990)
6. Dixon, B., Rauch, M., Tarjan, R.: Verification and sensitivity analysis of minimum spanning trees in linear time. SIAM J. Comput. 21(6), 1184–1192 (1992)
7. Dooms, G., Katriel, I.: The not-too-heavy spanning tree constraint. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 59–70. Springer, Heidelberg (2007)
8. Fischer, J., Heun, V.: Theoretical and practical improvements on the rmq-problem, with applications to lca and lce. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 36–48. Springer, Heidelberg (2006)
9. Gabow, H., Bentley, J., Tarjan, R.: Scaling and related techniques for geometry problems. In: Proc. of STOC, pp. 135–143 (1984)
10. Harel, D., Tarjan, R.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. 13(2), 338–355 (1984)
11. Manku, G.: An o(m + n log* n) algorithm for sensitivity analysis of minimum spanning trees (1994), citeseer.ist.psu.edu/manku94om.html
12. Régin, J.-C.: Maintaining arc consistency algorithms during the search without additional space cost. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 520–533. Springer, Heidelberg (2005)
13. Schieber, B., Vishkin, U.: On finding lowest common ancestors: Simplification and parallelization. SIAM J. Comput. 17(6), 1253–1262 (1988)
14. Tarjan, R.: Sensitivity analysis of minimum spanning trees and shortest path trees. Information Processing Letters 14(1), 30–33 (1982)
15. Tarjan, R.E.: Data Structures and Network Algorithms. CBMS-NSF Regional Conference Series in Applied Mathematics (1983)

Stochastic Satisfiability Modulo Theories for Non-linear Arithmetic*

Tino Teige and Martin Fränzle

Carl von Ossietzky Universität, Oldenburg, Germany
{teige,fraenzle}@informatik.uni-oldenburg.de

Abstract. The stochastic satisfiability modulo theories (SSMT) problem is a generalization of the SMT problem on existential and randomized (aka. stochastic) quantification over discrete variables of an SMT formula. This extension permits the concise description of diverse problems combining reasoning under uncertainty with data dependencies. Solving problems with various kinds of uncertainty has been extensively studied in Artificial Intelligence. Famous examples are stochastic satisfiability and stochastic constraint programming. In this paper, we extend the algorithm for SSMT for decidable theories presented in [FHT08] to non-linear arithmetic theories over the reals and integers which are in general undecidable. Therefore, we combine approaches from Constraint Programming, namely the iSAT algorithm tackling mixed Boolean and non-linear arithmetic constraint systems, and from Artificial Intelligence handling existential and randomized quantifiers. Furthermore, we evaluate our novel algorithm and its enhancements on benchmarks from the probabilistic hybrid systems domain.

1 Introduction

Papadimitriou [Pap85] proposed the idea of uncertainty for propositional satisfiability by introducing *randomized* quantification in addition to existential quantification. This yields the stochastic propositional satisfiability (SSAT) problem where randomly quantified variables (randomized variables for short) are set to `true` with a certain probability. The solution of an SSAT problem Φ is a strategy to assign values to the existential variables that maximizes the overall satisfaction probability of Φ. Since the quantifier ordering of Φ, called prefix, allows an alternating sequence of existential and randomized quantifiers, the value of an existential variable depends on the values of the randomized variables with earlier appearance in the prefix. Consequently, in general such a solution is a tree of assignments to the existential variables depending on the values of preceding randomized variables. The SSAT framework is –at least theoretically– able to tackle many problems from Artificial Intelligence (AI) exhibiting uncertainty, e.g. stochastic planning problems. We just briefly note that there is

* This work has been partially supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, `www.avacs.org`).

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 248–262, 2008.

a lot of work done on efficiently transforming AI problems into SSAT formulae, e.g. cf. [LMP01, ML98, ML03]. Littman [Lit99][1] proposed an algorithm for SSAT which extends the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [DP60, DLL62] (DPLL is the basic algorithm of most modern propositional satisfiability solver) with acceleration techniques like *unit resolution*, *purification*, and *thresholding*. For a very comprehensive survey about stochastic satisfiability confer [LMP01]. More recently, Majercik further improved the DPLL-style SSAT algorithm by introducing *non-chronological backtracking* [Maj04].

There are several attempts to extend the stochastic framework beyond the purely propositional case. Doing so yields stochastic constraint programming [Wal02, TMW06, BS06, BS07] in which the domains for all variables, also non-quantified variables, are so far still finite. In [BS07] it was shown that the stochastic constraint satisfaction problem (SCSP) is PSPACE-complete also for multiple objectives by describing an algorithm for SCSPs in non-prenex form. The authors of [FHT08] introduced the stochastic satisfiability modulo theories (SSMT) problem and its application for the reachability analysis of probabilistic hybrid automata. Moreover, they described an algorithm for SSMT for decidable theories, e.g. linear arithmetic over the reals and integers. Although quantified variables in an SSMT problem still have finite domains, this restriction is relaxed for non-quantified variables or, equivalently, the innermost set of existentially quantified variables.

In this paper, we extend and benchmark the ideas from [FHT08]. First, we propose an SSMT algorithm for *non-linear* arithmetic over the reals and integers. (Note that for the non-linear case the SSMT problem becomes undecidable in general.) Second, we implement this algorithm and prove its concept by presenting empirical results. Third, in addition to the *thresholding* pruning rules we adapt the promising idea of *solution-directed backjumping* [Maj04] to our setting. The algorithm described in this paper is strongly based on the iSAT algorithm [FHT+07] for solving non-linear arithmetic constraint systems (involving transcendental functions) with complex Boolean structure over real- and integer-valued variables.[2] The iSAT approach tightly integrates the DPLL algorithm with interval constraint propagation (ICP, cf. [BG06] for an extensive survey) enriched by enhancements like conflict-driven clause learning and non-chronological backtracking. For a very detailed description of the iSAT algorithm the reader is referred to the original paper, in particular to the example on pages 217–219. As the core algorithm, iSAT is implemented in the constraint solver HySAT-II[3] which has been specifically designed for bounded model checking of hybrid (discrete-continuous) systems.

[1] We remark that the problem in this paper, called P-SAT, additionally contains universal quantification.

[2] Note that the input formula of iSAT is rewritten into conjunctive normal form beforehand and all arithmetic constraints are decomposed into primitive constraints [FHT+07, Section 2].

[3] A HySAT-II executable, the tool documentation, and benchmarks can be found on `http://hysat.informatik.uni-oldenburg.de`.

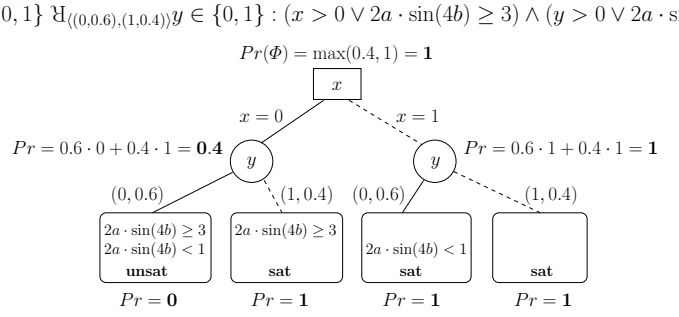


Fig. 1. Semantics of an SSMT formula depicted as a tree

Structure of the paper. In Section 2 we recall the definition of an SSMT problem while Section 3 presents an algorithm for SSMT for non-linear arithmetic theories. An experimental evaluation of that algorithm is given in Section 4. Section 5 concludes the paper and lists some directions for future work.

2 Stochastic Satisfiability Modulo Theories

The *satisfiability modulo theories* (SMT) problem (cf., e.g., [RT06]) is a decision problem for logical formulae wrt. combinations of background theories. Thus, SMT generalizes the well-known propositional satisfiability (SAT) problem. The stochastic SMT (SSMT) problem extends SMT to support *randomized quantification* over discrete variables as known from SSAT and SCSP.

Let φ be an SMT formula in conjunctive normal form (CNF) over some quantifier-free potentially non-linear arithmetic theory T over the reals, integers, and Booleans. I.e., φ is a logical *conjunction* of clauses, and a *clause* is a logical *disjunction* of (atomic) arithmetic predicates from T, as in $\varphi = (x > 0 \vee 2a \cdot \sin(4b) \geq 3) \wedge (y > 0 \vee 2a \cdot \sin(4b) < 1)$. An SSMT problem

$$\Phi = Q_1 x_1 \in \mathrm{dom}(x_1) \ldots Q_n x_n \in \mathrm{dom}(x_n) : \varphi$$

is specified by a *prefix* $Q_1 x_1 \in \mathrm{dom}(x_1) \ldots Q_n x_n \in \mathrm{dom}(x_n)$ binding the variables x_i to the quantifier Q_i,[4] and an SMT formula φ, also called the *matrix*. We require that the domains $\mathrm{dom}(x)$ of quantified variables x are finite (and thus discrete). A quantifier Q_i, associated with variable x_i, is either *existential*, denoted as $\exists$, or *randomized*, denoted as $\mathcal{U}_{d_i}$ where d_i is a discrete probability distribution over $dom(x_i)$. The value of a variable x_i bound by a randomized quantifier (randomized variable for short) is determined stochastically by the corresponding distribution d_i, while the value of an existentially quantified variable can be set arbitrarily. We usually denote such a probability distribution d_i by a list $\langle (v_1, p_1), \ldots, (v_m, p_m) \rangle$ of value pairs, where p_j is understood as the probability of setting variable x_i to v_j. The list satisfies $v_j \neq v_k$ for $j \neq k$, $\forall j : p_j > 0$, $\sum_{j=1}^{m} p_j = 1$, and $\mathrm{dom}(x_i) = \{v_1, \ldots, v_m\}$. For instance,

[4] Not all variables occurring in the formula φ need to be bound by a quantifier.

$\mathfrak{U}_{\{(0,0.2),(1,0.5),(2,0.3)\}}x \in \{0,1,2\}$ means that the variable x is assigned the value 0, 1, or 2 with probability 0.2, 0.5, and 0.3, respectively.

The semantics of an SSMT problem is defined by the *maximum probability of satisfaction*. Intuitively, for an SSMT formula $\Phi = \exists x_1 \in \mathrm{dom}(x_1)\ \mathfrak{U}_{d_2} x_2 \in \mathrm{dom}(x_2)\ \exists x_3 \in \mathrm{dom}(x_3)\ \mathfrak{U}_{d_4} x_4 \in \mathrm{dom}(x_4) : \varphi$ determine the maximum probability s.t. there is a value for x_1 s.t. for random values of x_2 there is a value for x_3 s.t. for random values of x_4 the SMT formula φ is satisfiable. (As standard, an SMT formula φ (in CNF) is *satisfiable* iff there exists a valuation σ of the variables in φ s.t. each clause is satisfied under σ, i.e., iff at least one atom in each clause is satisfied under σ. Otherwise, φ is *unsatisfiable*.) More formally, the maximum probability of satisfaction $Pr(\Phi)$ of an SSMT formula Φ is defined recursively by the following rules where φ denotes the matrix.

1. $Pr(\varphi) = 0$ if φ is unsatisfiable.
2. $Pr(\varphi) = 1$ if φ is satisfiable.
3. $Pr(\exists x_i \in \mathrm{dom}(x_i) \ldots Q_n x_n \in \mathrm{dom}(x_n) : \varphi)$
   $= \max_{v \in \mathrm{dom}(x_i)} Pr(Q_{i+1} x_{i+1} \in \mathrm{dom}(x_{i+1}) \ldots Q_n x_n \in \mathrm{dom}(x_n) : \varphi[v/x_i])$.
4. $Pr(\mathfrak{U}_{d_i} x_i \in \mathrm{dom}(x_i) \ldots Q_n x_n \in \mathrm{dom}(x_n) : \varphi)$
   $= \sum_{(v,p) \in d_i} p \cdot Pr(Q_{i+1} x_{i+1} \in \mathrm{dom}(x_{i+1}) \ldots Q_n x_n \in \mathrm{dom}(x_n) : \varphi[v/x_i])$.

For an example see Fig. 1.

3 SSMT Algorithm for Non-linear Arithmetic

In this section we present our algorithm SiSAT for calculating the maximum probability of satisfaction of an SSMT formula. More precisely, for a given SSMT formula Φ and a lower and upper target threshold $t_l, t_u \in [0,1]$ with $t_l \leq t_u$, the algorithm returns a witness value $p \leq Pr(\Phi)$ s.t. $p > t_u$ iff $Pr(\Phi) > t_u$, a value $p < t_l$ iff $Pr(\Phi) < t_l$, or otherwise (i.e., if $t_l \leq Pr(\Phi) \leq t_u$) the value $p = Pr(\Phi)$. If we wish to compute the exact value of $Pr(\Phi)$ we may thus simply set $t_l = 0$ and $t_u = 1$. SiSAT is an extension of the iSAT algorithm with an additional tightly integrated top layer for dealing with existential and randomized quantifiers. In the iSAT context, and thus in SiSAT, variables are interpreted over *interval valuations* which are manipulated during the proof search. As the iSAT algorithm is employed as the underlying core engine, we have to decompose all arithmetic predicates into so called *primitive constraints* by introducing additional auxiliary variables. A primitive constraint consists of exactly one relational operator, at most one arithmetic operator, and at most three variables. Note that for each (arithmetic) SMT formula there is an equi-satisfiable linearly-sized SMT formula in CNF just containing primitive constraints. For the input syntax of iSAT confer [FHT+07, Section 2]. As an example, the matrix of Φ from Fig. 1 can be rewritten to, e.g., $(x > 0 \vee h_1 \cdot h_2 \geq 3) \wedge (y > 0 \vee h_1 \cdot h_2 < 1) \wedge (h_1 = 2a) \wedge (h_2 = \sin(h_3)) \wedge (h_3 = 4b)$. All algorithmic enhancements of iSAT are naturally inherited, such as *conflict-driven clause learning & non-chronological backtracking*, the *two-watching scheme*, as well as the combined *unit* and *interval*

Algorithm 1. SiSAT(Pre, t_l, t_u)

In: A prefix Pre, lower and upper thresholds t_l, t_u.
Out: The satisfaction probability of the SSMT formula wrt. the thresholds.

1: **while** `true` **do**
2: **while** `true` **do**
3: $result := deduce()$. {Deducing.}
4: **if** $result =$ `CONFLICT` **then**
5: $resolved := analyze\_conflict()$. {Learning & Backjumping.}
6: **if not** $resolved$ **then**
7: **return** 0. {No solution for subproblem.}
8: **end if**
9: **else if** $result =$ `SOLUTION` **then**
10: **return** 1. {Solution found.}
11: **else**
12: **break**. {Leave loop for branching.}
13: **end if**
14: **end while**
{Existential quantifier.}
15: **if** $head(Pre) = \exists x \in \mathrm{dom}(x)$ **then**
16: $v \in \mathrm{dom}(x), set(x = v), \mathrm{dom}(x) := \mathrm{dom}(x) - \{v\}$.
17: $p_0 = \mathrm{SiSAT}(tail(Pre), t_l, t_u)$.
18: **if** $p_0 > t_u$ or $p_0 = 1$ or $\mathrm{dom}(x) = \emptyset$ **then**
19: **return** p_0. {Upper threshold exceeded or maximum possible probability reached or all branches investigated.}
20: **end if**
21: $p_1 = \mathrm{SiSAT}(Pre, \max(p_0, t_l), t_u)$. {Neglect probabilities less than p_0.}
22: **return** $\max(p_0, p_1)$. {Return maximum probability.}
23: **end if**
{Randomized quantifier.}
24: **if** $head(Pre) = \mathfrak{U}_d x \in \mathrm{dom}(x)$ **then**
25: $v \in \mathrm{dom}(x), (v, p_v) \in d, set(x = v), \mathrm{dom}(x) := \mathrm{dom}(x) - \{v\}$.
26: $p_{remain} = \sum_{v' \in \mathrm{dom}(x), (v', p') \in d} p'$.
27: $p_0 = \mathrm{SiSAT}(tail(Pre), (t_l - p_{remain})/p_v, t_u/p_v)$.
28: **if** $(p_v \cdot p_0) > t_u$ or $(p_v \cdot p_0) = 1$ or $\mathrm{dom}(x) = \emptyset$ **then**
29: **return** $p_v \cdot p_0$. {Upper threshold exceeded or maximum possible probability reached or all branches investigated.}
30: **end if**
31: **if** $p_{remain} < (t_l - p_v \cdot p_0)$ **then**
32: **return** $p_v \cdot p_0$. {Lower threshold cannot be reached by remaining branches.}
33: **end if**
34: $p_1 = \mathrm{SiSAT}(Pre, t_l - p_v \cdot p_0, t_u - p_v \cdot p_0)$. {Update thresholds.}
35: **return** $p_v \cdot p_0 + p_1$. {Return weighted sum.}
36: **end if**
{No quantifier left. Start iSAT branching.}
37: **if not** $decide\_next\_branch()$ **then**
38: **return** 1. {Approximative solution found.}
39: **end if**
40: **end while**

constraint propagation. For more details about iSAT the reader is referred to [FHT+07, THF+07].

Although we implemented SiSAT in an iterative manner, we present the basic ideas in a more intuitive recursive fashion (cf. Algorithm 1). Let $\Phi = Pre : \varphi$ be the SSMT formula to be solved and t_l, t_u be the lower and upper target thresholds, respectively. For the initial call SiSAT(Pre, t_l, t_u) the matrix φ, i.e. the clauses, of the SSMT formula Φ will be stored in a global database. New learned conflict clauses will be added to this database and, thus, will be public for all subproblems to be solved. The main loop of the SiSAT algorithm consists of the *deduction phase*, *conflict resolution*, and *branching*. Within the deduction phase the algorithm tries to conclude tighter intervals for the variables by chopping off non-solutions, starting from the domains of the variables as initial intervals. This is done by *unit propagation* and *interval constraint propagation*. Whenever a conflict occurs during search, i.e. if all constraints in a clause of the matrix are inconsistent with the current interval valuation, SiSAT analyzes the conflict. If the conflict can be resolved without revoking any assignment to a quantified variable then clause learning and backjumping are performed. Otherwise, i.e. if conflict resolution calls for undoing assignments to quantified variables, the function *analyze_conflict*() returns `false` indicating unsatisfiability of the current subproblem. Further backtracks concerning quantified variables are handled by the recursive nature of the algorithm. The branching step in the SiSAT framework corresponds to splitting an interval of a non-quantified variable or selecting a value for a quantified variable from its current domain. If a subproblem is decided to be satisfiable or unsatisfiable, the algorithm returns the probability 1 or 0 for that subproblem, resp., according to the semantics of Section 2. For the soundness of Algorithm 1, we require that the *deduce*() function returns `SOLUTION` *only* if the current quantifier prefix Pre is empty, i.e. branching for all quantified variables was performed beforehand.

The quantification issue is mainly treated within the branching step. In conformity with the semantical definition of the maximum probability, the branches for the quantified variables of the prefix are explored from left to right, and the resulting probabilities are combined correspondingly. The functions $head(Pre)$ and $tail(Pre)$ return the leftmost element $Q\,x \in \mathrm{dom}(x)$ of prefix Pre and the prefix originating from Pre where the leftmost element, i.e. $head(Pre)$, is eliminated, respectively. For a quantified variable x, we first select a value v from $\mathrm{dom}(x)$, assign v to x, and exclude v from $\mathrm{dom}(x)$. Then, we compute the probability for the branch $x = v$ by recursively calling the SiSAT procedure where the head element $Q\,x \in \mathrm{dom}(x)$ of the prefix is removed and the target thresholds are updated as follows: If x is existential then we simply preserve t_l, t_u. If x is randomized then we take the probability p_v for the value v and the maximum possible remaining probability $p_{remain} = \sum_{v' \in \mathrm{dom}(x), (v', p') \in d} p'$ for all remaining values $v' \neq v$ of x into account. I.e., the lower and upper target thresholds for this call are $(t_l - p_{remain})/p_v$ and t_u/p_v, resp., since if $t_l - p_{remain}$ cannot be reached by branch $x = v$ then t_l cannot be reached at all. (We remark that $t_l - p_{remain}$ can be a negative number and thus the new lower thresholds can

be negative. This fact, however, does not influence the correctness since the termination criterion concerning lower thresholds applies only if the remaining probability $p_{remain} \geq 0$ is strictly less than the (updated) lower threshold.)

We exploit some pruning rules concerning the target thresholds which allow to return a result without visiting all branches. These rules are generalizations of the *thresholding* rules for the propositional case from [LMP01]. Let p_0 be the result of the SiSAT call. Whenever the computed probability for the branch $x = v$, i.e. either p_0 or $p_v \cdot p_0$, exceeds the upper threshold t_u, we can skip investigation of all other branches and return the (positive) result. Note that the same holds if the domain $\mathrm{dom}(x)$ becomes empty or the maximum possible probability 1 is computed. For the randomized case, it could also happen that the maximum possible probability of all remaining branches p_{remain} cannot reach the new lower target threshold $t_l - p_v \cdot p_0$. Then we are also allowed to immediately return the (negative) result without further exploration of the remaining subtree. For the remaining subtree, i.e. $\forall v' \neq v : x = v'$, we set the target thresholds as follows: If x is existential then the new lower and upper thresholds are $\max(p_0, t_l)$ and t_u, resp., since we can neglect probabilities of the remaining subtree less than the already computed value p_0. If x is randomized then both new thresholds decrease by the computed probability $p_v \cdot p_0$. Let p_1 be the result of the second SiSAT call, then we combine the computed probabilities in accordance with the SSMT semantics, namely $\max(p_0, p_1)$ for the existential and $p_v \cdot p_0 + p_1$ for the randomized case, and return the result.

If all quantified variables are currently assigned to some values, i.e. the prefix Pre is empty ($Pre = head(Pre) = \emptyset$), the algorithm applies the usual iSAT branching for all non-quantified (Boolean, integer, and real-valued) variables by splitting their intervals. Note that the iSAT algorithm is in general not able to find a solution of any mixed Boolean and non-linear arithmetic constraint formula or to prove its absence, since search algorithms based on interval splitting and interval constraint propagation over the reals are incomplete deduction systems. In order to avoid a potentially infinite sequence of splitting intervals, branching stops if for each (non-quantified) variable z the width $\omega(z)$ of the current interval of z is less than a predefined value $\varepsilon > 0$, i.e. $\omega(z) < \varepsilon$. In such a case, the algorithm found a *hull consistent* interval valuation (for more details cf. [FHT+07]) which we consider as an approximative solution. Thus, we return the probability 1.

3.1 Solution-Directed Backjumping

For stochastic Boolean satisfiability, *solution-directed* and *conflict-directed backjumping* was introduced by Majercik [Maj04]. We note, however, that this idea was first proposed for quantified Boolean satisfiability in [GNT03]. We adapt the promising technique of solution-directed backjumping to the stochastic mixed Boolean and (non-linear) arithmetic framework. The idea of solution-directed backjumping (SDB) is to avoid exploring the remaining branches of a quantified variable x, whenever the truth value of the formula remains the same if the current value of x changed. I.e., the probability of all such subtrees are the same as for the current branch.

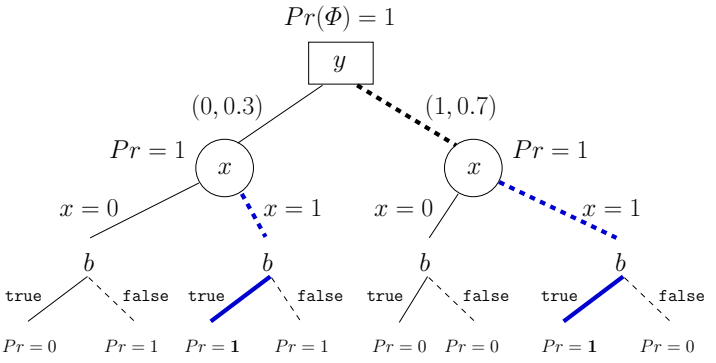


Fig. 2. Decision tree for Φ

Motivating this heuristics we first consider an example. Given the following SSMT formula

$$\Phi = \mathfrak{U}_{\langle(0,0.3),(1,0.7)\rangle} y \in \{0,1\}\ \exists x \in \{0,1\} : (\neg b\ \vee\ x \geq 1)\ \wedge\ (b\ \vee\ y < 1)$$

where $b \in \mathbb{B}$ is a Boolean variable[5]. The decision tree for Φ is depicted in Fig. 2. Calling the SiSAT algorithm on Φ, branching for the randomized variable y, say (1) $y = 1$, implies that $b = \texttt{true}$ (i.e. $b = 1$) by the second clause. Hence, the domain of b is narrowed to $[1, 1]$ by SiSAT's *deduce*() procedure. Then, by the first clause it follows that $x \geq 1$ has to hold, i.e. the domain of x is contracted to $\{1\}$. Thus, the only possibility for branching on the existential variable x is (2) $x = 1$. Here, *deduce*() returns `SOLUTION`. Consequently, the probability of branch (2) is 1. Since 1 is the maximum possible probability, SiSAT returns value 1 as the result for branch (1). I.e., the intermediate maximum satisfaction probability of Φ is $0.7 \cdot 1 = 0.7$. At this point, we take the idea of solution-directed backjumping into account: The assignment $y = 1$ has no impact on the satisfaction of the matrix (cf. Fig. 2). I.e., all other assignments to y also satisfy the formula and lead to the same probability. Hence, also the branch $y = 0$ results in probability 1 which means that we are able to conclude that $Pr(\Phi) = 0.7 + 0.3 \cdot 1 = 1$ without visiting the subtree for $y = 0$.

To be more formal, we first define a reason for a solution (analogously to a reason for a conflict). Given an SSMT formula $\Phi = Pre : \varphi$. Let ρ be a satisfying interval valuation of the matrix φ, i.e. $\rho(\varphi) = \texttt{true}$. If we consider hull consistency as an approximative solution then it is sufficient that ρ is hull consistent with φ, denoted as $\rho(\varphi) = \texttt{hc}$. We call a set $r \subseteq \{a : a \in c \in \varphi\}$ of atoms from φ a *reason for the satisfaction of* φ *under* ρ if the following hold:

1. $\forall c \in \varphi\ \exists a \in c : a \in r$, and
2. $\forall a \in r : \rho(a) = \texttt{true}$ (resp. $\rho(a) = \texttt{hc}$)

[5] The Boolean domain $\mathbb{B}$ is represented by the integer interval $[0, 1]$, where the values 0 and 1 correspond to the truth values `false` and `true`, respectively.

where $\rho(a)$ for an atom a gives the truth value of a under the interval valuation $\rho(x)$ of its variables x. Note that such a set r exists (while not being unique) whenever $\rho(\varphi) = \texttt{true}$ (resp. $\rho(\varphi) = \texttt{hc}$) holds. By $sat\_reasons(\varphi, \rho)$ we denote the set of all reasons r for the satisfaction of φ under ρ. In our example above, the only reason for the satisfaction is $\{x \geq 1, b\}$ where ρ is given by $\rho(y) = [1,1], \rho(x) = [1,1], \rho(b) = [1,1]$.

Given a reason $r \in sat\_reasons(\varphi, \rho)$, a quantified variable x, and the current domain $\mathcal{D}_x$ of x, the predicate $no\_impact(r, \rho, x, \mathcal{D}_x)$ returns $\texttt{true}$ only if the current interval $\rho(x)$ of x has no impact on the satisfaction. More precisely,

$$no\_impact(r, \rho, x, \mathcal{D}_x) = \begin{cases} \texttt{true} & ; \forall a \in r\ \forall v_x \in \mathcal{D}_x : \\ & \quad x \notin vars(a) \vee \\ & \quad \rho[v_x/x](a) = \texttt{true}\ (\text{resp. } \rho[v_x/x](a) = \texttt{hc}) \wedge \\ & \quad \forall y \in vars(a)\ \text{s.t. } y \neq x : y \notin qvars(\Phi) \\ \texttt{false} & \text{otherwise} \end{cases}$$

where $vars(a)$ gives the set of all variables occurring in atom a, $qvars(\Phi)$ gives the set of all quantified variables occurring in the SSMT formula Φ, and $\rho[v_x/x]$ is the modified interval valuation ρ defined by $\rho[v_x/x](x) = [v_x, v_x]$ and $\forall y \neq x : \rho[v_x/x](y) = \rho(y)$.

If $no\_impact(r, \rho, x, \mathcal{D}_x) = \texttt{true}$ then each assignment $x = v_x$ with $v_x \in \mathcal{D}_x$ for x also satisfies each atom a from r. If x occurs in an atom $a \in r$ together with another quantified variable y, e.g. $a = (x \geq y)$, the return value is always $\texttt{false}$. This definition allows to perform solution-directed backjumping for each quantified variable locally without considering the mutual interplay with other quantified variables. For $x \geq y$, the solution $\rho(x) = [1,1], \rho(y) = [0,0]$, and the current domains $\mathcal{D}_x = [0,1], \mathcal{D}_y = [0,1]$, we could otherwise wrongly conclude that the values 1 for x and 0 for y have no impact on the satisfaction, since $\forall v_x \in \mathcal{D}_x : v_x \geq 0$ and $\forall v_y \in \mathcal{D}_y : 1 \geq v_y$. However, the assignment $x = 0, y = 1$ does not satisfy $x \geq y$. For our set of benchmarks, the SSMT formulae do not contain atoms with more than one quantified variable as we will see in Section 4. Thus, the definition of $no\_impact(r, \rho, x, \mathcal{D}_x)$ is sufficient for our application domain. However, in future work we will develop a more general and more global reasoning mechanism to tackle this issue.

The extended SiSAT algorithm supporting solution-directed backjumping is enriched by two more pruning rules which are only applied if a solution ρ with a fixed reason $r \in sat\_reasons(\varphi, \rho)$ was found. Let x be an existential variable in rule 1 and a randomized variable in rule 2, $\mathrm{dom}(x)$ be the updated domain of x, and p_0 be the currently computed probability. If x is randomized then p_v is the probability of the currently processed branch and p_{remain} the sum of the probabilities of the remaining branches (cf. Algorithm 1). The solution-directed-backjumping rules are as follows:

1. **if** $no\_impact(r, \rho, x, \mathrm{dom}(x))$ **then return** p_0.
2. **if** $no\_impact(r, \rho, x, \mathrm{dom}(x))$ **then return** $p_v \cdot p_0 + p_{remain} \cdot p_0$.

4 Evaluation of the Algorithm

In this section, we evaluate our algorithm on SSMT formulae encoding discrete-time probabilistic hybrid automata. A probabilistic hybrid automaton (PHA) as described, e.g., in [FHT08] extends the notion of a hybrid automaton, where the non-deterministic selection of a transition is enriched by a probabilistic choice according to a distribution over variants of the transition. I.e., each transition carries a (discrete) probabilistic distribution. Each probabilistic choice within such a distribution leads to a potentially different successor mode while performing some discrete actions. For our case study, we are especially interested in k-bounded model checking (BMC) problems, i.e., we want to prove or disprove whether a given property P is satisfied with probability greater or equal p in a probabilistic hybrid automaton $\mathcal{H}$ along all its traces of length up to k. The automata considered for the experiments are shown in Fig. 3. These benchmarks are hand-made and serve as a first indicator for proving the concept of the approach and showing its current limits as well as the impact of the suggested algorithmic enhancements.

4.1 Description and Encoding of the Case Studies

Let us consider the probabilistic automaton $\mathcal{H}_1$ depicted in Fig. 3. $\mathcal{H}_1$ is not hybrid since it lacks continuous state components but serves as an illustration of the idea of a probabilistic choice. The initial mode of $\mathcal{H}_1$ is s_1 (indicated by the incoming edge). The system can change its current mode by taking an outgoing transition if its transition guard evaluates to true. In our example, there is just one outgoing transition t_1 with the trivially satisfied guard `true`. After nondeterministically selecting a transition, the follower mode and action performed is given by a discrete distribution. Taking t_1 in $\mathcal{H}_1$, the probability of reaching s_1 and s_2 are 0.9 and 0.1, respectively. For a given reachability property P, say reaching mode s_2 in $\mathcal{H}_1$, the problem is to determine the maximum probability of satisfying P in k steps. I.e., the underlying problem is to find a strategy s.t. selecting a transition maximizes the probability of satisfying P. For 1 step, the probability Pr of reaching s_2 obviously is 0.1, for 2 steps $Pr = 0.1 + (0.9 \cdot 0.1) = 0.19$, and in general for $k \geq 1$ steps $Pr = \sum_{i=0}^{k-1} \left(0.1 \cdot 0.9^i\right)$. For $\mathcal{H}_1$ there are no alternative transitions over which a maximization could be achieved. However, the initial mode s_1 in $\mathcal{H}_2$ has two outgoing transitions. Assume that $k_y = 1$ and $c = 0$, then both transitions are enabled, i.e. the guards $y > c$ of t_1 and `true` of t_3 are true. Thus, we have to opt for either t_1 or t_3. For each step depth, we cannot reach the target state s_2 without taking t_1. Hence, the selection of t_3 does never yield the maximum probability of satisfying the reachability property.

We encoded the next state relation of $\mathcal{H}_1$, $\mathcal{H}_2$, and $\mathcal{H}_3$ as SSMT formulae and unwind these up to some depth k. To gain an impression of that encoding, we exemplify it for $\mathcal{H}_1$. For more details confer [FHT08]. Let k be the unwinding depth. Then, for each step $i = 1, \ldots, k$ and for the transitions t_1, t_2 we introduce existential variables $e^i_{t_1}, e^i_{t_2}$ encoding the nondeterministic choice and

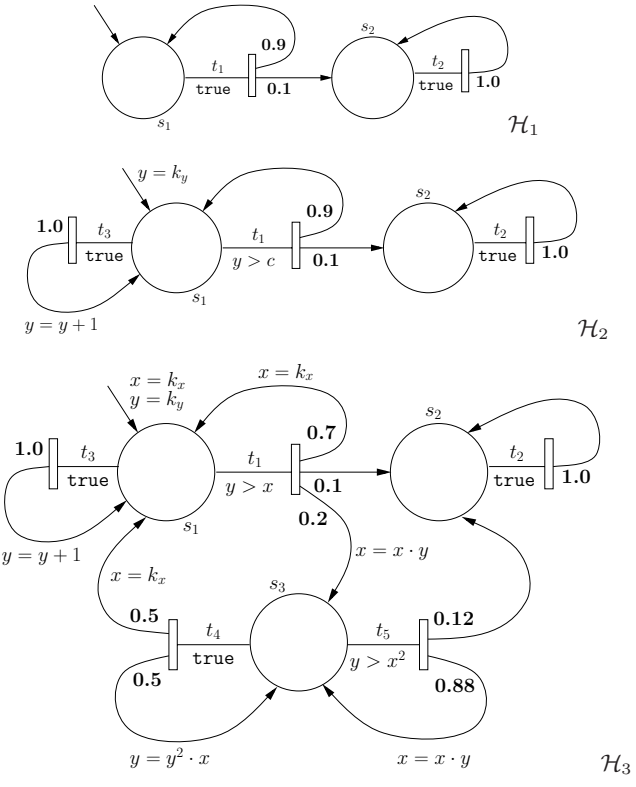


Fig. 3. Probabilistic hybrid automata $\mathcal{H}_1$ (top), $\mathcal{H}_2$ (middle), and $\mathcal{H}_3$ (bottom)

randomized variables $r^i_{t_1}, r^i_{t_2}$ encoding the probabilistic choice.[6] I.e., the prefix of the SSMT formula for step i is given by $\exists e^i_{t_1} \in \{0,1\}\ \mathfrak{U}_{\langle(0,0.9),(1,0.1)\rangle} r^i_{t_1} \in \{0,1\}\ \exists e^i_{t_2} \in \{0,1\}\ \mathfrak{U}_{\langle(0,1.0)\rangle} r^i_{t_2} \in \{0\}$. The matrix is constructed as follows. The initial condition is $s^0_1 \wedge \neg s^0_2$, and the target property is $(s^0_2 \vee \ldots \vee s^k_2)$, where s^i_n is a Boolean variable encoding whether $\mathcal{H}_1$ is in mode s_n before step $i+1$ is executed. At each point of time, the system has to be in exactly one mode, and exactly one transition has to be taken for a mode change. I.e., $s^i_1 + s^i_2 = 1$ and $e^i_{t_1} + e^i_{t_2} = 1$. The transition relation is encoded as:

$$
\begin{array}{l}
(s^{i-1}_1 \wedge (e^i_{t_1} = 1) \wedge (r^i_{t_1} = 0) \wedge s^i_1) \vee \\
(s^{i-1}_1 \wedge (e^i_{t_1} = 1) \wedge (r^i_{t_1} = 1) \wedge s^i_2) \vee \\
(s^{i-1}_2 \wedge (e^i_{t_2} = 1) \wedge (r^i_{t_2} = 0) \wedge s^i_2)
\end{array}
$$

Note that an equi-satisfiable linearly-sized formula in CNF can be obtain efficiently. Moreover, we can simply arrange all sub-prefixes in front of the formula

[6] Note that [FHT08] describes an alternative approach where only one existential and one randomized variable are required per step i. For the sake of clarity, we opt for the simpler one here.

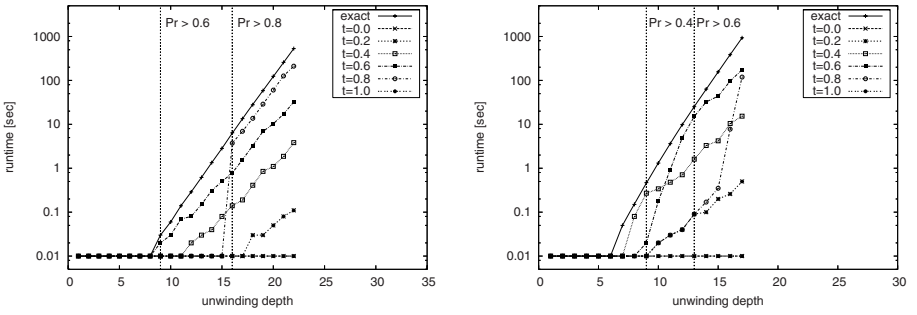


Fig. 4. Impact of thresholding for $\mathcal{H}_1$ (left) and $\mathcal{H}_2$ where $k_y = 1, c = 4$ (right)

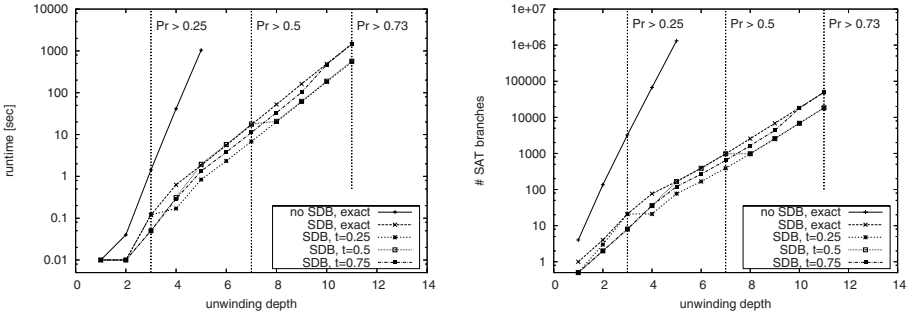


Fig. 5. Impact of solution-directed backjumping for $\mathcal{H}_3$: runtime (left) and number of found SAT branches (right)

(in ascending index-order), since all quantified variables in the transition relation for i do not occur in any other transition relation $j \neq i$. This yields an SSMT formula as required in Section 2.

For the *hybrid* case the encoding follows the same idea but we have to take account of the potentially non-linear real arithmetic *guards* of the transitions and *actions* to be performed for the probabilistic distributions. E.g., transition t_5 of $\mathcal{H}_3$ is encoded as:

$$(s_3^{i-1} \wedge (e_{t_5}^i = 1) \wedge (y_{i-1} > (x_{i-1})^2) \wedge (r_{t_5}^i = 0) \wedge s_2^i) \quad \vee$$
$$(s_3^{i-1} \wedge (e_{t_5}^i = 1) \wedge (y_{i-1} > (x_{i-1})^2) \wedge (r_{t_5}^i = 1) \wedge (x_i = x_{i-1} \cdot y_{i-1}) \wedge s_3^i)$$

where the real-valued variables x_{i-1} and y_{i-1} represent the values of the real-valued system variables x and y, resp., before step i is executed.

4.2 Experimental Results

This subsection compiles empirical results of the implemented algorithm SiSAT for the benchmarks from Subsection 4.1 encoded as SSMT formulae. The property to be checked for all automata is whether the mode s_2 can be reached.

Table 1. Empirical results for $\mathcal{H}_3$ where $k_x = 0, k_y = 2$

	exact	$t = 0.0$	$t = 0.2$	$t = 0.4$	$t = 0.6$	$t = 0.8$	$t = 1.0$
unwinding depth **1**: 38 vars + 10 quantified vars, 111 clauses							
witness value	0.1	0.006	0.0	0.0	0.0	0.0	0.0
#SATs	4	1	0	0	0	0	0
#conflicts	0	0	0	0	0	0	0
runtime (sec)	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
unwinding depth **2**: 69 vars + 20 quantified vars, 212 clauses							
witness value	0.194	0.000252	0.092944	0.0392	0.0392	0.0042	0.0
#SATs	136	1	66	24	24	8	0
#conflicts	0	0	0	0	0	0	0
runtime (sec)	0.04	< 0.01	0.02	< 0.01	< 0.01	< 0.01	< 0.01
unwinding depth **3**: 100 vars + 30 quantified vars, 313 clauses							
witness value	0.2809	1.058e-05	0.201	0.1492	0.07734	0.02022	0.0
#SATs	3,248	1	2,743	1,390	832	320	0
#conflicts	6	0	6	2	0	0	0
runtime (sec)	1.43	< 0.01	1.22	0.63	0.37	0.15	< 0.01
unwinding depth **4**: 131 vars + 40 quantified vars, 414 clauses							
witness value	0.3603	4.445e-07	> 0.2	0.242	0.1339	0.04349	0.0
#SATs	67,360	1	16,167	42,891	21,088	8,380	0
#conflicts	21	0	6	20	10	10	0
runtime (sec)	41.48	< 0.01	9.81	26.42	13.02	5.13	< 0.01
unwinding depth **5**: 162 vars + 50 quantified vars, 515 clauses							
witness value	0.4323	1.867e-08	0.2002	0.4001	0.1908	0.0844	0.0
#SATs	1,322,700	1	213,560	1,126,492	447,616	201,252	0
#conflicts	35	0	21	35	29	29	0
runtime (sec)	1,044.0	< 0.01	167.7	903.6	352.2	158.6	< 0.01

All benchmarks were performed on an 1.83 GHz Intel Core 2 Duo machine with 1 GByte physical memory running Linux. Concerning the issue of the *approximative* nature of solutions obtained by interval constraint propagation, we remark here that due to the deterministic assignments and the use of rational functions in the considered PHAs (cf. Fig. 3), we have obtained exact solutions on all benchmark runs. Hence, the computed probabilities are exact.

Concerning the performance of the SiSAT algorithm, Fig. 4 and 5 show that the runtimes dramatically grow over the BMC unwinding depths. As one can expect, the length of the quantifier prefix determines the runtimes. One acceleration technique we considered to battle against the high complexity is thresholding. Fig. 4 and Table 1 show a comparison for different thresholding parameters where *exact* means $t_l = 0$ and $t_u = 1$, and $t = k$ means $t_l = k$ and $t_u = k$. These results empirically prove the expected fact that thresholding leads to significant performance gains if the threshold parameters are *not* close to the exact maximum probability of satisfaction. Consider, e.g., the results for unwinding depth 5 of $\mathcal{H}_3$ in Table 1. The exact satisfaction probability is 0.4323. To compute this, SiSAT needed 1044 seconds, thereby visiting more than 1.3 millon satisfying branches. Setting $t_l = t_u = t = 0.4$ yields nearly the same performance while for thresholds $t < 0.4$ and $t > 0.4$ the runtimes quickly decrease. For the extreme values $t = 0$, i.e. finding just one solution, and $t = 1$, i.e. randomized quantifiers change to universal quantifiers, SiSAT terminates within fractions of a second.

While the impact of thresholding strongly depends on the pre-defined lower and upper target thresholds, solution-directed backjumping is independent from such settings but exploits the structure of the formula. Surprisingly, solution-directed backjumping yields performance gains of multiple orders of magnitude. The results for the more complex PHA $\mathcal{H}_3$ are illustrated in Fig. 5. For unwinding depth 5, the speedup factor obtained for the exact version is 567. This shows that the idea of skipping branches for which the probability remain the same actually works for our case studies. As shown on the right in Fig. 5, an enormous number of satisfying branches to be visited could be skipped when SDB was enabled. While the exact version *without* SDB was just able to solve the first 5 BMC unwindings of $\mathcal{H}_3$ within 100 minutes, the exact version *with* SDB solved 11 instances in the same time. The SSMT formula for depth 11 contains 110 quantified variables, 348 non-quantified variables, and 1121 problem clauses. Fig. 5 also indicates that on most of the BMC instances the combination of SDB and thresholding further increases the efficiency of the solver.

5 Conclusion and Future Work

In this paper, we presented an algorithm for stochastic SMT problems for non-linear arithmetic over the reals and integers together with experimental results from the reachability analysis of probabilistic hybrid automata. We showed that algorithmic enhancements like thresholding and solution-directed backjumping have a significant impact on the performance of the tool.

In future work, we will explore further techniques and heuristics to accelerate the SiSAT tool: For instance, further forms of backjumping within the quantified part of the decision tree. Another important aspect to improve the performance of search algorithms is to find suitable value and variable orderings. In the context of bounded model checking PHAs, we will work on an automatic translation of PHAs into SSMT formulae and bounded-model-checking optimizations like clause reusing and shifting [FH07]. Concerning the issue of approximate solutions, we will modify SiSAT to handle confidence intervals of probabilities instead of values s.t. we are able to obtain safe lower and upper bounds on the satisfaction probability when using also transcendental functions like sin or exp. Within the AVACS project[7], we will apply the SiSAT solver on benchmarks which deal with the impact of cooperative, distributed traffic management on flow of road traffic. These benchmarks are representative for a large number of hard scheduling and allocation problems and naturally show uncertain behavior.

Acknowledgements

The authors would like to thank Christian Herde, Holger Hermanns, Ralf Wimmer, Joost-Pieter Katoen, and Stephen Majercik for valuable discussions on SMT, probabilistic systems, and stochastic SAT algorithms. Furthermore, the authors are very grateful to the anonymous reviewers for their helpful comments.

[7] www.avacs.org

References

[BG06] Benhamou, F., Granvilliers, L.: Continuous and interval constraints. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming. Foundations of Artificial Intelligence, pp. 571–603. Elsevier, Amsterdam (2006)

[BS06] Balafoutis, T., Stergiou, K.: Algorithms for Stochastic CSPs. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 44–58. Springer, Heidelberg (2006)

[BS07] Bordeaux, L., Samulowitz, H.: On the stochastic constraint satisfaction framework. In: SAC, pp. 316–320. ACM, New York (2007)

[DLL62] Davis, M., Logemann, G., Loveland, D.: A Machine Program for Theorem Proving. CACM 5, 394–397 (1962)

[DP60] Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of the ACM 7(3), 201–215 (1960)

[FH07] Fränzle, M., Herde, C.: HySAT: An Efficient Proof Engine for Bounded Model Checking of Hybrid Systems. FMSD 30, 179–198 (2007)

[FHT+07] Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient Solving of Large Non-linear Arithmetic Constraint Systems with Complex Boolean Structure. JSAT Special Issue on SAT/CP Integration 1, 209–236 (2007)

[FHT08] Fränzle, M., Hermanns, H., Teige, T.: Stochastic Satisfiability Modulo Theory: A Novel Technique for the Analysis of Probabilistic Hybrid Systems. In: Proceedings of the 11th International Conference on Hybrid Systems: Computation and Control (HSCC 2008) (2008)

[GNT03] Giunchiglia, E., Narizzano, M., Tacchella, A.: Backjumping for quantified Boolean logic satisfiability. Artif. Intell. 145(1-2), 99–120 (2003)

[Lit99] Littman, M.L.: Initial Experiments in Stochastic Satisfiability. In: Proc.of the 16th National Conference on Artificial Intelligence, pp. 667–672 (1999)

[LMP01] Littman, M.L., Majercik, S.M., Pitassi, T.: Stochastic Boolean Satisfiability. Journal of Automated Reasoning 27(3), 251–296 (2001)

[Maj04] Majercik, S.M.: Nonchronological backtracking in stochastic Boolean satisfiability. Ictai 00, 498–507 (2004)

[ML98] Majercik, S.M., Littman, M.L.: MAXPLAN: A New Approach to Probabilistic Planning. Artificial Intelligence Planning Systems, pp. 86–93 (1998)

[ML03] Majercik, S.M., Littman, M.L.: Contingent Planning Under Uncertainty via Stochastic Satisfiability. Artificial Intelligence Special Issue on Planning With Uncertainty and Incomplete Information 147(1-2), 119–162 (2003)

[Pap85] Papadimitriou, C.H.: Games against nature. J. Comput. Syst. Sci. 31(2), 288–301 (1985)

[RT06] Ranise, S., Tinelli, C.: Satisfiability modulo theories. IEEE Intelligent Systems 21(6) (2006)

[THF+07] Teige, T., Herde, C., Fränzle, M., Kalinnik, N., Eggers, A.: A Generalized Two-watched-literal Scheme in a mixed Boolean and Non-linear Arithmetic Constraint Solver. In: Neves, J., Santos, M.F., Machado, J.M. (eds.) EPIA 2007. LNCS (LNAI), vol. 4874, pp. 729–741. Springer, Heidelberg (2007)

[TMW06] Tarim, A., Manandhar, S., Walsh, T.: Stochastic constraint programming: A scenario-based approach. Constraints 11(1), 53–80 (2006)

[Wal02] Walsh, T.: Stochastic constraint programming. In: Proc. of the 15th European Conference on Artificial Intelligence (ECAI 2002), IOS Press, Amsterdam (2002)

A Hybrid Constraint Programming / Local Search Approach to the Job-Shop Scheduling Problem

Jean-Paul Watson[1] and J. Christopher Beck[2]

[1] Discrete Math and Complex Systems Department,
Sandia National Laboratories,
Albuquerque, New Mexico, USA
jwatson@sandia.gov
[2] Department of Mechanical and Industrial Engineering,
University of Toronto, Toronto, Ontario, Canada
jcb@mie.utoronto.ca

Abstract. Since their introduction, local search algorithms – and in particular tabu search algorithms – have consistently represented the state-of-the-art in solution techniques for the classical job-shop scheduling problem. This is despite the availability of powerful search and inference techniques for scheduling problems developed by the constraint programming community. In this paper, we introduce a simple hybrid algorithm for job-shop scheduling that leverages both the fast, broad search capabilities of modern tabu search and the scheduling-specific inference capabilities of constraint programming. The hybrid algorithm significantly improves the performance of a state-of-the-art tabu search for the job-shop problem, and represents the first instance in which a constraint programming algorithm obtains performance competitive with the best local search algorithms. Further, the variability in solution quality obtained by the hybrid is significantly lower than that of pure local search algorithms. As an illustrative example, we identify twelve new best-known solutions on Taillard's widely studied benchmark problems.

1 Introduction

Local search algorithms for the traditional makespan-minimization formulation of the job-shop scheduling problem (JSP) have dominated the state-of-the-art for at least the past 15 years. These include Nowicki and Smutnicki's landmark TSAB tabu search algorithm [13], Balas and Vazacopoulos' guided local search algorithm [1], Nowicki and Smutnicki's follow-on i-TSAB tabu search algorithm [14], and most recently Zhang et al.'s hybrid tabu search / simulated annealing algorithm [25]. These algorithms are all built upon a foundation of one or more powerful, problem-specific move operators, which are able to efficiently identify promising feasible and high-quality solutions in the neighborhood of a given solution. Metaheuristic search strategies then leverage these move operators to perform global search for minimal-cost solutions; the complexity of these strategies ranges from simple tabu search (in the case of TSAB) to highly intricate hybridizations of tabu search, path relinking, and elite pool maintenance schemes (in the case of i-TSAB).

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 263–277, 2008.

On established benchmark problems [19,20], the progression of local search algorithms has consistently established new upper bounds over time, perhaps leading one to question the utility of further research in the area. We address such criticism with the following observations. First, on the most difficult benchmark instances, there is no indication that the upper bounds are necessarily close to the optimal solutions, i.e., there likely remains significant room for performance improvements. Second, although the aforementioned local search algorithms collectively have established the best-known solutions to benchmark instances, no single algorithm can consistently generate these solutions. Consequently, improved algorithms yielding reductions in performance variability are desirable. Third, proportionally little research is dedicated to understanding why local search algorithms are so effective on the job-shop scheduling problem; developing such knowledge is foundational to consistently achieving high-performance in other problem domains.

Perhaps somewhat paradoxically, constraint programming (CP) algorithms are more commonly used than their local search counterparts to obtain solutions to real-world scheduling problems, e.g., using ILOG's Scheduler software library [17]. This is widely attributed to a combination of the ability to easily incorporate various idiosyncratic "side" constraints that are pervasive in real-world scheduling problems (such constraints can require significant redesign of local search algorithms) and to effectively deduce, via powerful domain-specific constraint propagators, the implications of various scheduling decisions. However, despite the level of research effort dedicated to the development of scheduling-specific constraint propagation and search techniques (e.g., see [2,4]), the performance of CP algorithms on the traditional JSP has significantly lagged that of their local search counterparts. To date, the strongest CP-based algorithm is solution-guided multi-point constructive search [3], although the performance of even this algorithm lags that of modern tabu search algorithms for the JSP [9] in terms of both time and final solution quality.

Hybridization of local search and CP on JSPs without side constraints does not, therefore, immediately appear to be a promising research direction. However, the following two unexplored aspects of these algorithms provide what we feel to be contrary evidence, and motivate the line of research developed in this paper:

- The strong propagation techniques in CP are more efficient in constrained search states. That is, the polynomial time inference algorithms are more likely to be able to find implied constraints, and to reduce the search space, in states that are already highly constrained. When a good solution has been found, strong "back-propagation" from the upper bound on the makespan results in such a highly constrained search state. Therefore, we conjecture that while CP is unable to competitively find good solutions, once given a good solution, it may be able to improve on it more quickly than a local search approach.
- A popular conceptualization of the power of modern local search algorithms is that they balance intensification with diversification [9]. Intensification, which can loosely be understood as searching "near" an existing good solution, is often implemented by repeatedly restarting search from a good solution that has been found earlier. Diversification, in contrast, tries to distribute the search effort in unexplored areas of the search space. It is often implemented by maintaining a varied set of

promising solutions and combining them in a variety of ways, such as via path relinking [7]. However, modern tabu search algorithms seem to do a relatively poor job of intensification. Watson [22] showed that a relatively small number of iterations after restarting from a good solution, tabu search is a considerable distance from the starting solution. Further, *a posteriori* analysis of algorithmic traces indicates that tabu search often fails to locate high-quality solutions that are quite close to high-quality solutions located by tabu search [22]. In contrast, solution-guided constructive search performs a much more focused search around its guiding solution [3]. Therefore, we conjecture that improved performance may result from using CP to strongly intensifying around a diverse set of high-quality solutions generated by tabu search.

The remainder of this paper is organized as follows. We begin in Section 2 with a brief discussion of the job-shop scheduling problem and the benchmark instances used in our analysis. The foundational algorithms for our hybrid approach – iterated simple tabu search (i-STS) and solution-guided multi-point constructive search (SGMPCS) – and our simple hybrid are described in Section 3. Our experimental methodology is introduced in Section 4, followed by a description of empirical performance results in Section 5. Section 6 details some implications of our results, followed by our conclusions in Section 7.

2 Problem Description and Benchmark Instances

We consider the well-known $n \times m$ static, deterministic JSP in which n jobs must be processed exactly once on each of m machines [5]. Each job i ($1 \leq i \leq n$) is routed through each of the m machines in a pre-defined order π_i, where $\pi_i(j)$ denotes the jth machine ($1 \leq j \leq m$) in the routing order of job i. The processing of job i on machine $\pi_i(j)$ is denoted o_{ij} and is called an operation. An operation o_{ij} must be processed on machine $\pi_i(j)$ for an integral duration $\tau_{ij} > 0$. Once initiated, processing cannot be preempted and concurrency on individual machines is not allowed, i.e., the machines are unit-capacity resources. For $2 \leq j \leq m$, o_{ij} cannot begin processing until $o_{i(j-1)}$ has completed processing. The scheduling objective is to minimize the makespan C_{max}, i.e., the maximal completion time of the last operation of any job. Makespan-minimization for the JSP is NP-hard for $m \geq 2$ and $n \geq 3$ [6].

An instance of the $n \times m$ JSP is uniquely defined by the set of nm operation durations τ_{ij} and n job routing orders π_i. In nearly all benchmark instances, the τ_{ij} are uniformly sampled from the interval $[1, 99]$, while the π_i are given by random permutations of the integer sequence $1, \ldots, m$. As discussed in Section 4, our experimental results are generated using a subset of Taillard's benchmark instances, specifically those labeled `ta11` through `ta50` [19]. This subset contains 10 instances of each of the following problem sizes: 20×15, 20×20, 30×15, and 30×20. We have selected these instances because they are widely studied (all competitive algorithms introduced since 1995 have been tested on these instances), are known to be very challenging (even state-of-the-art algorithms fail to consistently find solutions with makespans equal to the best known solutions), and there remains "headroom" for improvement in best-known makespans (as illustrated by the often large gap between those values and the best-known lower

bounds). For these same reasons, we ignore the easier instances in Taillard's problem suite, in addition to many historical instances (e.g., the "ft", "la", and "orb" instances) for which modern JSP algorithms can consistently locate optimal solutions.

3 Algorithms

In this section, we discuss the two foundational algorithms in detail before presenting the simple hybridization we investigate in this paper.

3.1 Iterated Simple Tabu Search

Beginning with an early approach by Taillard [21], tabu search algorithms have consistently represented the state-of-the-art in obtaining high-quality solutions for the JSP. A variety of researchers have introduced tabu search algorithms of increasing effectiveness and complexity. Specific algorithmic advances of note in this progression include the introduction of (1) the highly restrictive $N5$ critical path-based move operator [13], (2) search intensification mechanisms in conjunction with sets of "elite" or high-quality solutions [13], and (3) search diversification mechanisms in the form of path relinking [14]. These techniques are simultaneously embodied in Nowicki and Smutnicki's i-TSAB algorithm, which has represented the state-of-the-art since 2003. With the exception noted below, the sole competitor is a hybrid tabu search / simulated annealing algorithm introduced by Zhang et al. [25]. The Zhang et al. algorithm uses simulated annealing to generate an initial set of elite solutions, which are then processed via tabu search-driven intensification. The primary differences between the Zhang et al. algorithm and i-TSAB are the lack of an explicit diversification mechanism (path relinking is used in i-TSAB) and the use of the $N6$ move operator introduced by Balas and Vazacopoulos [1] (i-TSAB employs the $N5$ move operator).

Although remarkably effective, i-TSAB is an extremely intricate and complex algorithm. Such complexity is a significant drawback to researchers, as in practice it impedes reproducibility, adoption, and subsequent study. In the specific case of i-TSAB, the intricacy makes it difficult to assess the contribution of the various algorithmic components to its overall performance. Toward this goal, we previously introduced a simplified version of i-TSAB, which we denote iterated simple tabu search, or i-STS [9]. As discussed below, i-STS contains the key algorithmic ingredients of i-TSAB while reducing the overall complexity and maintaining near-equivalent performance.

A basic tabu search lies at the core of i-STS, built around the $N5$ move operator. Short-term memory is used to prevent inversion of recently swapped pairs of adjacent operations on a critical path. Following [21], the tabu tenure is periodically and randomly sampled from a fixed interval $[L, U]$. Search in i-STS proceeds in two phases. In the first phase, the basic tabu search algorithm is executed for a small, fixed number of iterations from each of $|E|$ different random initial solutions. The best solution from each iteration-limited run is saved, and forms the initial set E of elite solutions.

In the second phase of i-STS, the elite solutions, E, are iteratively processed by both intensification and diversification mechanisms, each selected at any given iteration with respective probabilities p_i and p_d, $p_i + p_d = 1$. To perform search intensification, a

single elite solution $e \in E$ is selected at random and an iteration-limited tabu search is executed from e. Due to tie-breaking during move selection, facilitated by the pervasiveness of plateaus of equally fit neighboring solutions in the JSP [23], different trajectories can locate solutions of variable quality. If a solution e' with a lower makespan than e is located, e' replaces e in E. To perform diversification, two elite solutions $e_1, e_2 \in E$ are selected at random. Path relinking is then performed to generate a solution e' that is approximately equi-distant from both e_1 and e_2. Iteration-limited tabu search is then executed from e', as is performed in the intensification process. If a solution e'' is identified with a lower makespan than e_1, then e'' replaces e_1 in E. The second phase of i-STS continues until an aggregate number of basic tabu search iterations M have been executed, with the best solution $e \in E$ returned upon completion.

3.2 Solution-Guided Multi-point Constructive Search

Solution-guided multi-point constructive search (SGMPCS) is a recently proposed algorithm that combines constructive tree search, randomized restart, and heuristic guidance from good solutions found earlier in the search [3]. The basic approach is a CP tree search with a limit on the number of dead-ends ("fails") that are encountered before restarting. Each tree search is guided by using an existing sub-optimal solution as a value-ordering heuristic. Once a variable to be assigned has been chosen (see below), the value chosen is the one in the guiding solution, provided that value is still in the domain of the chosen variable. Otherwise, any other value-ordering heuristic may be used. As in i-STS, a small set of "elite" solutions is maintained, one of which is chosen with uniform probability to guide a given tree search. When a tree search exhausts its fail limit, it returns the best solution it has found (if any). That solution, if it exists, replaces the guiding solution in the elite pool.

Beck [3] showed that SGMPCS has strong, but not state-of-the-art, performance on job shop scheduling, makespan minimization problems. While finding significantly better solutions than chronological backtracking and randomized restart (using the same propagators, heuristics, and, in the latter case, fail limit sequences), SGMPCS was not able to perform as well as i-STS.

Details. A simplified version of SGMPCS is used in this paper. This version fixes a number of the parameters in the full algorithm. As the version presented here is a particular parametrization of the full version, we continue to refer to it as SGMPCS. Readers interested in the full version are referred to Beck [3].

Pseudocode for SGMPCS is shown in Algorithm 1. The algorithm initializes a set, E, of elite solutions and then enters a while-loop. In each iteration, a chronological backtracking search is guided with a randomly selected elite solution (line 6). If a solution, s, is found during the search, it replaces the starting elite solution, r. Each individual search is limited by a fail bound: a maximum number of fails that can be incurred. The entire process ends when the problem is solved, proved insoluble within one of the tree searches, or when some overall bound on the computational resources (e.g., CPU time or number of fails) is reached.

More formally, a search tree is created by asserting a series of choice points of the form: $\langle V_i = x \rangle \vee \langle V_i \neq x \rangle$, where V_i is a variable and x is the value assigned to V_i.

Algorithm 1. SGMPCS: Solution-Guided Multi-Point Constructive Search

```
SGMPCS():
1 initialize elite solution set E
2 while not solved and termination criteria unmet do
3     r := randomly chosen element of E
4     set upper bound on cost function
5     set fail bound, b
6     s := search(r, b)
7     if s is better than r then
8         replace r with s
9 return best(E)
```

SGMPCS can use any variable-ordering heuristic to choose the variable to assign. The choice point is formed using the value assigned in the guiding solution or, if the value in the guiding solution is inconsistent, a heuristically chosen value. Let a guiding solution, r, be a set of variable assignments, $\{\langle V_1 = x_1\rangle, \langle V_2 = x_2\rangle, \ldots, \langle V_m = x_m\rangle\}, m \leq n$, where n is the number of decision variables. Let $dom(V_i)$ be the set of possible values (i.e., the *domain*) of variable V_i. The variable-ordering heuristic has complete freedom to choose a variable, V_i, to be assigned. If $x_i \in dom(V_i)$, where $\langle V_i = x_i\rangle \in r$, the choice point is made with $x = x_i$. Otherwise, if $x_i \notin dom(V_i)$, any value-ordering heuristic can be used to choose $x \in dom(V_i)$.

At line 4 in the pseudocode, an upper bound is placed on the cost function for the subsequent search. We use two different methods in our experiments. The *local upper bound* is one less than cost of the guiding solution (i.e., cost(r)-1). The *global upper bound* is one less than the best solution that has been found (i.e., cost(best(E))-1). Intuitively, the local upper bound allows a more heuristic search since local improvements will be accepted into the elite set while the global upper bound tries to maximize the impact of the inference algorithms as it always searches in the most constrained space possible.

Given a large enough fail limit (line 5), an individual search can exhaust the search space. Therefore, completeness depends on the policy for setting the fail limit. In our experiments, we will use two fail sequence polices: Fixed and Luby. The Fixed limit simply uses a constant fail-limit for each search. Obviously, such a policy is not complete. The Luby limit is an evolving sequence that has been shown to be the optimal sequence for satisfaction problems under the condition of no knowledge about the solution distribution [12]. The sequence is as follows: 1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, ... That is, the fail limit for the first and second searches is 1, for the third search is 2, and so on. Following [24] and our own preliminary experimentation, we multiply the elements of the sequence by a fixed constant. As the sequence increases without limit, a single search will eventually have a fail limit that is sufficient to search the entire search space and therefore the overall algorithm using the Luby fail limit is complete.

SGMPCS is a general framework for constructive tree search. To apply SGMPCS to the JSP, solutions are encoded using the well known disjunctive graph representation.

Texture-based heuristics [4] are used to identify a machine and time point with maximum contention among the operations and to then choose a pair of unordered operations. The heuristic is randomized by specifying that the ⟨machine, time point⟩ pair is chosen with uniform probability from the top 10% most critical pairs. The ordering found in the guiding solution is asserted. Note that because the decisions are binary, the pair in the solution must be locally consistent, otherwise the pair of operations would already be sequenced in the opposite order. The standard constraint propagation techniques for scheduling [15, 10, 11] (available via the ILOG Scheduler library) are also used.

3.3 A Very Simple Hybrid Approach

Given the complexities of the two foundational algorithms, our approach to hybridization in this paper is the most concise that we could envision. We begin by executing i-STS for a fixed number of total iterations of the underlying tabu search algorithm. The intent of this phase is to quickly generate a set of high-quality solutions. At the end of these iterations, the best $|E|$ solutions that have been found are used as the initial elite set for SGMPCS (line 1 of the SGMPCS pseudocode). SGMPCS is then run for a fixed, comparatively larger (in contrast to i-STS) CPU time and the best solution found is returned.

There is clearly much more we could do. However, this simple (and perhaps, simple-minded) hybrid directly and concisely addresses the two motivations we had for this research, as described in Section 1.

4 Experimental Methodology

Our analysis is based on multiple runs of the hybrid on each of the Taillard benchmark instances we consider. Each run consists of executing i-STS for 5M iterations, which requires a few minutes of CPU time, depending on the problem size. For each run, we use an elite pool size of 8, and $p_i = p_d = 0.5$; these parameter settings were chosen based on prior empirical studies of both i-TSAB [14] and i-STS [9]. All remaining free parameters of i-STS are set identically to that reported in [9]. At the end of 5M iterations, the best $|E|$ elite solutions are used as the starting elite set for SGMPCS. SGMPCS is then run for 30 CPU minutes and reports the best solution found. Each problem instance is run 10 times independently for a given parameter configuration. i-STS is implemented in C++ while SGMPCS uses ILOG Scheduler 6.5 (also in C++). All code was compiled using the GNU gcc compiler. Experiments were executed on a cluster containing 2GHz Dual Core AMD Opteron 270 nodes, each with 2GB RAM running Red Hat Enterprise Linux 4.

Given an experimental setup for a problem instance (10 runs per instance, fixing the behavior of i-STS), we next consider the various configurations of the remaining free SGMPCS parameters. Following [3], we perform a full factor experimental design, considering: (1) $|E| \in \{1, 4, 8\}$ (for SGMPCS; in i-STS, $|E|$ is fixed to 8), (2) local and global upper bounds, and (3) a fixed fail limit of 500, in addition to scaled Luby limits with parameters 100 and 200. We observe that when $|E| = 1$, local and global upper

bounding strategies are equivalent. Consequently, we execute a set of 10 runs on each of Taillard's instances under fifteen distinct configurations of SGMPCS.

5 Results

Our analysis is broken into four components: parameter sensitivity (Section 5.1), performance relative to state-of-the-art algorithms for the JSP (Section 5.2), best-known upper bounds (Section 5.3), and proving optimality (Section 5.4).

5.1 Parameter Sensitivity

We first analyze the performance of the various SGMPCS parameterizations relative to one another, in an effort to determine parameter sensitivity and a sole candidate for comparison of the hybrid algorithm with other state-of-the-art JSP algorithms. We performed a two-way (factorial) ANOVA on the resulting data.[1] The three independent variables are elite pool size, fail limit, and upper bound method, while the sole dependent variable is the makespan of the best solution obtained. The outcome of this experiment indicated that there were *no* significant main *or* interaction effects between the SGMPCS parameters and the quality of the final solution obtained. The largest p value obtained was for $|E|$, and was equal to 0.499. This result is in direct contrast to [3], in which all main factors and interactions were statistically significant. The sole difference in the experimental designs is the mechanism used to initialize the elite solution set for SGMPCS (i-STS versus random solutions). It appears that while different parameterizations of SGMPCS influence the degree to which the algorithm can improve upon random initial solutions, this sensitivity disappears once solution quality is "sufficiently" good, e.g., as is the case for i-STS solutions.

Next, we examine the absolute difference in performance between the various algorithm configurations. For a given problem instance and solution makespan M, we define the relative error as $RE = (M - LB)/LB * 100$, where *LB* is the largest known lower bound for the instance. For our analysis, we take *LB* from [20], where such values have been recorded by Taillard since the introduction of these problem instances. Consider a specific parameterization of SGMPCS in our hybrid algorithm, and one of the following statistics defined over the set of 10 runs on a given problem instance: best makespan, average makespan, and worst makespan. The mean relative error, or MRE, for the given parameterization and makespan statistic is then computed simply as the mean *RE* taken over the 40 problem instances.

The resulting MRE statistics for our hybrid algorithm are shown in Table 1. Boldfaced entries indicate that the corresponding SGMPCS parameterization yielded the best performance with respect to the given makespan statistic. We exclude results for parameterizations with the Luby 100 fail limit, as they generally, though marginally, under-perform the respective Luby 200 fail limit strategy. Consistent with the two-way ANOVA analysis, the differences in all makespan statistics are minimal in absolute terms, varying at most by 0.20% in any given column. Although no parameterization stands out as a winner, the parameterization with $|E| = 8$, a fixed fail limit of 500,

[1] All statistical analyses were performed using the publicly available R software package.

Table 1. Mean relative error (MRE) statistics for the i-STS / SGMPCS hybrid on Taillard's benchmark instances for various parameter configurations. Bold-faced entries in an MRE column indicate the configuration obtaining the best performance.

$\|E\|$	Fail Limit	Upper Bound	Best MRE	Mean MRE	Worst MRE
1	Fixed 500	Local/Global	3.146	3.490	3.897
1	Luby 200	Local/Global	3.178	3.502	3.886
4	Fixed 500	Local	3.134	3.392	3.705
4	Luby 200	Local	3.143	3.424	3.718
4	Fixed 500	Global	3.129	3.408	3.726
4	Luby 200	Global	3.136	3.425	3.746
8	Fixed 500	Local	3.123	**3.369**	3.706
8	Luby 200	Local	3.142	3.397	**3.691**
8	Fixed 500	Global	**3.103**	3.400	3.709
8	Luby 200	Global	3.148	3.409	3.694

Table 2. MRE statistics for i-TSAB, Zhang et al.'s hybrid tabu search / simulated annealing algorithm, and our hybrid i-STS / SGMPCS algorithm, on Taillard's benchmark instances

Instance Group	Best Known	i-TSAB	Zhang		Hybrid		
			Best	Mean	Best	Mean	Worst
ta11-20	2.29	2.81	2.37	2.92	2.26	2.45	2.83
ta21-30	5.38	5.68	5.44	5.97	5.52	5.71	6.00
ta31-40	0.46	0.78	0.55	0.93	0.50	0.68	0.85
ta41-50	4.02	4.70	4.07	4.84	4.22	4.63	5.15
Overall	3.04	3.49	3.11	3.67	3.13	3.37	3.71

and the local upper bound obtained the most consistent performance, as measured in terms of average makespan of solutions obtained. For this reason, we emphasize this parameterization of SGMPCS in subsequent analyses.

5.2 Performance Relative to the State-of-the-Art

Having established the relative insensitivity of our hybrid algorithm performance to SGMPCS parameter settings, we now analyze performance relative to state-of-the-art search algorithms for the JSP. We select two baselines for comparison: Nowicki and Smutnicki's i-TSAB tabu search algorithm [14] and Zhang et al.'s hybrid tabu search / simulated annealing algorithm [25]. The i-TSAB algorithm represents the state-of-the-art from 2003 onwards, while Zhang et al.'s algorithm is a recently introduced competitor. A single "winner" is not easily determined, lacking carefully controlled experiments and availability of the source code of the two algorithms. However, it is clear from published MRE performance analysis that these two algorithms are superior to all predecessors. Finally, we do not compare the performance of our hybrid with that of previously published CP algorithms, e.g., [16], as those algorithms have not historically proved competitive on the standard JSP; to the best of our knowledge, SGMPCS is the best-performing pure CP algorithm for the JSP, as reported in [3].

As indicated previously in Section 5.1, we consider the performance of our hybrid algorithm obtained with the best overall *mean* performance, obtained with $|E| = 8$, a fixed fail limit of 500, and the local upper bound. While it may be argued that the comparison should be based on the mean MRE obtained across all parameter settings, Nowicki and Smutnicki document significant parameter tuning in the development of i-TSAB, and Zhang et al. undoubtedly performed similar experimentation, although it is not explicitly documented in [25]. The MRE performance statistics for the two comparative baselines and our hybrid algorithm are shown in Table 2; in addition, we compute the MRE for the best-known solutions recorded in [20] as of November 30, 2007. Unfortunately, Nowicki and Smutnicki [14] only report results for a single run of i-TSAB, complicating interpretation. Absent a rigorous alternative, we treat the corresponding MRE results as representative of mean i-TSAB performance. The Zhang et al. statistics are taken over 10 independent runs of their algorithm on each problem instance. Without the actual sample populations, it is not possible to make statistical inferences regarding the relative superiority of the Zhang et al. algorithm and our hybrid algorithm. Consequently, we proceed with a qualitative analysis.

First, we compare the performance of our hybrid with that of i-TSAB. On all but the ta21-30 problem group, the hybrid outperforms i-TSAB in terms of mean MRE. Overall, the hybrid outperforms i-TSAB by 0.12% in terms of mean MRE; again, we are treating the individual i-TSAB samples as representative of mean performance. While the percentage advantage is small in absolute terms, we observe that due to the difficulty of these instances, apparently small differences have historically differentiated state-of-the-art algorithms from second-tier competitors. Although we cannot rigorously determine whether our hybrid performance dominates that of i-TSAB, it is clear that the performance is, *at a minimum*, indistinguishable.

Next, we compare the performance of our hybrid with that of Zhang et al.'s algorithm, hereafter referred to simply as Zhang's algorithm. In terms of mean MRE, the hybrid algorithm dominates the Zhang algorithm both overall and in each problem subgroup; overall, the advantage is 0.30%. In terms of best MRE, each algorithm dominates on two of the four problem groups, with Zhang holding a slight 0.02% advantage overall. Of particular interest is the excellent worst MRE performance of our hybrid algorithm. On two of the problem groups, the worst MRE of the hybrid is better than the *mean* MRE of the Zhang algorithm. Overall, the hybrid worst MRE performance is only slightly worse that the Zhang mean MRE performance, with a difference of only 0.04%. Clearly, a significant advantage of our hybrid algorithm is the consistency of the state-of-the-art performance, which is often elusive (e.g., in the case of the Zhang algorithm) on very difficult benchmark problems.

A major issue in comparative assessment of state-of-the-art algorithms for the JSP involves quantification of computational effort. In addition to issues involving the use of disparate computing hardware, software engineering decisions and coding skill make such comparisons notoriously problematic. We do not address these issues here. Rather, we observe that from analyses of published performance reports [14,25], all three test algorithms were executed on modern computing hardware (Pentium III or greater) and the allocated run-times on the larger problem instances were all within a factor of three.

Table 3. The makespan of new best-known solutions identified by the hybrid i-STS / SGMPCS algorithm for Taillard's benchmark problems

Instance	Prev. Best-Known	New Best-Known	Instance	Prev. Best-Known	New Best-Known
ta11	1359	1357	ta19	1335	1332
ta21	1644	1643	ta24	1646	1645
ta32	1795	1794	ta34	1829	1828
ta40	1674	1671	ta41	2018	2006
ta46	2015	2011	ta47	1903	1899
ta49	1967	1966	ta50	1926	1924

Finally, an obvious question is: Does our hybrid outperform the basic i-STS algorithm? In other words, does the premature termination of i-STS followed by SGMPCS outperform the full-length i-STS algorithm. Drawing from our previously analysis of i-STS [9], the best and mean MREs of i-STS are respectively 3.30% and 3.55%. From Table 2, this represents an under-performance of of 0.17% and 0.18% for best and worst MRE, respectively, relative to our hybrid algorithm. Such large differences provide very strong evidence that our hybrid algorithm significantly outperforms the original i-STS baseline, as is confirmed by subsequent non-parametric two-sample tests.

5.3 Best-Known Upper Bounds

Given the strong performance of our hybrid i-STS / SGMPCS algorithm, it is worth noting that various runs, under various parameterizations of SGMPCS, yielded a remarkable twelve new best-known solutions to Taillard's benchmark instances. Although our main research goal is not to enter "horse-race" competitions of the type that are particularly common in Operations Research [8], the ability of an algorithm to establish new best-known solutions in a given domain is a common (albeit heuristic, because it fails to account for factors such as run-time, coding ability, machine, and related factors) benchmark for establishing the state-of-the-art in performance. At the very least, the ability of an algorithm to establish new best-known solutions with reasonable computing effort provides strong evidence of general effectiveness. Consequently, we record both the previous and our newly obtained best-known solutions to Taillard's benchmark instances in Table 3. Of particular note is the ability of our algorithm to establish new best-knowns for five of the ten 30×20 instances, which are among the most difficult JSP benchmarks – especially given that we did not scale allocated CPU time in proportion to problem instance size.

For the single parameterization of SGMPCS used in Table 2 ($|E| = 8$, Fixed 500, local upper bound), Table 4 indicates the number of problem instances in each group for which the best solution found by the parameterization over its 10 runs is better than, equals, or is worse than the best-known solutions. The best-known solutions are the lowest makespans in Taillard's table [20] and Zhang et al's results [25]. As can be observed, this single parameterization of the hybrid algorithm is able to meet or improve upon the current best known solutions in 33 of the 40 instances. This is an impressive result given that the best-known solutions are the best solutions found by a wide variety of algorithms rather than those of a single algorithm.

Table 4. The number of instances in each group for which the best solution found by the hybrid (with parameters $|E| = 8$, Fixed 500, local upper bound) is better than, equal to, or worse than the current best known. Each instance group contains 10 instances.

Instance Group	# New Best	# Equal Best Known	# Worse
ta11-20	0	10	0
ta21-30	1	6	3
ta31-40	3	7	0
ta41-50	5	1	4
Overall	9	24	7

Table 5. The number of runs (out of 10) for which the hybrid algorithm found and proved the optimal solution. Note that no use is made of existing lower bounds for these proofs.

Hybrid Parameters	ta14	ta31	ta34	ta35	ta36	ta38	ta39
$\|E\| = 8$, Fixed 500, Local	10	10	1	0	10	4	10
$\|E\| = 8$, Luby 200, Local	10	10	1	1	10	2	9

5.4 On Proving Optimality

Unlike previous state-of-the-art algorithms for JSP, our hybrid (when using the Luby bound) is a complete algorithm. It is therefore possible to find and prove an optimal solution directly rather than based on previously known lower bounds. Even when using a fixed fail limit, it may be possible to find a proof of optimality on some instances.

Table 5 displays the number of runs (out of 10) for which two parameterizations of SGMPCS were able to find and prove optimal solutions. That is, in each case, an individual tree search exhausted the search space without reaching its fail limit. The other parameterizations had similar performance. Again, we observe relatively consistent performance in proving optimality across different runs of the same parameterization and instance. Of particular note is ta34 as, according to [20], this is the first time that the optimal solution for this problem has been found and proved.

6 Discussion

Our strong empirical results are somewhat surprising given the very simple nature of the hybrid algorithm. We have achieved state-of-the-art results by running two strong, but not necessarily state-of-the-art, algorithms in sequence, using the best solutions found by the first algorithm to initialize the second. While the underlying algorithms are complex, the effort to hybridize them consisted almost entirely of writing a translation between the solution representations of the two algorithms.

As noted in Section 1, this work was motivated by non-formalized ideas about differences in the search styles of the two foundational algorithms. While our results are positive, it is important to note that this paper *does not test* these ideas. The ideas need to be examined through careful formalization and experimental design. It is possible that

there are other underlying explanations of our results, unrelated to these motivations. More rigorous testing of these ideas will be the focus on follow-on research.

There are a number of other interesting observations arising from this study, which we address in the balance of this section.

The Performance of $|E| = 1$*:* The strong performance of the hybrid with $|E| = 1$ is quite surprising, though it is consistent with previous SGMPCS results [3]. When $|E| = 1$, the best solution found by i-STS is used as the only elite solution for SGMPCS. We would expect that such an undiversified search would result in high variance: if we were unlucky, the elite solution would not be in the vicinity of a better solution. However, the results in Table 1 show that the worst MRE for $|E| = 1$ is not significantly worse than that for the other values of $|E|$. We believe that for an explanation we will need to understand more about both the distribution of solutions in the JSP search space and the behavior of SGMPCS in searching that space.

The Impact of Solution Guidance: The main innovation in SGMPCS is the use of elite solutions to guide constructive search. To evaluate the importance of this aspect of the algorithm, we also ran the hybrid but replaced SGMPCS with chronological backtracking and randomized restart. In both cases, we ran i-STS for 5M iterations, as above. In the case of chronological backtracking, one less than the cost of the best solution found by i-STS was used as the upper bound on the cost function and the same randomized heuristics and propagators described above were used. The only differences are that the value ordering was always determined by the min-slack heuristic [18] and there was no restarting of the search (i.e., the fail limit was infinite). For randomized restart, the upper bound on the cost function, the propagators, and the heuristics were identical to that of chronological backtracking. We experimented with the same three fail limit sequences used for SGMPCS.

Overall, performance was poor. Over the 400 runs of chronological backtracking (10 runs per instance), an improvement over the i-STS starting solution was found in only one run. The improvement reduced the makespan by one time-unit. Similarly, over the 1200 runs of randomized restart (10 runs by 3 fail limits by 40 instances), an improvement was only found in 6 runs. Again each of these improvements only reduced the makespan by one time-unit.

The randomized restart results are particularly interesting because the only difference with SGMPCS (with global upper bound) is the value-ordering heuristic. We conclude, therefore, that elite solution guidance is *a* critical component of the hybrid.

7 Conclusions and Future Research Directions

Historically, the performance of constraint programming approaches – despite the availability of strong, domain-specific propagation and heuristic search techniques – has lagged that of local search algorithms on the classical job-shop scheduling problem. We introduced a simple hybrid algorithm that leverages the broad search capabilities of a high-performance tabu search algorithm for the JSP (i-STS) with the domain-specific inference capabilities of the state-of-the-art constraint programming algorithm for the

JSP (SGMPCS). The performance of the hybrid algorithm is at least competitive with the two state-of-the-art algorithms for the JSP: Nowicki and Smutnicki's i-TSAB tabu search algorithm and Zhang et al.'s hybrid tabu search / simulated annealing algorithm. While various factors outside our immediate control prevent us from making a more rigorous and precise statement regarding relative performance, we additionally observe that our hybrid algorithm was able to locate 12 new best-known solutions to Taillard's notoriously difficult benchmark instances, providing additional evidence of the effectiveness of our approach. Further, our hybrid algorithm provides two additional advantages over the i-TSAB and Zhang et al. algorithms. First, we demonstrate that performance is largely insensitive to the choice of the fundamental parameters underlying the algorithm. Second, and perhaps most importantly, the hybrid is able to *consistently* achieve excellent performance, e.g., the worst-case performance is roughly equivalent to the mean performance of the Zhang algorithm.

While this paper focuses on the introduction and analysis of a hybrid algorithm in terms of performance, our original motivation was to better understand why constraint programming algorithms for the JSP – in particular, SGMPCS – generally underperform their local search counterparts. Although it is now clear that SGMPCS has a niche relative to local search in state-of-the-art algorithms for the JSP, we have only begun preliminary investigations into understanding this niche and how SGMPCS exploits it. For example, we have preliminary evidence that SGMPCS acts primarily as an intensification mechanism for the elite solutions generated by i-STS, and is empirically more efficient than tabu search in that role. The insensitivity of SGMPCS performance to parameter settings also raises a number of issues, for example, the need to better understand why elite pool size is not a major factor in CP-based search, while it appears fundamental in local search. Overall, the present contribution establishes the hybrid i-STS / SGMPCS algorithm as an interesting test subject; future research will analyze these and other questions raised by this performance analysis.

Acknowledgments

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada, the Canadian Foundation for Innovation, the Ontario Research Fund, Microway, Inc., and ILOG, S.A.. Sandia is a multipurpose laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000

References

1. Balas, E., Vazacopoulos, A.: Guided local search with shifting bottleneck for job-shop scheduling. Management Science 44(2), 262–275 (1998)
2. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-based Scheduling. Kluwer Academic Publishers, Dordrecht (2001)
3. Beck, J.C.: Solution-guided multi-point constructive search for job shop scheduling. Journal of Artificial Intelligence Research 29, 49–77 (2007)
4. Beck, J.C., Fox, M.S.: Dynamic problem structure analysis as a basis for constraint-directed scheduling heuristics. Artificial Intelligence 117(1), 31–81 (2000)

5. Blażewicz, J., Domschke, W., Pesch, E.: The job shop scheduling problem: Conventional and new solution techniques. European Journal of Operational Research 93(1), 1–33 (1996)
6. Garey, M.R., Johnson, D.S., Sethi, R.: The complexity of flowshop and jobshop scheduling. Mathematics of Operations Research 1(2), 117–129 (1976)
7. Glover, F., Laguna, M., Martí, R.: Scatter search and path relinking: Advances and applications. In: Glover, F., Kochenberger, G.A. (eds.) Handbook of Metaheuristics, Kluwer Academic Publishers, Dordrecht (2003)
8. Hooker, J.N.: Testing heuristics: We have it all wrong. Journal of Heuristics 1, 33–42 (1996)
9. Howe, A.E., Watson, J.P., Whitley, L.D.: Deconstructing nowicki and smutnicki's i-tsab tabu search algorithm for the job-shop scheduling problem. Computers and Operations Research, Anniversary Focused Issue on Tabu Search 33(9), 2623–2644 (2006)
10. Laborie, P.: Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. Artificial Intelligence 143, 151–188 (2003)
11. Le Pape, C.: Implementation of resource constraints in ILOG Schedule: A library for the development of constraint-based scheduling systems. Intelligent Systems Engineering 3(2), 55–66 (1994)
12. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. Information Processing Letters 47, 173–180 (1993)
13. Nowicki, E., Smutnicki, C.: A fast taboo search algorithm for the job shop problem. Management Science 42(6), 797–813 (1996)
14. Nowicki, E., Smutnicki, C.: An advanced tabu search algorithms for the job shop problem. Journal of Scheduling 8(2), 145–159 (2005)
15. Nuijten, W.P.M.: Time and resource constrained scheduling: a constraint satisfaction approach. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology (1994)
16. Nuijten, W.P.M., Le Pape, C.: Constraint-based job shop scheduling with ILOG Scheduler. Journal of Heuristics 3, 271–286 (1997)
17. Scheduler. ILOG Scheduler 6.5 User's Manual and Reference Manual. ILOG, S.A (2007)
18. Smith, S.F., Cheng, C.C.: Slack-based heuristics for constraint satisfaction scheduling. In: Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI 1993), pp. 139–144 (1993)
19. Taillard, E.D.: Benchmarks for basic scheduling problems. European Journal of Operational Research 64, 278–285 (1993)
20. Taillard, É.D.: (November 2007), http://ina.eivd.ch/collaborateurs/etd/default.htm
21. Taillard, É.D.: Parallel taboo search technique for the jobshop scheduling problem. Technical Report ORWP 89/11, DMA, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland (1989)
22. Watson, J.-P.: On metaheuristic "Failure Modes": A case study in tabu search for job-shop scheduling. In: Proceedings of the Fifth Metaheuristics International Conference (2005)
23. Watson, J.P.: Empirical Modeling and Analysis of Local Search Algorithms for the Job-Shop Scheduling Problem. PhD thesis, Department of Computer Science, Colorado State University (2003)
24. Wu, H., van Beek, P.: On universal restart strategies for backtracking search. In: Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming, pp. 681–695 (2007)
25. Zhang, C.Y., Li, P., Rao, Y., Guan, Z.: A very fast TS/SA algorithm for the job shop scheduling problem. Computers and Operations Research 35(1), 282–294 (2008)

Counting Solutions of Integer Programs Using Unrestricted Subtree Detection

Tobias Achterberg[1], Stefan Heinz[2,⋆], and Thorsten Koch[2]

[1] ILOG Deutschland, Ober-Eschbacher Str. 109, 61352 Bad Homburg, Germany
tachterberg@ilog.de
[2] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
{heinz,koch}@zib.de

Abstract. In the recent years there has been tremendous progress in the development of algorithms to find optimal solutions for integer programs. In many applications it is, however, desirable (or even necessary) to generate *all* feasible solutions. Examples arise in the areas of hardware and software verification and discrete geometry.

In this paper, we investigate how to extend branch-and-cut integer programming frameworks to support the generation of all solutions. We propose a method to detect so-called *unrestricted subtrees*, which allows us to prune the integer program search tree and to collect several solutions simultaneously. We present computational results of this *branch-and-count* paradigm which show the potential of the unrestricted subtree detection.

1 Introduction

In the last decades much progress has been made in finding optimal solutions to integer linear programs (IP) [6]. Recently, more attention has been given to the task of finding all feasible solutions to a given IP, since it arises in applications, for instance, in the context of hardware and software verification and the analysis of polyhedra (see De Loera et al. [9] and references therein). Furthermore, for IP problems that evolve from industry applications, it is desirable to find multiple or even all optimal solutions as discussed in [8].

A common way to solve IP counting or enumeration problems is to transform them into an equivalent binary representation and use specialized solvers. For Boolean satisfiability instances an algorithm for counting solutions is introduced in [13]. A method based on binary decision diagrams is stated in [4]. This algorithm is capable of counting or enumerating all feasible solutions of binary linear programs (BP), which are IPs containing only binary variables. Alternative methods for these type of problems are given in [7] and [10]. Both approaches make use of a search tree. The first one additionally uses linear programming (LP) relaxations to detect infeasible subproblems.

⋆ Supported by the DFG Research Center Matheon *Mathematics for key technologies* in Berlin.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 278–282, 2008.

There are only few algorithms that count or enumerate all feasible solutions of a general IP and work on the integer variable space. In [8] a branch-and-cut based algorithm is introduced which is able to generate multiple or even all (near) optimal solutions of a given IP (available in CPLEX). Setting the objective function to zero forces this algorithm to enumerate all feasible solutions. Another method which operates on the integer variable space is Barvinok's algorithm [3]. This algorithm counts all lattice points inside a convex polytope in polynomial time when the dimension is fixed.

In this paper, we introduce a *branch-and-count* method based on a branch-and-cut framework to generate all solutions of a given IP. This method works on the integer domain. Furthermore, we state a technique called *unrestricted subtree detection* which collects several solutions simultaneously.

2 Problem Definition

We consider *integer programs* (IP) of the form

$$\min\{\boldsymbol{c}^T\boldsymbol{x} \mid A\,\boldsymbol{x} \leq \boldsymbol{b},\, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u},\, \boldsymbol{x} \in \mathbb{Z}^n\}$$

with $A \in \mathbb{R}^{m\times n}$, $\boldsymbol{b} \in \mathbb{R}^m$, and $\boldsymbol{c}, \boldsymbol{l}, \boldsymbol{u} \in \mathbb{R}^n$. Note that all variables are bounded and of integer type. We are addressing the task of computing the finite set $X_{\mathrm{IP}} = \{\boldsymbol{x} \mid A\,\boldsymbol{x} \leq \boldsymbol{b},\, \boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u},\, \boldsymbol{x} \in \mathbb{Z}^n\}$ of all feasible solutions of a given IP. We denote by $X^*_{\mathrm{IP}} \subseteq X_{\mathrm{IP}}$ the set of all optimal solutions of the integer program, that is, $X^*_{\mathrm{IP}} = \operatorname{argmin}\{\boldsymbol{c}^T\boldsymbol{x} \mid \boldsymbol{x} \in X_{\mathrm{IP}}\}$. If $\boldsymbol{c} = \boldsymbol{0}$, both sets are equal.

It is known that the above formulation is quite general. Maximization problems can be transformed to minimization problems by multiplying the objective function coefficients by -1. Similarly, "$\geq$" constraints can be multiplied by -1 to obtain "$\leq$" constraints. Equations can be replaced by two opposite inequalities.

In the next section, we discuss an approach to compute X_{IP}. With this method it is also possible to generate X^*_{IP}. There are two natural ways to do this: one is to first compute X_{IP} and subsequently X^*_{IP} by only keeping those elements of X_{IP} that minimize the objective function. The other possibility is to solve the underlying IP to optimality, add an additional constraint of the form $\boldsymbol{c}^T\boldsymbol{x} \leq c^*$ to the IP, with c^* being the optimal value of the IP, and finally, compute the set $X_{\mathrm{IP}'}$ for the resulting IP$'$. Obviously, $X_{\mathrm{IP}'}$ is equal to X^*_{IP}.

3 Branch-and-Count Approach

Currently, the most successful general technique to solve IPs (to optimality) is branch-and-cut using LP-relaxations. For a detailed description of the work-flow of branch-and-cut algorithms in general, we refer to Nemhauser and Wolsey [11].

Branch-and-cut algorithms can be adapted to enumerate all feasible solutions of a given integer program, by traversing the whole search tree and collecting all feasible solutions step-by-step. In this section we introduce a technique to speed up the enumeration process of a brach-and-cut based algorithm.

3.1 Pruning by Detecting Unrestricted Subtrees

The basic idea of our approach is to find a way to deduce and construct *all* solution vectors contained in a subtree. If this is possible, the whole subtree can be pruned without explicitly enumerating all leaves. The two most simple structures are subtrees which have no solutions and subtrees where any variable assignment constitutes a feasible solution. We call these subtrees *infeasible subtrees* and *unrestricted subtrees*, respectively.

The infeasible subtree detection is also an issue for a standard branch-and-cut based algorithm focusing on optimal solutions. One way to improve infeasible subtree detection is *conflict analysis*, see [1,12]. Unrestricted subtrees can be detected in the following way: at every node S in the search tree, it is checked whether each constraint is *locally redundant*, i.e., whether it is always satisfied in the local domains.

Definition. A constraint is called *locally redundant at subproblem* S if it is satisfied by all possible variable assignments of values in the local domains at subproblem S.

Lemma 1. *The subtree at a node* S *of the search tree is unrestricted if and only if all constraints are locally redundant at node* S*.*

Proof. Let x be an arbitrary vector in the local domains of subproblem S. If all constraints are locally redundant, each constraint is satisfied by x and thus, x is a feasible solution. Hence, the subtree below node S is unrestricted. On the other hand, if the subtree below S is unrestricted, x must be feasible. Therefore, it satisfies each individual constraint. It follows that each constraint is locally redundant at node S. □

A similar observation was made by Morgado et al. [10] with respect to BPs. Their search algorithm detects feasible solutions if all constraints are locally redundant (through previous variable fixings). Additionally, they have to add so-called *blocking clauses* to prevent the algorithm to count the same solutions several times. Branch-and-cut based algorithms find feasible solutions without checking each constraint for locally redundancy. Therefore, the redundancy check has to be performed explicitly in every search node to find unrestricted subtrees.

Example 1. Consider the following IP:

$$\begin{aligned} \min\{\mathbf{0}^T \boldsymbol{x} \mid\ & x_0 + x_1 + x_2 \leq 2, \\ & x_0 - x_1 + x_2 \leq 1, \\ & x_0 + x_1 - x_2 \leq 1, \\ & x_0 - x_1 - x_2 \leq 0, \\ & \boldsymbol{x} \in \{0,1\}^3\}. \end{aligned}$$

In Figure 1 we depict different branching possibilities for the root node. Only in the first case, where we branch on variable x_0, the resulting subproblems constitute an unrestricted and an infeasible subtree. More precisely, if variable x_0 is fixed to zero, all constraints are locally redundant; setting variable x_0 to one, leads to an infeasible subproblem.

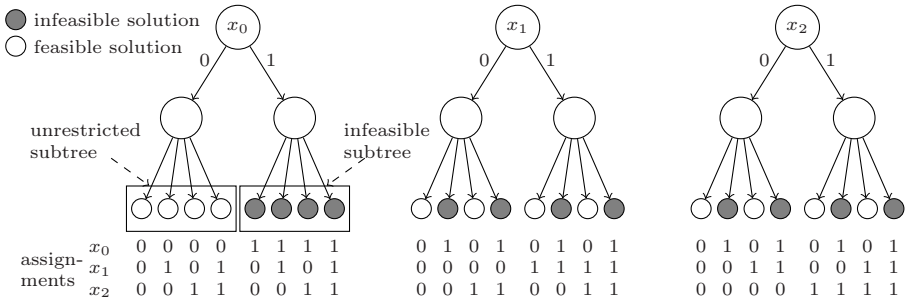


Fig. 1. Possible branching decisions in the root node for Example 1

Table 1. Results for chip verification instances

Instance				basic approach			unrestricted subtree detection			
Name	Cons	Vars	$\lvert X_{\text{IP}}\rvert$	time	nodes	depth	time	nodes	depth	unrest.
veri1	1589	1251	809 424	12.9	1 618 847	26	0.3	17 639	19	8 448
veri2	854	691	655 360	16.8	1 310 762	30	9.6	491 567	29	245 766
veri3	219	138	573 440	18.2	1 146 948	29	15.0	860 227	29	143 360
veri4	748	623	2 097 152	33.0	4 194 319	23	4.9	327 687	20	163 840
veri5	1631	1294	260 096	4.5	520 207	22	0.7	41 011	19	20 487
veri6	1140	901	100 980	1.6	201 959	22	0.1	2 087	13	1 044
veri7	2123	1683	>68 749 M	>1800	>272 M	50	>1800	>111 M	42	>55 684 k
veri8	43	53	264 241 407	>1800	>237 M	34	77.1	4 316 909	31	2 122 366

3.2 Computational Results

We integrated the unrestricted subtree detection into the branch-and-cut framework SCIP (Version 1.00.5) [2]. As an LP-solver we used SOPLEX 1.3.3 [14]. All computations presented in this section were run on computers with an Intel Core 2 Quad CPU with 2.66 GHz, 4 MB cache, and 4 GB of RAM. A time limit of 30 minutes was employed.

Due to the lack of space we first restrict our self to 8 real-world instances which contain several ten-thousand solutions each. These instances arise from chip verification problems and have been provided by OneSpin Solutions[1]. The results are given in Table 1. The first four columns contain the problem instance information, namely the name ("Name"), the number of constraints and variables ("Cons", "Vars"), and the number of feasible solutions ("$\lvert X_{\text{IP}}\rvert$"). Columns labeled with "basic approach" and "unrestricted subtree detection" report the individual results for the branch-and-count framework without and with unrestricted subtree detection, respectively; the first subcolumns report the running time in seconds, the total number of search nodes, and the maximum search tree depth. For the unrestricted subtree detection we further state the number of detected (non-trivial) unrestricted subtrees ("unrest.").

[1] http://www.onespin-solutions.com

The unrestricted subtree detection leads to a substantial decrease in the number of needed search nodes. This comes along with a reduction in the total running time and the maximum depth level of the search tree.

We also applied our method to the MIPLIB [5] instances that do not have continuous variables to compute the sets X^*_{IP} of all optimal solutions. The generation of all optimal solutions can be performed in less than 5 minutes for each instance, except for *cracpb1* and *p2756*. For *p0548* the unrestricted subtree detection was necessary to solve the instance within the time limit.

We compared our approach (SCIP) to existing methods, in particular AZOVE [4], CPLEX [8], LATTE [9], and ZERONE [7]. Our approach clearly dominates the other solvers on the chip verification instances. For the MIPLIB instances the branch-and-cut based algorithms, i.e., CPLEX, ZERONE, and SCIP, are similar in efficiency, while the other solvers are inferior.

References

1. Achterberg, T.: Conflict analysis in mixed integer programming. Discrete Optim. 4, 4–20 (2007)
2. Achterberg, T.: Constraint Integer Programming, PhD thesis, TU Berlin (2007)
3. Barvinok, A.I.: A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. Math. Oper. Res. 19, 769–779 (1994)
4. Behle, M., Eisenbrand, F.: 0/1 vertex and facet enumeration with BDDs. In: Workshop on Algorithm Engineering and Experiments (ALENEX) (2007)
5. Bixby, R.E., Boyd, E.A., Indovina, R.R.: MIPLIB: A test set of mixed integer programming problems. SIAM News 25, 16 (1992)
6. Bixby, R.E., Fenelon, M., Gu, Z., Rothberg, E., Wunderling, R.: MIP: Theory and practice – closing the gap. In: Powell, M., Scholtes, S. (eds.) Systems Modelling and Optimization: Methods, Theory, and Applications, pp. 19–49. Kluwer, Dordrecht (2000)
7. Bussieck, M.R., Lübbecke, M.E.: The vertex set of a 0/1-polytope is strongly P-enumerable. Comput. Geom. 11, 103–109 (1998)
8. Danna, E., Fenelon, M., Gu, Z., Wunderling, R.: Generating multiple solutions for mixed integer programming problems. In: Fischetti, M., Williamson, D.P. (eds.) IPCO 2007. LNCS, vol. 4513, pp. 280–294. Springer, Heidelberg (2007)
9. De Loera, J.A., Hemmecke, R., Tauzer, J., Yoshida, R.: Effective lattice point counting in rational convex polytopes. J. Symb. Comput. 38, 1273–1302 (2004)
10. Morgado, A., Matos, P.J., Manquinho, V.M., Silva, J.P.M.: Counting models in integer domains. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 410–423. Springer, Heidelberg (2006)
11. Nemhauser, G.L., Wolsey, L.A.: Integer and Combinatorial Optimization. John Wiley & Sons, New York (1988)
12. Sandholm, T., Shields, R.: Nogood learning for mixed integer programming, Tech. Report CMU-CS-06-155, Carnegie Mellon University, Computer Science Department (2006)
13. Thurley, M.: sharpSAT – Counting Models with Advanced Component Caching and Implicit BCP. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 424–429. Springer, Heidelberg (2006)
14. Wunderling, R.: Paralleler und objektorientierter Simplex-Algorithmus, PhD thesis, TU Berlin (1996)

Rapidly Solving an Online Sequence of Maximum Flow Problems with Extensions to Computing Robust Minimum Cuts

Doug Altner and Özlem Ergun

H. Milton Stewart School of Industrial and Systems Engineering,
Georgia Institute of Technology. Atlanta, Georgia

Abstract. We investigate how to rapidly solve an online sequence of maximum flow problems (MFPs). Such sequences arise in a diverse collection of settings including stochastic network programming and constraint programming. In this paper, we formalize the study of solving a sequence of MFPs, introduce a maximum flow algorithm designed for "warm starts" and extend our work to computing a robust minimum cut. We demonstrate that our algorithms reduce the running time by an order of magnitude when compared similar codes that use a black-box MFP solver. In particular, we show that our algorithm for robust minimum cuts can solve instances in seconds that would require over four hours using a black-box maximum flow solver.

Keywords: Maximum Flow; Reoptimization; Robust Minimum Cut.

1 Introduction

The Maximum Flow Problem (MFP) is a fundamental problem in discrete optimization. Efficient, network algorithms exist to solve instances with thousands of nodes in a matter of seconds. However, despite the existence of large sequences of MFPs in a diverse selection of papers, there does not exist a formalized study of solving a large sequence of MFPs. Given the existence of rapid and scalable algorithms, it seems intuitive that there would be no substantial cost to using a black-box maximum flow solver to solve a sequence of MFPs. The goal of this paper, however, is to convince the reader that this may lead to an enormous number of unnecessary computations.

We list a few examples of procedures that require solving a sequence of MFPs in the operations research literature: repeatedly checking if the `alldifferent` constraint is satisfied during constraint programming [9], computing market clearing prices [4], bicriteria maximum flow network interdiction [10], estimating the physical difference between two proteins [11], scheduling jobs on a dual-processor machine in real-time [12], computing a robust minimum capacity s-t cut using a polyhedral model of robustness, separating valid inequalities for the Traveling Salesman Problem [2], and stochastic network programming [13].

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 283–287, 2008.

In the applications above, the MFPs are typically similar. Moreover, the time it takes to sort the MFPs in advance usually exceeds any benefits from the sorting. Thus, to model this, we study *online* sequences of MFPs.

To obtain an algorithm for the aforementioned, we modify the preflow-push algorithm of Goldberg and Tarjan [7] to enable "warm starts." We focus on the algorithm of Goldberg and Tarjan because it is considered the fastest maximum flow algorithm in practice [3] and because there are nice properties of the algorithm that can be exploited for efficient reoptimization techniques.

As a selected application, we choose to focus on the Robust Minimum Capacity s-t Cut Problem (RobuCut). This problem models choosing a minimum s-t cut under data uncertainty. This framework can be applied to open-pit mining [8] or assigning jobs to two processors [12].

The contributions of this research are as follows: First, we have formalized the study of solving an online sequence of MFPs. Second, we have designed an effective algorithm for solving the maximum flow single arc reoptimization problem that demonstrates to outperform a black-box maximum flow solver by an order of magnitude. Third, we have designed a powerful algorithm for solving RobuCuts. Our algorithm can solve instances in seconds that typically take over four hours when using a black-box solver.

2 The Maximum Flow Single Arc Reoptimization Problem

In this section, we study the special case of an online sequence of MFPs when each MFP differs from the previous MFP in that exactly one arc has changed. In addition to being a logical place to begin our study, this problem also has applications in real-time scheduling on a dual-processor machine and in computing a robust minimum capacity s-t cut.

We informally define the Maximum Flow Single Arc Reoptimization Problem (MFSAROP) as given a ground network and an online series of sub-networks, find the maximum flow in each of the sub-networks. We note that this framework also allows for changing the capacity of a single arc, which can be done with the usage of parallel arcs, as well as for the addition or deletion of a single node, which can be done with a split-node network.

Our algorithm for MFSAROP exploits the Maximum Flow Minimum Cut Theorem of Ford and Fulkerson [5]. Simply put, this theorem states that the value of the maximum flow in a network equals the value of the minimum capacity cut.

Assume that we must solve the ith MFP and that we have already solved the $(i-1)$st MFP. If a single arc is to be modified before we must compute the new maximum flow, then there are only four possible cases for reoptimization that may be encountered:

1. *An added arc is contained in all minimum cuts.* In this case, a modified maximum flow computation is necessary to recompute the new maximum flow, which will increase in value.

2. *An added arc is not contained in all minimum cuts.* In this case, the maximum flow in the network will not change. No further computations are needed.
3. *A removed arc was contained in at least one minimum cut.* In this case, the maximum flow will decrease by a known amount and the new maximum flow can be computed with breadth-first search.
4. *A removed arc was not contained in any minimum cut.* It is possible that a new minimum cut was created. In this case, we first try to compute the maximum flow that was on the removed arc that can be recovered. All flow that cannot be redirected is removed with breadth-first search.

To determine which reoptimization case we have encountered, we created a data structure called a *cut tripartition*. For a given maximum flow, this data structure stores a unique tripartition of the nodes: all of the nodes reachable from the source in an optimal residual network form one partition, all of the nodes that can reach the sink in an optimal residual network form another partition and all other nodes are in the third partition. A cut tripartition contains two minimum cuts. It also allows one to heuristically identify the appropriate reoptimization case in constant time.

Our algorithm consists of three steps which are iteratively performed for each subsequent MFP after the first MFP is solved. The first step is to transition from the previous network to the next network. The second step is to use the cut tripartition to heuristically identify the reoptimization case and to take the appropriate action to recompute the maximum flow. The third step is to update the cut tripartition.

3 Robust Minimum Capacity s-t Cuts

In this section, we discuss how to use maximum flow reoptimization techniques to design an algorithm for the Robust Minimum Capacity s-t Cut Problem (RobuCut). In [1], Bertsimas and Sim introduced a general model for robust combinatorial optimization problems (RobuCOPs) along with an algorithm to solve an arbitrary RobuCOP by solving a linear number of nominal COPs. Building on their algorithm, we offer an algorithm for RobuCut that uses our reoptimization heuristics.

RobuCut can be viewed as a Stackelberg game. Assume that a user has initially selected Γ to specify his desired level of conservative planning. First, the user will choose a s-t cut. Then, an adversary will choose Γ arcs to set to their highest capacity so as to maximize the capacity of the chosen cut. All other arcs will assume their lowest capacity. The user's objective is to choose a cut that will be of minimum capacity after the adversary exercises his negative influence. We formally define the Robust Minimum s-t Cut Problem as follows:

Robust Minimum s-t Cut Problem: Let $N = (V, A)$ be a network with source s and sink t and let ζ be the family of all minimum capacity s-t cuts in N. Assume arc capacities $\tilde{u_e}$ are uncertain but are known to take value in

$[u_e, u_e + d_e]\ \forall\ e \in A$. Compute a minimum cut under the assumption that at most $\Gamma > 0$ of the arcs are assigned their highest capacity so as to maximally adversely influence the objective value while all other arcs assume their lowest capacity.

$$Minimize \quad \sum_{e \in C} u_e + max_{\{S|S \subseteq A, |S| \leq \Gamma\}} \sum_{j \in S \cap C} d_j$$

$$Subject\ to \quad C \subseteq \zeta$$

Theorem 1. *RobuCut may be solved by computing exactly* $|A| + 1$ *maximum flow computations.*

Proof. Follows from Theorem 3 in [1] and the Maximum Flow Minimum Cut Theorem. □

We can enumerate the $|A|+1$ nominal MFPs (NMFPs) such that the arc capacities are always increasing throughout the sequence of NMFPs. When we need to solve the ith NMFP, we store an *incremental network*, which is an induced subnetwork on the set of all arcs whose capacity changed compared to the $(i-1)$st network of the NMFP. This network is stored as a heuristic technique to accelerate the computation time. Reoptimizing the maximum flow over the induced sequence of incremental networks can be modeled as the MFSAROP.

Our strategy for solving RobuCut is as follows: First we solve the first NMFP. Then, to solve the ith NMFP, we first use the $(i-1)$st incremental network to to recompute the new maximum flow value in the ith incremental network. Then we use the ith incremental network's maximum flow and the maximum flow in the $(i-1)$st NMFP to construct a feasible solution for the ith NMFP to warm start our maximum flow reoptimization algorithm.

4 Computational Results

We conducted a series of experiments on randomly generated instances of both MFSAROP and RobuCut. The purpose of these experiments is to demonstrate the computational savings from using our algorithms as opposed to using a maximum flow solver as a black-box subroutine. For a black-box solver, we implemented our own version of the Goldberg-Tarjan algorithm employing both the gap and global relabeling heuristics described in [3]. Since using Goldberg's maximum flow code [6] as a black-box usually outperforms warm starting a commercial linear programming solver for solving a sequence of maximum flow problems, we restricted our computational study to comparing network-based algorithms.

For MFSAROP, we created over 300 randomly generated instances. The number of nodes ranged from 100 to 1,000, and the number of reoptimizations required ranged from 100 to 500. In these instances, our reoptimization algorithm demonstrated to require about 15% of the time that the black-box solver required. The larger instances took about 25 minutes with a black-box solver but required only around 4 minutes with our reoptimization algorithm.

For RobuCut, we created close to 300 randomly generated instances where the number of nodes ranged from 100 to 500. In these instances, our implementation always solved the instances in a matter of seconds while the black-box solver required over 4 hours.

5 Conclusions

We have demonstrated that through the use of simple reoptimization heuristics one can design effective algorithms for rapidly solving an online sequence of MFPs and for rapidly computing RobuCuts.

Acknowledgements

Özlem Ergun was partially supported by the NSF Career grant DMI-0238815.

References

1. Bertsimas, D., Sim, M.: Robust Discrete Optimization and Network Flows. Mathematical Programming 98(1), 49–71 (2003)
2. Carr, R.: Separating Clique Trees and Bipartition Inequalities Having a Fixed Number of Handles and Teeth in Polynomial Time. Mathematics of Operations Research 22(2), 257–265 (1997)
3. Cherkassky, B., Goldberg, A.: On Implementing Push-Relabel Method for the Maximum Flow Problem. Algorithmica 19(4), 390–410 (1994)
4. Devanur, N., Papadimitriou, C., Saberi, A., Vazirani, V.: Market Equilibrium via a Primal-Dual Algorithm for a Convex Program. In: Proceedings of the 43rd Annual Symposium on Foundations of Computer Science (2002)
5. Ford, L.R., Fulkerson, D.R.: Maximal Flow Through a Network. Canadian Journal of Mathematics 8, 399–404 (1956)
6. Goldberg, A.: Andrew Goldberg's Network Optimization Library, http://avglab.com/andrew/soft.html
7. Goldberg, A., Tarjan, R.: A New Approach to the Maximum Flow Problem. Journal of Associated Computing Machinery 35 (1988)
8. Hochbaum, D., Chen, A.: Improved Planning for the Open - Pit Mining Problem. Operations Research 48, 894–914 (2000)
9. Régin, J.C.: A Filtering Algorithm for Constraints of Difference in Constraint Satisfaction Problems. In: The Proceedings of the Twelfth National Conference on Artificial Intelligence, vol. 1, pp. 362–367 (1994)
10. Royset, J., Wood, R.K.: Solving the Bi-objective Maximum-Flow Network-Interdiction Problem. INFORMS Journal on Computing 19, 175–184 (2007)
11. Strickland, D., Barnes, E., Sokol, J.: Optimal Protein Structure Alignment Using Maximum Cliques. Operations Research (to appear, 2008)
12. Stone, H.S.: Multiprocessor Scheduling with the Aid of Network Flow Algorithms. IEEE Transactions on Software Engineering 3(1), 85–93 (1977)
13. Wallace, S.: Investing in Arcs in a Network to Maximize the Expected Max Flow. Networks 17, 87–103 (1987)

A Hybrid Approach for Solving Shift-Selection and Task-Sequencing Problems

Ada Barlatt[1], Amy M. Cohn[1], and Oleg Gusikhin[2]

[1] University of Michigan, Ann Arbor MI 48109, USA
abarlatt@umich.edu, amycohn@umich.edu
[2] Ford Motor Company, Dearborn MI 48121, USA
ogusikhi@ford.com

Abstract. A common problem in production planning is to sequence a series of tasks so as to meet demand while satisfying operational constraints. This problem can be challenging to solve in its own right. It becomes even more challenging when higher-level decisions are also taken into account, such as which shifts should be selected to accommodate production. In this paper, we introduce the *Shift-Selection and Task Sequencing (SS-TS)* problem, develop the hybrid *Test-and-Prune algorithm (T&P)* to solve SS-TS, and present computational experiments based on a real-world problem in *automotive stamping* to demonstrate its effectiveness. In particular, we are able to solve, in very short run times, a number of problem instances that could not be solved through traditional integer programming methods.

1 Introduction

Production planning problems are typically premised on the assumption of a predefined, fixed set of shifts in which tasks can be scheduled. Decreasing the number of shifts used or the available capabilities of those shifts can result in significant savings in overhead costs, which typically dominate the cost of performing the actual tasks. We call the integration of the higher-level *Shift-Selection (SS)* decisions with the more detailed *Task-Sequencing (TS)* decisions the *Shift-Selection and Task-Sequencing (SS-TS)* problem.

TS problems are often quite challenging to solve by themselves (eg. [3], [6], [8] ,[9]); integrating them with SS decisions yields even greater challenges ([4]), including very large *mixed integer programs (MIPs)* with weak *linear programming* (LP) relaxations. Due to these challenges, this problem has historically been disaggregated to achieve tractability (eg. [5] and [10]), often at the expense of solution quality.

We propose an alternative approach to SS-TS to overcome these challenges, which we call *Test-and-Prune* (*T&P*). This algorithm leverages three key facts common to many SS-TS problems. First, the SS costs greatly dominate the costs associated with TS. Second, the set of SS decisions is discrete and fairly limited. Third, when the shifts are pre-determined, we need only test the feasibility of the corresponding TS problem.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 288–292, 2008.

In such cases, instead of modeling and solving SS-TS as a single optimization problem, we propose to solve it by enumerating all solutions to SS (i.e. all sets of SS decisions) and, for each of these, determining whether the corresponding TS problem instance is feasible. We show that this approach, in conjunction with pruning techniques (to limit the number of feasibility problems actually solved), enables us to solve problems that are computationally intractable under traditional IP-based approaches.

2 Problem Statement

2.1 Formal Problem Statement

- S is the set of SS decisions to be made – whether or not to operate each given shift and, more broadly, decisions about what characteristics these shifts might have.
- y is the decision vector associated with SS; y_s is the s^{th} element of this vector, i.e. the decision variable associated with SS decision s.
- $\mathcal{Y}$ is the set of valid SS solution, capturing any broader constraints that span multiple decisions (for example, budgetary constraints). Note that we do not require that $\mathcal{Y}$ can be represented as a mathematical program.
- $f(y)$ is the cost function applied to SS solution $y \in \mathcal{Y}$. There are no restrictions on f so long as it is easy to compute.
- x is the decision vector associated with TS.
- $\mathcal{X}$ defines any non-negativity constraints, integrality constraints, and upper/lower bounds on the TS decisions x.
- H is the coefficient matrix defining any constraints on x which do not depend on y. We restrict these constraints to be linear.
- I is the set of *linking constraints* between x and y.
- g^i and d^i are the row vectors defining the i^{th} linking constraint.

Formulation

$$\begin{aligned} \min \quad & f(y) && (1)\\ s.t. \quad & \\ & g^i * x - d^i * y \leq 0 \quad \forall i \in I && (2)\\ & H * x \leq 0 && (3)\\ & x \in \mathcal{X} && (4)\\ & y \in \mathcal{Y} && (5) \end{aligned}$$

The objective function (1) computes the value of SS solution y. Constraints (2) enforce limitations on the TS decisions relative to the upper limits provided by the SS solution. For example, tasks cannot be performed during shift i unless shift i has been selected. Constraints (3) are additional constraints to ensure the feasibility of x. Finally, (4) and (5) enforce integrality, non-negativity, and other restrictions on the vectors x and y.

There are several structural aspects of SS-TS that can make this problem difficult to solve. **First**, the set $\mathcal{Y}$ may be non-linear, discrete, non-convex, or have some other characteristic that makes it difficult (if not impossible) to represent as a MIP. **Second**, the cost function $f(y)$ may be non-linear, non-convex, or possibly even not a closed-form function. **Third**, even when SS-TS can be formulated as a MIP, it typically will poses a weak LP relaxation. This is because the objective value can be decreased in a fractional solution by matching the capacity of y to the exact value required by the decisions x. In practice, very poor convergence of the branch-and-bound tree is often observed in such problems. [See [1] and [7] for examples of this in other problem domains.] **Finally**, the TS problem represented by $H * x \leq 0$, $x \in X$ may itself be a large MIP. The integration of SS and TS can exacerbate this, because it may prevent TS from being decomposed.

3 *Test-and-Prune* Algorithm

Our approach to overcome the challenges outlined in Section 2 is premised on the simple idea of solving many easy feasibility problems instead of one difficult optimality problem. Specifically, we enumerate all SS solutions. We select an SS solution and solve its corresponding TS feasibility problem, then we use the result to reduce the feasible region by removing the dominated SS solutions. We repeat this process until we have determined the lowest-cost SS solution that is TS feasible.

There are two phases to the T&P algorithm. First, we *build* the list of candidate SS solutions; we then *process* this list until it is empty and the optimal solution has been found. The Build phase begins by looping through all combinations of the binary SS decisions in S. For each such set of decisions (i.e. SS solution vector y), we check its *validity* – testing to see if it is a member of the set $\mathcal{Y}$. If the solution is valid we compute its cost and add it to the Pending list. If not, we delete it.

In the Process phase, we begin by removing a candidate solution y from the Pending list and testing its TS feasibility. If it is feasible, we prune from the list any pending SS solution with equal or higher cost; clearly such a solution is sub-optimal. We then update the Current Best list with solution y. Conversely, if y is infeasible, then we prune from the Pending list any SS solution $\widehat{y}$ for which $\widehat{y}_s \leq y_s$ for all individual SS decisions s; such SS solutions constrain TS even more tightly and thus will also be infeasible. We then choose another SS solution vector from the (reduced) Pending list and repeat until the list is empty. [In our experience, selecting from the middle of the list greatly enhances the impact of pruning, although one open area of research is to evaluate strategies for processing the list.]

4 Computational Results, Conclusions and Future Research

To evaluate the performance of T&P, we considered the problem of scheduling production in an automotive stamping facility. In this problem, body parts

(hoods, fenders, doors, etc.) are stamped from sheet metal. The sequencing and scheduling of the different part types is itself a difficult task ([2]), with complex rules about when changeovers between part types can take place. In addition, labor rules require that when someone is hired for a given shift on one day (there are up to three eight-hour shifts operated each day), they must be hired for that shift for the full planning horizon (here, two weeks). Thus, substantial savings can be gained by scheduling all production into a reduced set of common shifts (eg. only first and third shift).

Solving these two problems concurrently as a traditional MIP poses substantial challenges. We considered an instance with 22 different presslines collectively producing 130 different part types. Three different demand levels (high, medium, and low) were evaluated. In all three cases, run time was limited to 10.5 hours. For the low demand, the final solution after this run time had an optimality gap of over 16%. The two other instances did not find any integer-feasible solutions in this time period.

We then solved the problem using T&P. For each of the three shifts, there are two characteristics: Is production permitted during this shift? Can changeovers take place in this shift? [This is because there are two different types of labor needed for these tasks.] As a result we have $2^6 = 64$ candidate SS solutions to evaluate. In contrast to the traditional approach, which could not even find feasible solutions in over ten hours, the run times to find optimal solutions for the three instances using T&P were 25, 77, and 509 seconds.

We then tested a larger problem instance, assuming fifteen candidate labor types, for a total of $2^{15} = 32,768$ SS solutions. In this case, the largest number of feasibility problems that actually had to be solved across the three instances was 18 (the remaining 32,750 were all pruned), and the longest runtime was 5 hours; the other two instances each solved in under 10 minutes.

These results suggest that T&P may be a viable option for solving complex SS-TS problems.

There are several ways in which this research can be extended to provide further benefits.

First, we need not restrict the shift attributes to binary decisions, but may consider integer characteristics as well. For example, we might want to consider not only whether a shift is "on" or "off," but also *how many* laborers to staff in that shift.

Second, T&P can naturally be extended to a broader class of problems in which some high level set of (discrete) resources are being allocated (which dominate system cost), while lower level decisions about how to utilize these resources to complete a set of tasks (which dominate system complexity) must be made.

Third, as parallel computing capabilities become increasingly more accessible and affordable, these capabilities can naturally be applied to improve the performance of T&P, by solving multiple TS feasibility problems concurrently. The challenge is then to effectively control the interaction between sub-problems.

Acknowledgments

The authors gratefully acknowledge Yakov Fradin, Sean Little, Mary Jo Luppino, and Craig Morford for their help on this project.

This work was supported by the National Science Foundation Graduate Research Fellowship Program, the Engineering Research Center for Reconfigurable Manufacturing Systems of the National Science Foundation under Award Number EEC-9529125 and a Ford Motor Company University of Michigan Alliance Grant.

References

1. Armacost, A., Barnhart, C., Ware, K.: Composite Variable Formulations for Express Shipment Service Network Design. Transportation Science 36, 1–20 (2002)
2. Barlatt, A., Cohn, A., Guisikin, O., Fradin, Y., Morford, C.: Using Composite Variable Modeling to Achieve Realism and Tractability in Production Planning: An Example from Automotive Stamping. Technical Report TR07-01 (2007), http://ioe.engin.umich.edu/techrprt/pdf/TR07-01.pdf
3. Bernstein, D., Rodeh, M., Gertner, I.: On the Complexity of Scheduling Problems for Parallel/Pipelined Machines. IEEE Transactions on Computers 39, 1308–1313 (1989)
4. Blazewicz, J., Lenstra, J., Rinnooy Kan, A.: Scheduling Subject to Resource Constraints: Classification and Complexity. Discrete Applied Mathematics 5, 11–24 (1983)
5. Gabbay, H.: Multi-Stage Production Planning. Management Science 25, 1138–1148 (1979)
6. Garey, M., Johnson, D., Sethi, R.: The Complexity of Flowshop and Jobshop Scheduling. Mathematics of Operations Research 1, 117–129 (1976)
7. Klose, A.: An LP-Based Heuristic for Two-Stage Capacitated Facility Location Problems. The Journal of the Operational Research Society 50, 157–166 (1999)
8. Monma, C., Potts, C.: On the Complexity of Scheduling with Batch Setup Times. Operations Research 37, 798–804 (1989)
9. Potts, C., Kovalyov, M.: Scheduling with Batching: A Review. European Journal of Operational Research 120, 228–249 (2000)
10. Qiu, M.M., Burch, E.E.: Hierarchical Production Planning and Scheduling in a Multi-Product, Multi-Machine Environment. International Journal of Production Research 35, 3023–3042 (1997)

Solving a Log-Truck Scheduling Problem with Constraint Programming

Nizar El Hachemi, Michel Gendreau, and Louis-Martin Rousseau

Interuniversity Research Centre on Enterprise Networks,
Logistics and Transportation (CIRRELT)
C.P. 6128, succursale centre-ville, Montreal, Canada H3C 3J7
{nizar,michelg,louism}@crt.umontreal.ca

Abstract. Scheduling problems in the forest industry have received significant attention in the recent years and have contributed many challenging applications for optimization technologies. This paper proposes 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 non-productive activities such as waiting time of trucks and forest log-loaders and the empty driven distance of vehicles. We propose a constraint programming model to address the combined scheduling and routing problem and an integer programming model to deal with the optimization of deadheads.

1 Introduction

The forest industry plays an important role in the economy of several countries such as Chile, Canada, Sweden, Finland and New Zealand. Planning problems in forestry cover a wide scope of activities ranging from planting and harvesting to road building and transportation. Furthermore, in most problems, it is critical to pay attention to important environmental issues, as well as to company-specific goals and operating rules. In Quebec, transportation represents more than 30% of the cost of provisioning for wood transformation mills.

The Log-Truck Scheduling Problem (LTSP) is closely related to some routing problems encountered in other industries, in particular, so-called "pick-up and delivery problems" (see for instance [6]). In our case, we consider a pick-up and delivery problem such that for each request exactly one load of wood has to be transported from its pick-up location (forest area) to its delivery location (woodmill). A truck visits only one forest area and one mill on any given trip, i.e., requests are served individually by trucks. After unloading at a mill from its previous trip, a truck is usually sent back empty to its next forest destination. All requests are also assumed to be known in advance. We first assume that at each mill and each forest location, there is a single log loader that ensures the loading and unloading of all trucks. When a truck arrives at a location, if the loader is busy, then the truck has to wait until the loader becomes available. These waiting

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 293–297, 2008.

times can severely delay trucks and thus increase the cost of transportation; they should therefore be avoided as much as possible.

Since the mid-1990s, several companies in the forestry sector have initiated major projects aimed at improving the transportation portion of their activities, in particular, the control and quality of truck scheduling [9,3,4,2]. Rönnqvist [5] gives a detailed description of optimization in forestry.

However none of these techniques have addressed the issue of synchronizing the schedules of both log-loaders and trucks. The contribution of this paper is therefore a Constraint Programming (CP) model for the LTSP, which minimizes all non-productive activities such as deadhead trips and waiting times. To speed up the resolution process, we model the truck circulation as an Integer Programming (IP) problem that, once solved, generates global cardinality constraints for the CP scheduling model. To our knowledge this is the first time that IP and CP models communicate through the use of structured global constraints.

2 Modelling the LTSP

Over the last years, constraint-based scheduling has become an important tool for modeling and solving scheduling problems [1] and complex transportation problem [8]. We thus present a CP model for the LTSP along the lines of this paradigm. The decision variables of the problem are based on the ILOG Scheduler component of OPL studio 3.7 (see [7]). For each *activity* A, two finite domain variables are created, A^s and A^e, which are associated respectively to the beginning and the end of the activity. Let R be the set of requests, the model is based on the following variables and definitions:

V : set of vehicles defined as *alternative unary resources*.
L^m : log loader in a mill m defined as a *unary resource*.
L^f : log loader in a forest area f defined as a *unary resource*.
P_r : pickup *activity* of duration d^p associated to request r.
D_r : delivery *activity* of duration d^d associated to request r.
C_r : combined *activity* of pickup, traveling and delivery.
$V_r \in V, \forall r \in R$: Vehicle assigned to request r.
$S_r \in O, \forall r \in I$: Successor of request r on the same vehicle.

In this first model, the objective function consists in minimizing the sum of deadheads costs (empty trips) and the cost of the waiting time of trucks and log-loaders. The problem is essentially[1] constrained by the fact that 1) C_r *requires* V, 2) P_r *requires* L^f, 3) D_r *requires* L^m, 4) P_r *precedes* D_R, 5) $P_r^s = C_r^s$, 6) $D_r^e = C_r^e$, 7) $AllDifferent(S)$, and 8) $V_{S_r} = V_r$. Furthermore the scheduling part of the model (activities and resources) is linked with the transportation variables (S and V) through the use of the OPL constraint $activityHasSelectedResource(C_r, V, v)$, which holds if activity C_r has selected resource v in alternative resources V. The search strategy consist of two steps: first, we generate a value for each successor

[1] The complete model cannot be included here, for space consideration.

variable $S_r, \forall r \in I$, secondly, once the successor variables are fixed, we try to schedule each unloading request as soon as possible.

3 An Hybrid Approach

Preliminary experiments showed that the model had some difficulties identifying good values for the successor variables. We thus proposed a decomposition approach, where we modeled the circulation of trucks between the mills and the forest as a network flow problem (with some additional constraints). This model can be easily solved as an Integer Program (IP) where x_{ij} is number of empty trucks driving from woodmill i to forest area j. It yields an optimal solution with respect to the deadhead component of the objective functions, but it is, however, not able to schedule the trucks and the log-loader. To link both IP and CP models, we have considered three different approaches: 1) Solve the IP, note the optimal *objective value*, and use it as a constraint on the deadhead component of the objective function in the CP model. 2) Solve the IP, note the optimal *solution*, and use it to fix the successor variables in the CP model. 3) Solve the IP, look at the structure of the optimal solution, and impose it in the CP model through the introduction of global constraints on successor variables.

Since the objective function of CP model is basically a large sum of small elements, it unfortunately does not allows for good back propagation (bounding the objective does not really trigger propagation and domain reduction). For that reason, method 1 would not be very useful in our context. On the other hand, since the IP model completely ignores the important scheduling aspects of the problem, fixing the sequence of all requests in the CP model would be over constraining the scheduler. Once the complete sequence is given, there is not enough flexibility left to avoid waiting times. The objective thus remains to identify a good solution to the deadhead problem, while still giving the CP model enough flexibility to minimize waiting times in the final schedule.

For this reason we chose to migrate from the IP model only the minimal information that would allow achieving the minimal deadhead value. Since all full loads must be transported from their origin to their destination, we observed that the optimal value is completely determined by the arcs representing empty trips in the solution. It is thus not the global sequence that is important, but rather the number of empty trips performed between each mill and forest site.

These numbers can be extracted from the IP optimal solution and imposed in the CP model through the introduction of Global Cardinality Constraints (GCC). Let us consider x^*_{ij} the optimal solution of the IP model. We define x^*_i as the vector composed of the $|F|$ entries of x^*_{ij}. To introduce these news constraints, we need to define a new variable J^i_r which specifies which forest area will be visited just after unloading request r at mill i. The added constraints are thus $\text{GCC}(x^*_i, F, J^i_r), \forall i \in M$. We thus constrain the CP model to use the correct number of deadhead trips between each mill-forest pair. Imposing this structure considerably reduces the search space and speeds up the resolution.

4 Experimental Results

We compared the two approaches presented and evaluated their respective performances on two different case studies (only one is reported here). These cases where provided by the Forest Engineering Research Institute of Canada (FERIC). Some results for the first case study are reported in Table 1 where the value of a solution is presented in terms of unproductive costs. We report (in dollars) the deadhead costs, the waiting cost of trucks queuing to get loaded or unloaded, the waiting cost of log-loaders while waiting for a truck to arrive, and finally the total cost of all these unproductive activities.

Table 1. Impact of the hybrid approach

$\|V\|$	$\|R\|$	deadhead($)	truck($)	log-loaders($)	total($)
CP only					
15	32	4439	55	2942	7436
15	45	6498	110	4509	11117
15	55	7927	230	7050	15207
15	70	10150	370	7725	18245
Hybrid					
15	32	2222	15	2475	4712
15	45	3202	100	5367	8669
15	55	3914	40	6700	10654
15	70	4982	20	8733	13735

Looking at the table we note that although the smaller instances are relatively easy, the difficulty to synchronize efficiently the trucks and the log loaders increases rapidly with the size of the instances. In several cases, the decomposition method provides a better overall solution than the straightforward approach. However, for some larger instances, the log loader waiting time considerably increases. This behaviour occurs as a result of the particular structure that is imposed in the scheduling problem through the cardinality constraints. We also have developed a perturbation technique that attempts to resolve this situation by exploring other circulations that have less impact on the log loader scheduling, but that still have a very low deadhead cost.

5 Conclusion

We have presented a Log-Truck Scheduling Problem with an objective function to minimize the cost of unproductive time. We proposed a decomposition approach involving a Constraint Programming model and an Integer Programming model that allows us to compute the optimal global circulation of the vehicles. This circulation is then communicated to the CP model by introducing global cardinality constraints.

References

1. Baptiste, P., LePape, C., Nuijten, W.: Constraint-Based Scheduling. Kluwer Academic Publishers, Dordrecht (2001)
2. Gronalt, M., Hirsch, P.: Log-Truck Scheduling with Tabu Search Strategy. Metaheuristics 39 (2007)
3. Linnainmaa, S., Savalo, J., Jokinen, O.: EPO: A knowledge based system for wood procurement management. In: The 7th Annual Conference on Artifical Intelligence, Montreal (1995)
4. Palmgren, M., Rönnqvist, M., Värbrand, P.: A near-exact method for solving the log-truck scheduling problem. International Transactions in Operational Research 11, 447–464 (2004)
5. Rönnqvist, M.: Optimization in forestry. Mathematical Programming 97, 267–284 (2003)
6. Ropke, S., Cordeau, J.-F., Laporte, G.: Models and Branch-and-Cut Algorithm for Pick-up and Delivery Problems with Time Windows. Networks 49(4), 258–272 (2007)
7. Van Hentenryck, P.: The OPL Optimization Programming Language. MIT Press, Cambridge (1999)
8. Simonis, H., Charlier, P., Kay, P.: Constraint Handling in an Integrated Transportation Problem. IEEE Intelligent Systems and their Applications 15(1) (January 2000)
9. Weintraub, A., Epstein, R., Morales, R., Seron, J., Traverso, P.: A truck scheduling system improves efficiency in the forest industries. Interfaces 26(4), 1–12 (1996)

Using Local Search to Speed Up Filtering Algorithms for Some NP-Hard Constraints

Philippe Galinier, Alain Hertz, Sandrine Paroz, and Gilles Pesant

École Polytechnique de Montréal
C.P. 6079, succ. Centre-ville
Montreal, Canada H3C 3A7
{philippe.galinier,alain.hertz,sandrine.paroz,gilles.pesant}@polymtl.ca

1 Introduction

Constraint programming relies heavily on identifying key substructures of a problem, writing down a model for it using the corresponding constraints, and solving it through powerful inference achieved by the efficient filtering algorithms behind each constraint. But sometimes these individual substructures are still too difficult to handle because we do not have any efficient filtering algorithm for them. In other words, deciding satisfiability for some substructures is NP-hard.

Besides breaking them up into smaller tractable pieces and thereby sacrificing the possibility of more global inference, a few researchers have proposed ways to preserve such substructures (e.g., [8,3]). In this paper, we propose a new approach which relies on using a local search heuristic. Local search has been very successful at solving difficult, large-scale combinatorial problems. Applied to a particular substructure, it may quickly find some solutions, each solution acting as a witness for the variable-value pairs appearing in it. In this way, a collection of diverse solutions can offer a support for many variable-value pairs. Local search offers however no help in general to confirm those variable-value pairs that should be filtered. If only a few unsupported candidates remain, a complete method can very well be affordable to decide about them.

To illustrate this approach, we consider the `SomeDifferent` constraint which states that some pairs of variables are restricted to take different values. Richter et al. [7] have studied this substructure, and proposed a filtering algorithm for it. The `SomeDifferent` constraint can be described with the following graph coloring model. Consider a set $X = \{x_1, \cdots, x_n\}$ of variables with domains $D = \{D_1, \cdots, D_n\}$, and a graph $G = (V, E)$ with vertex set $V = \{1, \cdots, n\}$ and edge set E. We denote $D(U) = \bigcup_{v \in U} D_v$ for any $U \subseteq V$, and D_v is called the *color set* of v. A D-coloring of G is a function $c : V \rightarrow D(V)$ that assigns a color $c(v) \in D_v$ to each vertex so that $c(u) \neq c(v)$ for all edges $(u, v) \in E$. The graph G is *D-colorable* if such an assignment exists. The *list coloring problem* is to determine if a given graph G with color sets D is D-colorable. It is NP-complete, even when restricted to interval graphs [1] or bipartite graphs [4].

Following Richter et al. [7], a *point coloring* is defined as a pair (v, i) with $v \in V$ and $i \in D_v$. A point coloring (v, i) is *supported* in G if there exists a D-coloring c of G with $c(v) = i$. If a point coloring (v, i) is *unsupported* in G,

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 298–302, 2008.

then color i can be suppressed from D_v, and we say that the point coloring (v, i) can be *filtered* out from G. The role of our filtering algorithm is to achieve domain consistency which corresponds to finding new domains $D'_1, \cdots, D'_n$ so that $i \in D'_v$ if and only if (v, i) is a supported point coloring in G.

2 Description of the Filtering Algorithm

The proposed filtering procedure follows the following three steps. First, during Step 1 (*colorability testing*), we determine if the graph G is D-colorable. If it is not the case, the algorithm stops. Otherwise, we go to Step 2. During Step 1, we apply some preliminary reduction techniques, then decompose the graph into connected components and test the colorability of each connected component. The colorability test procedure used in Step 1 is detailed in Section 2.1. Next, during Step 2 (*marking*), we generate the largest possible set L of supported point colorings in the graph. This task is achieved by applying our local search procedure to each connected component. The local search procedure is described in Section 2.2. Last, during Step 3 (*filtering*), we test each point coloring (v, i) that does not belong to L and determine whether it is supported. Each connected component G_j is considered in turn. For each vertex v in G_j and each color i in $D_v - L$, we build a copy H of the connected component, assign in H value i to v, and apply to H a colorability test similar to the one used in Step 1. If H is not colorable, point coloring (v, i) is filtered out in G.

2.1 A Colorability Test Procedure

The input of the colorability testing procedure is a graph G and a set D of domains. We first apply some simple reduction techniques in order to achieve a preliminary filtering and to remove some "superfluous" edges in the graph. For example, when the color set of a vertex v contains a single color i, color i is filtered out from the color sets of the vertices u adjacent to v, and all edges incident to v are removed. In addition, we remove edges with both endpoints having disjoint color sets. Then, we decompose the graph into connected components $G_1, \cdots, G_r$. Finally, we test the colorability of each component G_j. If G_j is not colorable, the algorithm stops immediately because the initial graph is not colorable. In order to determine if a particular connected component G_j is colorable, we first apply our tabu search procedure (called `TabuSat`, see Section 2.2). If the tabu procedure does not find a solution, we use the exact graph coloring algorithm DSATUR [6]. The list coloring problem is transformed into a graph coloring problem using the technique proposed in [7].

2.2 A Tabu Search Heuristic Used for Marking

The role of the `TabuSD` algorithm is, when applied to a graph G, to return the largest possible set of supported point colorings. It can be seen as a natural extension of the `Tabucol` algorithm [2] that solves the classical vertex coloring problem. The solution space S is the set of all functions $c : V \rightarrow D(V)$ with

$c(v) \in D_v$ for all $v \in V$. Hence, a solution is not necessarily a D-coloring since adjacent vertices u and v in G can have the same color. In such a situation, we say that the edge linking u to v is a *conflicting edge*. When `TabuSD` visits a D-coloring c (i.e., a solution without conflicting edges), all pairs $(v, c(v))$ are introduced in a list L which contains all point colorings for which we have a proof that they are supported.

Let $f_1(c)$ denote the number of conflicting edges in c, and let $f_2(c)$ denote the number of point colorings $(v, c(v))$ in L. The objective function to be minimized by `TabuSD` is defined as $f(c) = \alpha f_1(c) + f_2(c)$. Parameter α is initially set equal to 1 and is then adjusted every ten iterations : if the ten previous solutions were all D-colorings of G then α is divided by 2; if instead they all had conflicting edges, then α is multiplied by 2; otherwise, α remains unchanged.

A neighbor solution $c' \in N(c)$ is obtained by assigning a new color $c'(v) \neq c(v)$ to a vertex v so that either v is adjacent to a vertex u with $c(u) = c(v)$, or $(v, c(v))$ belongs to L. When performing such a move from c to c', we forbid to reassign color $c(v)$ to v for $K + \lambda\sqrt{|\ N(c)\ |}$ iterations : if $(v, c(v)) \in L$, $(v, c'(v)) \notin L$, and v is not adjacent to any vertex with color $c(v)$, then K is uniformly chosen in the interval $[30, 40]$ and we set $\lambda = 50$; otherwise, K is uniformly chosen in $[20, 30]$ and we set $\lambda = 1$.

We use a first improvement strategy. More precisely, when evaluating the solutions in $N(c)$, it may happen that a non tabu neighbor c' is reached with $f(c') < f(c)$. In such a case, we stop evaluating the neighbors of c and move from c to c'. Otherwise, `TabuSD` moves from c to the best non tabu neighbor. The algorithm stops as soon as L contains all point colorings of G or is not modified for `maxiter` iterations. The `TabuSat` procedure used during the colorability test procedure is very similar to the `TabuSD` procedure, except that it stops as soon as it finds a D-coloring (i.e. a solution c with $f_1(c) = 0$) or after `maxiter` iterations. For our experiments, we set `maxiter` to 2000.

3 Computational Experiments

We evaluated our algorithm on three types of instances: real data, random graphs and graphs with a unique D-coloring. All tests were performed on a 2.80GHz Pentium D with 1024K cache running Linux CentOS 2.6.9. We have done five runs on each instance, and we report average results.

3.1 Workforce Management Data

The real life problem studied in [7] is a workforce management problem in a certain department at IBM. We are given a set of jobs with dates during which each job was to be performed, and a list of people qualified to perform these jobs. Jobs that overlap in time cannot be performed by the same person. This is a typical `SomeDifferent` situation which can be modeled by a D-coloring problem where jobs are vertices of the graph, colors correspond to people, and there is an edge between two vertices if the corresponding jobs overlap in time. The color

set D_v of a vertex v is the set of people qualified to perform v. In all there are 290 instances with a number n of jobs varying from 20 to 300. Our results are similar to those reported [7], both in the overall behavior for satisfiable versus unsatisfiable instances and in the computing times, which never exceed 0.35s. For comparison, without using our tabu search marking procedure, computing times were on average 34 times higher, reaching 10.61s on one instance. This data set actually has very few unsupported point colorings (less than 1%) so only a few values are filtered out, if any.

3.2 Random Graphs

For our second test set, we considered the same random graphs as in [7]. These graphs are defined with a quadruplet (n, p, d, max_k) of parameters: n is the number of vertices, p is the probability of having an edge between two vertices, $d = |D(V)|$ is the number of different colors, and max_k is the maximum size of an individual color set. For each vertex v, an integer k_v is uniformly chosen in the interval $[1, \cdots, max_k]$, and D_v is generated by randomly choosing k_v colors in the interval $[1, \cdots, d]$. The instances in [7] have $n = 20, 30, \cdots, 100$, $p = 0.1, 0.3, 0.6$, $d = 300$, and $max_k = 10, 20$. We have generated additional instances considering also $n = 200, 500$, $p = 0.9$, and $max_k = 5, 40, 80$. For the original instances, our method is much faster since the maximum cpu time reported in [7] is 608.73s while our filtering algorithm never requires more than 0.058s. For the additional instances with up to 100 vertices, the maximum cpu time is 0.2s, which can be considered as reasonable for an algorithm used to achieve domain consistency. For random graphs with 200 vertices, the maximum cpu time grows up to 1.5s, and it reaches 41.5s for $n = 500$. The increase of the cpu time is mainly due to the use of `TabuSD` which requires many iterations to find a support for almost all point colorings. Again the percentage of unsupported point colorings is very small, with a maximum of 0.6%.

3.3 Graphs with a Unique D-Coloring

Mahdian and Mahmoodian [5] have described a family of graphs with a unique D-coloring. More precisely, given an integer k, they define a graph with $3k - 2$ vertices and in which every color set D_v contains exactly k colors. For each vertex v there is exactly one supported point coloring (v, i) with $i \in D_v$, which means that $(k - 1)(3k - 2)$ point colorings can be filtered out.

These instances are difficult, mainly because `Dsatur` is required to confirm that unsupported point colorings can indeed be filtered out. Computing times exceed 100 seconds for graphs with 25 vertices (for $k = 9$), which shows the limit of applicability of our filtering algorithm. While most point colorings are shown supported with `TabuSD` for small values of k, `Dsatur` does the job for larger values since `TabuSD` is not able to find the unique D-coloring.

With the hope of generating graphs with a significant yet more realistic number of point colorings to filter out, we created some variations of these graphs by deleting a certain percentage p of edges. We performed tests for $k = 5, 6, 7, 8, 9$

and $p = 0.02, 0.05, 0.1, 0.15$ and for each pair of parameters we generated ten different graphs. The results were not homogeneous. For example, while 113s are needed to solve the original graph with $k = 9$, the time needed for the graphs with $k = 9$ and $p = 0.05$ ranged from 0.32s to 182.7s. Moreover, with $p = 0.15$, no point coloring could be filtered out, which defeated our purpose.

4 Conclusion and Future Work

We presented a filtering algorithm for the `SomeDifferent` constraint (i.e. the list coloring problem) which combines a tabu search to quickly find a supporting solution for as many point colorings as possible, and an exact algorithm to validate or filter out the remaining point colorings. Our filtering algorithm turned out to be about as fast as the one of Richter et al. [7] when tested on data from a workforce management problem, and significantly faster for random data. In future work, we intend to test our implementation of the `SomeDifferent` constraint within a constraint programming model, in conjunction with other types of constraints, in order to measure the potential increased efficiency.

The general principles of the proposed approach are not specific to the graph coloring substructure. Indeed, the technique can be adapted to other NP-hard constraints in order to obtain a filtering procedure that enforces domain consistency. This can be done by developing mainly two specific (problem-dependent) low-level procedures: an exact algorithm and a local search procedure. We plan to investigate this approach for other NP-hard constraints.

References

1. Biro, M., Hujter, M., Zsolt, T.: Precoloring extension 1. interval graphs. Discrete Mathematics 100, 267–279 (1992)
2. Hertz, A., de Werra, D.: Using tabu search for graph coloring. Computing 39, 345–351 (1987)
3. Katriel, I.: Expected-case analysis for delayed filtering. In: Beck, J.C., Smith, B.M. (eds.) CPAIOR 2006. LNCS, vol. 3990, pp. 119–125. Springer, Heidelberg (2006)
4. Jansen, K., Scheffler, P.: Generalized coloring for tree-like graphs. Discrete Applied Mathematics 75, 135–155 (1997)
5. Mahdian, M., Mahmoodian, E.S.: A characterization of uniquely 2-list colorable graphs. Ars Combinatoria 51, 295–305 (1999)
6. Peemoeller, J.: A correction to Brélaz's modification of Brown's coloring algorithm. Communications of the ACM 26(8), 593–597 (1983)
7. Richter, Y., Freund, A., Naveh, Y.: Generalizing alldifferent: The somedifferent constraint. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 468–483. Springer, Heidelberg (2006)
8. Sellmann, M.: Approximated consistency for knapsack constraints. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 679–693. Springer, Heidelberg (2003)

Connections in Networks: A Hybrid Approach

Carla P. Gomes[1], Willem-Jan van Hoeve[2], and Ashish Sabharwal[1]

[1] Department of Computer Science, Cornell University, Ithaca NY 14853, U.S.A.
{gomes,sabhar}@cs.cornell.edu
[2] Tepper School of Business, Carnegie Mellon Univ., Pittsburgh PA 15213, U.S.A.
vanhoeve@andrew.cmu.edu

Abstract. This paper extends our previous work by exploring the use of a hybrid solution method for solving the connection subgraph problem. We employ a two phase solution method, which drastically reduces the cost of testing for infeasibility and also helps prune the search space for MIP-based optimization. Overall, this provides a much more scalable solution than simply optimizing a MIP model of the problem with Cplex. We report results for semi-structured lattice instances as well as on real data used for the construction of a wildlife corridor for grizzly bears in the Northern Rockies region.

In recent work [2], we investigated the *connection subgraph* problem, which seeks to identify a cost bounded connected subgraph of a given undirected graph connecting certain pre-specified terminal nodes, while maximizing the overall utility. Here costs and utilities are non-negative numbers assigned to each node of the graph, and the cost (or utility) of a subgraph is the sum of the costs (utilities, resp.) of the nodes in it. This problem is a variant and generalization of the familiar Steiner tree problem, and occurs in natural settings such as wildlife conservation and social networks.[1] Our experimental results [2] identified an interesting easy-hard-easy pattern in a pure optimization version of the problem. They also brought out some surprising issues with respect to the hardness of proving infeasibility versus the hardness of proving optimality. Specifically, using a mixed integer programming (MIP) model for the problem and solving it to optimality using Cplex 10.1 [3] revealed that in median terms, Cplex took orders of magnitude longer to prove infeasibility of infeasible instances than it took to find optimal solutions to the feasible instances. This naturally raises the question, can one do better on infeasible instances?

This paper reports our results obtained using a hybrid technique for solving the connection subgraph problem, beginning with results on certain semi-structured grid graphs also considered previously. We use a two phase solution method. The first phase employs a minimum Steiner tree based algorithm to test for infeasibility and to produce a greedy (and often sub-optimal) solution for feasible instances. This phase runs in polynomial time for a constant number of terminal nodes. The second phase refines this greedy solution to produce an optimal solution with Cplex, also using shortest path information generated by

[1] Due to lack of space, we refer the reader to our previous paper [2] for a formal definition and detailed discussion of the problem.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 303–307, 2008.

the first phase to prune the search space significantly (often by 40-60%). With this hybrid approach, the time to test for infeasibility is drastically reduced, and in fact becomes negligible compared to the cost of running Cplex on feasible instances (the runtime for which is also significantly reduced due to the starting solution and pruning). The hardness profiles still show a clear easy-hard-easy pattern in the feasible region.

We also apply this technique to the original resource economics problem that motivated this work—designing a "wildlife conservation corridor" in the Northern Rockies for preserving grizzly bears. The scale of this real-world problem precludes computing optimal solutions in well over a month of CPU time, even with our hybrid approach. We therefore introduce a *streamlined* model, where we seek to compute the optimal (i.e., highest utility) solution which is restricted to include all nodes that form part of a minimum cost solution, which is also computed in the first phase. We are able to solve this "extended-mincost solution" problem significantly faster, and to near optimality within a month of CPU time on the real wildlife corridor data.

The extended-mincost solution is interesting to compute only if it does not dramatically limit the utility one might achieve in the end. To obtain further insights into this, we study how the extended-mincost solution compares in quality (i.e., attained utility) against the true optimal solution for a given budget, for both grid graphs and coarse granularity (and thus easier) versions of the actual corridor construction problem. We show that the *utility gap* between the optimal and extended-mincost solutions itself follows a fairly narrow low-high-low pattern as the budget increases, indicating that for a large range of budgets, solving the streamlined extended-mincost problem yields a fairly good approximation to the true optimal solution.

The Two Phase Approach. In Phase I, we compute a minimum cost Steiner tree for the terminal nodes of the graph, ignoring all utilities. While there are fixed parameter tractable (FPT) algorithms for computing a minimum cost Steiner tree, we used a simpler "enumeration" method (see, e.g., [4]) based on computing all-pairs-shortest-paths with respect to vertex costs. The idea behind this algorithm, which runs in polynomial time for a constant number of terminal nodes, is to compute a minimum Steiner tree for the "complete shortest distance graph" using the fact that in such a graph, there exists a minimum Steiner tree all whose non-terminal nodes have degree at least three, thereby limiting the total number of nodes in the tree. A minimum Steiner tree of the complete shortest distance graph yields a minimum Steiner tree for the original graph as well, by replacing edges by shortest paths.

The computation of the Steiner tree either classifies the problem instance as infeasible for the given budget or provides a feasible (but often sub-optimal) "mincost" solution. In the latter case, we use a very efficient *greedy method* to improve the quality of the solution by using any residual budget as follows. We consider those nodes that are adjacent to the current solution and have cost lower than the residual budget, and identify one whose *gain*, defined as the utility-to-cost ratio, is the highest. If there is such a vertex, we add it to the current solution, appropriately reduce the residual budget, and repeat until no

more nodes can be added. This process often significantly increases the solution quality. We call the resulting solution an *extended-mincost solution*. We will also be interested in computing the optimal extended-mincost solution, by "freezing" the vertices in the mincost solution to be part of all solutions of Phase II.

After Phase I, which always took almost negligible time compared to Phase II on our problem instances, we either know that the instance is infeasible or already have a greedily extended feasible solution. In the latter case, Phase II of the computation translates the problem into a MIP instance (see [2] for details of the encoding), and solves it using Cplex. Solving using Cplex is the most computationally-intensive part of the whole process. The greedy solution obtained from Phase I is passed on to Cplex as a starting solution, providing a major boost to its efficiency. Further, the all-pairs-shortest-paths matrix computed in Phase I is also passed on to Phase II. It is used to statically (i.e., at the beginning) prune away all nodes that are easily deduced to be too far to be part of a solution (e.g., if the minimum Steiner tree containing that node and all of the terminal vertices already exceeds the budget). This significantly reduces the search space size, often in the range of 40-60%. Overall, Phase II computes an optimal solution (or the optimal extended-mincost solution) to the utility-maximization version of the connection subgraph problem.

Experimental Results. For a varying budget, we investigate the computational hardness of the problem with respect to computing the optimal solution or the optimal extended-mincost solution. Our experiments were conducted on a 3.8 GHz Intel Xeon machine with 2 GB memory running Linux 2.6.9-22.ELsmp. We used Cplex 10.1 [3] to solve the MIP problems in Phase II.

For the first set of experiments, we make use of semi-structured lattice graphs of order m, with 3 terminal vertices, and with uniform random costs and utilities (see [2] for details). In Figure 1, each data point is based on 500 random instances for $m = 10$; similar results, peaking at identical x-axis values, were obtained for $m = 6$ and 8 as well, and are available from the authors. The hardness curves are represented by median running times over all instances per data point. In order to normalize for the small but non-negligible variation in the characteristics of various randomly generated instances with the same parameters, we use for the x-axis of most of our plots the 'budget slack percentage', rather than simply the budget, computed as follows. For every instance, we consider its *mincost*, the cost of the cheapest solution. The *budget slack %* with respect to mincost is defined as: $100 \times (budget - mincost)/mincost$. In other words, we consider computational hardness and other measured quantities as a function of the extra budget available for the problem beyond the minimum required.

In the left half of Figure 1, we show the hardness profiles for the lattices, which exhibit an easy-hard-easy pattern, the peak of which is to the right of the mincost point (shown as 0 on the relative x-scale). As one might expect, computing the optimal extended-mincost solution (lower curve) is significantly easier than computing the true optimal solution (upper curve). How much "better" are the true optimal solutions compared to the easy-to-find extended solutions? The right half of the figure shows the *relative utility gap %* between the solution

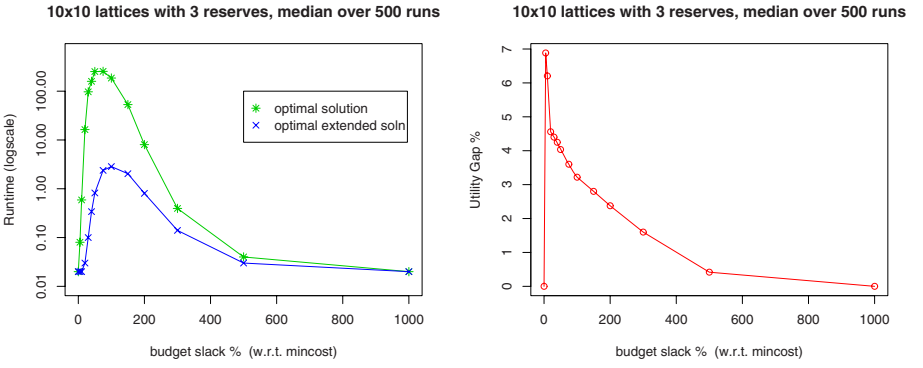


Fig. 1. Left: Hardness profile (runtime, log-scale) for lattices of order 10 with 3 terminal nodes; upper curve: optimal solution; lower curve: optimal extended-mincost solution. Right: Percentage gap in the utility of optimal and extended-mincost solutions.

qualities (i.e., attained utilities) in the two cases, defined as $100 \times (optimal - extended)/optimal$. We see that when budget equals mincost, both optimal and extended solutions have similar quality. The gap between the qualities reaches its maximum shortly thereafter, and then starts to decrease rapidly, so that the extended solution at 100% budget slack is roughly 3.2% worse than the optimal solution for order 10 grids, and at 500% budget slack, only around 0.4% worse.

For the second set of experiments, we used real data for the design of a wildlife conservation corridor for grizzly bears in the Northern Rockies, connecting the Yellowstone, Salmon-Selway, and Northern Continental Divide Ecosystems in Idaho, Wyoming, and Montana. To measure the utility of each parcel, we use grizzly bear habitat suitability data [1]. The cost is taken to be the land value estimate provided by the U.S. Department of Agriculture. We experimented with various granularities for the problem, going from County level regions down to 5 km $\times$ 5 km square grid regions. Going to finer granularities reduces the cost of the **cheapest corridor** from \$1.9 B for the County level, to \$1 B for a 40 km square grid, to as low as \$11.8 M for the 5 km grid. Using a 25 square km *hexagonal grid* allows for better connectivity than the 5 km $\times$ 5 km square grid, since each hexagonal parcel is connected to 6 other parcels rather than 4, and results in a further decrease in cost to only \$7.3 M. A hexagonal grid also yields a wider corridor on average. As the granularity of the parcels is increased, the problem size grows rapidly. For example, while the County level abstraction has only 67 parcels, the 40 km square grid already has 242 parcels, and the 25 square km hexagonal grid has 12,889 parcels. As a result, solving the connection subgraph model in a naïve manner (as in [2]) using the Cplex solver quickly becomes infeasible: in fact, Cplex even had difficulty finding *any* feasible solution at all for a 40 km square grid or finer.

The left half of Figure 2 shows the relative gap between the optimal and extended solution utilities for the 40 km square abstraction (both were solved

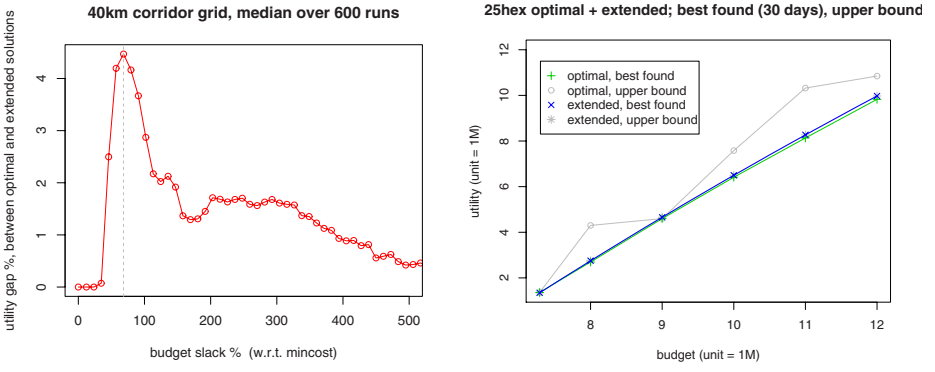


Fig. 2. Left: Utility gap % of optimal and extended-mincost solutions for 40 km grid. Right: Best found optimal and extended-mincost solutions for the 25 sq. km hexagonal grid, 30 day cutoff. Upper bound computed from the optimality gap reported by Cplex.

optimally). The relative gap is under under 5% when it is at its peak, and is usually within 2% of the optimal. This suggests that for this problem, one does not lose too much by solving only for the extended-mincost solution.

The right plot in Figure 2 depicts results on our best grid: the 25 square km hexagonal grid. This grid is significantly harder to solve. While the County level and the 50 km square grid were solved to optimality within seconds, even the extended-mincost solution for the hexagonal grid could not be solved optimally in over 10 days. Fortunately, the eventual optimality gap for the best extended-mincost solutions found after 30 days was only 0-0.07% (the "best found" curve for extended solutions is visually right on top of the corresponding "upper bound" curve). The best true optimal solutions, on the other hand, had an optimality gap of up to 27% (in one case 59%), as seen from the top curve. Interestingly, the best extended solutions found in this case were in fact of better quality than the best optimal solutions found (the green line is slightly *lower* than the blue line). This is in line with the concept of streamlining, where restricting the problem to only extended-mincost corridors allowed Cplex to compute better quality solutions within a limited amount of computation time.

References

[1] CERI. Grizzly bear habitat sustainability data, Craighead Environmental Research Institute, Bozeman, MT (2007)
[2] Conrad, J., Gomes, C.P., van Hoeve, W.-J., Sabharwal, A., Suter, J.: Connections in networks: Hardness of feasibility versus optimality. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 16–28. Springer, Heidelberg (2007)
[3] ILOG, SA. CPLEX 10.1 reference manual (2006)
[4] Prömel, H.J., Steger, A.: The Steiner Tree Problem: A Tour Through Graphs, Algorithms, and Complexity. Vieweg (2002)

Efficient Haplotype Inference with Combined CP and OR Techniques

Ana Graça[1], João Marques-Silva[2], Inês Lynce[1], and Arlindo L. Oliveira[1]

[1] IST/INESC-ID, Technical University of Lisbon, Portugal
{assg,ines}@sat.inesc-id.pt,aml@inesc-id.pt
[2] School of Electronics and Computer Science, University of Southampton, UK
jpms@ecs.soton.ac.uk

Abstract. Haplotype inference has relevant biological applications, and represents a challenging computational problem. Among others, pure parsimony provides a viable modeling approach for haplotype inference and provides a simple optimization criterion. Alternative approaches have been proposed for haplotype inference by pure parsimony (HIPP), including branch and bound, integer programming and, more recently, propositional satisfiability and pseudo-Boolean optimization (PBO). Among these, the currently best performing HIPP approach is based on PBO. This paper proposes a number of effective improvements to PBO-based HIPP, including the use of lower bounding and pruning techniques effective with other approaches. The new PBO-based HIPP approach reduces by 50% the number of instances that remain unsolvable by HIPP based approaches.

1 Introduction

Haplotype inference is a challenging computational problem, with a significant number of applications in genetics. Current DNA sequencing technology is not able to sequence independently the two copies of each chromosome which define the genetic inheritance of each diploid organism, such as humans. However, diagnosis and prevention of genetically related diseases requires, in many cases, the identification of the exact DNA sequences of each chromosome. This leads to the development of computational methods that can infer the haplotypes from the now easily obtained genotype information.

Over the last few years, Boolean satisfiability (SAT) and pseudo-Boolean optimization (PBO) techniques have been used to speed up one particular haplotype inference approach, based on pure parsimony [4]. Despite the success, the haplotype inference by pure parsimony (HIPP) problem is computationally hard, and there are several test cases that no HIPP solver is able to tackle. As a result, either alternative criteria or approximate algorithms are commonly used. With the objective of generalizing the use of HIPP solvers in haplotyping, it is important to increase the robustness of HIPP solvers, by increasing the number of instances HIPP solvers can solve efficiently. This paper pursues this objective, and combines CP and OR techniques that further reduce the search space, thus being able to solve some of the most difficult problem instances.

The paper is organized as follows. Section 2 introduces the HIPP problem. Section 3 describes the PBO-based HIPP approach, RPoly, and section 4 introduces the new techniques for improving the RPoly model. Afterwards, experimental results show that the new PBO model is able to solve a larger number of problem instances.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 308–312, 2008.

2 Haplotype Inference by Pure Parsimony (HIPP)

A haplotype is as a sequence of single nucleotide polymorphisms (SNPs) within a single chromosome. SNPs correspond to DNA nucleotides where mutations have occurred. Hence, for sites in the chromosome corresponding to SNPs we may either have the wild type (represented by 0) or the mutant type (represented by 1). Genotypes represent the conflated data contained in haplotypes. Each genotype is explained by two haplotypes. Unlike haplotypes, genotypes may be obtained using sequencing techniques.

Haplotype inference is the problem of identifying a set of haplotypes that may explain a given set of genotypes. A formal definition follows.

Definition 1. *Given a set of n genotypes $\mathcal{G}$, each genotype $g \in \mathcal{G}$ is represented by a string of size m over the alphabet $\{0, 1, 2\}$. The j^{th} element of the i^{th} genotype is referred to as g_{ij} with $1 \leq i \leq n$ and $1 \leq j \leq m$. Genotype g_i is heterozygous at site j if $g_{ij} = 2$ and is homozygous if $g_{ij} = 0$ or $g_{ij} = 1$. The* haplotype inference problem *consists in identifying a set of n pairs of haplotypes $\mathcal{H}$, not necessarily disjoint, with each haplotype h being represented by a string of size m over the alphabet $\{0, 1\}$, such that each pair of haplotypes explains a given genotype. A pair of haplotypes (h_i^a, h_i^b) is said to explain a genotype g_i ($g_i = h_i^a \otimes h_i^b$) if the following holds (with $1 \leq j \leq m$):*

$$\begin{aligned} h_{ij}^a = h_{ij}^b = 0, \ \textit{if } g_{ij} = 0; \\ h_{ij}^a = h_{ij}^b = 1, \ \textit{if } g_{ij} = 1; \\ h_{ij}^a = 1 - h_{ij}^b, \ \textit{if } g_{ij} = 2. \end{aligned}$$

It is clear that there is some freedom when selecting pairs of haplotypes for explaining genotypes with more than one site with value 2. For example, genotype 022 may be explained either by the pair of haplotypes (001,010) or by the pair of haplotypes (000,011). However, there is a biological motivation for selecting among the possible solutions to a set of genotypes the one with the smallest number of distinct haplotypes. Given that individuals from the same population have common ancestors and that mutations do not occur often, it is expected that individuals from the same population share a significant percentage of haplotypes.

Definition 2. *The approach that restricts the solutions to the haplotype inference problem such that the required number of haplotypes is minimum is called* pure parsimony *[4]. Finding a solution with a minimum number of haplotypes is a NP-hard problem [5].*

3 RPoly: A Pseudo-boolean HIPP Model

The most well-known tools for solving the HIPP problem can be divided into four categories: (i) RTIP [4], PolyIP [1] and HybridIP [1] are integer linear programming (ILP) formulations, (ii) Hapar [8] is a branch and bound algorithm, (iii) SHIPs [6] is a SAT-based model for the HIPP problem and (iv) RPoly [3] is a pseudo-Boolean model.

The pseudo-Boolean optimization model, referred to as *Reduced Poly model (RPoly)* [3], is currently the best performing algorithm for the HIPP problem. RPoly is based on the PBO model for PolyIP and further enhanced with key optimizations.

The RPoly model associates two haplotypes, h_i^a and h_i^b, with each genotype g_i, and these haplotypes are required to explain g_i. Moreover, RPoly associates a variable t_{ij} with each heterozygous site g_{ij}, such that $t_{ij} = 1$ indicates that $h_{ij}^a = 1$ and $h_{ij}^b = 0$, whereas $t_{ij} = 0$ indicates that $h_{ij}^a = 0$ and $h_{ij}^b = 1$. The values of h_i^a and h_i^b at homozygous sites are implicitly assumed.

Furthermore, let $x_{i\,k}^{p\,q}$, with $p, q \in \{a, b\}$ and $1 \leq k < i \leq n$, be 1 if haplotype p of genotype g_i and haplotype q of genotype g_k are different. The conditions on the $x_{i\,k}^{p\,q}$ variables are based on the values of variables t_{ij} and t_{kj} for heterozygous sites.

Moreover, two genotypes are said to be *incompatible* if there exists a site for which the value of one genotype is 0 and the other is 1; otherwise they are *compatible*. Clearly, candidate haplotypes for each genotype are related with candidate haplotypes for other genotypes only if the two genotypes are compatible. Then, incompatible genotypes g_i and g_k are guaranteed not to be explained by the same haplotype and so the value of $x_{i\,k}^{p\,q}$ is 1 for the four possible combinations of p and q.

In addition, the model uses variables u to denote whether one of the haplotypes, associated with a given genotype, is different from all previous haplotypes. Hence, u_i^p, with $p \in \{a, b\}$ and $1 \leq i \leq n$, is 1 if haplotype p of genotype g_i is different from all previous haplotypes. Then, the conditions on the u_i^p variables are based on the conditions for the $x_{i\,k}^{p\,q}$ variables, with $1 \leq k < i$ and $q \in \{a, b\}$.

Finally, the cost function minimizes the number of distinct haplotypes used, which is given by the sum of variables u_i^p. The next section describes new improvements to the RPoly model, which allow significant additional performance improvements.

4 Optimizations to the RPoly Model

This section describes optimizations to the RPoly model, the state of the art HIPP solver. The resulting model is called New RPoly (NRPoly for short).

The first optimization consists in integrating the lower bounds of SHIPs [6,7] in the NRPoly model. SHIPs is a SAT-based HIPP approach that, starting from a lower bound on the number of haplotypes, generates a SAT instance for each candidate number of haplotypes. SHIPs most recent lower bound procedure [7] provides a list of genotypes with an indication of the contribution of each genotype to the lower bound. Each genotype either contributes with +2, indicating that 2 new haplotypes will be required for explaining this genotype, or with +1, indicating that 1 new haplotype will be required for explaining this genotype.

In practice, for each genotype with an associated haplotype, the corresponding u variable, denoting whether a haplotype used for explaining a genotype is different from the haplotypes considered so far, is assigned value 1, and the clauses used for constraining the value of u *need not* be generated. The NRPoly model needs to be generated in such a way that the first genotypes correspond to genotypes used in the lower bound.

Similarly to the advantages of using lower bounds in SHIPs, the integration of lower bounds in NRPoly offers a few relevant advantages. First, several variables u become fixed with value 1, allowing the solver to focus on the remaining variables. Second, the size of the generated PBO problem instances is significantly reduced. The integration of lower bound information can reduce the generated PBO instances up to a factor of 3.

The second optimization is based on a key simplification introduced in the RTIP model [4], which consists in not considering all pairs of haplotypes that can explain a genotype. If a genotype can be explained by a pair of haplotypes such that none of these two haplotypes can explain any other genotype, then this pair of haplotypes needs not be considered.

Inspired by the pruning in RTIP, new constraints can be added to the NRPoly model. First, observe that each genotype that is not incompatible with all other genotypes must be explained by at least one haplotype that also explains some other genotype. Therefore, if a genotype g_i is explained by a pair of haplotypes (h_i^a, h_i^b) such that neither h_i^a nor h_i^b have been used to explain a genotype with lower index, then at least one of the haplotypes, h_i^a or h_i^b, must be used to explain one of the genotypes with higher index.

Consider genotypes compatible with at least one other genotype in $\mathcal{G}$. Define the predicate $\kappa(i,k)$ to be true if g_i and g_k are compatible. Formally, for all $1 \leq i \leq n$ such that g_i is compatible with at least another genotype in $\mathcal{G}$:

$$\text{If } u_i^a \wedge u_i^b, \text{ then } \exists_{k>i,\kappa(i,k)} \exists_{p,q\in\{a,b\}} \neg x_{k\,i}^{p\,q}. \tag{1}$$

Finally, an additional improvement consists in enriching the model with cardinality constraints on the x variables. For many combinatorial problems, adding new constraints to a model prunes the search and it is therefore likely to contribute to the solver being more efficient at finding solutions.

Clearly, unless genotypes g_i and g_k are equal, they cannot be explained by the same pair of haplotypes. Therefore, two different genotypes must be explained by at most one common haplotype. In practice, this constraint is integrated in the model by adding cardinality constraints on the variables x which capture the number of distinct haplotypes used to explain a pair of genotypes. Moreover, for incompatible pairs of genotypes, the constraint on the x variables is automatically guaranteed. Hence, for each pair of distinct non-homozygous compatible genotypes, at least three of their four pairwise haplotypes must be different:

$$\text{If } \kappa(i,k) \wedge g_i \neq g_k \wedge \exists_{j,j'}(g_{ij} = 2 \wedge g_{ij'} = 2), \text{ then } \sum_{p,q\in\{a,b\}} x_{i\,k}^{p\,q} \geq 3. \tag{2}$$

5 Experimental Results

A comprehensive evaluation was performed, using a set of 1183 problem instances (described in [3]), that include real and artificially generated problem instances. NRPoly has been compared against the other HIPP solvers. NRPoly uses the PBO solver MiniSat+ [2]. For the models using ILP, CPLEX version 11 was used. All HIPP solvers were run on a Intel Xeon 5160 server (3.0GHz, 1333Mhz, 4GB) running Red Hat Linux.

Figure 1 (left) provides a table with the number of aborted instances by NRPoly and the other HIPP algorithms, including the approaches in which NRPoly has been directly inspired: RTIP, SHIPs and RPoly. The total number of instances not solved within the time limit of 1000 seconds is given for each solver. We should note, however, that for RTIP many of the aborted instances exhausted the memory resources before the time limit. For SHIPs, the most recent version [7], which includes the lower bound used

Algorithms	# Aborted
NRPoly	**18**
RPoly	36
SHIPs	67
RTIP	378
Hapar	603
HybridIP	708
PolyIP	709

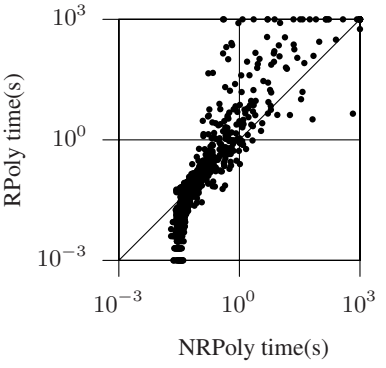


Fig. 1. Instances aborted by HIPP solvers within 1000s and performance of RPoly *vs* NRPoly

by NRPoly, was considered. As can be concluded, the HIPP algorithms based on SAT or PBO are the most effective. NRPoly is the most robust algorithm aborting only 18 problem instances, thus reducing in half the number of instances aborted by RPoly.

Figure 1 (right) compares NRPoly with the best performing tool RPoly. For very easy instances RPoly is clearly faster (mainly due to the additional constraints of NRPoly) but for difficult instances NRPoly is consistently faster. There is only one exception for one problem instance that RPoly is able to solve a few seconds before the timeout and NRPoly is not. However, we have observed that NRPoly would be able to solve the same instance if it was allowed a few more seconds. Overall, we may conclude that NRPoly is more robust and more effective on solving the hardest instances.

Acknowledgments This work is partially funded by FCT research projects POSC/EIA/61852/2004 and PTDC/EIA/64164/2006 and PhD grant SFRH/BD/28599/2006.

References

1. Brown, D., Harrower, I.: Integer programming approaches to haplotype inference by pure parsimony. IEEE/ACM Transactions on Computational Biology and Bioinformatics 3(2), 141–154 (2006)
2. Eén, N., Sörensson, N.: Translating pseudo-Boolean constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation 2, 1–26 (2006)
3. Graça, A., Marques-Silva, J., Lynce, I., Oliveira, A.: Efficient haplotype inference with pseudo-Boolean optimization. In: Anai, H., Horimoto, K., Kutsia, T. (eds.) Ab 2007. LNCS, vol. 4545, pp. 125–139. Springer, Heidelberg (2007)
4. Gusfield, D.: Haplotype inference by pure parsimony. In: Baeza-Yates, R., Chávez, E., Crochemore, M. (eds.) CPM 2003. LNCS, vol. 2676, pp. 144–155. Springer, Heidelberg (2003)
5. Lancia, G., Pinotti, C., Rizzi, R.: Haplotyping populations by pure parsimony: complexity of exact and approximation algorithms. INFORMS Journal on Computing 16, 348–359 (2004)
6. Lynce, I., Marques-Silva, J.: Efficient haplotype inference with Boolean satisfiability. In: National Conference on Artificial Intelligence (AAAI) (2006)
7. Lynce, I., Marques-Silva, J., Prestwich, S.: Boosting haplotype inference with local search. Constraints 13(1) (2008)
8. Wang, L., Xu, Y.: Haplotype inference by maximum parsimony. Bioinformatics 19(14), 1773–1780 (2003)

Integration of CP and Compilation Techniques for Instruction Sequence Test Generation

Boris Gutkovich

Intel Corporation, Haifa, Israel
boris.gutkovich@intel.com

Abstract. This paper gives a short description of a novel approach to automatic generation of high probing random instruction sequences that are used in validation of processors. The approach is based on integration of constraint programming (CP) with techniques from operations research (OR) and compilation of programming languages.

1 Introduction

Simulation-based validation is one of the main methodologies currently used to verify processors. Validation engineers put much effort in developing and maintaining test programs to cover different scenarios of the processor behavior. Much more effort is required to specify and cover the extremely improbable events in the modern processors where potential bugs usually reside [1]. Automatic Test Generation (ATG) tools are being developed and used in order to make this possible. Constraint-driven random generation is a technology by means of which these tools are able to create high-probing tests. A comprehensive overview article can be found in [2].

The complexity of the test generation process prevents it from being represented and solved as a single constraint solving problem (CSP). Usually the process is performed iteratively. At each stage the architectural state is maintained by the reference simulator along with different kinds of tracking servers and different CSP models are formulated for each instruction generation [3]. Such CSP model is solved by Instruction Generator (IG). In general case it comprises three main sub-models:

1. The architectural specification model, which is stable during the process of the whole test generation.
2. The architectural state model, which is incrementally changed after generation of each instruction.
3. The test scenario model, which can be stable for a snippet of the instruction sequence or may change from one instruction to another.

The described instruction-by-instruction generation flow is a compromise that makes it possible to handle a very complex problem. The main deficiency in this flow comes from the difficulties with handling cross-instruction constraints. Only a very small subset of such constraints can be relatively easily modeled by means of the architectural state. To deal with the cross-instruction constraints (or constraints on a sequence of instructions) the multi-level solving scheme can be used:

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 313–317, 2008.

1. Create CSP model that represents only constraints on a sequence. This model does not contain IG constraints listed above.
2. Find a random solution (or a set of random solutions) for this model.
3. Transform the solution to constraints that are added to each single instruction CSP.
4. Use IG to find random solutions for these CSP models. In case of failure return to step 2 with fixed values for the accepted solutions and try to find another solution for the sequence CSP.

The main problem with the described algorithm is the loop from step 4 to step 2 which causes substantial performance degradation. To avoid this loop we fulfilled analysis of typical sequence constraints in the domain of ATG for processors and developed one-pass algorithms for a few special cases of constraints. This paper studies the case of a sequence which can be described by constraints on arithmetic (or Boolean) expressions.

2 Instruction Sequence from Constraints on Expression: Example

Sometimes there is a need to produce a sequence of instructions that leads to specific result, e.g. a particular data value in a register or in the memory. Often this value may be required to trigger an architectural exception event. For example, one may wish to generate an interesting sequence that leads to arithmetic overflow. Constraint on expression is also used in generation of self-check tests. For example, we may want to generate an interesting sequence that checks the most famous (or infamous) FDIV bug in Intel Pentium floating-point (FP) unit (FPU).

Let us look at the latter use-case in more detail. It is known now that one of the simplest ways to check the FDIV bug is to execute FDIV instruction on the particular operands, e.g. 4195835.0/3145727.0. It might also be 5505001.0/294911.0 or 8391667.0/1572863.0, etc. The incorrect behavior can be detected even with the single precision of FP calculations. For example, the correct answer in the first case should be no less than 1.3338. We are also aware from the validation experience that uncovered bugs often reside in the vicinity of the known bugs. So it might be useful to generate an interesting sequence of instructions that creates the required values on the FPU stack and then to execute FDIV on these values. A method to create such sequence is described in the next section.

3 Instruction Sequence from Constraints on Expression: Method

The idea of the method is based on the multi-stage solving scheme. It can be outlined as follows:

1. Produce an arithmetic (or Boolean) expression, defined on a set of variables.
2. Solve the constraint, implying that the expression must be evaluated to the required value. Assign the solution values to the variables of the expression.
3. Compile the expression to a set of constraints for each individual instruction of the sequence
4. Generate Assembler instructions by means of Instruction Generator

Let us demonstrate the method on the use-case of testing the FDIV bug (see the previous section). On the first step we have a choice of many different arithmetic expressions. The particular one can be picked randomly from the Test Knowledge Base (TKB) or it can be written by a validation engineer in the test-scenario spec. Suppose quadratic and linear polynomials were chosen to create the required operands of FDIV instruction. Therefore, we may have the following constraints to be sent to the dedicated solver, which can work in the domain of FP numbers:

a*x**2 + b*x + c == 4195835.0 a*x + b*x + c == 3145727.0

It is up to the sequence generator to decide upon the following questions:

- Which elements of the polynomials are constants and which are constrained variables (CV)?
- Do we have two independent constraint solving problems or a combined one - a system of the equations?
- Do these polynomials share some of the variables?

Usually the best choice is to leave as much freedom as possible to the solver, but sometimes the restrictions of the available solvers, the performance considerations or constraints external to the instruction sequence may require making some fixed decisions. For example we may need to set the values of the coefficients to the values that already reside in the registers. Such dependency, which links one part of the test with the other parts, usually helps making interesting tests.

Let us suppose the sequence generator decided to handle the two constraints, given above, independently. It also decided to handle *a, b* and *x* as the floating-point CV with appropriate domains, but to fix the coefficient *c* to the 0.0 constant. One of the possible solutions we can get from a solver (in our case it was ILOG Solver [4]) is:

a = 2.48266683628885e-006	a = 2.41978999989436
b = 2.72288327722903e-010	b = 1.05637770432692e-010
x = 1300019.83683147	x = 1300000.00001892

The first part of the possible sequences of constraints, generated at step 3 of the proposed flow, is given in the Appendix. The constraints are represented in IG constraint language. All constraints require generating instructions for FPU. It is important to note that this sequence of constraints can be easily mixed with a set of constraints to generate an instruction sequence which does not use FPU, e.g. SIMD or integer instruction sequences. Such mixture will have a good opportunity in finding bugs of micro-architecture implementation.

4 Feasibility of the Proposed Method

What are the conditions that should be met to make the proposed method successful?

First of all the repertoire of expressions should be consistent with the repertoire of solvers. In case of FP, solver must be able to handle the required precision of the real numbers while finding a solution for the chosen equations. It should avoid falling into pitfalls of the floating-point calculations. More details on such pitfalls can be found in [5]. In our experiments, we used ILOG Solver [4] and ILOG CPLEX [6]. Both

libraries treat FP values in the double-precision mode. On top of these libraries we developed a set of specialized solvers. These solvers were tuned to handle a specific domain (FP of different precision modes, including the extended-precision mode, and integers of different lengths). These solvers were also tuned to a specific class of expressions. A set of expressions for TKB consists of two subsets. The first subset was created manually; the second one was generated automatically by means of the Prolog-based expression generator. A multi-valued decision tree was built to link expressions and solvers.

The sequence generator must have knowledge about capabilities of the instruction generator. It should solve constraints on a sequence in such a way that guarantees or maximizes the probability to achieve the final solution. This is the common requirement for the multi-stage solving schema. In our particular case of the instruction sequence for expressions, there exists a rather simple way to obey this requirement. The constraint generator actually works as a code generator of a compiler for expressions. The regular compiler guarantees by construction that generated Assembler code running under specific OS will produce expected result on the target machine. The constraint generator can guarantee the generated sequence of constraints will produce such Assembler code with the support from the available Instruction Generator.

Let us demonstrate the last statement on the "FDIV bug" example given above. The constraint generator was asked to generate commands for Intel x87 FPU. It may decide to evaluate the expression as a sequence of operations on FPU stack. From the expression solver it receives the correct data values which guarantee the evaluation will not cause any numeric exceptions. The constraint generator knows that the request to load these data values from memory to FPU stack can be handled by Instruction Generator without problems. To prevent the stack overflow the constraint generator must be capable to calculate the maximal number of the stack registers for the expression evaluation. Given the number of the valid stack entries before the evaluation, it can choose the generation strategy. For example, if there is enough free entries in the stack, the generator can convert the expression into RPN form and directly use this form as a template for constraint generation. In other case the generator will use another compilation strategy, e.g. it can use the general purpose registers to store the temporary results.

References

1. Bentley, B.: High Level Validation of Next-Generation Microprocessors. In: Proc. of the 7th IEEE International Workshop on High Level Design Validation and Test (October 2002)
2. Naveh, Y., et al.: Constraint-Based Random Stimuli Generation for Hardware Verification. AI Magazine 28(3), 13 (2007)
3. Gutkovich, B., Moss, A.: CP with Architectural State Lookup for Functional Test Generation. In: 11th Annual IEEE International Workshop on High Level Design Validation and Test, pp. 111–118 (2006)
4. ILOG Solver 6.5 Reference Manual (October 2007)
5. Aharoni, M., et al.: FPgen - A Test Generation Framework for Datapath Floating-Point Verification. In: Proc. of the 8th IEEE International Workshop on High Level Design Validation and Test, pp. 17–22 (2003)
6. ILOG CPLEX C++ API 11.0 Reference Manual (2007)

Appendix: A Sequence of Constraints for Expression Evaluation

```
// a*x**2 + b*x + c
fp(load, memory, 2.48266683628885e-006)
fp(load, memory, 1300019.83683147)
fp(mult, pop)
fp(load, memory, 1300019.83683147)
fp(mult, pop)
fp(load, memory, 2.72288327722903e-010)
fp(load, memory, 1300019.83683147)
fp(mult, pop)
fp(add, pop)
fp(load, 0.0)
fp(add, pop)
```

Propagating Separable Equalities in an MDD Store

T. Hadzic[1], J.N. Hooker[2], and P. Tiedemann[3]

[1] University College Cork
t.hadzic@4c.ucc.ie
[2] Carnegie Mellon University
john@hooker.tepper.cmu.edu
[3] IT University of Copenhagen
petert@itu.dk

Abstract. We present a propagator that achieves MDD consistency for a separable equality over an MDD (multivalued decision diagram) store in pseudo-polynomial time. We integrate the propagator into a constraint solver based on an MDD store introduced in [1]. Our experiments show that the new propagator provides substantial computational advantage over propagation of two inequality constraints, and that the advantage increases when the maximum width of the MDD store increases.

In [1] we proposed a width-limited *multivalued decision diagram* (MDD) as a general constraint store for constraint programming. We demonstrated the potential of MDD-based constraint solving by developing MDD-propagators for *alldiff* and *inequality* constraints. In this paper, we describe an MDD-propagator for the *separable equality* constraint that uses a pseudo-polynomial algorithm to achieve *MDD consistency*. We show the computational advantage of the new propagator over the existing approach of modeling equalities with two inequality propagators.

Preliminaries. A *constraint satisfaction problem* is specified with a set of constraints $\mathcal{C} = \{C_1, \ldots, C_m\}$ on the variables $X = \{x_1, \ldots, x_n\}$ with respective finite domains $D_1, \ldots, D_n$. An MDD M is a tuple (V, r, E, var, D), where V is a set of vertices containing the special terminal vertex 1 and a root $r \in V$, $E \subseteq V \times V$ is a set of edges such that (V, E) forms a directed acyclic graph with r as the source and 1 as the sink for all maximal paths in the graph. Further, $var : V \rightarrow \{1, \ldots, n+1\}$ is a labeling of all nodes with a variable index, with $var(1) = n + 1$. D is a set containing an *edge domain* D_{uv} for each edge (u, v). We require that $\emptyset \neq D_{uv} \subseteq D_{var(u)}$ for all edges in E, and for convenience we take $D_{uv} = \emptyset$ if $(u, v) \notin E$. We work only with *ordered* MDDs. A total ordering $<$ of the variables is assumed, and all edges (u, v) respect the ordering; that is, $var(u) < var(v)$. For convenience, we assume that the variables in X are ordered according to their indices. Ordered MDDs can be viewed as arranged in n *layers* of vertices, with the vertices on each layer labeled with the same variable index. The *width* k of the MDD is the size of the largest layer. While MDDs

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 318–322, 2008.

in general allow edges to skip layers, for the simplicity of representation in this paper we consider only MDDs without long edges; that is, for each $(u, v) \in E$, $var(v) = var(u) + 1$. Thus, if an $r \to 1$ *path* is defined to be a path $u_1, \ldots, u_{n+1}$ in which $u_1 = r$ and $u_{n+1} = 1$, then each $r \to 1$ path *represents* the subset of solutions $\prod_{i=1}^{n}(D_{u_i u_{i+1}})$. Let C be a constraint on the variables $\{x_1, \ldots, x_n\}$. For a given MDD M we have a notion of consistency that generalizes the well known *generalized arc consistency* (GAC) [2].

Definition 1 (MDD consistency). *A constraint C is* MDD consistent *with respect to M if, for every edge $(u, v) \in E$ with $i = var(u)$ and every value $\alpha_i \in D_{uv}$, there exists a tuple $(\alpha_1, \ldots, \alpha_n)$ satisfying C that is represented by an $r \to 1$ path passing through (u, v).*

1 A Propagator for the *Separable Equality* constraint

Unlike a standard domain-store propagator, which is specified only by the way it *prunes* infeasible values from a domain-store, an MDD-store propagator also *refines* the MDD representing the store by adding new vertices and edges. We develop such a propagator for the *separable equality* constraint, which for a set of unary functions $f_1, \ldots, f_n$ and a constant c is defined as

$$f_1(x_1) + f_2(x_2) + \ldots + f_n(x_n) = c. \tag{1}$$

1.1 Pruning

One simple way to perform pruning on the constraint (1) is to do so for the two inequality constraints $\sum_{i=1}^{n} f_i(x_i) \leq c$ and $\sum_{i=1}^{n} f_i(x_i) \geq c$. We can achieve MDD consistency in linear time in the size of the MDD for each of these separately, using the inequality propagator described in [1]. Yet this ensures only that each remaining edge is on a shortest path with cost at most c and on a longest path with cost at least c. It therefore does not achieve MDD consistency for the equality constraint.

To achieve MDD consistency we use the following procedure. In the first phase, for each node u the algorithm computes the cost $L_{down}(u)$ of the cheapest path and the cost $H_{down}(u)$ of the most expensive path leading from u to the terminal. In the second phase it marks the edges in the MDD store that are on at least one $r \to 1$ path representing a solution of the constraint. In the final phase all unmarked edges are removed from the MDD.

The pseudo-code for the algorithm Mark-Support implements the second phase with a dynamic programming recursion and is shown in Figure 1. It is initially invoked on the root r and the right-hand side c. When invoked on a node u it searches for a path through the MDD from u with the given cost. For each edge (u, u') and $\alpha \in D_{u,u'}$, the algorithm recursively checks if there exists a path of cost $c - f_{var(u)}(\alpha)$ from u' to the terminal, and the result of this query is cached as $cache(u', c - f_{var(u)}(\alpha))$. If the result is positive the edge is marked to indicate that it must not be pruned. For each node u the previously computed

$L_{down}(u)$ and $H_{down}(u)$ values provide an early cutoff, because there can be no path of cost c from u if the cheapest path from u is too expensive or the most expensive path is too cheap. Note that if the width of the MDD store is 1 then the algorithm is essentially the domain store filter of [3].

```
MARK-SUPPORT(u, cost)
 1  if cache(u, cost) ≠ UNKNOWN
 2     then return cache(u, cost)
 3     else if L_down(u) > cost ∨ H_down(u) < cost
 4             then return false
 5     else cache(u, cost) ← false
 6          for (u, u') ∈ E and α ∈ D_{u,u'}
 7              do if MARK-SUPPORT(u', cost − f_{var(u)}(α))
 8                    then marked ← marked ∪ (u, u', α)
 9                         cache(u, cost) ← true
10          return cache(u, cost)
```

Fig. 1. The algorithm shown above, initially invoked as MARK-SUPPORT(r, c) for a constraint $\sum_{1 \leq i \leq n} f_i(x_i) = c$, ensures that an edge along with a value from its edge domain is in *marked* iff there is a path through that edge, using the corresponding value with cost exactly c

Complexity. The complexity of a propagation step is dominated by the execution of MARK-SUPPORT as the other phases can be done in linear time. Since each call to MARK-SUPPORT only does constant work in addition to the recursive calls, we can evaluate the time based on the number of recursive calls alone. A call to MARK-SUPPORT on a node u only results in recursive calls if the given cost has not been processed before for u. Let $L_{up}(u)$ and $H_{up}(u)$ be the cost of the cheapest and most expensive path from the root to u. Then an upper bounds on the number $q(u)$ of distinct costs that any node u will be queried for is $q(u) = c - L_{up}(u) - (c - H_{up}(u)) + 1 = H_{up}(u) - L_{up}(u) + 1$. Hence the total number of recursive calls over all nodes can be bounded by

$$\sum_{u \in V} q(u)|D_{var(u)}| = O\left(q(1) \max_{x_i \in X}\{|D_i|\}|V|\right) = O\left(q(1) \max_{x_i \in X}\{|D_i|\}nk\right).$$

Note that this bound increases linearly with the width k. This is very pessimistic, however, as a larger width makes the store a more precise approximation that allows fewer candidate solutions. This results in fewer paths to a given node, and therefore in most cases fewer distinct costs of these paths, which translates into fewer recursive calls per node. Thus, a larger width can *decrease* the time required to execute MARK-SUPPORT. Additionally, a more refined store will allow more edges to be pruned. Hence a larger width could be expected to reduce the overall solution time. We verify this behavior in our empirical results.

1.2 Refining

An MDD propagator refines the MDD through *node splitting* [1]. We first select a node $u \in V$ and create an isomorphic copy $u' \in V$ by copying every outgoing edge (u, t) into (u', t) along with all edge labels $D_{u',t} = D_{u,t}$. We then copy ingoing edges (s, u) into (s, u') along with a *subset* $D_{s,u'} \subseteq D_{s,u}$ of edge labels that are then removed from the original edges: $D_{s,u} \leftarrow D_{s,u} \setminus D_{s,u'}$. Fig. 2 shows an example of a node split and subsequent propagation for an *alldiff* constraint.

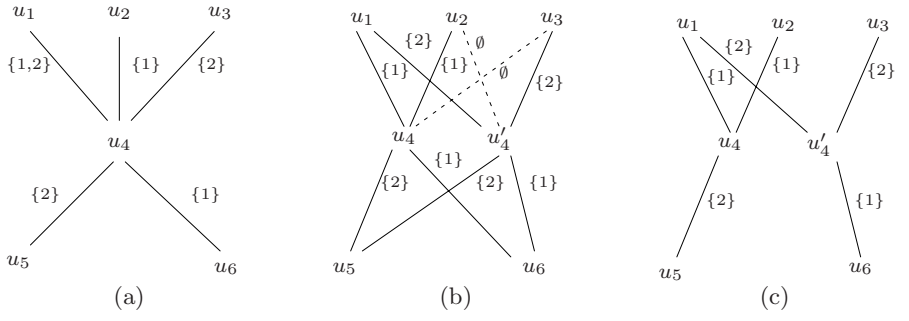


Fig. 2. (a) Part of an MDD store representing a relaxation of a global alldiff, just before splitting on the node u_4. Note that while there are obvious inconsistencies between the edge domains (such as label 1 in domains of (u_1, u_4) and (u_4, u_6)), we cannot remove any value. (b) A new node u'_4 has been created and some of the edge domain values to u_4 have been transferred to u'_4. There are no labels on (u_2, u'_4) and (u_3, u_4), so the edges need not be created. (c) After the split we can prune inconsistent values and as a result remove edges (u_4, u_6) and (u'_4, u_5).

Our splitting strategy selects a splitting node and a subset of incoming edges to be redirected by heuristically estimating the *quality* of the resulting split. For an *equality* constraint we try to increase the potential for subsequent pruning by maximizing the shortest path $L(u')$ and minimizing the longest path $H(u')$ passing through u'. In particular, we try to minimize the expected difference $H(u') - L(u')$. This splitting strategy is used with both the pseudo-polynomial pruning and pruning based on two inequalities.

2 Empirical Results

We randomly generated a number of problem instances involving 3 separable equalities over 15 variables with a domain of size 3. We measured the time necessary to find a solution using two inequality propagators and our equality propagator. For each x_i and $v \in D_i$, we randomly selected $f_i(v) \in [-10000, 10000]$. The results are shown in Figure 3. We can observe that enforcing the stronger MDD consistency through an equality constraint consistently outperforms the weaker consistency enforced by two inequalities. The computation time for two inequalities increases with larger width, while reducing for the new equality propagator.

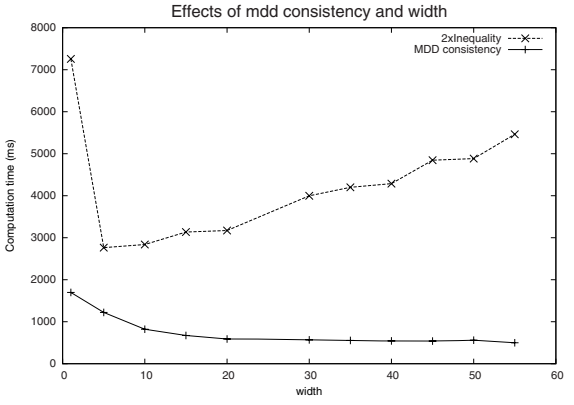


Fig. 3. The effect of MDD width (horizontal axis) on computation time (in ms, vertical axis) when using (a) two inequality propagators to propagate the equality constraint and (b) the MDD consistent equality propagator introduced in this paper

3 Conclusions and Future Work

We presented a propagator for the MDD store that achieves MDD consistency for a separable equality constraint in pseudo-polynomial time. From our empirical results we observed that the extra overhead is worthwhile in practice. In particular, the benefit increases as the width of the MDD store increases. In future work we intend to develop an approximate propagation scheme based on caching for small ranges of cost instead of a single precise cost.

References

1. Andersen, H.R., Hadzic, T., Hooker, J.N., Tiedemann, P.: A constraint store based on multivalued decision diagrams. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 118–132. Springer, Heidelberg (2007)
2. van Hoeve, W.J., Katriel, I.: Global constraints. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming. Foundations of Artificial Intelligence, pp. 169–208. Elsevier Science Publishers, Amsterdam (2006)
3. Trick, M.: A dynamic programming approach for consistency and propagation for knapsack constraints. In: Proceedings of the Third International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR-01), pp. 113–124 (2001)

The Weighted CFG Constraint

George Katsirelos, Nina Narodytska, and Toby Walsh

University of New South Wales and NICTA, Sydney, Australia

Abstract. We introduce the weighted CFG constraint and propose a propagation algorithm that enforces domain consistency in $O(n^3|G|)$ time. We show that this algorithm can be decomposed into a set of primitive arithmetic constraints without hindering propagation.

1 Introduction

One very promising method for rostering and other domains is to specify constraints via grammars or automata that accept some language. We can specify constraints in this way on, for instance, the number of consecutive night shifts or the number of days off in each 7 day period. With the REGULAR constraint [4], we specify the acceptable assignments to a sequence of variables by a deterministic finite automaton. One limitation of this approach is that the automaton may need to be large. For example, there are regular languages which can only be defined by an automaton with an exponential number of states. Researchers have therefore looked higher up the Chomsky hierarchy. In particular, the CFG constraint [8,6] permits us to specify constraints using any context-free grammar. In this paper, we consider a further generalization to the weighted CFG constraint. This can model over-constrained problems and problems with preferences.

2 The Weighted CFG Constraint

In a context-free grammar, rules have a left-hand side with just one non-terminal, and a right-hand side consisting of terminals and non-terminals. Any context-free grammar can be written in Chomsky form in which the right-hand size of a rule is just one terminal or two non-terminals. The weighted WCFG$(G, W, z, [X_1, \ldots, X_n])$ constraint holds iff an assignment X forms a string belonging to the grammar G and the minimal weight of a derivation of X less than or equal to z. The matrix W defines weights of productions in the grammar G. The weight of a derivation is the sum of production weights used in the derivation. The WCFG constraint is domain consistent iff for each variable, every value in its domain can be extended to an assignment satisfying the constraint.

We give a propagator for the WCFG constraint based on an extension of the CYK parser to probabilistic grammars [3]. We assume that G is in Chomsky normal form and with a single start non-terminal S. The algorithm has two stages. In the first, we construct a dynamic programing table $V[i, j]$ where an element A of $V[i, j]$ is a potential non-terminal that generates a substring $[X_i, \ldots, X_{i+j}]$. We compute a lower bound $l[i, j, A]$ on the minimal weight of a derivation from A. In the second stage, we move from $V[1, n]$ to the bottom of table V. For an element A of $V[i, j]$, we compute an

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 323–327, 2008.

upper bound $u[i, j, A]$ on the maximal weight of a derivation from A of a substring $[X_i, \ldots, X_{i+j}]$. We mark the element A iff $l[i, j, A] \leq u[i, j, A]$. The pseudo-code is presented in Algorithm 1. Lines 2–5 initialize l and u. Lines 6–16 compute the first stage, whilst lines 20–29 compute the second stage. Finally, we prune inconsistent values in lines 30–31. Algorithm 1 enforces domain consistency in $O(|G|n^3)$ time.

Algorithm 1. The weighted CYK propagator

```
1:  procedure WCYK-ALG(G, W, z, [X_1, ..., X_n])
2:      for j = 1 to n do
3:          for i = 1 to n − j + 1 do
4:              for each A ∈ G do
5:                  l[i, j, A] = z + 1; u[i, j, A] = −1;
6:      for i = 1 to n do
7:          V[i, 1] = {A|A → a ∈ G, a ∈ D(X_i)}
8:          for A ∈ V[i, 1] s.t A → a ∈ G, a ∈ D(X_i) do
9:              l[i, 1, A] = min{l[i, 1, A], W[A → a]};
10:     for j = 2 to n do
11:         for i = 1 to n − j + 1 do
12:             V[i, j] = ∅;
13:             for k = 1 to j − 1 do
14:                 V[i, j] = V[i, j] ∪ {A|A → BC ∈ G, B ∈ V[i, k], C ∈ V[i + k, j − k]}
15:                 for each A → BC ∈ G s.t. B ∈ V[i, k], C ∈ V[i + k, j − k] do
16:                     l[i, j, A] = min{l[i, j, A], W[A → BC] + l[i, k, B] + l[i + k, j − k, C]};
17:     if S ∉ V[1, n] then
18:         return 0;
19:     mark (1, n, S); u[1, n, S] = z;
20:     for j = n downto 2 do
21:         for i = 1 to n − j + 1 do
22:             for A such that (i, j, A) is marked do
23:                 for k = 1 to j − 1 do
24:                     for each A → BC ∈ G s.t. B ∈ V[i, k], C ∈ V[i + k, j − k] do
25:                         if W[A → BC] + l[i, k, B] + l[i + k, j − k, C] > u[i, j, A] then
26:                             continue;
27:                         mark (i, k, B); mark (i + k, j − k, C);
28:                         u[i, k, B] = max{u[i, k, B], u[i, j, A] − l[i + k, j − k, C] − W[A → BC]};
29:                         u[i + k, j − k, C] = max{u[i + k, j − k, C], u[i, j, A] − l[i, k, B] − W[A → BC]};
30:     for i = 1 to n do
31:         D(X_i) = {a ∈ D(X_i)|A → a ∈ G, (i, 1, A) is marked and W[A → a] ≤ u[i, 1, A]};
32:     return 1;
```

3 Decomposition of the Weighted CFG Constraint

As an alternative to this monolithic propagator, we propose a simple decomposition with which we can also enforce domain consistency. A decomposition has several advantages. For example, it is easy to add to any constraint solver. As a second example, decomposition gives an efficient incremental propagator, and opens the door to advanced techniques like nogood learning and watched literals. The idea of the decomposition is to introduce arithmetic constraints to compute l and u. Given the table V obtained by Algorithm 1, we construct the corresponding AND/OR directed acyclic graph (DAG) as in [7]. We label an OR node by $n(i, j, A)$, and an AND node by $n(i, j, k, A \rightarrow BC)$. We denote the parents of a node nd as $PRT(nd)$ and the children as $CHD(nd)$. For each node two integer variables are introduced to compute l and u. For an OR-node nd, these are $l_O(nd)$ and $u_O(nd)$, whilst for an AND-node nd, these are $l_A(nd)$, $u_A(nd)$.

For each AND node $nd = n(i, j, k, A \rightarrow BC)$ we post a constraint to connect nd to its children $CHD(nd)$:

$$l_A(nd) = \sum_{n_c \in CHD(nd)} l_O(n_c) + W[A \rightarrow BC] \quad (1)$$

For each OR node $nd = n(i, j, A)$ we post constraints to connect nd to its children $CHD(nd)$:

$$l_O(nd) = \min_{n_c \in CHD(nd)} \{l_A(n_c)\} \quad (2)$$

$$u_O(nd) = u_A(n_c),\ n_c \in CHD(nd) \quad (3)$$

For each OR node $nd = n(i, j, A)$ we post a set of constraints to connect nd to its parents $PRT(nd)$ and siblings:

$$u_O(nd) = max_{n_p \in PRT(nd)}\{u_A(n_p) - l_O(n_{sb}) - W[P]\}, \quad (4)$$

where $P = B \rightarrow AC$ or $B \rightarrow CA$, $n_p = n(r, q, t, P)$ is the parent of $nd = n(i, j, A)$ and $n_{sb} = n(i_1, j_1, C)$.

Finally, we introduce constraints to prune X_i. For each leaf of the DAG that is an OR node $nd = n(i, 1, a)$, we introduce:

$$a \in D(X_i) \Rightarrow 0 \leq l_O(nd) \leq z \quad (5)$$

$$a \notin D(X_i) \Leftrightarrow l_O(nd) > z \quad (6)$$

$$l_O(nd) > u_O(nd) \Rightarrow a \notin D(X_i) \quad (7)$$

As the maximal weight of a derivation is less than or equal to z we post:

$$u_O(n(1, n, S)) \leq z \quad (8)$$

Bounds propagation will set the lower bound of $l_O(n(i, j, A))$ to the minimal weight of a derivation from A, and the upper bound on $u_O(n(i, j, A))$ to the maximum weight of a derivation from A. We forbid branching on variables $l_{A|O}$ and $u_{A|O}$ as branching on $l_{A|O}$ would change the weights matrix W and branching on $u_{A|O}$ would add additional restrictions to the weight of a derivation. Bounds propagation on this decomposition enforces domain consistency on the WCFG constraint. If we invoke constraints in the decomposition in the same order as we compute the table V, this takes $O(n^3|G|)$ time. For simpler grammars, propagation is faster. For instance, as in the unweighted case, it takes just $O(n|G|)$ time on a regular grammar.

We can speed up propagation by recognizing when constraints are entailed. If $l_O(nd) > u_O(nd)$ holds for an OR node nd then constraints (4) and (2) are entailed. If $l_A(nd) > u_A(nd)$ holds for an AND node nd then constraints (1) and (3) are entailed. To model entailment we augmented each of these constraints in such a way that if $l_O(nd) > u_O(nd)$ or $l_A(nd) > u_A(nd)$ hold then corresponding constraints are not invoked by the solver.

4 The Soft CFG Constraint

We can use the WCFG constraint to encode a soft version of CFG constraint which is useful for modelling over-constrained problems. The soft $\text{CFG}(G, z, [X_1, \ldots, X_n])$ constraint holds iff the string $[X_1, \ldots, X_n]$ is at most distance z from a string in G. We consider both Hamming and edit distances. We encode the soft $\text{CFG}(G, z, [X_1, \ldots, X_n])$ constraint as a weighted $\text{CFG}(G', W, z, [X_1, \ldots, X_n])$ constraint. For Hamming distance, for each production $A \to a \in G$, we introduce additional unit weight productions to simulate substitution:

$$\{A \to b, W[A \to b] = 1 | A \to a \in G, A \to b \notin G, b \in \Sigma\}$$

Existing productions have zero weight. For edit distance, we introduce additional productions to simulate substitution, insertion and deletion:

$$\begin{gathered}\{A \to b, W[A \to b] = 1 | A \to a \in G, A \to b \notin G, b \in \Sigma\} \cup \\ \{A \to \varepsilon, W[A \to \varepsilon] = 1 | A \to a \in G, a \in \Sigma\} \cup \\ \{A \to Aa, W[A \to Aa] = 1 | a \in \Sigma\} \cup \\ \{A \to aA, W[A \to aA] = 1 | a \in \Sigma\}\end{gathered}$$

To handle ε productions we modify Alg. 1 so loops in lines (13),(23) run from 0 to j.

5 Experimental Results

We evaluated these propagation methods on shift-scheduling benchmarks [2,1]. A personal schedule is subject to various regulation rules, e.g. a full-time employee has to have a one-hour lunch. This rules are encoded into a context-free grammar augmented with restrictions on productions [7,5]. A schedule for an employee has $n = 96$ slots represented by n variables. In each slot, an employee can work on an activity (a_i), take a break (b), lunch (l) or rest (r). These rules are represented by the following grammar:

$$\begin{array}{ll} S \to RPR, f_P(i,j) \equiv 13 \leq j \leq 24, & P \to WbW,\ L \to lL|l, f_L(i,j) \equiv j = 4 \\ S \to RFR, f_F(i,j) \equiv 30 \leq j \leq 38, & R \to rR|r,\ W \to A_i, f_W(i,j) \equiv j \geq 4 \\ A_i \to a_i A_i | a_i, f_A(i,j) \equiv open(i), & F \to PLP \end{array}$$

where functions $f(i,j)$ are restrictions on productions and $open(i)$ is a function that returns 1 if the business is opened at ith slot and 0 otherwise. To model labour demand for a slot we introduce Boolean variables $b(i,j,a_k)$, equal to 1 if jth employee performs activity a_k at ith time slot. For each time slot i and activity a_k we post a constraint $\sum_{j=1}^{m} x(i,j,a_k) > d(i,a_k)$, where m is the number of employees. The goal is to minimize the number of slots in which employees worked.

We used Gecode 2.0.1 for our experiments and ran them on an Intel Xeon 2.0Ghz with 4Gb of RAM[1]. In the first set of experiments, we used the weighted $\text{CFG}(G, z_j, X)$, $j = 1, \ldots, m$ with zero weights. Our monolithic propagator gave similar results to the unweighted CFG propagator from [7]. Decompositions were slower than decompositions of the unweighted CFG constraint as the former uses integers instead of Booleans.

[1] We would like to thank Claude-Guy Quimper for his help with the experiments.

Table 1. All benchmarks have one-hour time limit. $|A|$ is the number of activities,m is the number of employees, $cost$ shows the total number of slots in which employees worked in the best solution, $time$ is the time to find the best solution, bt is the number of backtracks to find the best solution, BT is the number of backtracks in one hour, Opt shows if optimality is proved, Imp shows if a lower cost solution is found by the second model.

			Monolithic				Decomposition				Decomption+entailment					
$\|A\|$	#	m	cost	time	bt	BT	cost	time	bt	BT	cost	time	bt	BT	Opt	Imp
1	2	4	**107**	**5**	**0**	8652	**107**	7	**0**	5926	**107**	7	**0**	**11521**		
1	3	6	**148**	**7**	**1**	5917	**148**	34	**1**	1311	**148**	9	**1**	**8075**		
1	4	6	**152**	1836	**5831**	11345	**152**	**1379**	**5831**	**14815**	**152**	1590	**5831**	13287		
1	5	5	**96**	6	**0**	8753	**96**	6	**0**	2660	**96**	**3**	**0**	**45097**		
1	6	6	–	–	–	10868	**132**	3029	**11181**	13085	**132**	**2367**	**11181**	**16972**		
1	7	8	**196**	16	**16**	10811	**196**	18	**16**	6270	**196**	**15**	**16**	**10909**		
1	8	3	**82**	11	**9**	**66**	**82**	13	**9**	**66**	**82**	**5**	**9**	**66**	√	√
1	10	9	–	–	–	10871	–	–	–	9627	–	–	–	**18326**		
2	1	5	**100**	523	**1109**	7678	**100**	634	**1109**	6646	**100**	**90**	**1109**	**46137**		
2	2	10	–	–	–	**11768**	–	–	–	10725	–	–	–	6885		
2	3	6	**165**	3517	**9042**	9254	168	2702	4521	6124	**165**	**2856**	**9042**	**11450**		√
2	4	11	–	–	–	**8027**	–	–	–	6201	–	–	–	5579		
2	5	4	**92**	**37**	**118**	**12499**	**92**	59	**118**	6332	**92**	49	**118**	10329		
2	6	5	**107**	**9**	**2**	6288	**107**	22	**2**	1377	**107**	14	**2**	**7434**		
2	8	5	**126**	422	**1282**	12669	**126**	1183	**1282**	3916	**126**	**314**	**1282**	**16556**		√
2	9	3	**76**	1458	**3588**	8885	**76**	2455	**3588**	5313	**76**	**263**	**3588**	**53345**		√
2	10	8	–	–	–	3223	–	–	–	3760	–	–	–	**8827**		

In the second set of experiments, we assigned weight 1 to activity productions, like $A_i \rightarrow a_i$, and post an additional cost function $\sum_{j=1}^{m} z_j$ that is minimized. $\sum_{j=1}^{m} z_j$ is the number of slots in which employees worked. Results are presented in Table1. We improved on the best solution found in the first model in 4 benchmarks and proved optimality in one. The decomposition of the weighted CFG constraint was slightly slower than the monolithic propagator, while entailment improved performance in most cases.

References

1. Cote, M.-C., Bernard, G., Claude-Guy, Q., Louis-Martin, R.: Formal languages for integer programming modeling of shift scheduling problems. Technical Report, Center for Research on Transportation, Montreal (2007)
2. Demassey, S., Pesant, G., Rousseau, L.-M.: Constraint programming based column generation for employee timetabling. In: Barták, R., Milano, M. (eds.) CPAIOR 2005. LNCS, vol. 3524, Springer, Heidelberg (2005)
3. Ney, H.: Dynamic programming parsing for context-free grammars in continuous speech recognition. IEEE Trans. on Signal Processing 39(2), 336–340 (1991)
4. Pesant, G.: A regular language membership constraint for finite sequences of variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, Springer, Heidelberg (2004)
5. Quimper, C.-G., Louis-Martin, R.: A large neighbourhood search approach to the multi-activity shift scheduling problem. Technical Report, Center for Research on Transportation, Montreal (2007)
6. Quimper, C.-G., Walsh, T.: Global Grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, Springer, Heidelberg (2006)
7. Quimper, C.-G., Walsh, T.: Decomposing Global Grammar constraints. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, Springer, Heidelberg (2007)
8. Sellmann, M.: The theory of Grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, Springer, Heidelberg (2006)

CP with ACO

Madjid Khichane[1,2], Patrick Albert[1], and Christine Solnon[2]

[1] ILOG SA, 9 rue de Verdun, 94253 Gentilly cedex, France
{mkhichane,palbert}@ilog.fr
[2] LIRIS CNRS UMR 5205, University of Lyon I, France
christine.solnon@liris.cnrs.fr

The Ant Colony Optimization (ACO) meta-heuristic [1] has proven its efficiency to solve hard combinatorial optimization problems. However most works have focused on designing efficient ACO algorithms for solving specific problems, but not on integrating ACO within declarative languages so that solving a new problem with ACO usually implies a lot of procedural programming. Our approach is thus to explore the tight integration of Constraint Programming (CP) with ACO. Our research is based upon ILOG Solver, and we use its modeling language and its propagation engine, but the search is guided by ACO. This approach has the benefit of reusing all the work done at the modeling level as well as the code dedicated to constraint propagation and verification.

1 Description of Ant-CP

Some ACO algorithms have been previously proposed for solving Constraint Satisfaction Problems (CSPs), e.g., [2,3]. In these algorithms, ants iteratively build complete assignments (that assign a value to every variable) that may violate constraints, and their goal is to minimize the number of constraint violations; a solution is found when the number of constraint violations is null. In this paper, we investigate a new ACO framework for solving CSPs: ants iteratively build partial assignments (such that some variables may not be assigned to a value) that do not violate constraints, and their goal is to maximize the number of assigned variables; a solution is found when all variables are assigned. This new ACO framework may be combined with the propagation engine of ILOG Solver in a very straightforward way. It can be viewed as a generalization of [4] to CSPs.

More precisely, our approach is sketched in Algorithm 1. First, pheromone trails are initialized to some given value τ_{max}. Then, at each cycle (lines 2-12), each ant k constructs a consistent assignment $\mathcal{A}_k$ (lines 4-10): starting from an empty assignment, the ant iteratively chooses a variable which is not yet assigned and a value to assign to this variable; this variable assignment is added to $\mathcal{A}_k$, and constraints are propagated; this process is iterated until either all variables have been assigned or the propagation step detects a failure. Once every ant has constructed an assignment, pheromone trails are updated. The algorithm stops iterating either when an ant has found a solution, or when a maximum number of cycles has been performed.

Pheromone structure. The pheromone structure, denoted by Φ, is a parameter which defines the set of pheromone trails that are used to guide ants

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 328–332, 2008.

Algorithm 1. Ant-CP procedure

Input: A CSP (X, D, C), a pheromone structure Φ, and a heuristic factor η
Output: A (partial) consistent assignment for (X, D, C)

```
Initialize all pheromone trails of Φ to τ_max
repeat
    foreach k in 1..nbAnts do
        A_k ← ∅
        repeat
            Select a variable X_j ∈ X so that X_j ∉ var(A_k)
            Choose a value v ∈ D(X_j)
            Add ⟨X_j, v⟩ to A_k
            Propagate constraints
        until var(A_k) = X or Failure ;
    Update pheromone trails of Φ using {A_1, ..., A_nbAnts}
until var(A_i) = X for some i ∈ {1..nbAnts} or max cycles reached ;
return the largest constructed assignment
```

during assignment constructions. The default pheromone structure associates a pheromone trail with every variable-value couple, i.e., $\Phi_{default} = \{\tau_{\langle X_i, v_i\rangle} / X_i \in X, v_i \in D(X_i)\}$. Each pheromone trail $\tau_{\langle X_i, v_i\rangle}$ represents the learnt desirability of assigning value v_i to variable X_i. For specific problems, one may design other pheromone structures and we shall propose and compare two other pheromone structures for the car sequencing problem in the next section.

Selection of a variable. When constructing an assignment, the order in which the variables are assigned is rather important and variable ordering heuristics have been studied widely in the context of backtrach search. These heuristics can be used as well in our context of greedy construction of assignments.

Choice of a value. Once a variable X_j has been selected, the value v to be assigned to this variable is randomly chosen within $D(X_j)$ with respect to a probability $p(X_j, v)$ which depends on a pheromone factor $\tau_{\mathcal{A}}(X_j, v)$ —which reflects the past experience of the colony regarding the addition of $\langle X_j, v\rangle$ to the partial assignment $\mathcal{A}$— and a heuristic factor $\eta_{\mathcal{A}}(X_j, v)$ —which is problem-dependent, i.e.,

$$p(X_j, v) = \frac{[\tau_{\mathcal{A}}(X_j, v)]^{\alpha}[\eta_{\mathcal{A}}(X_j, v)]^{\beta}}{\sum_{w \in D(X_j)}[\tau_{\mathcal{A}}(X_j, w)]^{\alpha}[\eta_{\mathcal{A}}(X_j, w)]^{\beta}} \quad (1)$$

where α and β are two parameters that determine the relative weights of pheromone and heuristic information. The definition of the pheromone factor depends on the pheromone structure. For the default structure $\Phi_{default}$, this pheromone factor is defined by $\tau_{\mathcal{A}}(X_j, v) = \tau_{\langle X_j, v\rangle}$.

Constraint propagation. Each time a variable is assigned to a value, a propagation algorithm is called. This algorithm narrows the domains of the variables that are not yet assigned. If the domain of a variable becomes a singleton, then

the partial assignment $\mathcal{A}_k$ is completed by the assignment of this variable and the propagation process is continued. At the end of the propagation process, if the domain of a variable becomes empty or if some inconsistency is detected, then *Failure* is detected.

Pheromone updating step. Once every ant has constructed an assignment, each pheromone trail of the pheromone structure Φ is decreased and then the best ants of the cycle deposit pheromone, i.e., $\forall \tau_i \in \Phi$,

$$\tau_i \leftarrow (1-\rho) \cdot \tau_i + \sum_{\mathcal{A}_k \in BestOfCycle} \Delta\tau(\mathcal{A}_k, \tau_i)$$
$$\text{if } \tau_i < \tau_{min} \text{ (resp.} \tau_i > \tau_{max}) \text{ then } \tau_i \leftarrow \tau_{min} \text{ (resp. } \tau_i \leftarrow \tau_{max})$$

where

- ρ is the evaporation parameter, such that $0 \leq \rho \leq 1$,
- τ_{min} and τ_{max} are two parameters for bounding pheromone trails,
- *BestOfCycle* is the set of the best assignments constructed during the cycle,
- $\Delta\tau(\mathcal{A}_k, \tau_i)$ is the quantity of pheromone deposited on the pheromone trail τ_i by the ant that has built assignment $\mathcal{A}_k$. This quantity depends on the chosen pheromone structure. For the default structure $\Phi_{default}$, this quantity is equal to zero if X_i is not assigned to v in $\mathcal{A}_k$; otherwise, it is proportionally inverse to the gap of sizes between $\mathcal{A}_k$ and the largest assignment $\mathcal{A}_{best}$ built since the beginning of the search (including the current cycle).

2 Using Ant-CP to Solve the Car Sequencing Problem

The car sequencing problem involves scheduling cars along an assembly line in order to install options on them. Each car requires a set of options; all cars requiring a same subset of options are grouped in a same car class. For each option i, a capacity constraint p_i/q_i imposes that there are at most p_i cars requiring option i every q_i consecutive cars. We refer the reader to [5] for more details on this problem.

CP model and variable ordering heuristic. The CP model for the car sequencing problem has been defined with the CP modeling language of ILOG Solver and corresponds to the first model proposed in the user's manual of ILOG Solver. In our experiments, we have used a classical sequential variable ordering heuristic, which consists in assigning variables associated with positions in the sequence of cars in the order defined by the sequence.

Pheromone structures for the car sequencing problem. As pheromone is at the core of the efficiency of any ACO implementation, we explore the impact of its structure: besides the default pheromone structure $\Phi_{default}$, we propose and compare two structures called $\Phi_{classes}$ and Φ_{cars}. To run Ant-CP with a new pheromone structure, one basically has to define the set Φ of pheromone components, define the pheromone factor $\tau_{\mathcal{A}}(X_j, v)$ with respect to these pheromone components, and define which pheromone components must be rewarded during the pheromone updating step. Due to lack of space, we do not describe into details the two pheromone structures $\Phi_{classes}$ and Φ_{cars}, but just give the intuition:

- $\Phi_{classes}$ is used in [6] and associates a pheromone trail $\tau_{(v,w)}$ with every couple of car classes (v, w); this pheromone trail represents the learnt desirability of sequencing a car of class w just after a car of class v;
- Φ_{cars} is used in [3] and associates a pheromone trail $\tau_{(v,i,w,j)}$ with every couple of classes (v, w) and every $i \in [1; \#v]$ and every $j \in [1; \#w]$ where $\#v$ and $\#w$ respectively are the number of cars in classes v and w. This pheromone trail represents the learnt desirability of sequencing the j^{th} car of class w just after the i^{th} car of class v.

Heuristic factors for the car sequencing problem. In the transition probability defined by eq. (1), the pheromone factor is combined with a heuristic factor $\eta_{\mathcal{A}}(X_j, v)$ which is problem-dependent. We have considered here the DSU heuristic of [7], which is based on the sum of the dynamic utilization rates of options: $\eta_{\mathcal{A}}(X_j, v) = \sum_{i \in \text{ options}(v)} \frac{n_i \cdot q_i}{N \cdot p_i}$, where n_i is the number of cars that are not yet sequenced and that require option i and N is the number of cars that are not yet sequenced.

Experimental results. We now experimentally compare Ant-CP($\Phi_{default}$), Ant-CP(Φ_{cars}), and Ant-CP($\Phi_{classes}$), which respectively use the pheromone structures $\Phi_{default}$, Φ_{cars} and $\Phi_{classes}$, with Ant-CP($\emptyset$) which ignores pheromone (i.e., $\Phi = \emptyset$ and the pheromone factor $\tau_{\mathcal{A}}(X_j, v)$ is set to 1). All experiments have been

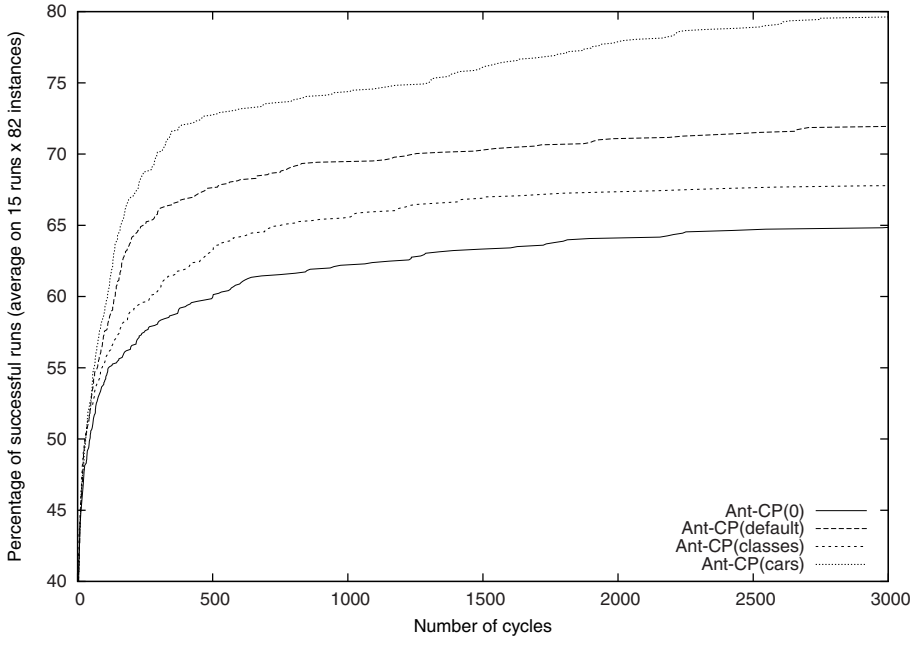


Fig. 1. Comparison of pheromone strategies

performed with the following parameter setting: $\tau_{min} = 0.01$, $\tau_{max} = 4$, $\alpha = 1$, $\beta = 6$, $\rho = 2\%$, $nbAnts = 30$, and $nbMaxCycles = 3000$.

Let us first note that the 70 instances of the test suite provided by Lee and available in CSPLib [8] are all very quickly solved by all instanciations of Ant-CP in a very few assignment constructions. It is worth mentionning here that some of these instances are still considered as difficult ones for complete branch-and-propagate based solvers [9,10].

Fig. 1 displays the evolution of the percentage of successful runs with respect to the number of cycles on a more difficult benchmark provided by Perron and Shaw. This benchmark contains 82 instances that have between 100 and 500 cars to sequence. Fig. 1 shows that guiding the search with pheromone increases the success rate, after 3000 cycles, from 64.82% for Ant-CP($\emptyset$) to 79.62% for Ant-CP(Φ_{cars}), 71.86% for Ant-CP($\Phi_{default}$), and 67.77% for Ant-CP($\Phi_{classes}$).

These first results show that pheromone significantly improves the solution process, even when considering the default structure. Further works will mainly concern the validation of our approach on other CSPs and the integration of a reactive scheme in order to automatically adapt parameters during the search process.

References

1. Dorigo, M., Stuetzle, T.: Ant Colony Optimization. MIT Press, Cambridge (2004)
2. Solnon, C.: Ants can solve constraint satisfaction problems. IEEE Transactions on Evolutionary Computation 6(4), 347–357 (2002)
3. Solnon, C.: Combining two pheromone structures for solving the car sequencing problem with Ant Colony Optimization. EJOR (to appear, 2008)
4. Meyer, B., Ernst, A.: Integrating aco and constraint propagation. In: Dorigo, M., Birattari, M., Blum, C., Gambardella, L.M., Mondada, F., Stützle, T. (eds.) ANTS 2004. LNCS, vol. 3172, pp. 166–177. Springer, Heidelberg (2004)
5. Solnon, C., Cung, V., Nguyen, A., Artigues, C.: The car sequencing problem: overview of state-of-the-art methods and industrial case-study of the ROADEF 2005 challenge problem. EJOR (to appear, 2008)
6. Gravel, M., Gagné, C., Price, W.: Review and comparison of three methods for the solution of the car-sequencing problem. JORS (2004)
7. Gottlieb, J., Puchta, M., Solnon, C.: A study of greedy, local search and aco approaches for car sequencing problems. In: Raidl, G.R., Cagnoni, S., Cardalda, J.J.R., Corne, D.W., Gottlieb, J., Guillot, A., Hart, E., Johnson, C.G., Marchiori, E., Meyer, J.-A., Middendorf, M. (eds.) EvoIASP 2003, EvoWorkshops 2003, EvoSTIM 2003, EvoROB/EvoRobot 2003, EvoCOP 2003, EvoBIO 2003, and EvoMUSART 2003. LNCS, vol. 2611, Springer, Heidelberg (2003)
8. Gent, I., Walsh, T.: Csplib: a benchmark library for constraints. Technical report (1999), http://csplib.cs.strath.ac.uk/
9. van Hoeve, W.J., Pesant, G., Rousseau, L.M., Sabharwal, A.: Revisiting the sequence constraint. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 620–634. Springer, Heidelberg (2006)
10. Brand, S., Narodytska, N., Quimper, C.G., Stuckey, P.J., Walsh, T.: Encodings of the sequence constraint. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 210–224. Springer, Heidelberg (2007)

A Combinatorial Auction Framework for Solving Decentralized Scheduling Problems (Extended Abstract)

Hoong Chuin Lau[1], Kong Wei Lye[2], and Viet Bang Nguyen[1]

[1] Singapore Management University, 80 Stamford Road, Singapore 178902
[2] Singapore Institute of Manufacturing Technology, 71 Nanyang Drive, Singapore 638075
{hclau,vbnguyen}@smu.edu.sg, kwlye@simtech.a-star.edu.sg

1 Introduction

Computing an optimal solution to an integer program for a realistic scheduling problem can often be very time consuming. As a result, ad-hoc and hand-crafted heuristics have become popular alternatives. These methods, however, suffer from the inability to guarantee good or optimal solutions. Our objective in this work is to leverage on prior theoretical results from the OR literature and agent technology from the AI literature to derive a computational framework that can be easily implemented for solving decentralized scheduling problems. In decentralization, there is an issue that information and control are inherently private to individual agents, even though a solution has to be jointly derived. In this paper, we are concerned about the following class of decentralized scheduling problems:

1. There is a central pool of limited resources that comprises multiple units of resources for each machine type;
2. There are multiple self-interested agents and each has to obtain resources from the central pool to solve its own scheduling problem (job shop, flow shop, etc).

Leveraging on the work by Kutanoglu and Wu [1], we propose a generic auction-based agent framework that enables a given decentralized scheduling algorithm to be readily designed and implemented. We then demonstrate the application of our proposed framework on a real-world supply chain problem. Kutanoglu and Wu [1] (and [2]) have shown that solving the Job Shop Scheduling problem using Lagrangian relaxation can be viewed as an auction. In this auction, job agents iteratively bid for resources and an auctioneer responds with new prices for these resources in a price adjustment process (also known as "*tâtonnement*"). Economic principles dictate that given a restricted setting such as gross substitutability, the tâtonnement process converges to an equilibrium [3]. Our contributions in this paper are: 1) decentralization, and 2) genericity. In particular, we show how an arbitrary decentralized scheduling problem can be generically tackled with an auction approach and demonstrate how a generic framework can be constructed that allows designers to quickly design and implement a decentralized algorithm. For benchmarking purposes, we will compare the solutions obtained against a conventional exact approach. We also surface the attractiveness of our approach by measuring the level of agent "fairness" which we define as the extent to which resources are allocated according to individual agent demands.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 333–337, 2008.

2 Lagrangian Relaxation and Combinatorial Auction

Kutanoglu and Wu [1] formulated the Job Shop Scheduling problem as an integer program and dualized the machine capacity constraints with a vector of non-negative Lagrangian multipliers λ. The resulting relaxed problem LR_λ can be decomposed into independent sub-problems $LR_{\lambda,i}$ corresponding to the individual jobs' local scheduling problems. A lower bound on the optimal cost of the original problem can be computed using λ. The best lower bound corresponds to the solution of the dual of LR_λ, *LRD*. To find λ for the dual problem's optimal solution, a sub-gradient search procedure can be used. Taking this idea to the auction context, they showed that the iterative process for finding λ actually corresponds to a combinatorial auction:

1. The auctioneer initializes the prices for the machine-time resources λ^0.
2. At auction round *r*, each job agent solves its local scheduling problem $LR_{\lambda,i}$ using λ^r in its cost minimization objective function and submits its bid for resources (local schedule) to the auctioneer.
3. The auctioneer collates all the bids from the job agents and generates a global schedule. It resolves resource conflicts in this schedule and computes new prices for the resources using a tâtonnement scheme. This corresponds to an iteration in solving *LRD*. Sub-gradient search can be used to update λ^r.
4. The auctioneer starts the next round of auction (by announcing the new prices λ^{r+1} to the bidders) if a best feasible schedule has not been found.

3 Decentralized Lot-Sizing-Cum-Transportation Problem

In this section, we show how a real-world supply chain problem can be modeled as a decentralized scheduling problem and solved using our proposed auction approach and framework. We are given multiple global manufacturing plants (MPs) and a global distribution centre (DC) that sends item parts to the various MPs. Each MP has its own production plan that requires item parts for fulfilling the production across multiple periods within a prescribed time horizon. Items can be shipped via multiple routes from the DC to each MP. Each route has a finite capacity, operates on a regular schedule, and is associated with a concave cost function. This problem can be modeled as an auction problem: Each MP is represented as a bidder agent, while the DC acts as the auctioneer. The auctioneer (DC) has to decide how to price the resources so that all the MPs can meet their production plans. For each time period, the DC consolidates the demands for items and routes, and computes the unit price for each item in each period on each route. This price is then broadcast to all MPs. Each MP has a production plan which specifies the quantity of each part required for each time period. The MP needs to order sufficient parts to meet its production. Excess units ordered incur holding costs. There are upper and lower bounds for each part inventory at each time period and a requirement on inventory level for the last time period. Hence, each MP has to decide on the quantity of each part to order for each time period on each route, in order to minimize the total cost (inventory holding cost plus shipping cost). Each MP has to solve a local scheduling problem that minimizes the holding and shipping costs, *plus* the current prevailing cost imposed by the auctioneer

on the items and routes it decides to bid on. The optimal solution for this problem is a bid that specifies resource requirement bundle. This decentralized lot-sizing cum transportation problem can be formulated as an integer program:

$$\min \sum_{s=1}^{S}\sum_{r\in R_s}\sum_{t=1}^{T} f_{r,t}(x_{r,t}^s) + \sum_{s=1}^{S}\sum_{t=1}^{T} g_s(I_t^s) \qquad (\mathrm{P}_X)$$

$$\text{s.t.} \quad I_{t+1}^s = I_t^s + \sum_{r\in R_s} x_{r,t}^s - q_t^s \ , \qquad t=1,..,T\text{-}1,\ s=1,\ldots,S$$

$$I_t^s \geq q_{t+1}^s \ , \qquad t=1,..,T\text{-}1,\ s=1,\ldots,S$$

$$l_t^s \leq I_t^s \leq u_t^s \ , \qquad t=1,..,T,\ s=1,\ldots,S$$

$$\sum_{s=1}^{S} x_{r,t}^s \leq C_t^r \ , \qquad \text{for each } r\in R,\ t=1,\ldots,T \qquad (*)$$

$$x_{r,t}^s \geq 0 \ , \qquad \text{for each } r\in R,\ s=1,\ldots,S,\ t=1,\ldots,T$$

where T is the time horizon, S is the number of MPs, R_s is the set of routes that can serve MP s. $R = R_1 \cup R_2 \ldots \cup R_s$ is the set of all routes originating from DC, $Q_s = \{q_1^s,\ldots,q_T^s\}$ is a vector representing the production plan of MP s, I_t^s is the inventory level of MP s at the end of time period t, $U_s = \{u_1^s,\ldots,u_T^s\}$ and $L_s = \{l_1^s,\ldots,l_T^s\}$ are upper and lower bounds on inventories respectively. $X_s = \{x_{r,t}^s\}$ $\forall r\in R_s$, $t=1,\ldots,T$, $x_{r,t}^s \in \mathbb{Z}^+$ are decision variables for MP s on the quantities of parts to order for each time unit on each route. $f_{r,t}(\cdot)$ is the concave shipping cost function for route r at time period t and $g_s(\cdot)$ is the constant holding cost function for excess inventory. C_t^r is the capacity of shipping route r at time period t. By dualizing the capacity constraints (*), we arrive at the relaxed problem LR(λ):

$$\min \sum_{s=1}^{S}\sum_{r\in R_s}\sum_{t=1}^{T} f_{r,t}(x_{r,t}^s) + \sum_{s=1}^{S}\sum_{t=1}^{T} g_s(I_t^s) + \sum_{r\in R_s}\sum_{t=1}^{T}\lambda_{r,t}\sum_{s=1}^{S}(x_{r,t}^s - C_t^r) \qquad (\mathrm{LR}(\lambda))$$

subject to the above constraints in P_X, less the capacity constraints (*).

Following the decomposition procedure in Section 2, an MP thus solves the subproblem MP_X given λ, and DC solves DC_λ for each route r given X:

$$\min \sum_{r\in R_s}\sum_{t=1}^{T} f_{r,t}(x_{r,t}^s) + \sum_{t=1}^{T} g_s(I_t^s) + \sum_{r\in R_s}\sum_{t=1}^{T}\lambda_{r,t}(x_{r,t}^s - C_t^r) \qquad (\mathrm{MP}_X)$$

subject to its own subsets of constraints in P_X.

$$\max \sum_{s=1}^{S}\sum_{t=1}^{T}[f_{r,t}(x_{r,t}^s) + g_s(I_t^s)] + \sum_{t=1}^{T}\lambda_{r,t}\sum_{s=1}^{S}(x_{r,t}^s - C_t^r) \ , \text{ s.t. } \lambda\geq 0 \qquad (\mathrm{DC}_\lambda)$$

Each agent (MP) decides on the quantity of each item to order for each time period $X_s = \{x_{r,t}^s\}$, and solves an optimization problem MP_X. This problem is a classical

multi-period lot-sizing cum transportation problem which can be solved efficiently via dynamic programming. The auctioneer consolidates all the bid bundles from all the agents and computes the total demand for each route *r* for each time period *t*. Its role is to adjust the prices so as to influence future total demand so it does not exceed the capacity C_t^r. It needs to solve the problem DC_λ to (try to) find a better λ. In the auction context, this means it first computes the resource capacity violation costs, and then, for route-time resources that have demand exceeding capacity, it raises their prices, and for those that have demand less than capacity, it lowers their prices.

4 Experimental Results

In this section, we compare the results of our auction approach in terms of system performance versus an exact (branch-and-bound) approach. We also validate the robustness of our approach by testing on diverse scenarios, addressing different demand patterns from the MPs (bidders). The problem size (the total number of decision variables) is given by *(total number of time periods)* x *(total number of routes)* x *(total number of bidders).* We designed test cases having problem sizes of 32=4x2x4, 36=3x3x4, 48=6x2x4, 56=7x2x4 respectively. We generated test cases with varying route capacities and total demand / total supply ratio (α). The route capacity is set randomly within the interval [5, 12]. The total supply S is calculated as route capacity x number of routes x number of time periods. The total demand of all MPs D = S x α is then spread evenly to all MPs (for the purpose of measuring fairness, to be discussed later). The demand distribution across different periods varies in different cases – ranging from uniform distribution to skewed distribution. The shipping cost function for each time period on each route is a concave cost function, which generally takes the form $f_{r,t}(k_t) = 0$ if $k_t = 0$, and $\xi + \delta(x)$, where ξ is the fixed cost and $\delta(x) = a_0x^0 + a_1x^1 + \ldots + a_nx^n$. The holding cost takes the form of a linear function $g_s(i) = ui$, where *u* is the unit holding cost set as 10 in all test cases. We compare our auction approach (AUC) against an exact algorithm with branch and bound and forward constraint checking (OPT). AUC is set to run for 50 iterations for each test case. We used a 3GHz Pentium IV machine with 1GB RAM. The results are shown in Table 1. Table 1 shows that on an average, AUC produced results within an average of 5% from optimality with significant run time performance gain compared to OPT for large problems (minor exceptions for cases 5 and 6 where OPT happens to be more efficient because constraint checking enables pruning to take place early in the search tree). For sparse demands uniformly distributed across time periods (cases 3 and 10, highlighted in gray), AUC produced optimal solutions. In cases where the demand patterns are similar and under tight demand/supply constraint (case 4), there are frequent conflicts among the agents resulting in fewer solutions and thus less efficient resource allocation. We further investigate the positive effect of decentralization in terms of agent *fairness* in the sense of [5] - which is a desirable scheduling notion that measures the extent that each agent is allocated resources in proportion to its demand. We conjecture that a decentralized (auction) approach may lead to higher agent-centric fairness compared to a central approach where the concern is simply to optimize global (social) cost. For this purpose, given that demands are evenly spread out

among agents, we devise the fairness measure based on the standard deviation metric $\sigma = \Sigma\ (cost_i - average\ cost)^2$, where $cost_i$ is the objective cost for MP i, and *average cost* is the average cost of all agents. The results are reported in the last column in the table, where we observe that in the majority of cases, the fairness measure for our approach outperforms the centralized optimum approach.

Table 1. Comparison between auction approach and branch and bound

Problem size	Test case	Demand /Supply Ratio α	Objective Function			Run Time (seconds)		Std Deviation σ	
			AUC	OPT	Ratio	AUC	OPT	AUC	OPT
32	1	0.875	621	611	1.016	0.031	0.86	8.04	8.27
	2	0.854	618	617	1.002	0.031	0.953	7.40	8.12
	3	0.688	613	613	1.000	0.031	0.313	5.79	5.79
36	4	0.889	555	462	1.201	0.016	0.094	2.53	5.64
	5	0.870	551	548	1.005	0.110	0.032	3.33	2.21
	6	0.870	590	582	1.014	0.094	0.093	5.17	5.40
	7	0.833	649	628	1.033	0.016	0.047	5.00	6.74
48	8	0.717	776	675	1.150	0.032	14.922	16.01	20.00
	9	0.900	1049	1044	1.005	0.031	65.066	59.80	60.44
	10	0.442	828	828	1.000	0.203	5.563	16.02	16.02
56	11	0.900	1519	1498	1.014	0.187	5004.97	109.73	108.64
	12	0.884	1626	-	-	0.180	> 3 hrs	-	-

References

1. Kutanoglu, E., Wu, S.D.: On combinatorial auction and Lagrangean relaxation for distributed resource scheduling. IIE Transactions 31, 813–826 (1999)
2. Dewan, P., Joshi, S.: Auction-based distributed scheduling in a dynamic job shop environment. Int. Journal of Production Research 40(5), 1173–1191 (2002)
3. Arrow, K.J., Hurwicz, L.: Competitive Stability under Gross Substitutability: The Euclidean Distance Approach. Int. Economic Review 1, 38–49 (1960)
4. Fisher, M.L.: The Lagrangian Relaxation Method for Solving Integer Programming Problems. Management Science 27(1), 1–18 (1981)
5. Baruah, S.K., et al.: Proportionate Progress: A Notion of Fairness in Resource Allocation. Algorithmica 15, 600–625 (1996)

Constraint Optimization and Abstraction for Embedded Intelligent Systems

Paul Maier and Martin Sachenbacher

Technische Universität München, Institut für Informatik
Boltzmannstraße 3, 85748 Garching, Germany
{maierpa,sachenba}@in.tum.de
http://www9.in.tum.de

Abstract. Many tasks in artificial intelligence, such as diagnosis, planning, and reconfiguration, can be framed as constraint optimization problems. However, running constraint optimization within embedded systems requires methods to curb the resource requirements in terms of memory and run-time. In this paper, we present a novel method to control the memory requirements of message-passing algorithms that decompose the problem into clusters and use dynamic programming to compute approximate solutions. It can be viewed as an extension of the previously proposed mini-bucket scheme, which limits message size simply by omitting constraints from the messages. Our algorithm instead adaptively abstracts constraints, and we argue that this allows for a more fine-grained control of resources particularly for constraints of higher arity and variables with large domains that often occur in models of technical systems. Preliminary experiments with a diagnosis model of NASA's EO-1 satellite appear promising.

Keywords: Constraint optimization, decomposition, heuristic search, automated abstraction, embedded systems.

1 Introduction

New generations of technical devices are being developed that use models of themselves to implement self-awareness capabilities, for example in assistant automotive systems and cognitive manufacturing systems [1]. Many of the underlying computational tasks – such as automatically determining the most likely current state of the system (monitoring and diagnosis), or finding least-cost sequences of actions that drive the system towards desired states to compensate for contingencies (planning and reconfiguration) – can be framed as combinatorial optimization problems over sets of constraints [7]. Such a common representational basis enables tight integration of the different tasks.

However, constraint optimization is exponential in the number of system variables, which is especially a problem given the limited resources (memory and CPU time) available in embedded controllers. As models of technical systems often present some inherent modularity (for instance, a car's controller area

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 338–342, 2008.

network), approaches that decompose the constraint model off-line into clusters and infer solutions on-line by passing messages between the clusters [6,11] can provide a significant improvement. Actually, one of the first approaches to constraint-based diagnosis used such a message-passing scheme [8].

Unfortunately, this can still lead to an infeasible message size (exponential in the separator width of the decomposition). In the literature, mini-clustering [9] has been suggested as a scheme to limit the memory requirements of structure-based constraint optimization algorithms, by approximating messages instead of computing them exactly. The idea of mini-cluster tree elimination (MCTE) is to restrict the size of messages in the tree by partitioning the constraints in the clusters into subsets (called mini-clusters) involving at most i different variables. By combining only the constraints in a mini-cluster, one gets a set of messages that approximate the cost function of the original message with an upper (optimistic) bound, and this can be used as a heuristic for subsequent search. The complexity of mini-cluster tree elimination is therefore $O(r \cdot k^i)$, where k is the largest domain size, and r is the total number of occurring cost functions [6].

However, when we tried to use mini-clustering as a reasoning scheme in embedded constraint optimization, we encountered two problems. First, it is difficult to control the number of messages that need to be transmitted. If i is small, the messages themselves are computed faster but their total number (and thus r) will increase. Limiting the number of messages by suppressing some of them is hardly a solution, because constraints are unique to their respective mini-cluster and thus information about these constraints will be completely lost in the heuristic approximation. Second, the parameter i allows only limited control over the size of the resulting messages, because MCTE can only either omit a constraint or use it in a message, without any intermediate steps. While this works for models with small (binary) constraints and small variable domains, typical constraint models for diagnosis and planning tend to have large domains and big constraints. For instance, in a model of NASA's Earth Oberserving Satellite (EO-1) [4], variables have up to ten domain values and many constraints involve more than four variables. For such larger problems, mini-clustering offers only limited possibilities to form the mini-clusters and therefore it cannot control the size of the messages effectively. In fact, restricting the number of constraints occurring in messages (or even limiting the number of variables in a constraint by projecting them on a subset of their variables) will affect only the exponent of the space complexity; depending on k, this can lead to big jumps in the possible message size.

2 Adaptive Abstraction for Constraint-Based Models

Extending upon the mini-bucket scheme, we propose a more general approach to limit the size of messages exchanged between clusters. It allows for finer control over the message size, and therefore enables better adaption of message-passing constraint optimization algorithms to the tight resources in embedded systems.

Instead of omitting constraints from messages (as in mini-buckets), our approach adaptively reduces the size of constraints by abstracting them, similar

to the automated generation of search heuristics (pattern databases) in game analysis and path search [12,5]. The key idea is to choose appropriate aggregations of the domains of the variables that limit the worst-case complexity of messages by allowing only a limited number of total distinctions. This aggregation affects both the base and the exponent of the message complexity, and therefore allows for a more fine-grained control compared to the mini-bucket parameter i. It is worth noting that using such abstractions, one can reconstruct the behavior of the original mini-bucket algorithm as a special case where the identical abstraction (preserving all distinctions) is applied to the constraints inside a minibucket, and the trivial abstraction (eliminating all distinctions) is applied to the constraints outside the minibucket.

We have implemented this approach as a variant of the existing message-passing algorithm MCTE . The new algorithm, called Bucket Elimination with Domain Abstraction (BEDA), was integrated into the open-source constraint solver Toolbar [2] to allow comparison with MCTE and other constraint optimization algorithms. BEDA combines tree decomposition with automated aggregation of variable domains, which reduces domain sizes and yields smaller, abstract constraints that approximate the original constraints. The granularity of the domain abstraction is automatically adapted to a given message size limit $T_{\max}$ by determining appropriate sizes for the abstracted domains through solving a small optimization problem. In our preliminary implementation, this is achieved by greedily decreasing the domain sizes of the largest domains of the variables in an outgoing message, until $T_{\max}$ is met. Our current prototype determines only one such domain abstraction per cluster, and only a single outgoing message will be sent per cluster.

Figure 1 shows the results of a preliminary test with the EO-1 diagnosis model. The horizontal axis (logarithmic scale) is the size limit (as given by the embedded controller) imposed on the messages during the message-passing phase, whereas

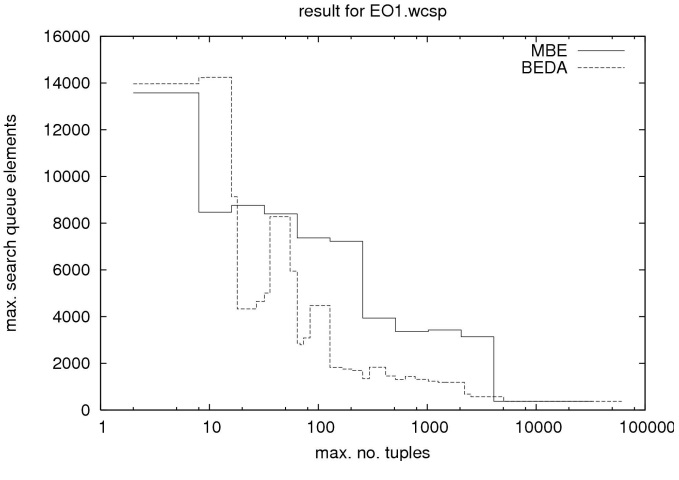


Fig. 1. Search nodes vs. size limit for the EO-1 diagnosis example

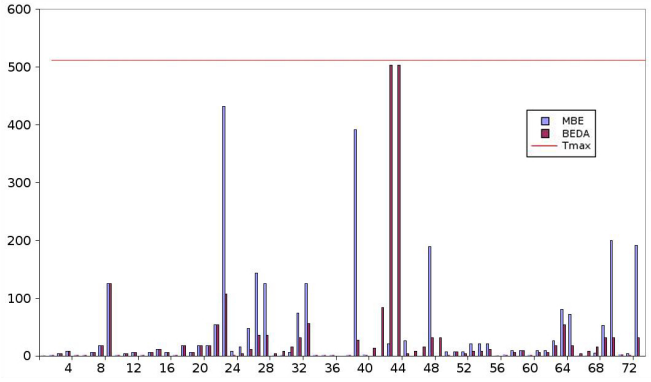


Fig. 2. Message sizes for $T_{\max} = 512$ for the EO-1 diagnosis example

the vertical axis shows the number of search nodes that are expanded in the subsequent search phase to find the optimal solution (corresponding to run-time). Clearly, the larger the size limit, the better the heuristic approximation achieved during the message-passing phase, and thus the shorter the search phase. This tendency can be observed both for MCTE and BEDA. However, for MCTE (solid line), the message size proceeds only in relatively coarse steps, as it can only indirectly be controlled through the parameter i (maximum number of variables in a minibucket). In comparison, BEDA (dotted line) enables finer control of the message size, potentially better adapting to the resources of the embedded system. Figure 2 further illustrates this point by comparing the sizes of the messages sent by each of the 73 clusters in the model for a given size limit of $T_{\max} = 512$ tuples.

Note also that for this example, at a certain minimum size of the messages BEDA yields a better heuristic than the mini-bucket approximation, given the same message size limit. This is intriguing considering that BEDA sends *fewer* messages per cluster than MCTE. We attribute this to the fact that at least in this real-world example, the approximation of a cluster message through local combinations of its constituents as in MCTE is less informative than a global (though coarse) approximation as used in BEDA. In all cases, the time for computing and passing along the BEDA messages was comparable to or even smaller than for MCTE, that is, computing the domain abstractions did not incur significant overhead over forming the mini-clusters.

3 Future Work Directions

Further tests and a more rigorous complexity analysis will be conducted to reveal the potential of the approach. Also, we are working on more refined strategies for the automatic adaption of the abstraction [13]. A particular promising direction is iterative refinement of the abstractions, based on ideas presented in [3,12,5]. Our main goal here is to achieve better utilization of embedded resources, by

enabling more fine-grained control over the message size in message-passing algorithms. Another goal in embedded diagnosis and planning is to focus the computation on the best solutions only, as in this application context the controller typically needs to know only a few best solutions [10]. Our abstraction-based approach can be helpful in this respect also, as we can easily bias the computation of the abstraction: the idea is to use a more fine-grained resolution for values with high utility, and a more coarse-grained resolution for values with lower utility. We are currently experimenting with such biased abstraction strategies for embedded diagnosis and planning applications.

References

1. Beetz, M., Buss, M., Wollherr, D.: Cognitive technical systems – what is the role of artificial intelligence? In: Hertzberg, J., Beetz, M., Englert, R. (eds.) KI 2007. LNCS (LNAI), vol. 4667, pp. 19–42. Springer, Heidelberg (2007)
2. Bouveret, S., Heras, F., de Givry, S., Larrosa, J., Sanchez, M., Schiex, T.: Toolbar: A state-of-the-art platform for wcsp, http://www.inra.fr/mia/T/degivry/ToolBar.pdf
3. Koster, A.: Frequency Assignment – Models and Algorithms. PhD thesis, Universiteit Maastricht, Maastricht, The Netherlands (1999)
4. Hayden, S.C., Sweet, A.J., Christa, S.E.: : Livingstone Model-Based Diagnosis of Earth Observing One. In: Proceedings AIAA 1st Intelligent Systems Technical Conference (September 2004)
5. Sturtevant, N.R., Jansen, R.: An analysis of map-based abstraction and refinement. In: Miguel, I., Ruml, W. (eds.) SARA 2007. LNCS (LNAI), vol. 4612, pp. 344–358. Springer, Heidelberg (2007)
6. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers, San Francisco (2003)
7. Sachenbacher, M., Williams, B.C.: Diagnosis as Semiring-based Constraint Optimization. In: Proceedings 16th European Conference on Artificial Intelligence (ECAI 2004), Valencia, Spain, pp. 873–877 (2004)
8. El Fattah, Y., Dechter, R.: Diagnosing tree-decomposable circuits. In: Proceedings 14th International Joint Conference on Artificial Intelligence (IJCAI 1995), Montreal, Canada, pp. 1742–1749 (1995)
9. Kask, K., Dechter, R.: Mini-Bucket Heuristics for Improved Search. In: Proceedings 15th Conference on Uncertainty in Artificial Intelligence (UAI 1999), Stockholm, Sweden, pp. 314–323 (1999)
10. O'Sullivan, B., Provan, G.M.: Approximate Compilation for Embedded Model-based Reasoning. In: Proceedings 21st National Conference on Artificial Intelligence (AAAI 2006), Boston, USA (2006)
11. Petcu, A., Faltings, B.: Superstabilizing, fault-containing distributed combinatorial optimization. In: Proceedings 20th National Conference on Artificial Intelligence (AAAI 2005), Pittsburgh, USA, pp. 449–454 (2005)
12. Holte, R., Hernadvolgyi, I.: Steps towards the automatic creation of search heuristics. Technical report, Computing Science Department, University of Alberta (2004)
13. Sachenbacher, M., Struss, P.: Task-dependent Qualitative Domain Abstraction. Artificial Intelligence 162(1-2), 121–143 (2005)

A Parallel Macro Partitioning Framework for Solving Mixed Integer Programs

Mahdi Namazifar and Andrew J. Miller*

Industrial and Systems Engineering Department, University of Wisconsin - Madison,
1513 University Avenue, Madison, Wisconsin, USA
{namazifar,ajmiller5}@wisc.edu

Abstract. Mixed Integer Programs are a class of optimization problems which have a vast range of applications in engineering, business, science, health care, and other areas. For many applications, however, problems of realistic size can take a an impractical amount of time to solve on a single workstation. However, using parallel computing resources to solve MIP is difficult, as parallelizing the standard branch-and-bound framework presents an array of challenges. In this paper we present a novel framework called a Parallel Macro Partitioning (PMaP) framework for solving mixed integer programs in parallel. The framework exploit ideas from modern MIP heuristics to partition the problem at a high-level into MIP subproblems, each of which can be solved on a separate processor by an MIP algorithm. Initial computational resources suggest that PMaP has significant promise as a framework capable of bringing many processors to bear effectively on difficult problems.

Keywords: High-Performance Computing, Mixed Integer Programming, Primal Heuristics, Branch-and-Bound.

1 Introduction

In recent year great strides have been made in our ability to solve mixed integer programming (MIP) problems. Much of this progress has come through theoretical and algorithmic improvements that enable LP-based branch-and-bound and branch-and-cut algorithms to solve much large problems than previously possible. In spite of this progress, however, there are many MIP problems that remain practically intractable. For example, the MIPLIB library http://miplib.zib.de/miplib2003.php has numerous problems that take several days to solve; and, also, problem swhich have not yet been solved to optimality. Such facts encourage the use of high-performance computing resources for solving MIP's.

Parallelizing branch-and-bound has been studied in several papers (e.g., [4,8]). In general, two classic strategies for parallelizing branch-and-bound are node-based strategies, in which computations at the nodes of branch-and-bound tree are parallelized, and tree-based strategies, in which building and exploring the branch-and-bound tree is done in parallel [4]. In the latter approach, each processor is assigned a node of the tree and builds up the tree rooted at its assigned

* This research was supported by NSF grant CMMI 0521953.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 343–348, 2008.

node. The domain of the problem is thus split into several disjoint pieces, and each piece is processed by a single processor. Several solvers such as CPLEX [2], Xpress [7], and SYMPHONY [5], use such techniques to parallelize the branch-and-bound algorithm.

In this paper we propose a new framework that partitions the domain of MIP by using concepts derived from recently developed primal heuristics. Primal heuristics are methods which are used in the course of branch-and-bound to find good feasible solutions; examples include LP-and-fix, RINS, and local branching. By using primal heuristic methods both to define subproblems and to generate complementary cuts, we partition the feasible region at a high level into a set of subproblem MIPs that can be solved simultaneously in parallel.

The rest of the this paper is organized as follows: in section 2 we describe primal heuristics. In section 3 we sketch the whole view of the parallel MIP solver framework that we propose. Section 4 gives some notes about the implementation of the framework and some numerical results, and section 5 concludes the paper.

2 Primal Heuristics

Finding feasible solutions for mixed integer programs is important for at least two reasons: 1) for many applications identifying a good feasible solution is the primary goal of the MIP model; and 2) a feasible solution provides the branch-and-bound algorithm with an upper bound (considering a minimization problem) which accelerates the pruning process of a branch-and-bound algorithm.

Generally primal heuristics fall into two categories: construction heuristics and improvement heuristics. Construction heuristics try to produce a feasible solution from scratch. Improvement heuristics try to improve a given solution (or set of solutions) to find a better feasible solution. Here we briefly review some commonly used, powerful heuristics.

2.1 LP-and-FIX

This heuristic is very simple and is conceptually related to diving. Given a branch-and-bound tree node, the idea is to explore a sub-space defined by the current linear programming relaxation (LP) solution of the node. To do this we look at the LP solution and fix those integer variables that happen to take integral values in the relaxation solution. I.e., if the MIP problem is

$$(P) \qquad \begin{aligned} \min_{(x,y)\in\mathbb{R}^n\times\mathbb{R}^p} \; & cx + fy \\ \text{s.t. } & Ax + Gy \geq b \\ & x \geq 0, y \in \{0,1\}^p \,, \end{aligned}$$

let the the LP relaxation solution be $(\hat{x}, \hat{y})$. The LP-and-FIX problem is

$$(LP - FIX) \qquad \begin{aligned} \min_{(x,y)\in\mathbb{R}^n\times\mathbb{R}^p} \; & cx + fy \\ \text{s.t. } & Ax + Gy \geq b \\ & x \geq 0, y \in \{0,1\}^p \\ & y_j = \hat{y}_j \text{ for all } j \text{ with } \hat{y}_j \in \{1,0\} \,, \end{aligned}$$

2.2 Relaxation Induced Neighborhood Search (RINS)

Another highly useful primal heuristic is RINS [3], which can be seen as the improvement analog of LP-and-fix. Here we explore a sub-space defined by fixing those integer variables that take the same value in both the LP relaxation solution and a MIP feasible solution. Again letting $(\hat{x}, \hat{y})$ be and LP relaxation solution, and letting $(\bar{x}, \bar{y})$ is a MIP feasible solution for the problem. the RINS problem is

$$
\begin{aligned}
(RINS) \qquad & \min_{(x,y)\in\mathbb{R}^n\times\mathbb{R}^p} cx + fy \\
& \text{s.t. } Ax + Gy \geq b \\
& \quad x \geq 0, y \in \{0,1\}^p \\
& \quad y_j = \bar{y}_j \text{ for all } j \text{ with } \bar{y}_j = \hat{y}_j \ .
\end{aligned}
$$

2.3 Local Branching

Another type of improvement primal heuristic is Local Branching. In this heuristic we try to search a neighborhood around a MIP feasible solution. To do this search first an integer k is chosen. Then the neighborhood around a MIP feasible solution is defined as those y vectors that do not differ from the MIP feasible solution in more than k coordinates. As a result the local branching problem is defined as follows:

$$
\begin{aligned}
(LocalBranching) \qquad & \min_{(x,y)\in\mathbb{R}^n\times\mathbb{R}^p} cx + fy \\
& \text{s.t. } Ax + Gy \geq b \\
& \quad x \geq 0, y \in \{0,1\}^p \\
& \quad \sum_{j:\bar{y}_j=0} y_j + \sum_{j:\bar{y}_j=1} (1 - y_j) \leq k \ .
\end{aligned}
$$

3 The Parallel Macro Partitioning (PMaP) Framework

Here we explain the Parallel Macro Partitioning (PMaP) framework and its elements. We assume we have n processors, and we will discuss the work each processor does. We also have some data pools in the framework which can be considered as simple databases. We will briefly explain these pools in this section.

3.1 Processors

Brancher Processor. This processor generates subproblems and partitions the feasible region of the MIP. The brancher starts solving the MIP using branch-and-bound algorithm. At each node of the branch-and-bound tree, if there exists any MIP feasible solution in the pool of feasible solutions, the brancher generates a RINS problem and puts it in the subproblem pool. Then it adds the complement of the RINS cut (1) to the problem that it is solving:

$$
\sum_{j\in S^0} y_j + \sum_{j\in S^1} (1 - y_j) \geq 1 \tag{1}
$$

S^0: Set of variable indices with value of 0 for the variable in the feasible solution.
S^1: Set of variable indices with value of 1 for the variable in the feasible solution.

This cut guarantees that the part of the feasible space which is being processed in this RINS problem won't overlap with the problem which the brancher is solving. If there is no feasible solution in the feasible solution pool, the brancher generates a LP-and-FIX problem, puts it in the subproblem pool, and adds the complement of the LP-and-FIX cut

$$\sum_{j \in S^0} y_j + \sum_{j \in S^1} (1 - y_j) \geq 1 \tag{2}$$

S^0: Set of variable indices with value of 0 in the LP relaxation solution.
S^1: Set of variable indices with value of 1 in the LP relaxation solution.

The same process is done for Local Branching, and both new subproblems and complement cuts are generated at each node of branch-and-bound tree.

The primary purpose of the brancher is not to find feasible solutions, but rather to quickly create work for the worker processors. However, the brancher may well find feasible solutions during the branch-and-bound process. Whenever it finds a feasible solution, it writes it into the feasible solutions pool.

Worker Processors. They are $n-2$ processors that solve subproblems that are assigned to them using a branch-and-cut algorithm. During the solution process, they frequently check the feasible solution pool to see if any new solution has been found by other processors. If so, they update their cutoff value. Whenever they find a feasible solution they write it to the feasible solutions pool, and when they finish solving a subproblem, they send a message to the assigner processor to let it know that they are idle.

Assigner Processor. This processor looks for new subproblems in the feasible solutions pool. It also keeps track of the status of worker processors (busy or idle). As soon as a new subproblem appears in the subproblem pool, the assigner gets that and looks at the status of the workers. If there is an idle worker, the assigner gives the subproblem to the worker to solve. Otherwise, it waits until one of the workers declares that it is free.

3.2 Data Pools

Sub-problem Pool. All the subproblems generated by the brancher processor go to this pool. As was described earlier, the subproblems wait here until they are assigned to a worker processor to be solved.

Feasible Solution Pool. All the feasible solutions found by the brancher and worker processors are put here to be shared between all the processors. The brancher uses them to update it's cutoff value and generate subproblems. Workers use them to update their cutoff value.

4 Implementation and Numerical Results

4.1 Implementation

We have implemented PMaP using the free MIP solvers MINTO [6] and Coin-Cbc [1]. The brancher is implemented using MINTO and the workers are implemented using Coin-Cbc. The communication between the processors is done using either MPI (Message Passing Interface) or text files. The feasible solutions and subproblems pools are basically text files. We have currently implemented RINS and LP-and-FIX subproblems and cuts; we intend to implement local branching subproblems and cuts in the very near future. In general, PMaP is still at a preliminary stage, and we expect that we will be able to significantly improve its results in the very near future.

4.2 Results

Here we present the result of runs of PMaP on some of hard problems from MIPLIB 2003 (`http://miplib.zib.de/miplib2003.php`). As a benchmark, we use parallel CPLEX 10. The runs were performed on the Datastar machine in the San Diego Supercomputer Center (`http://www.sdsc.edu/us/resources/datastar/`). Because of the multi-threaded nature of parallel CPLEX, we could run it on at most 32 processors (maximum number of processors with shared memory) on Datastar. For fair comparison, we used the same number of processors for PMaP. Each instance ran for 30 minutes and Table 1 shows the best feasible solution each solver could find. As suggested by the table, the performance of PMaP is competitive with that of parallel CPLEX. This is significant, since serial CPLEX significantly outperforms serial Cbc on a single machine.

Table 1. The best feasible solution found by PMaP and parallel CPLEX 10 using 32 processors over 30 minutes

Problem	CPLEX	PMaP	Optimal Solution
glass4	1.60001e+09	1.65000e+09	1.20001e+09
markshare1	7	4	1
markshare2	25	16	1
portfold	-20	-21	-31
atlanta-ip	-	95.0098	90.0099
sp97ar	6.62541e+08	6.8753e+08	?
seymor	425	425	423
danoint	65.6667	65.6667	65.6667
dano3mip	698.6296	709.9629	?
swath	730.1	577.367659	467.407
net12	255	-	214

5 Conclusions

Our initial results suggest that PMaP is competitive with state-of-the-art commercial parallel softwares on the same architecture, even though PMaP itself is written entirely using open source code. Moreover, PMaP can, in principle, make use of hundreds or even thousands of processors simultaneously, and we intend to test its performance on truly massively parallel systems in the very near future. In addition, we expect that further development of PMaP will only enhance its performance. For example, the use of local branching subproblems and cuts may considerably enhance its performance. Moreover, it will likely be possible to use concepts of evolutionary methods (similarly to [9]) to quickly define even more subproblems that can simultaneously improve a upon a set of feasible solutions.

References

1. Coin-or project, https://projects.coin-or.org/Cbc
2. CPLEX. CPLEX User's Manual. CPLEX: a division of ILOG, Version 10 (2005)
3. Danna, E., Rothberg, E., Le Pape, C.: Exploring relaxation induced neighborhoods to improve MIP solutions. Mathematical Programming Series A 102, 71–90 (2005)
4. Gendron, B., Crainic, T.G.: Parallel brach-and-bound algorithms: Survey and synthesis. Operations Research 42, 1042–1066 (1994)
5. Ladányi, L., Ralphs, T.K., Trotter Jr., L.E.: Branch, cut, and price: Sequential and parallel. In: Jünger, M., Naddef, D. (eds.) Computational Combinatorial Optimization. LNCS, vol. 2241, pp. 223–260. Springer, Heidelberg (2001)
6. Nemhauser, G.L., Savelsbergh, M.W.P., Sigismondi, G.: Functional description of MINTO, a mixed integer optimizer. Operations Research Letters 15, 47–58 (1994)
7. Dash Optimization. Xpress optimizer (2008), http://www.dashoptimization.com/
8. Ralphs, T.K., Ladanyi, L., Saltzman, M.J.: Parallel branch, cut, and price for large-scale discrete optimization. Mathematical Programming 98, 253–280 (2003)
9. Rothberg, E.: Exploring relaxation induced neighborhoods to improve MIP solutions. INFORMS Journal on Computing 19, 1060–0189 (2007)

Guiding Stochastic Search by Dynamic Learning of the Problem Topography

Yehuda Naveh

IBM Haifa Research Lab, Haifa University Campus, Haifa 31905, Israel
naveh@il.ibm.com

Abstract. We study experimentally the effect of dynamic learning of the problem topographic characteristics on stochastic search.

1 Introduction

In this short paper we present experimental results showing that a general-purpose stochastic solver can be beneficially guided by dynamic learning of the topographic characteristics of the search space. We show that those characteristics are a property of the problem (and not of a particular run), and that different problems feature distinctively different characteristics.

2 Results

We study a few constraint-satisfaction and optimization problems taken from different domains. For each problem, we list the set of variables and the set of constraints used to model the problem. Each constraint is associated with a cost function. The cost of the problem on a particular state of variables is the sum of costs of constraints acting on those variables.

The experiments were conducted in the following manner: On each problem, we ran SVRH (Simulated Variable Range Hopping [1]) which is a general-purpose hill-climbing algorithm used by IBM's stochastic constraint solver Stocs. The main feature of SVRH relevant to this work is that it checks complete assignments which are, *a-priori*, at an arbitrary distance from the current complete assignment. We call such a check an 'attempt', and the distance of the checked assignment from the current one is the 'attempt size'. If the checked assignment is of lower cost than the current assignment then the attempt is 'successful', otherwise it is 'unsuccessful'.

For each problem, we plot the number of successful attempts as a function of the attempt size. This defines the characteristic step sizes for the problem, or its 'signature'. The results are obtained with the 'domain knowledge' and 'learn' features of SVRH turned off, so they represent the actual, unbiased, successes of attempts. At the end of the section, we compare runtime of Stocs on several problems, with the 'learn' feature turned off and on. All experiments were performed on a single-core Intel (TM) 3GHz PC running Red-hat Linux.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 349–354, 2008.

Social Golfer: This is problem 10 from CSPLib. Given number of weeks W, number of groups n, and group size k, we define nW variables, each a bit-string of length kn. The on-bits of each variable represent all persons playing in a given group at a given week. The constraints are: (1) nW unary constraint on each of the variables, forcing the number of on-bits $n_{\mathbf{on}}$ to be exactly k. Cost function is $|n_{\mathbf{on}} - k|$; (2) W n-ary constraints, each operating on the set of variables corresponding to a given week, forcing that all golfers be present in at least one group. Cost function is number of off-bits in the bitwise-OR of the set of variables; and (3) $nW(nW-1)/2$ binary constraints on each pair of variables, forcing that the scalar product between the variables is equal or less than 1. Cost function is $\max(V_1 \cdot V_2 - 1, 0)$. Results for this model are presented in Fig. 1. In this and the subsequent figures, attempt size is measured as the number of bit-flips realizing the attempt, i.e. the Hamming distance between the current state and the checked state.

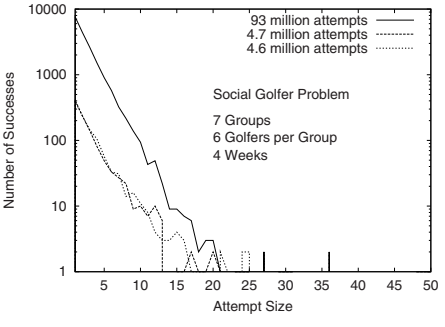


Fig. 1. Successful attempts for the social golfer problem

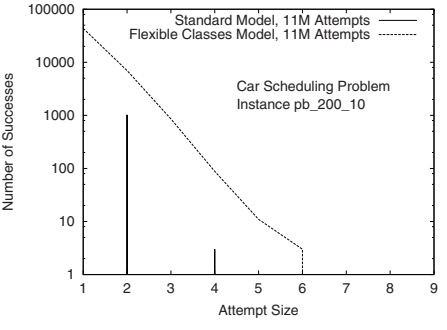


Fig. 2. Successful attempts for the two models of the car scheduling problem

A striking conclusion can be drawn from Fig. 1. Namely, the characteristic step size is a solid property of the problem, and is not related to the particular run of the algorithm. This is manifested in two ways. First, the curves of two different runs with the same number of attempts almost coalesce. Second, even when we raise the number of attempts by more than an order of magnitude, the characteristic length scale for escape remains the same — at around 20. We show below that this conclusion is indeed general: Even though the success curves of different problems behave very differently, the curve for each particular problem is highly reproducible.

From a practical point of view, this means that for this problem instance there is absolutely no sense in attempting step sizes larger than 20. On the other hand, when stuck in a persistent local minima, step sizes of the order of 15-20 *must* be tried in order to escape.

Car Scheduling: This is problem 1 from CSPLib. We define a series of N variables V_q $(1 \leq q \leq N)$, each representing an individual car, ordered in a sequence. Each variable is a bit-string of length k, where bit i is on iff the

car requires option i. The constraints on these variables are: (1) One N-ary constraint, forcing the number of cars in each class j to be equal to n_j. Cost function of the constraint is $F\sum_j |a_j - n_j|$ where a_j is the actual number of cars having the options of class j under the current state. F is a flexibility factor discussed below; and (2) k N-ary constraints, each forcing the sequence of variables to adhere to the capacity of the corresponding option. Cost function is $\sum_{p=1}^{N} \max\left(\sum_{q=p}^{\min(p+l_i-1,N)} V_{q,i} - m_i, 0\right)$, where i is the option tied to the constraint, and $V_{q,i}$ is the value of bit i of variable q.

In the first, "standard", model we take $F \to \infty$, meaning that the number of cars in each class remains fixed throughout the search. In the second, "flexible classes" model, $F = 1$. Then search proceeds through states which violate either the capacity constraints, or the constraints on the number of cars per class, or both. Both models describe equivalent CSP's (because they are identical when no constraint is violated), but distinct optimization problems (for infeasible problems, the optimal solution may be different for the different models).

Results for the two models are shown in Fig. 2. In the standard model, only even step-sizes are possible, meaning that the topography is characterized by a checker-board pattern, with single-bit local minima separated by single-bit barriers in all principal directions. It may take a while to infer this topography directly from the model, and so the automatic learning is crucial here when using a general-purpose solver. The second model is characterized by a very different topography, reminiscent of the 'social-golfer' topography (Fig. 1), but with a significantly smaller cutoff value (6 instead of 20).

Low Autocorrelation Binary Sequence (LABS): This is problem 5 from CSPLib. We define a single variable V, represented by a bit-string of length N. The constraints on these variables are: $N-1$ unary constraints, each forcing the corresponding correlator to vanish. Cost function of the k'th constraint is C_k^2, where $C_k = \sum_{i=0}^{n-k-1} s_i s_{i+k}$, $s_i = 2V_i - 1$, and V_i is the value of the ith bit of V.

Results are shown in Fig. 3. Two features are immediately observed. First, the characteristic curves are dramatically different than those of the social-golfer and car-sequencing problems. In fact, there is no step-size cutoff in the LABS problem, and cost-reducing steps are observed on all length scales. Second, the curves of the two different runs virtually coalesce (though, by examining the actual data, one sees that they are not identical). This implies that even the finest structures of the characteristic curves, such as the small cusp around attempt size of 12, should not be treated as noise, and are a result of some distinct feature of the problem's topography.

Floating Point Unit Verification: This is an industrial problem described in detail in [1,2]. We define two variables V_1 and V_2, represented as bit-strings of length N. Given a mask M defining a set of bits fixed to either 0 or 1, and a number k, we add the following constraints: (1) Two unary constraints forcing V_1, V_2 to have exactly k bits on, with cost $|n_{\text{on}} - k|$, where n_{on} is the number of on-bits; and (2) One binary mask constraint on the product of the two variables,

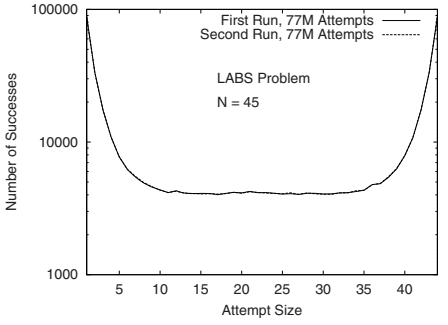


Fig. 3. Successful attempts for two runs of the LABS problem with $N = 45$

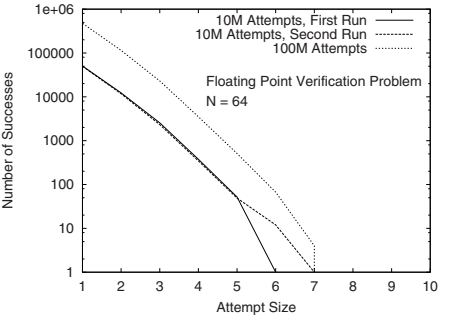


Fig. 4. Successful attempts for the floating point verification problem

forcing $A = V_1 \times V_2$ to be masked by M, with cost $\sum_{i|i \text{ is a masked bit}} |A_i - M_i|$, where A_i is the value of the ith bit of A, and M_i the value fixed by M for bit i. Results are shown in Fig. 4 for the parameters $N = 64$, $k = 10$, and M = 11110000xxxxxxxx11110000xxxxxxxx11110000xxxxxxxx11110000xxxxxxxx, where 'x' in M signifies an unconstrained bit. The results appear quite similar to those of the flexible model of the car-sequencing problem. Once again, we see that the results are robust to different runs on the same problem, and the inferred cutoff length-scale is practically independent of the absolute number of attempts.

Random Expression Problems: [3] introduced the concept of random-expression CSP's. It argued the this form of randomness can emulate real-world problems more realistically than random-table CSP's, while still maintaining all the benefits of a random ensemble. One ensemble of random-expression problems, corresponding to a particular random grammar was provided at `http://www.haifa.il.ibm.com/projects/verification/octopus/random/`.

Fig. 5 shows results on problems from this ensemble. In contrast to the previous results, here different curves correspond to runs on different problems, not to different runs on the same problem. The key in the figure specifies the compound probability of each run (i.e., the probability of following a rule which creates a compound formula, rather than an atomic formula), the number of variables, and the domain of all variables. The number of constraints was 50 in all runs. Two runs on the same problem ($P = 0.1$, $|V| = 50$, $D = [0, 10]$) are also shown.

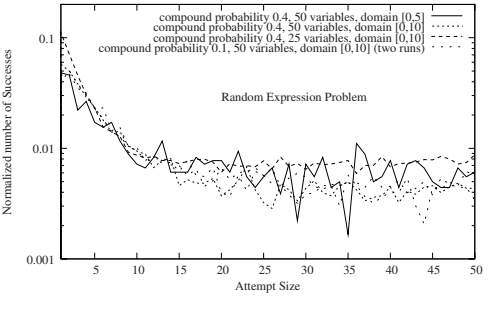


Fig. 5. Successful attempts for the random expression problem. The y-axis is normalized to all successes (see text).

The y-axis in Fig. 5 shows the *normalized* number of successes, which is the number of successes at each attempt size, divided by the total number of successes. For the hardest problems, total number of attempts was around two million, and the total number of successes around ten thousand. Those numbers were an order of magnitude smaller for the easier problems.

What is remarkable about the results of this sub-section is that even though the random expression problems are significantly different from each other, their signatures appear to be more or less the same. In some way, this is a little disappointing because in [3] it was postulated that different instances of the random grammar may emulate different real-world problems. Here we see that this in not the case, at least for the grammar presented in the above URL. On the other hand, the results here again confirm the main point of this paper, namely that similar problems have a similar signature. We still suggest that different random grammars (as opposed to the same grammar with different random parameters) can produce distinctively different problems, but this needs to be verified in a more comprehensive work on random grammars.

Comparison: We present a comparison between results obtained by SVRH running with learn and without learn. The results obtained with learn are identical to those of [1]. In the experiments without learn, everything was kept identical to the experiments with learn, including the availability of domain knowledge [1], even though domain knowledge was disabled in the experiments of the previous subsections. When learning was disabled, each time the algorithm needed to create a "learned" attempt, it instead chose a random attempt with size 1, 2, 3, or 4 with probability 0.5, 0.3, 0.15, 0.05, respectively. The results are summarized in Tables 6 and 7.

	learn enabled	learn disabled
Instances solved	133	35
Average solution time	0.97 sec	12.1 sec

Fig. 6. Results on floating-point verification benchmark with 133 instances. Time-out is 1 minute.

N	learn enabled	learn disabled
45	1.32 hours	time-out
46	1.74 hours	time-out
47	2.35 hours	time-out

Fig. 7. Results on three instances of LABS problem. Time-out is 24 hours.

3 Conclusions

We have studied the effect on stochastic search of learning the high-level characteristics, or signatures, of the cost-function-induced topography representing CSPs and optimization problems. We have shown that different problems have well-defined and distinct characteristics, and learning these characteristics may have a drastic positive effect on the ease of solvability by general-purpose stochastic solvers. This work complements and reinforces the modern body of work on

hyper-heuristics [4], in which general-purpose solvers are enhanced with a capability to dynamically learn and decide the suitable heuristics for a given specific problem.

References

1. Naveh, Y.: Stochastic solver for constraint satisfaction problems with learning of high-level characteristics of the problem topography. In: Proceedings of the 1st International Workshop on Local Search Techniques in Constraint Satisfaction (LSCS 2004). (2004)
2. Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., Shurek, G.: Constraint-based random stimuli generation for hardware verification. AI Magazine 28, 13–30 (2007)
3. Sabato, S., Naveh, Y.: Preprocessing expression-based constraint satisfaction problems for stochastic local search. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 244–259. Springer, Heidelberg (2007)
4. Burke, E.K., Kendall, G.N., Newall, J., Hart, E., Ross, P., Schulenburg, S.: Hyper-Heuristics: An Emerging Direction in Modern Search Technology. In: Handbook of Meta-Heuristics, pp. 457–474. Springer, Heidelberg (2003)

Hybrid Variants for Iterative Flattening Search

Angelo Oddi[1], Amedeo Cesta[1], Nicola Policella[2], and Stephen F. Smith[3]

[1] Institute for Cognitive Science and Technology, CNR, Rome, Italy
{angelo.oddi,amedeo.cesta}@istc.cnr.it
[2] European Space Agency, Darmstadt, Germany
nicola.policella@esa.int
[3] Carnegie Mellon University, Pittsburgh, PA, USA
sfs@cs.cmu.edu

1 Introduction

Iterative Flattening Search (IFS) [1] is an iterative improvement heuristic schema for makespan minimization in scheduling problems. Given an initial solution, IFS iteratively interleaves a *relaxation-step*, which randomly retracts some search decisions, and an incremental solving step (or *flattening-step*) to recompute a new solution. The process continues until a stop condition is met and the best solution found is returned. In two subsequent works the performance of the original IFS procedure has been improved through refinements of the basic search schema. [2] proposes a simple and effective extension of IFS, which iterates the relaxation step multiple times. The resulting algorithm found many new upper bounds for the reference benchmark of Multi Capacity Job Shop Scheduling (MCJSSP) problems and produced solutions within 1% of the best upper bounds on average. Additional optimal solutions and improvements were obtained in [3]: this approach follows the IFS schema using different engines for the flattening and relaxation steps. In this paper we combine basic 'component' strategies to obtain hybrid variants (until now, no attempt has been made in this direction) and perform a detailed experimental evaluation of their performance. Specifically, we examine the utility of: (1) operating with different relaxation strategies; (2) using different searching strategies to built a new solution. We present a two-step experimental evaluation: (a) an extensive explorative evaluation with a spectrum of parameter combination; (b) a time-intensive evaluation of the best IFS combinations emerged from the previous. The experimental results shed light on weaknesses and strengths of the different variants improving the current understanding of this family of meta-heuristics.

2 Iterative Flattening Search

Figure 1 shows the IFS algorithm which alternates Relaxation and Flattening steps until a better solution is found or a maximal number of iterations is executed. The procedure takes two parameters as input: (1) an initial solution S; (2) a positive integer $MaxFail$, which specifies the maximum number of non-makespan-improving steps that the algorithm will tolerate before terminating. After initialization (Steps 1-2), a solution is repeatedly modified within the while loop (Steps 3-9) by the application of the RELAX and FLATTEN procedures. In the case that a better makespan solution is found (Step 6), the new solution is stored in S_{best} and the *counter* is reset to 0.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 355–360, 2008.

Otherwise, if no improvement is found after $MaxFail$ moves, the algorithm terminates and returns the best solution found. In this work we consider an uniform framework to combine and experimentally evaluate different IFS strategies. A preliminary introduction of the framework is given in [4]. Specifically, in the following we examine the utility of (1) operating with different relaxation strategies, one targeted to remove decisions on the solution critical path and another one that considers all decisions as candidates for retraction; (2) using different strategies for constructing a new solution, one posting precedence constraints among the activities and another one based on setting activity start times.

```
IFSSEARCH(S,MaxFail)
1.  S_best ← S
2.  counter ← 0
3.  while (counter ≤ MaxFail) do
4.      RELAX(S)
5.      S ←FLATTEN(S)
6.      if Mk(S) < Mk(S_best) then
7.          S_best ← S
8.          counter ← 0
9.      else counter ← counter + 1
10. return (S_best)
```

Fig. 1. The IFSSEARCH general schema

Given a solution S, a *Relaxation* procedure transforms a feasible schedule into a possibly resource infeasible, but temporally feasible, by removing some search decisions. We have considered two of these strategies. The first one, used in [1,2], removes at random precedence constraints between pair of activities on the critical path of the solution, hence is called *cp-based relaxation*. This procedure takes as input the probability p_r of removing a decision on the critical path and the number of time ($MaxRlxs$) the process is iterated on the current critical path. The second procedure, introduced in [3], starts from a solution S represented as a Partial Order Schedule (POS). Roughly speaking, in a POS solution activities which require the same resource units are linked via precedence constraints into precedence *chains*. Given this structure, each precedence constraint represents a *producer-consumer* relation, allowing each activity 'to know' the set of predecessors that supply the units of resource it requires for execution. In this way, the resulting network of chains can be interpreted as a flow of resource units through the schedule; each time an activity terminates its execution, it passes its resource unit(s) on to its successors. The relaxation procedure randomly selects some activities with probability p_r and *breaks* some of its chains by removing the activities from the created resource flows (hence the name *chain-based relaxation*).

We have implemented two general solution schema. First, the flattening step used in [1], *Precedence Constraint Posting Search* or *PCPS*, is inspired by prior work on the Earliest Start Time Algorithm (ESTA). The algorithm is characterized by a two-phase solution generation process. The first step *constructs an infinite capacity solution*. The second step *levels resource demand by posting precedence constraints*. Resource constraints are super-imposed by projecting "resource demand profiles" over time. Detected resource conflicts are then resolved by iteratively posting simple precedence constraints between pairs of competing activities. The second solving procedure, *Set Start Time Search* or *SSTS*, is based on the idea of searching the set of possible assignments to the activity start-times. In particular, our implementation of SSTS can be seen as a *serial scheduling schema* adopting the *latest finish time* (LFT) priority rule, which branches the search on the possible earliest start times. Both the procedures are implemented as a depth-first backtracking search using an input parameter α, which is used to limit

the number of backtracking steps. In particular, the procedures return the solution found with minimal makespan, within αn steps, where n is the number of problem's activities.

Search Variants. At present we are working at uniformly evaluating strengths and weaknesses of the single IFS component strategies proposed so far. According to this idea we propose the following IFS procedures:

- Two procedures based on PCPS search, one uses the *critical-path* relaxation – named PCPS+CPRELAX – and another one that uses the *chain-based* relaxation – named PCPS+CHAINRELAX.
- Two IFS procedures based on SSTS search, each of them using different relaxation schema hence called SSTS+CPRELAX and SSTS+CHAINRELAX.
- A new IFS procedure – called PCPS+COMBORELAX – which combines PCPS and a relaxation phase that mixes the two relaxation options: this procedure uses the *chain-based* relaxation, except when an improved solution is found within the IFS loop (Step 6 in Fig. 1). In this case, the relaxation procedure is temporary switched to CPRELAX.
- A new IFS procedure – called SSTS+COMBORELAX – which coincides with the previous one, but uses the SSTS search procedure.

It is worth reminding that the first four IFS strategies, combines already known procedures. Alternatively, the last two procedures propose new algorithms.

3 Experimental Analysis

We consider the Multi-Capacity Job-Shop Scheduling Problem, MCJSSP, as a basis for evaluating the performance of our search procedures. In the analysis below we refer to the benchmarks used also in previous works (benchmark of 80 problems subdivided in Sets A, B, C and D with a number of activities ranging from 100 to 900). We conducted a two-step experimental evaluation. First, we performed an explorative evaluation on Set C (a quite representative subset of the MCJSSP instances with a number of activities ranging from 300 to 600). This phase aims at selecting the best variants and parameters for the second phase. The second more intensive set of experiments is performed on the whole benchmark. All algorithms were implemented in Allegro Common Lisp and were run on a Pentium 4 processor 2.6 GHz, under Linux.

Explorative Experiments. The settings for the IFS strategies used at this stage were the following: (1) for both PCPS and SSTS, different amounts of backtracking were considered by setting α to the percentage values $\alpha \in \{0, 5, \ldots, 30\}$ – for example, the value 10 means that the procedure executes a maximum number of backtracking steps equal to 10% of the number of problem's activities; (2) the probability values p_r for the critical-path relaxation and the chain-based relaxation strategies were set to the same percentage values $p_r \in \{10, 15, 20\}$; (3) the parameter $MaxRlxs$ was set to $MaxRlxs = 6$ for the critical-path based relaxation; (4) a 400 second timeout values was imposed for each problem instance; (5) for each strategy we set $MaxFail = 3200$ (the maximum number *non improving* steps that the algorithm tolerates before

Table 1. Set C – preliminary experiments

	α	PCPS			SSTS		
		$p_r = 10$	$p_r = 15$	$p_r = 20$	$p_r = 10$	$p_r = 15$	$p_r = 20$
cpRelax	0	9.4	9.2	8.7	15.4	15.4	16.4
	5	6.8	6.5	6.8	16.0	16.3	16.3
	10	7.3	6.2	6.3	16.8	16.3	15.9
	15	6.9	6.6	6.4	15.5	15.7	16.3
	20	7.3	6.0	6.4	16.1	16.2	16.9
	25	7.2	6.6	6.9	15.8	15.5	15.3
	30	6.8	6.7	6.7	16.7	15.9	15.8
chainRelax	0	5.8	6.9	8.3	15.5	16.1	15.1
	5	3.0	3.8	5.4	15.7	15.1	15.1
	10	2.8	3.9	5.4	14.8	15.8	15.0
	15	2.6	3.4	5.3	15.1	15.2	14.7
	20	**2.5**	3.4	5.3	14.5	14.5	15.3
	25	**2.5**	3.5	5.5	15.7	15.1	15.0
	30	2.6	3.9	5.2	14.8	15.2	14.4
comboRelax	0	5.7	6.8	8.3	15.4	16.3	15.2
	5	3.0	4.1	5.5	14.8	14.4	15.1
	10	3.0	4.3	5.1	15.8	15.8	15.6
	15	**2.1**	3.4	5.7	15.2	15.2	15.5
	20	2.7	3.6	4.9	15.1	15.1	14.9
	25	2.6	3.3	5.1	14.7	14.7	14.6
	30	**2.5**	3.4	5.1	15.0	15.0	15.4

termination). Table 1 compares the performance of the proposed IFS strategies with respect to the value $\Delta LWU_{\%}$, which represents the average percentage deviation from the Lawrence upper bound. In particular, given a numeric value in the table (for example 2.1), the corresponding IFS strategy is given by reading the column's label (PCPS or SSTS – the solving strategy), and the row's label (one among CPRELAX, CHAINRELAX or COMBORELAX – the adopted relaxation strategy). Hence, within the identified sub-table, given the numeric value, we have the corresponding values for the parameters α and p_r (for example, performance value 2.1 is obtained by using the combination PCPS+COMBORELAX with values $\alpha = 15$ and $p_r = 10$).

The results in Table 1 show evidence of the fact that, within the same computational framework, PCPS search performs better than SSTS. CHAINRELAX variants perform much better than CPRELAX. A possible explanation of the latter trend is that critical-path relaxation is more targeted to directly reduce the makespan of a solution becoming more prone to be trapped in local minima. On the contrary, the chain-based relaxation removes activities independently from the critical path, hence its search has an inherently higher degree of *diversification* that explains the better performance observed. COMBORELAX performs slightly better than CHAINRELAX and the best performance is obtained by combining PCPS and COMBORELAX (value 2.1 obtained with $\alpha = 15$ and $p_r = 10$). This is a first confirmation of the intuition behind the definition of COMBORELAX. That is, the use of a critical path relaxation has the function of converging more quickly to a local minima when the default chain-based relaxation strategy has driven the search close to a local minima. For this reason in PCPS+COMBORELAX, the improvement of the objective function is used, within the IFS loop, to trigger the critical-path based strategies.

Table 2. $\Delta LWU_{\%}$ values on the complete benchmark

	Set A	Set B	Set C	Set D	All
PCPS + CHAINRELAX	**-0.13**	-1.62	0.81	0.43	-0.12
PCPS + COMBORELAX	-0.11	**-1.66**	**0.48**	**0.23**	**-0.26**

Intensive Experimentation. We use the settings which give the best results in the preliminary analysis, that is we have chosen the parameters $\alpha = 15$ and $p_r = 10$ (as above, we use the same parameter p_r for both the relaxation strategies), we imposed a timeout of 3200 seconds for each problem instance, and for each strategy we set $MaxFail = 3200$. The results shown in Table 2 reasonably confirm the behavior found in the first phase. In fact, in average (column *All*) the hybrid schema COMBORELAX outperforms the chain-based relaxation strategy.

However, we can notice some differences when we consider the results for each subset (A, B, C, and D) separately. In particular, CHAINRELAX outperforms COMBORELAX on the Set A, the subset with the smallest problem sizes (they range from 100 to 225 activities). COMBORELAX maintains a positive advantage on each of the other subsets, the maximum gap is for the set C and D (0.81 vs. 0.48 and 0.43 vs. 0.23). Our explanation is that for the Set A (and also B), after 3200 seconds of run-time the behavior of the algorithm PCPS+CHAINRELAX and PCPS+COMBORELAX enters into a *stable* phase, where they show similar performance, whereas for the more challenging and greater in size problems, the algorithms are still in a transient phase, where COMBORELAX maintains a performance advantage on CHAINRELAX.

4 Conclusions

This paper presents our current results on a framework to combine and experimentally evaluate different IFS strategies. It specifically examines the utility of: (i) operating with different relaxation strategies; (ii) using different strategies to built a new solution. The presented experimental results clarify some weaknesses and strengths of the ideas proposed over the past years and suggest new effective and efficient IFS procedures. From the current experimentation we have empirical evidence of the fact that within the same computational framework PCPS search performs better than SSTS and the best performing procedures are PCPS+CHAINRELAX and PCPS+COMBORELAX. In a future work, we will consider a more sophisticated version for the solving strategy SSTS and plan to study the effect of constraint propagation within the considered backtracking search procedures.

Acknowledgments. Amedeo Cesta and Angelo Oddi's work is partially supported CNR under RSTL funds 2007. Nicola Policella is currently supported by a Research Fellowship of the European Space Agency, Human Spaceflight and Explorations Department. Stephen F. Smith's work is supported in part by the Department of Defense Advanced Research Projects Agency under contract #FA8750-05-C-0033 and by the CMU Robotics Institute.

References

1. Cesta, A., Oddi, A., Smith, S.F.: Iterative Flattening: A Scalable Method for Solving Multi-Capacity Scheduling Problems. In: AAAI/IAAI. 17^{th} National Conference on Artificial Intelligence, pp. 742–747 (2000)
2. Michel, L., Van Hentenryck, P.: Iterative Relaxations for Iterative Flattening in Cumulative Scheduling. In: ICAPS 2004. Proceedings of the 14^{th} International Conference on Automated Planning & Scheduling, pp. 200–208 (2004)
3. Godard, D., Laborie, P., Nuitjen, W.: Randomized Large Neighborhood Search for Cumulative Scheduling. In: ICAPS-2005. Proceedings of the 15^{th} International Conference on Automated Planning & Scheduling, pp. 81–89 (2005)
4. Oddi, A., Cesta, A., Policella, N., Smith, S.: Iterative Improvement Strategies for Multi-Capacity Scheduling Problems. In: COPLAS 2007. Proceedings of CP/ICAPS Joint Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems (2007)

Global Propagation of Practicability Constraints

Thierry Petit and Emmanuel Poder

École des Mines de Nantes, LINA FRE CNRS 2729
Thierry.Petit@emn.fr, Emmanuel.Poder@emn.fr

Abstract. This article shows the advantages of a variable-based framework for solving over-constrained problems with practicability constraints. The case-study is a cumulative scheduling problem with over-loads.

1 Introduction

When a problem is over-constrained, the issue is to search for solutions that remain feasible in practice despite some violations of constraints [13]. We consider over-constrained cumulative scheduling problems. Experiments compare the two usual frameworks for solving over-constrained problems as optimization problems. The first one extends Constraint Programming (CP) to handle violations as *valuations* [4]. The second one express them by extra-variables in the problem [11]. We provide three contributions: (1) A model for a cumulative over-constrained problem with practicability constraints. (2) An extension of the `Cumulative` constraint of Choco [6] to unsatisfiable problems. (3) Theoretical and experimental comparisons of the two frameworks, in terms of solving.

Relaxed Cumulative Problem. An activity consumes an amount of resource, has a release date (min. starting time) and a due date (max. ending time).

Definition 1. *Given one resource R and a set A of n activities, each one consuming an amount of R, the* `Cumulative` *constraint [2] enforces: C1: $\forall a \in A$ origin(a)+duration(a)=end(a). C2: At each point in time i, cumulated height h_i of activities overlapping i is less than a limit max_capa on R (the capacity).*

We deal with human resource cumulative problems that may be relaxed w.r.t. the capacity. The makespan m (max. due date) is fixed, making some instances not satisfiable. Time unit is 1 hour. Each day has 7 hours. Durations of the n activities are fixed, from 1 to 4 hours. Due dates are initially m, and release dates are 0. An activity consumes a fixed amount of resource: the number of persons used to perform it. The resource is upper-bounded by the total number *ideal_capa* of employees. Given that extra-employees can be hired or some activities might be performed with smaller teams than the initial ones, to obtain solutions the resource may exceed *ideal_capa* at some points in time. However, to remain feasible in practice, solutions must satisfy some rules w.r.t. over-loads.

1. At any point in time, an over-load should not exceed a given margin.
2. The total *sum* of over-loads should be reasonable (ideally minimum).

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 361–366, 2008.

3. Practicability rule : At most 3 over-loaded hours by day, and among them at most one over-load greater than 1 extra-employee.
4. Practicability rule : In two consecutive days, if the last hour of work is over-loaded then the first hour of the next day should not be over-loaded.

Motivations. CP researchers have studied large over-loaded cumulative problems without practicability rules [3], a soft global constraint for unsatisfiable one machine problems [1], and multi-objective relaxed time-tabling [13]. More generally, when not all constraints can be satisfied, we search for a compromise solution which satisfies as much as possible the constraints. This is an optimization problem with a criterion on constraint violations. Some hard constraints must be satisfied in any solution, whereas soft constraints which may be violated. *Practicability rules* specify solutions which have a practical interest.

◇ *Extending CP.* The first way to express an over-constrained problem is to extend CP with an external structure, which handles relaxation of constraints [4]. The principle is to associate a valuation with each tuple of a constraint, which is null when satisfied. Valuations are aggregated into optimization criteria. Soft constraints are violation functions, hard constraints have an "infinite" valuation for each violating tuple. Practicability rules are expressed by objective criteria.

◇ *Handling Relaxation within the CP Framework.* This approach consists of associating with each possible violation a new variable [11]. Usually integer variables are used. Values in the domain express different degree of violation for a constraint. Such domains are maintained by classical constraints. Optimization criteria are related to the whole set of violation variables. Practicability rules can be expressed by (global) constraints, integrating several violation variables.

◇ *Comparison.* The use of extra-variables to express violations does not entail any solving penalty when domains of possible values for violations are reduced monotonically during search. All search heuristics used with valued paradigms [4] can be used in variable-based paradigms [11]. All violation variables will have then a value by propagation (branching is performed only on problem variables), leading to the same search tree. For sake of modelling, two existing solving techniques specific to valued frameworks would require some research effort to be suited to our case-study. In the first one, solving is performed by variable elimination [7]. This algorithm is currently not suited to global constraints like `Cumulative` or its extensions. In the second one, valuations move from tuples to values and vice versa [8]. Even in an objective function, we don't know how to define practicability rules if valuations can increase and decrease and do not express over-loads.

◇ *Motivations for an Experimental Comparison.* Additionally to a global soft constraint expressing the core of an over-constrained problem, our goal is to investigate whether propagating globally practicabillity constraints improves the solving process or not.[1] These external constraints depend on each specific prob-

[1] The comparison of the variable-based and valued approaches in terms of modelling was performed in [11].

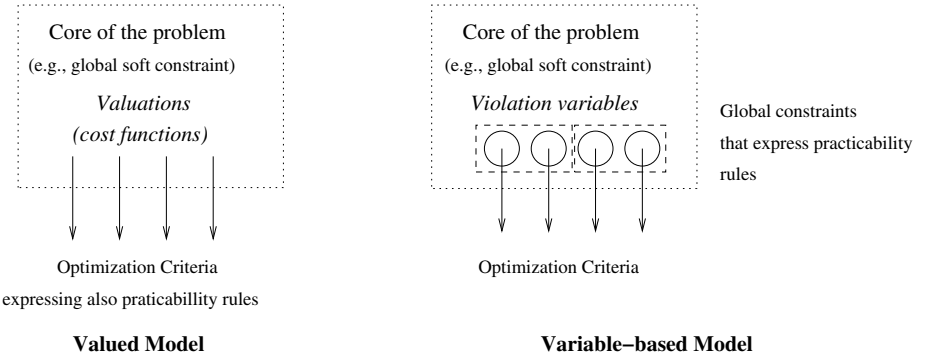


Fig. 1. Valuation-based and Variable-based models

lem, *e.g*, rules 3. and 4. in section 1. With a valuation-based encoding, practicability can be expressed by the optimization function. With a variable-based encoding, it is possible to use global constraints on violation variables.

2 Cumulative Constraint with Over-Loads

To add external practicability constraints on over-loads, it is mandatory to discretize time, while keeping a reasonable time complexity for pruning. The `SoftCumulativeSum` extends the `Cumulative` of Choco [6] by minimizing the sum of over-loads variables, and by pruning activities according to upper bounds of these variables instead of simply considering max_capa.

Definition 2. *The* `SoftCumulativeSum` *augments the* `Cumulative` *constraint with a second limit of resource* $ideal\_capa \leq max\_capa$*, with for each point in time* $i < m$ *an integer variable* $costVar[i]$*, and finally with an integer variable cost. It enforces: C1 and C2 (see* Definition 1*), C3: For each time point* i*,* $costVar[i] \geq max(0, h_i - ideal\_capa)$[2] *and C4: cost* $= \sum_{i \in \{0,\ldots,m-1\}} costVar[i]$.

Pruning techniques use a sweep algorithm, according to a cumulated profile build from compulsory parts of activities [2], and task intervals [9,5]. Please see our preliminary research report on this topic [10] for a detailed presentation of these algorithms. The main steps are the following.

- Determine the current cumulated profile and prune starts of activities according to this profile in order to avoid at any time i to be in excess of $ideal\_capa + max(D(CcostVar[i]))$.[3] The pruning is not complete but it is independent from the discretized points in time. Time complexity is $O(n \cdot \log n)$.
- Update the minimum of $costVar$ variables while computing the cumulated profile (no increase in complexity): at any time i, $min(D(costVar[i]))$ is greater or equal to the height of the profile at time i minus $ideal\_capa$.

[2] Moreover, it ensures that in ground solutions $costVar[i] = max(0, h_i - ideal\_capa)$.
[3] $min(D(x)$ and $max(D(x)$ are min. and max. values in the domain of variable x.

– (Back-)Propagate $C4$. Notably, update $min(D(cost))$ thanks to lower-bounds.
  * LB_1 is the sum of all $min(D(costVar[i]))$. Time complexity $O(m)$.
  * LB_2 is a lower bound stemming from a energetic reasoning on intervals deduced from the cumulated profile. LB_2 improves LB_1 by considering lower-bounds of *cost* variable which are local to each interval. Time complexity is $O(n^2 + n \cdot m)$. Our experiments highlighted the main importance of LB_2.

3 Variable-Based vs Valuation-Based Model

Here is the variable-based model in a pseudo-code syntax.

```
int[] ds, hs;  // fixed random durations and heights
int ideal_capa, max_capa;
IntDomainVar[] start, costVar; IntDomainVar cost;  // variables
// core of the problem
SoftCumulative(start, costVar, cost, ds, hs, ideal_capa, max_capa);
// practicability constraints
for each array "day", element of a partition of costVar[]:
  Gcc[4] ("day",...));                  // rule 3 (max 3 over-loads)
  AtMostKNotZeroOrOne("day",1));  // rule 3 (max 1 big over-load)
for(i:7..costVar.length-1):          // rule 4 (consecutive days)
  if(i%7==0): [costVar[i-1]==0 || costVar[i]==0];
// objective
minimize(cost)
```

Table 1. Number of nodes of **optimum** schedules with $n = 10$, $m = 14$, durations between 1 and 4, resource consumption between 1 and 3, $ideal\_capa = 3$, $max\_capa = 7$

Instance	*cost* value	Valuation-based Model	Variable-based Model
1	0	67 (0.08 s)	**67 (0.01 s)**
2	0	3151 (6.07 s)	**74 (0.02 s)**
3	5	372 (0.6 s)	**117 (0.1 s)**
4	0	2682 (4.4 s)	**120 (0.03 s)**
5	4	**116 (0.1 s)**	134 (0.1 s)
6	0	**62 (0.01 s)**	132 (0.04 s)
7	10	1796 (2.3 s)	**694 (0.9 s)**
8	4	391 (0.48 s)	**352 (0.45 s)**

The constraint `AtMostKNotZeroOrOne` imposes at most k values different from 0 or 1 into a set of variables. With valuations, rules 3. and 4. should be expressed in the objective, since no violation variables are available, which may be complex. We simulate this in a particular variable-based model. A valued model expresses violations by functions that, given a tuple, return a valuation. Valuations are aggregated in the objective. Given a set of valuations $\{v_1, \ldots, v_l\}$, a practibility rule answers 'yes' or 'no', to validate (or not) the current partial assignment. This can be simulated by implementing constraints that answer 'yes'

[4] Global Cardinality Constraints [12] on subsets of $costVar$.

Table 2. Number of nodes of **optimum** schedules with $n = 15$, $m = 21$, durations between 1 and 4, resource consumption between 1 and 3, $ideal\_capa = 3$, $max\_capa = 7$

Instance	*cost* value	Valuation-based Model	Variable-based Model
1	2	> 60 s	**211 (0.34 s)**
2	0	> 60 s	**178 (0.08 s)**
3	1	> 60 s	**200 (0.12 s)**
4	15	> 60 s	**27 (0.04 s)**
5	12	> 60 s	**546 (2 s)**
6	3	> 60 s	**1875 (6 s)**
7	6	2240 (11.4 s)	**160 (0.12 s)**
8	9	> 60 s	**79 (0.06 s)**

Table 3. Number of nodes of **optimum** one week schedules: $n = 25$, $m = 35$, durations between 1 and 4, resource consumption between 1 and 3, $ideal\_capa = 3$, $max\_capa = 7$

Instance	*cost* value	Variable-based Model
1	3	858 (5.8 s)
2	0	627 (1.2 s)
3	-	> 60 s
4	0	419 (0.6 s)
5	-	> 60 s
6	23	263 (1.6 s)
7	10	416 (0.9 s)
8	19	197 (0.6 s)

or 'no' given a set of values (no global propagation on $costVar$ domains). Here, values are lower bounds of $costVar$. Modified constraints are called `NFGcc` and `NFAtMostKNotZeroOrOne`. Except them the model remains identical, notably the `SoftCumulativeSum`, which is quite generic.[5] We assume our algorithms could be implemented with a valued solver dedicated to cumulative problems.

In our experiments, the search strategy was to assign minimum values of domains, first to start variables and after to $costVar$ and *cost* variables, which is consistent with a valued model (no branching on over-load variables). The processor was a 2.2 Ghz Intel Core 2, with 4 Go of 667 Mhz RAM.

References

1. Baptiste, P., Le Pape, C., Peridy, L.: Global constraints for partial CSPs: A case-study of resource and due date constraints. In: Proc. CP, pp. 87–102 (1998)
2. Beldiceanu, N., Carlsson, M.: A new multi-resource *cumulatives* constraint with negative heights. In: Proc. CP, pp. 63–79 (2002)
3. Benoist, T., Jeanjean, A., Rochart, G., Cambazard, H., Grellier, E., Jussien, N.: Subcontractors scheduling on residential buildings construction sites. ISS 2006 Int. Sched. Symposium, Technical Report JSME-06-203, 32–37 (2006)
4. Bistarelli, S., Montanari, U., Rossi, F., Schiex, T., Verfaillie, G., Fargier, H.: Semiring-based CSPs and valued CSPs: Frameworks, properties, and comparison. Constraints 4, 199–240 (1999)

[5] Conversely, practicability rules 3. and 4. depend on each problem instance.

5. Caseau, Y., Laburthe, F.: Cumulative scheduling with task intervals. In: Proc. JICSLP, pp. 363–377 (1996)
6. Choco: A Java library for CSPs, constraint programming and explanation-based constraint solving (2007), http://choco.sourceforge.net/
7. Larrosa, J., Dechter, R.: Boosting search with variable elimination in constraint optimization and constraint satisfaction problems. Constraints 8(3), 303–326 (2003)
8. Larrosa, J., Schiex, T.: Solving weighted csp by maintaining arc consistency. Artificial Intelligence 159(1-2), 1–26 (2004)
9. Lopez, P., Erschler, J., Esquirol, P.: Ordonnancement de tâches sous contraintes: une approche énergétique. Automatique, Productique, Informatique Industrielle 26(5-6), 453–481 (1992)
10. Petit, T., Poder, E.: Global propagation of practicability constraints. Research report 0702, Ecole des Mines de Nantes (2007), http://www.emn.fr/x-info/tpetit/TR0702tpetit.pdf
11. Petit, T., Régin, J.-C., Bessière, C.: Meta constraints on violations for over constrained problems. In: Proc. IEEE-ICTAI, pp. 358–365 (2000)
12. Régin, J.-C.: Generalized arc consistency for global cardinality constraint. In: Proc. AAAI, pp. 209–215 (1996)
13. Rudová, H., Vlk, M.: Multi-criteria soft constraints in timetabling. In: Proc. MISTA, pp. 11–15 (2005)

The Polytope of Tree-Structured Binary Constraint Satisfaction Problems*

Meinolf Sellmann

Brown University
Department of Computer Science
115 Waterman Street, P.O. Box 1910
Providence, RI 02912
sello@cs.brown.edu

Abstract. We correct a result that we recently published in this conference series on the polytope of Binary Constraint Problems (BCPs). We had claimed that the so-called "support formulation" would characterize the convex hull of all feasible solutions to tree-structured BCPs. We show that this claim is not accurate by providing a small counter example. We then show that the respective polytope defines a facet of the stable-set polytope of a perfect graph which allows us to perform LP inference in polynomial time.

1 Binary Constraint Satisfaction

Definition 1 (Binary Constraint Satisfaction Problem)

- *A* binary constraint problem (BCP) *is a triplet* $\langle V, D, C\rangle$*, where* $V = \{X_1, \ldots, X_n\}$ *denotes the finite set of variables,* $D = \{D_1, \ldots, D_n\}$ *denotes a set of* n *finite sets of possible values for these variables (*D_i *is called the* domain *of variables* X_i*), and* $C = \{C_1, \ldots, C_m\}$ *is the set of* constraints*, where* $C_j : D_{j_1} \times D_{j_2} \rightarrow Bool$ *specifies which simultaneous assignments of values to the variables* X_{j_1} *and* X_{j_2} *are allowed. The set* $\{X_{j_1}, X_{j_2}\}$ *is called the* scope *of constraint* C_j*.*
- *An* assignment *for a BCP* $\mathcal{P} = \langle V, D, C\rangle$ *is a function* $\sigma : V \rightarrow \bigcup_{i \leq n} D_i$*. A* solution *to a BCP* $\mathcal{P} = \langle V, D, C\rangle$ *is an assignment* σ *such that* $\sigma(X_i) \in D_i$ *for all* $1 \leq i \leq n$ *and such that* $C_j(\sigma(X_{j_1}), \sigma(X_{j_2})) = \mathbf{true}$ *for all* $1 \leq j \leq m$*. The set of all solutions to a BCP* $\mathcal{P}$ *is denoted by* $Sol(\mathcal{P})$*.*

Note how, in contrast to the custom in integer programming, in CP the term "binary" is used to express that all *constraints* affect just two variables, while the size of the *domain* of each variable is not limited! The fact that the arity of the constraints is limited to two allows us to state constraints simply as sets of allowed pairs $\mathcal{R}_{j_1,j_2} = \{(k,l) \mid X_{j_1} = k,\ X_{j_2} = l \text{ ok}\}$, or, alternatively, as sets of forbidden pairs $\overline{\mathcal{R}}_{j_1,j_2} = \{(k,l) \mid X_{j_1} = k,\ X_{j_2} = l \text{ forbidden}\}$.

It is easy to see that the general BCP is NP-hard. One simple way is to reduce from graph coloring where each node is modeled as a variable that must be assigned a color

* This work was supported by the National Science Foundation through the Career: Cornflower Project (award number 0644113).

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 367–371, 2008.

such that adjacent nodes are not colored identically (i.e., the corresponding constraint on each edge $\{i,j\}$ is a not-equal constraint $\overline{\mathcal{R}_{i,j}} = \{(k,k) \mid \forall\ k\}$). Conversely, every binary constraint problem can be visualized as a *constraint network* where each node corresponds to a variable and an edge connects two nodes iff there exists a constraint over the corresponding variables. Of course, the exact semantic of the constraints is lost in that visualization. However, it is a well-known fact that any BCP whose corresponding constraint network is a tree can be solved in polynomial time [4,5].

2 The Support Formulation

In [1], we devise an IP model for BCPs by using linear constraints to specify that, when a variable X_{j_1} takes value k, variable X_{j_2} must take a value that is consistent with $X_{j_1} = k$. In that way, we enforce that each variable assignment is *supported* by a correct assignment to adjacent variables (by which we mean variables that share a constraint). The IP then reads:[1]

$$
\begin{aligned}
S_{IP} = \max \quad & \sum p_{ik} y_{ik} \\
s.t. \quad & y_{j_1 k} - \sum_{l:(k,l)\in\mathcal{R}_{j_1,j_2}} y_{j_2 l} \leq 0 && \forall\ 1 \leq j \leq m, k \in D_{j_1} && (1)\\
& \sum_{k\in D_i} y_{ik} = 1 && \forall\ i \in \{1,\ldots,n\} && (2)\\
& y_{ik} \in \{0,1\} && \forall\ i \in \{1,\ldots,n\}, k \in D_i && (3)
\end{aligned}
$$

The above formulation dominates the traditional way of expressing constraints (1) by constraints $y_{j_1 k} + y_{j_2 l} \leq 1$ for all conflicting assignments $(k,l) \in \overline{\mathcal{R}_{j_1,j_2}}$: We show that $y \in S_{LP}$ (where S_{LP} denotes the linear continuous relaxation of the above S_{IP}) implies $y_{j_1 k} + y_{j_2 l} \leq 1$ for all $1 \leq j \leq m, (k,l) \in \overline{\mathcal{R}_{j_1,j_2}}$. Given $y \in S_{LP}$, it holds that $y \geq 0$ and $\sum_{k\in D_i} y_{ik} = 1$. Moreover, for all $1 \leq j \leq m, k \in D_{j_1}$, $0 \geq y_{j_1 k} - \sum_{l:(k,l)\in\mathcal{R}_{j_1,j_2}} y_{j_2 l} = y_{j_1 k} - (1 - \sum_{l:(k,l)\in\overline{\mathcal{R}_{j_1,j_2}}} y_{j_2 l})$. Consequently, $y_{j_1 k} + \sum_{l:(k,l)\in\overline{\mathcal{R}_{j_1,j_2}}} y_{j_2 l} \leq 1$, which implies $y_{j_1 k} + y_{j_2 l} \leq 1$ for all $1 \leq j \leq m, (k,l) \in \overline{\mathcal{R}_{j_1,j_2}}$.

On the other hand, assume we are given a BCP (V,D,C) with $V = \{X_1,\ldots,X_5\}$, $D = \{\{1,2,3,4\},\ \{1,2\},\ \{1,3\},\ \{2,3\},\ \{0,4\}\}$, and $C = \{(X_1 = 4\ \vee\ X_1 = X_2),\ (X_1 = 4\ \vee\ X_1 = X_3),\ (X_1 = 4\ \vee\ X_1 = X_4),\ (X_1 \neq X_5)\}$, and we are to maximize X_5. S_{LP} returns an optimal continuous solution with value 0, which happens to be the optimal value that any integer solution can achieve. Now consider $y_{11} = y_{12} = y_{13} = 1/3$, $y_{14} = 0$, $y_{21} = y_{22} = 1/2$, $y_{31} = y_{33} = 1/2$, $y_{42} = y_{43} = 1/2$, and $y_{50} = 0$, $y_{54} = 1$. It is easy to verify that this solution, which achieves an objective value of 4, is feasible when constraints (1) are replaced by the traditional constraints. Consequently, the support formulation is never worse but in general stronger that the traditional way of linearizing binary constraint networks.

Now, in [6] we claimed that, when the given BCP was tree-structured, the linear continuous relaxation S_{LP} of S_{IP} provided a perfect characterization of the convex hull

[1] Whereby, for simplicity in constraints (1), we assume that each constraint over variables i,j induces two truth tables $\mathcal{R}_{i,j}$ and $\mathcal{R}_{j,i}$.

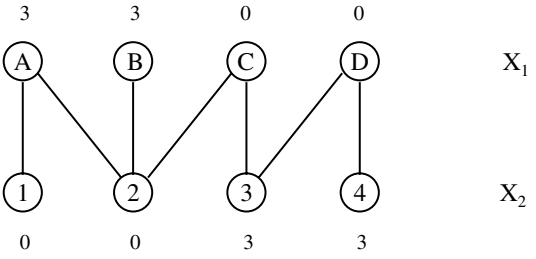


Fig. 1. A BCP with two variables X_1 and X_2. The figure shows the domains of the respective variables and an edge between values that the variables are allowed to take simultaneously. The numbers above each value show the profit that is achieved when a variable is set to this value.

of all integer feasible solutions. When trying to use the same proof-technique for the linearization of a different type of constraints, we realized the following: In the proof of Theorem 1 in [6] we consider a Lagrangian relaxation of S_{IP} and show that its value is the same as that of the linear relaxation S_{LP}, while the corresponding solution is integer and obeys all relaxed constraints. However, this does *not* imply that the corresponding solution is optimal for the original problem. Ergo, the proof is not complete. The following example shows that the proof can also not be corrected:

Example 1. Consider the BCP $\langle V, D, C\rangle$, where $V = \{X_1, X_2\}$, $D = \{D_1, D_2\}$ with $D_1 = \{A, B, C, D\}$ and $D_2 = \{1, 2, 3, 4\}$, and $C = \{C_1\}$ with $C_1 : D_1 \times D_2 \rightarrow \mathit{Bool}$ is given by the set of allowed pairs $\mathcal{R}_{1,2} = \{(A, 1), (A, 2), (B, 2),$-$(C, 2), (C, 3), (D, 3), (D, 4)\}$ (see Figure 1). Assume we achieve a profit of 3 when setting X_1 to A or B and 0 otherwise, and another profit of 3 when setting X_2 to 3 or 4 and 0 otherwise. Clearly, the maximum profit we can achieve is 3. However, when setting $y_{1A} = y_{1B} = y_{1D} = 1/3$, $y_{1C} = y_{21} = 0$, and $y_{22} = y_{23} = y_{24} = 1/3$, then y is feasible for S_{LP} and achieves a profit of 4 which is strictly greater than the optimal value of 3.

3 The Polytope of Tree-Structured BCPs

Theorem 1. *If the BCP that is given has a tree-structured constraint network, then the convex hull of all solutions to S_{IP} defines a facet of the stable-set polytope of a perfect graph.*

Proof. Consider the "conflict graph" that emerges from the given BCP: we have one node that corresponds to each variable assignment (similar to the nodes in Figure 1), and an edge between two nodes if and only if both corresponding assignments are incompatible (that is, one edge for each pair in $\overline{\mathcal{R}_{j_1,j_2}}$ and one for each pair of nodes that belong to the same variable domain). We claim that this graph is perfect. We prove this claim by showing that it is Berge [2], i.e., that it and its complement graph do not contain an odd cycle of length 5 or more with no shortcuts (a so-called "odd-hole").

Consider the conflict graph and assume that it contains a hole, i.e., a cycle with no shortcuts of length 5 or more. Wlog we may assume that this cycle is minimal and

involves nodes that belong to at least three BCP variables (otherwise there exists a shortcut between nodes that belong to the same variable). Because of the tree-structure of the given BCP, this implies that there exist two non-adjacent nodes in the cycle that belong to the same variable, which implies that there exists a shortcut in the cycle. Therefore, the cycle was not minimal.

Now consider the complement of the conflict graph. It contains edges for all pairs in $\overline{\mathcal{R}_{j_1,j_2}}$ and edges between all nodes that belong to BCP variables that have no constraint linking them. The given BCP is acyclic, thus any hole cannot involve nodes that belong to more than two BCP variables (otherwise there exists a shortcut). However, any subgraph on nodes from two variables is bipartite, which means that all cycles have even length. Consequently, there also exists no odd-hole in the complement of the conflict graph. □

According to [3], our result implies that we can characterize the polytope of tree-structured BCPs as a linear program where we the constraints enforce that the weight of each maximal clique in the conflict graph is lower or equal one. The support formulation comes close to enforcing these clique constraints, but with two shortcomings: First, in case that the support in D_{j_2} of a value $k \in D_{j_1}$ is a superset of the support in D_{j_2} of another value $h \in D_{j_1}$, the support formulation considers a clique that is not maximal (e.g., consider values A and B in Figure 1). Second, there may be other maximal clique-constraints that are not enforced.

The first shortcoming is easily addressed: Denote with $H_j(k) := \{l \in D_{j_2} \mid (k,l) \in \mathcal{R}_{j_1,j_2}\}$ the support of value $k \in D_{j_1}$ in D_{j_2}. Problematic are those $h \in D_{j_1}$ for which $H_j(h) \subseteq H_j(k)$. Then, $y_{j_1 h} \leq \sum_{l \in H_j(h)} y_{j_2 l} \leq \sum_{l \in H_j(k)} y_{j_2 l}$. And since X_{j_1} can only take either value k or h, with $y_{j_1 k} \leq \sum_{l \in H_j(k)} y_{j_2 l}$, we can enforce $y_{j_1 k} + y_{j_1 h} \leq \sum_{l \in H_j(k)} y_{j_2 l}$. We achieve a strengthened formulation:

$$SS_{IP} = \max \sum p_{ik} y_{ik}$$

$$s.t. \quad y_{j_1 k} + \sum_{h : H_j(h) \subseteq H_j(k)} y_{j_1 h} - \sum_{l \in H_j(k)} y_{j_2 l} \leq 0 \quad \forall 1 \leq j \leq m, k \in D_{j_1} \tag{4}$$

$$\sum_{k \in D_i} y_{ik} = 1 \quad \forall i \in \{1, \ldots, n\} \tag{5}$$

$$y_{ik} \in \{0,1\} \quad \forall i \in \{1, \ldots, n\}, k \in D_i \tag{6}$$

To address the second shortcoming, since there may be many other maximal cliques in the perfect conflict graph, it is not feasible to add them all to our formulation. We show another example in Figure 2. Note that there is a clique on nodes $y_{1B}, y_{1D}, y_{22}, y_{24}$ in the conflict graph whose corresponding constraint is not enforced in SS_{IP}: The solution $y_{1B} = y_{1C} = y_{1D} = y_{22} = y_{23} = y_{24} = 1/3, y_{1A} = y_{1E} = y_{21} = y_{25} = 0$ assigns a weight of 4/3 to the nodes in this clique but is feasible for SS_{IP}. The solution achieves a profit of 4 while the optimal integer solution has value 3.

In theory, for each problem it is possible to generate the relevant missing clique constraints in a lazy fashion [3]. To obtain polynomial guarantees on the number of additional constraints that need to be generated, we would have to use the Ellipsoid algorithm to solve our LPs, though. In practice, we may prefer to use a practically

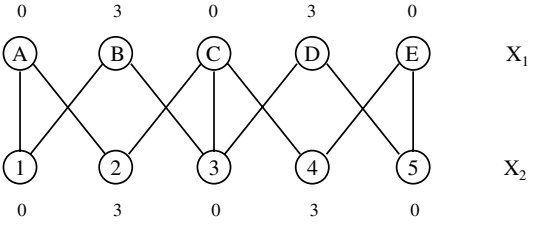


Fig. 2. A BCP with two variables X_1 and X_2. The figure shows the domains of the respective variables and an edge between values that the variables are allowed to take simultaneously. The numbers above each value show the profit that is achieved when a variable is set to this value.

efficient LP solver instead and terminate the generation of cuts early, knowing that the support formulation without any additional cuts is already strictly stronger than the traditional formulation.

References

1. Aron, I.D., Leventhal, D.H., Sellmann, M.: A Totally Unimodular Description of the Consistent Value Polytope for Binary Constraint Programming. In: Beck, J.C., Smith, B.M. (eds.) CPAIOR 2006. LNCS, vol. 3990, pp. 16–28. Springer, Heidelberg (2006)
2. Chudnovsky, M., Robertson, N., Seymour, P.D., Thomas, R.: Progress on Perfect Graphs. Mathematical Programming 97, 405–422 (2003)
3. Chvatal, V.: On certain polytopes associated with graphs. Combinatorial Theory B 18, 138–154 (1975)
4. Dechter, R., Pearl, J.: Tree clustering for constraint networks. Artificial Intelligence 38, 353–366 (1989)
5. Freuder, E.C.: Complexity of k-tree structured constraint satisfaction problems. In: AAAI, pp. 4–9 (1990)
6. Sellmann, M., Mercier, L., Leventhal, D.H.: The Linear Programming Polytope of Binary Constraint Problems with Bounded Tree-Width. In: CPAIOR, pp. 275–287 (2007)

A Tabu Search Method for Interval Constraints

Charlotte Truchet[1], Marc Christie[2], and Jean-Marie Normand[1]

[1] LINA, UMR 6241,
Université de Nantes, 2 rue de la Houssinière, 44322 Nantes, France
[2] IRISA/INRIA Rennes - Bretagne Atlantique, Campus de Beaulieu, 35042, Rennes, France
{charlotte.truchet,jean-marie.normand}@univ-nantes.fr,
marc.christie@irisa.fr

Abstract. This article presents an extension of the Tabu Search (TS) metaheuristic to continuous CSPs, where the domains are represented by floating point-bounded intervals. This leads to redefine the usual TS operators to take into account the special features of interval constraints: real variables encoded in floating points domains, high cardinality of the domains, nature of the CSP where constraints may be partially satisfied. To illustrate the expressiveness of the framework, we instantiate this method to compute an inner-approximation of a set of inequalities.

Metaheuristics, in particular Tabu Search (TS, [6,8]), have largely proven their efficiency on discrete optimization or constraint problems. In this article, we propose a framework to extend TS to continous problems on real variables. There exist a variety of optimization techniques to solve such problems, but they are usually dedicated to particular types of constraints (linear, polynomial, differentiable) and do not offer the guarantees of interval approaches [4,5]. Interval-based solvers like Realpaver [7] are dedicated to compute rigourous inner and outer-approximations of continuous problems, but often fail in efficiently computing a first solution due to the complete nature of the search process. We provide a unified way to express continuous CSPs in a TS framework that shares the reliability of interval techniques, and tackle a problem not well adressed by classical tools: the computation of a single solution for interval CSPs or optimisation problems, whatever the type of the constraints.

1 Interval Constraints

Interval arithmetics [9] offers a reliable solution to avoid rounding errors due to finite representation of real values (see IEEE754 norm). Real values are encompassed within *floating-point intervals*: a real value r may be represented by any interval I, with floating-point bounds, containing r. The set of all intervals is $\mathbb{I}$. The Cartesian product of intervals is called a *box*. The classical operators over $\mathbb{R}$ can be redefined over $\mathbb{I}$ by enforcing the fundamental property of containment[9]: the interval extension of a binary operator $\square$ is given by the smallest interval containing the results of the application of $\square$. This construct

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 372–376, 2008.

guarantees that no values are lost, but can lead to an over-estimation of the result. Real functions are extended on $\mathbb{I}$ in the same way.

A CSP defined over continuous domains is translated into intervals by taking interval variables over interval domains. The constraints are extended over $\mathbb{I}$ in the same way as functions or operators. The goal is to find a box $\boldsymbol{B} \subseteq D_1 \times \ldots \times D_n$ such that the constraints are satisfied. An interval constraint C on a box $\boldsymbol{B}$ can be *certainly satisfied* (every real vector in $\boldsymbol{B}$ satisfies C), *certainly not satisfied* (no real vector in $\boldsymbol{B}$ satisfies C). If neither of those two properties can be computed, C is said *partially satisfied*.

Several metaheuristics have been transposed to CSP on real variables. Some of them are adapted from TS [4,5,2]. However, these algorithms are based on real configurations, and employed to tackle optimization problems. Our method is closer to that of [3] for interval CSPs with non-linear differentiable constraints, or to [1] on multi-objective problems.

2 Tabu Search on Intervals

The basic components of a TS algorithm on discrete domains are well-known: a penalty function $f : D_1 \times \ldots \times D_n \rightarrow \mathbb{N}$, counting the number of violated constraints most of the times, a neighborhood function, and the exact definition of a tabu mechanism (basic tabu behaviour with size `t`, dynamic tabu, aspiration, ...). However, adapting the algorithm to new types of domains leads to a number of obstacles. Firstly, TS algorithms use many operators which are trivial over discrete configurations (equality, membership, random choice) but require specific attention over interval domains. Secondly, an interval CSP has a search space with specific properties: the domains are huge, and they are defined by two floating point numbers but include an infinite set of reals. Hence we dissect the algorithm into more precise atomic components.

2.1 Framework

Random on $\mathcal{S}$. A random function `random` : $\mathcal{S} \rightarrow \mathcal{S}$ is needed both for the random restart, and to choose a neighbor of a current configuration. An interval is characterized by its center, length and bounds. A uniform random law cannot be easily defined over these parameters; choosing the center (or one bound) first leads to a bias toward small intervals, and conversely, choosing the length first leads to bias toward centered intervals. To both ensure diversification and limit the number of possibly satisfied intervals, we choose the center first, and then the length L. L follows a repartition law $P(L \leq z) = 2b*(1-ln(2z/b-a))/(b-a)$ with an average of $(b-a)/8$.

Neighborhood exploration. A neighborhood is a subset of $\mathcal{S}$, supposedly close to the current configuration either in terms of Euclidean distance in the domain space of in terms of solution similarity. We divide this into two steps. A first operator `neighborhood` : $\mathcal{S} \rightarrow 2^{\mathcal{S}}$ computes a neighbor area, based on the same

principle than on discrete problems. Choosing configurations at a Hamming distance of 1, we can define for instance $\texttt{neighborhood}_1(< v_1 \dots v_j \dots v_n >) = \{\forall v'_j \subset D_j, < v_1 \dots v'_j \dots v_n >\}$, which tries to move a randomly chosen variable, or $\texttt{neighborhood}_2 = \{\forall 1 \leq j \leq n, \forall v'_j \subset D_j, < v_1 \dots v'_j \dots v_n >\}$ which moves every variable once.

Sampling the neighborhood. The second neighborhood operator is in charge of the neighbourhood sampling, which has to be done partially because of the cardinality of the results of `neighborhood`. This requests that $\mathcal{S}$ comes with an operator $\texttt{sample} : \mathcal{S} \to 2^{\mathcal{S}}$, to choose a good sample of boxes within a given box. This function is parameterized by the size of the sample, nb_{neigh}. We propose two possibilities for `sample`. The first one is merely random: let $\texttt{sample}_1(B)$ be a set of nb_{neigh} random configurations, with calls to `random`. The second one ensures that the box B is widely explored: $\texttt{sample}_2$ consists in cutting the box B into nb_{neigh} equal parts and choosing randomly a configuration in each part.

Penalty function. Technically, a penalty function needs to be computable in a reasonable time (if possible, incrementally), to have comparable values, and express the satisfaction of the problem. One can measure the satisfaction of a problem by counting the number of satisfied constraints. However this raises a problem on interval CSPs due to interval approximation: each configuration is either satisfied, non satisfied or partially satisfied, which questions the way to encompass the partially satisfied constaints in the count. The definition of f influences the semantic of the problem. In the problem we consider, we have focused on inner approximations of the solution set. The penalty function is thus defined as $f(s) = ps(s) + cns(s)$, where $ps(s)$ (resp. $cns(s)$) represents the cardinality of the partially satisfied constraints (resp. non-satisfied), for a configuration s. In such a case, $f(s) = 0$ iff $ps(s) + cns(s) = 0$, iff $cs(s) = p$, that is, all the constraints are satisfied by s.

Tabu mechanism. In the discrete case, the tabu mechanism relies on an equality test on the configurations. This cannot transpose to interval domains, where two intervals have a near to zero chance to be equal. A tabu configuration must forbid the search not only at a point, but in an area around it. A simple way to ensure this is to compare the configurations not w.r.t. equality, but w.r.t. intersection We add the possibility to resize the tabu configurations, in order to control their tabu influence. This `resize` function multiplies all lengths by a constant factor α. Intuitively, a configuration will be left out of the search if it crosses more than α of a tabu configuration. In the end, the tabu mechanism is enclosed into a single function $\texttt{cutTabu}(\mathcal{T}, V) = \{v \in V, \forall v_{\text{tabu}} \in \mathcal{T}, v \cap \texttt{resize}(v_{\text{tabu}})) = \emptyset\}$

2.2 Implementation

We have instantiated our framework to compute inner-approximations for two non-linear CSPs: continuous NQueens and Nlights[1]. Both have non-smooth con-

[1] Available online with implementation and results at `http://www.normalesup.org/~truchet/LSCont/`

nb_tries $\leftarrow 0$, $\mathcal{T} \leftarrow \emptyset$
repeat
  iter $\leftarrow 0$, $s \leftarrow$ `random`$(\mathcal{S})$
  repeat
    $N_0 \leftarrow$ `neighborhood`(s), $N_1 \leftarrow$ `sample`(N_0) // neighbourhood generation
    $N_2 \leftarrow$ `cutTabu`$(N_1, \mathcal{T})$ // suppression of tabu configurations
    $s \leftarrow$usualRandom$\{s' \in N_2, f(s')$ is minimal $\}$ // selection of a best neighbour
    $\mathcal{T} \leftarrow (s, \mathtt{t}) \cup$ `actualizeTabu`$(\mathcal{T})$, iter++
  until iter¿ max_iter **or** $f(s) = 0$
  nb_tries++
until nb_tries ¿ max_tries **or** $f(s) = 0$

Fig. 1. Continuous Tabu Search

straints, which impedes the use of classic derivative techniques. Neighborhood and sampling have been defined and tested with different strategies. The implementation enables to prove the expressiveness of the framework.

Table 1. Results over the Interval implementation of the Tabu Search Framework (average on 20 runs)

Benchmark	NbSteps max_iter	NbNeighbors nb_{neigh}	Tabu Tenure t	%Success	Nbrestarts max_tries=30	Time
NQueens 5	100	30	5	100%	1.0	0.08 s.
NQueens 8	500	50	5	100%	1.0	2.2 s.
NQueens 10	500	100	5	100%	1.5	5.07 s.
NLights 10	100	30	5	100%	2	0.95 s.
NLights 10	100	50	5	100%	1	0.61 s.
NLights 15	500	100	5	100%	1	7.56 s.

In terms of performances, a comparison is difficult to establish as most techniques do not compute inner-approximations of systems, and those who do, rely on a complete exploration of the search space (*e.g.* Realpaver). However, the system was able to solve instances with many variables on which Realpaver fails in a reasonnable time (10 min). Table 1 shows some results for different parameter settings.

2.3 Discussion

As a first remark, the framework allows to retrieve other algorithms (or part if them if hybridized), for instance ECTS [2] by implemeting its distance function in `neighborhood` and `resize`.

As often, the algorithm has some critical parameters. In addition to the usual TS ones, we introduce two parameters: nb_{neigh} and α. This is of course a drawback. However, nb_{neigh} can be tuned rather easily, by counting the time spent in neighborhood exploration w.r.t. the time spent in a single iteration. And α behaves in the same way as `t`.

The main advantage of the proposed framework is its genericity. All the key ingredients, which define the search strategy, are defined independantly of the CSP. Constraints can also be generic, although, truth be told, the TS algorithm is unlikely to challenge problems with well known constraint types (linear, polynomial, differentiable functions). But it offers to solve in an unified way constraints that could not be properly handled by classical approaches.

3 Conclusion

The proposed framework extends the TS metaheuristic to handle interval CSPs and their particular semantic in a generic way. Further work has to be done to try other possibilities for the new TS operators. The framework can also be extended to other kind of set CSPs, provided they come with a finite representation and computable constraints.

Some continuous problems are still a challenge for resolution methods, and the interval TS is an good alternative in that case. Experimentally, we provide two such CSPs. The interval TS works well on them because of their high number of variables and constraints, with high correlation within the variables in the constraints.

References

1. Barichard, V., Hao, J.-K.: A Population and Interval Constraint Propagation Algorithm. In: Fonseca, C.M., Fleming, P.J., Zitzler, E., Deb, K., Thiele, L. (eds.) EMO 2003. LNCS, vol. 2632, pp. 88–101. Springer, Heidelberg (2003)
2. Chelouah, R., Siarry, P.: Tabu search applied to global optimization. European Journal on Operational Research (2000)
3. Cruz, J.: Constraint Reasoning for Differential Models. IOS Press, Amsterdam (2005)
4. Fanni, A., Manunza, A., Marchesi, M., Pilo, F.: Tabu search metaheuristics for electromagnetic problemsoptimization in continuous domains. IEEE Transactions on Magnetics 35(3) (1999)
5. Franze, F., Speciale, N.: A tabu-search-based algorithm for continuous multiminima problems. International Journal for Numerical Methods in Engineering 50(3) (2001)
6. Glover, F., Laguna, M.: Tabu Search. Kluwer Academic Publishers, Dordrecht, The Netherlands (1997)
7. Granvilliers, L., FrédéricBenhamou: Algorithm 852: RealPaver: An interval solver using constraint satisfaction techniques. ACM Transactions on Mathematical Software 32(1) (2006)
8. Hansen, E.R.: Global Optimization Using Interval Analysis. Pure and Applied Mathematics. Marcel Dekker Inc, New York (1992)
9. Moore, R.E.: Interval Analysis. Prentice-Hall, Englewood Cliffs (1966)

The Steel Mill Slab Design Problem Revisited

P. Van Hentenryck[1] and L. Michel[2]

[1] Brown University, Box 1910, Providence, RI 02912
[2] University of Connecticut, Storrs, CT 06269-2155

Abstract. Recently, Gargani and Refalo (G&R) presented an elegant model for the Steel Mill Slab Design Problem (Problem 38 in the CSP LIB). Contrary to earlier approaches, their model does not use 0/1 variables but exploits the traditional expressiveness of constraint programming. G&R indicated that static symmetry-breaking constraints proposed earlier are not effective on this model, as these interact with their heuristic. Instead they use large neighborhood search to obtain solutions quickly. This paper shows that a simple search procedure breaking symmetries dynamically leads to a constraint program solving the problem in a few seconds, while maintaining the completeness of the approach and removing the need for large neighborhood search.

1 Introduction

The steel mill slab design problem (problem 38 in the CSP Library) has attracted significant interest in the community. The problem consists of packing a set of orders into slabs, minimizing the total capacity of the slabs needed while satisfying the capacity and order compatibility constraints on the slabs. The CSPLIB proposes an instance with 111 orders which could not be solved to optimality by constraint-programming approaches until last year. Earlier work included the presentation of different models in [1], the study of symmetry breaking in [2], the hybridization of constraint and mathematical programming in [4] which solves a sub-instance of the CSP Lib problem with 30 orders in about 1000s, and the local search solver WSAT(OIP) for pseudo boolean variables [7] which solves the decision problem with 111 orders in about 2000s.

Last year, Gargani and Refalo [3] reconsidered the problem using a constraint-programming approach. They stated that "*the models used for [earlier] constraint programming approaches to this problem were basically linear models over binary variables. While such models are suited for integer programming solvers that can tighten the formulation by cutting-plane generation, these models are notoriously not well suited to a constraint programming approach because of the limited domain reductions they produce.*" They introduced a natural constraint-programming model using logical and global constraints exploiting the structure of the problem. By designing a specific strategy for variable and value selection and combining the heuristic with a large-neighborhood search, they showed how to solve the largest instance with 111 orders in just 3s using the Ilog constraint-programming solver.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 377–381, 2008.

Gargani and Refalo also studied how to add (static) constraints in order to prevent the search strategy from producing symmetrical solutions. Their experimental results show that these constraints are useful for small instances but negatively impact performance on larger instances. They argued that "*the symmetry-breaking constraints prevent our strategy from finding good solutions causing the loss in performance*" and explained the interference between the search heuristics and the symmetric-breaking constraints. As a result, they stated: "*For these reasons, we have not used symmetry breaking constraints in our constraint programming solution, and we have dramatically improved the convergence of the search by using a local search approach.*"

This paper shows that their constraint-programming model with a simple, dynamic symmetry-breaking scheme leads to a constraint program solving the problem in a few seconds, while maintaining the completeness of the approach and removing the need for large neighborhood search.

2 The Steel Mill Slab Design Problem

The problem consists in producing n orders using a set of slabs. Each order o has a color c_o and a weight w_o representing the slab capacity it takes. Each slab has a capacity that must be chosen from the increasing set of capacities $\{u_1, u_2, ..., u_k\}$. A solution is an assignment of orders to slabs such that

1. the total weights of the orders in a slab must not exceed the slab capacity;
2. the orders in a slab can be of two different colors only.

The objective is to minimize the sum of the weights of the slabs used in the solution or, equivalently, the sum of losses (unused capacity) in the slabs used in the solution.

3 The Constraint Program

Figure 1 depicts the constraint program for solving the steel mill slab problem in COMET. Lines 1–16 are essentially the model of Gargani and Refalo, while lines 18–24 are the new search procedure including the dynamic symmetry breaking.

The ingenuity in their model is in the expression of the objective function. Indeed, the model uses two sets of decision variables: variable `x[o]` specifies the slab assigned to order `o`, while variable `l[s]` represents the load of slab `s`. Once the load of a slab is known, it is easy to compute its loss: simply take the smallest capacity supporting the load. Line 6 computes an array of losses for each possible capacity, while the objective function in line 12 uses the element constraint to compute the loss of each slab. Note that a slab with no order incurs no loss. Gargani and Refalo use a global packing constraint [5] for computing the weight: this constraint is semantically equivalent to

```
forall(s in Slabs)
  cp.post(sum(o in Orders) weight[o] * (x[o] == s) == l[s]);
```

```
1  int capacities[Caps] = ...;
2  int weight[Orders] = ...;
3  int color[Orders] =...;
4  set{int} colorOrders[c in Colors] = filter(o in Orders) (color[o] == c);
5  int maxCap = max(i in Caps) capacities[i];
6  int loss[c in 0..maxCap] = min(i in Caps: capacities[i] >= c) capacities[i] - c;
7
8  Solver<CP> cp();
9  var<CP>{int} x[Orders](cp,Slabs);
10 var<CP>{int} l[Slabs](cp,0..maxCap);
11
12 minimize<cp> sum(s in Slabs) loss[l[s]]
13 subject to {
14    cp.post(packing(x,weight,l));
15    forall(s in Slabs)
16       cp.post(sum(c in Colors) (or(o in colorOrders[c]) (x[o] == s)) <= 2);
17 } using {
18    forall(o in Orders) by (x[o].getSize(),-weight[o]) {
19       int ms = max(0,maxBound(x));
20       tryall<cp>(s in Slabs: s <= ms + 1)
21          cp.label(x[o],s);
22       onFailure
23          cp.diff(x[o],s);
24    }
25 }
```

Fig. 1. The Constraint-Programming Model in COMET

and the experimental results will discuss its importance. The second set of constraints are meta-constraints specifying that the orders can be of at most two different colors.

The search procedure in lines 18–24 is the main novelty here. It iterates on the orders, selecting first the orders with the smallest domains (first-fail principle) and breaking ties by choosing orders with the largest weight (line 18). The search procedure then considers the slabs to assign to the selected order: it only considers slabs in which some orders have been placed as well as *one* additional empty slab. Line 19 computes the already used slabs (i.e., $1..ms$), while line 20 is a nondeterministic instruction trying to assign the slabs to variable `x[o]`.

Observe that the entire model is 25 lines of COMET, does not include large neighborhood search, and is guaranteed to be complete, since two empty slabs are equivalent for allocating an order. These value symmetries were studied theoretically in [6] and have been used in several constraint programs for graph coloring, scene allocation, deployment of serializable services to name only a few. Note also that there are other symmetries that could be broken: two slabs with the same capacities and the same colors are also symmetric, but it was not necessary to break these symmetries to achieve good performance.

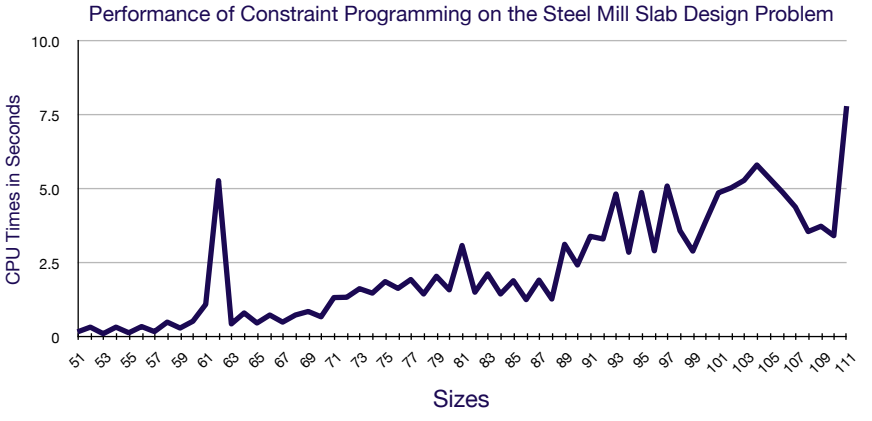


Fig. 2. Performance of the COMET Program on the Steel Mill Slab Design Problem

4 Experimental Results

Figure 2 depicts the experimental results on a 2.16 GHz Intel processor running Mac OS X 10.5.1. As can be seen, the COMET program solves all instances within less than 8 seconds, indicating that this problem has become extremely easy for constraint programming.

Readers may wonder how much of the efficiency is due to the global constraint for packing. To determine its contribution, it was replaced by the constraints

```
forall(s in Slabs)
  cp.post(sum(o in Orders) weight[o] * (x[o] == s) == l[s]);
cp.post(sum(o in Orders) weight[o] == sum(s in Slabs) l[s]);
```

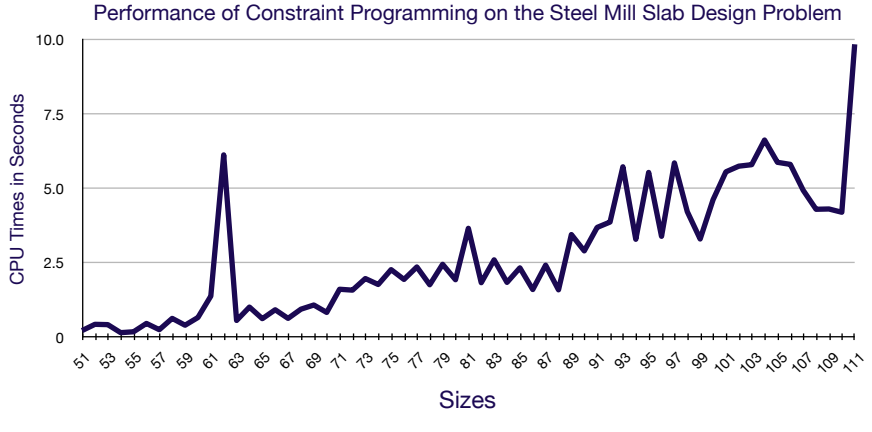


Fig. 3. Performance of the COMET Program with no Global Constraint

The first set of constraints was discussed earlier and captures the semantics of the global constraint, while the last constraint is semantically redundant and expresses that the total weight of the orders is equal to the total load of the slabs. Figure 3 depicts the experimental results. They indicate that all the instances are now solved within 10 seconds, showing that the global constraint is not strictly necessary here. Observe also the similar shape of the computation results.

5 Conclusion

In recent work, Gargani and Refalo (G&R) presented an elegant model for the Steel Mill Slab Design Problem (Problem 38 in the CSP LIB). Contrary to earlier approaches, their model does not use 0/1 variables but exploits the traditional expressiveness of constraint programming. G&R indicated that static symmetry-breaking constraints proposed earlier are not effective on this model, as these interact with their heuristic. Instead they use large neighborhood search to obtain solutions quickly. This paper showed that an simple search procedure using the first-fail principle and dynamic symmetry breaking leads to a constraint program solving the problem in a few seconds, while maintaining the completeness of the approach and removing the need for large neighborhood search.

It is interesting to observe that the steel mill slab design problem is now solved efficiently using technology and concepts which were all available in 1991.

Acknowledgments. This research is partially supported by NSF awards DMI-0600384 and ONR Award N000140610607.

References

1. Frisch, A., Miguel, I., Walsh, T.: Modelling a steel mill slab design problem. In: Proceedings of the IJCAI 2001 Workshop on Modelling and Solving Problems with Constraints (2001)
2. Frisch, A., Miguel, I., Walsh, T.: Symmetry and implied constraints in the steel mill slab design problem. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 77–92. Springer, Heidelberg (2001)
3. Gargani, A., Refalo, P.: An efficient model and strategy for the steel mill slab design problem. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, Springer, Heidelberg (2007)
4. Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Hybrid modelling for robust solving. Annals of Operations Research 130(1–4), 19–39 (2004)
5. Shaw, P.: A Constraint for Bin-Packing. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 648–662. Springer, Heidelberg (2004)
6. Van Hentenryck, P., Flener, P., Pearson, J., Ågren, M.: Tractable symmetry breaking for csps with interchangeable values. In: International Joint Conference on Artificial Intelligence (IJCAI 2003) (2003)
7. Walser, J.: Solving linear pseudo-boolean constraints with local search. In: Proceedings of the Eleventh Conference on Artificial Intelligence, pp. 269–274 (1997)

Filtering Atmost1 on Pairs of Set Variables

Willem-Jan van Hoeve[1] and Ashish Sabharwal[2,⋆]

[1] Tepper School of Business, Carnegie Mellon University
[2] Department of Computer Science, Cornell University

1 Introduction

Many combinatorial problems, such as bin packing, set covering, and combinatorial design, can be conveniently expressed using set variables and constraints over these variables [3]. In constraint programming such problems can be modeled directly in their natural form by means of *set variables*. This offers a great potential in exploiting the structure captured by set variables during the solution process, for example to break problem symmetry or to improve domain filtering.

We present an efficient filtering algorithm, establishing bounds consistency, for the `atmost1` constraint on pairs of set variables with fixed cardinality. Computational results on social golfer benchmark problems demonstrate that with this additional filtering, these problems can be solved up to 50 times faster.

2 Domain Filtering for Set Constraints

A *set variable* is a variable whose domain values are sets. As the number of possible values of a set variable can be enormous (the size of a power set, in the worst case), one usually represents the domain of a set variable S by an interval $[L(S), U(S)]$, where $L(S)$ and $U(S)$ are a 'lower' and 'upper' bound on the values that S can take. In addition, a lower bound $l(S)$ and upper bound $u(S)$ on the *cardinality* of S are maintained. A natural (and widely adopted) representation for the domain of set variables is based on the *subset ordering* of the domain. That is, the lower bound $L(S)$ represents all *mandatory* elements, while the upper bound $U(S)$ represents all *possible* elements, i.e., $D(S) = \{s \mid L(S) \subseteq s \subseteq U(S), l(S) \leq |S| \leq u(S)\}$. We refer to this representation as the subset+cardinality representation. It is applied in CP solvers such as ILOG Solver, Eclipse, and Gecode.

For constraints involving set variables, the filtering task is to increase the lower bounds and decrease the upper bounds of the domains such that we achieve *bounds consistency*, which should formally be called *subset+cardinality-bounds consistency* in our case:

Definition 1. *Let $S_1, \ldots, S_n$ be set variables. A constraint $C(S_1, \ldots, S_n)$ is called* subset+cardinality-bounds consistent *if for all $i = 1, \ldots, n$, $L(S_i)$ and*

⋆ This research was partly supported by the Intelligent Information Systems Institute, Cornell University under AFOSR Grant FA-9550-04-1-0151.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 382–386, 2008.

$U(S_i)$ are the intersection and the union, respectively, of all values in $D(S_i)$ that can be assigned to S_i in a solution to C, while in addition $l(S_i)$ and $u(S_i)$ are equal to the minimum and maximum cardinality over these values, respectively.

When a filtering algorithm for set constraints does not necessarily establish bounds consistency, we call it a *partial* filtering algorithm.

The Atmost1 Constraint on Pairs of Set Variables

The `atmost1` constraint was introduced by Sadler and Gervet [5] and specifies, for a collection of n set variables with given cardinalities, that each pair of variables overlaps in at most one element. Filtering the `atmost1` constraint to bounds consistency is NP-hard [1]. Therefore, [5] give in on bounds consistency and present a partial filtering algorithm. In this work, we given in on the number of variables instead, and consider the `atmost1` constraint involving two set variables only, which we will refer to as the `pair-atmost1` constraint. Formally, $\texttt{pair-atmost1}(S_1, S_2, c_1, c_2) = \{(s_1, s_2) \mid s_1 \in D(S_1), s_2 \in D(S_2), |s_1| = c_1, |s_2| = c_2, |s_1 \cap s_2| \leq 1\}$, where S_1 and S_2 are set variables and $c_1, c_2 \geq 1$ are integers representing the cardinalities of S_1 and S_2, respectively.

A natural way of implementing the `pair-atmost1` constraint is to use the following decomposition of $\texttt{pair-atmost1}(S_1, S_2, c_1, c_2)$ into three constraints: $|S_1| = c_1, \quad |S_2| = c_2, \quad |S_1 \cap S_2| \leq 1$. We will refer to this as the *decomposition* for `pair-atmost1`. Unfortunately, filtering these constraints separately does not establish bounds consistency on the `pair-atmost1` constraint, as illustrated by the following example:

Example 1. Let $D(S_1) = [\{1,2\}, \{1,2,3,5,6\}]$, $D(S_2) = [\{3\}, \{1,2,3,4\}]$, and $c_1 = c_2 = 3$. Establishing bounds consistency on $\texttt{pair-atmost1}(S_1, S_2, c_1, c_2)$ leads to $D(S_1) = [\{1,2\}, \{1,2,5,6\}]$, $D(S_2) = [\{3,4\}, \{1,2,3,4\}]$. This will not be achieved by the decomposition.

3 The Bounds Consistency Filtering Algorithm

We next present the filtering algorithm that establishes bounds consistency on the `pair-atmost1` constraint, which we call BC-FilterPairAtmost1 (shown as Algorithm 1).

First, we partition each of $D(S_1)$ and $D(S_2)$ into six disjoint sets. For this purpose we define L1= $L(S_1)$ and P1= $U(S_1) \setminus L(S_1)$, i.e., L1 represents the lower bound, and P1 the possible values, for S_1. We define L2 and P2 similarly for $D(S_2)$. Using these shorthands, we define the partition of $D(S_1)$ into L1only $= L1 \setminus U(S_2)$, L1L2 $= L1 \cap L2$, L1P2 $= L1 \cap P2$, P1L2 $= P1 \cap L2$, P1P2 $= P1 \cap P2$, and P1only $= P1 \setminus U(S_2)$. $D(S_2)$ is similarly partitioned into L2only, L2L1, L2P1, P2L1, P2P1, and P2only. Note that L1L2 = L2L1, P1L2 = L2P1, and P2L1 = L1P2. For these three pairs, we explicitly maintain only one set per pair, namely, L1L2, P1L2, and P2L1, respectively. (While P1P2 = P2P1 as well, we still need to maintain both of these sets.)

```
BC-FilterPairAtmost1(S_1, S_2, c_1, c_2)
begin
    Scan L(S_1), U(S_1), L(S_2), and U(S_2) to compute the cardinality of each of the 9 sets:
        L1only, L2only, L1L2, P1only, P2only, P1L2, P2L1, P1P2, P2P1
    Initialize the 'can-have' and 'not-necessary' flags of each of the 9 sets to False
    if |L1L2| > 1 then Fail
    if |L1L2| = 1 then
        Perform BC-Case0(c_1 − 1, c_2 − 1, nil)
        Perform BC-UpdateDomains
        Return
    // |L1L2| = 0
    Perform BC-Case0(c_1, c_2, nil)    // no shared element
    for each s ∈ { P1L2, P2L1, P1P2, P2P1 } do
        // possible solution has a shared element from s
        if BC-Case0(c_1 − 1, c_2 − 1, s) then s.can-have ← True
    Perform BC-UpdateDomains
end
sub BC-Case0(c_1, c_2, s)
begin
    k_1 ← c_1 − (|L1only| + |L1L2| + |P2L1|); if s = P2L1 then k_1++
    k_2 ← c_2 − (|L2only| + |L1L2| + |P1L2|); if s = P1L2 then k_2++
    slack1 ← (|P1only| + |P1P2|) − k_1
    slack2 ← (|P2only| + |P2P1|) − k_2
    slack3 ← (|P1only| + |P2only| + |P1P2|) − (k_1 + k_2)
    if (slack1 ≥ 0) and (slack2 ≥ 0) and (slack3 ≥ 0) then
        // solution exists
        P1only.can-have ← True; P2only.can-have ← True
        P1L2.not-necessary ← True; P2L1.not-necessary ← True
        if slack1 > 0 then
            P2P1.can-have ← True; P1P2.not-necessary ← True
            if slack3 > 0 then P1only.not-necessary ← True
        if slack2 > 0 then
            P1P2.can-have ← True; P2P1.not-necessary ← True
            if slack3 > 0 then P2only.not-necessary ← True
        return True;
    else
        return False;
end
sub BC-UpdateDomains
begin
    for each s ∈ { P1L2, P2L1, P1P2, P2P1 } do
        if s.can-have = False or s.not-necessary = False then
            for all y ∈ s computed by re-scanning L(S_1), U(S_1), L(S_2), U(S_2) do
                if s.can-have = False then Remove y from U(S_i) for corresponding i
                if s.not-necessary = False then Add y to L(S_i) for corresponding i
end
```

Algorithm 1. Bounds consistency domain filtering for `pair-atmost1`

Example 2. For the scenario of Example 1, we have L1 = {1, 2}, P1 = {3, 5, 6}, L2 = {3}, and P2 = {1, 2, 4}. The 9 sets in this case are: L1only = ∅, L2only = ∅, L1L2 = ∅, P1only = {5, 6}, P2only = {4}, P1L2 = {3}, P2L1 = {1, 2}, P1P2 = ∅, and P2P1 = ∅.

For each of the 9 sets, we maintain two Boolean flags: The "can-have" flag and the "not-necessary" flag, that are all initialized to False. Some of them will be set to True during the course of the algorithm when we find a solution. If

at the end, for a set s, s.can-have is still FALSE, we remove s from the upper bound of the corresponding domain. If s.not-necessary is still FALSE, we add s to the lower bound.

We find a solution by comparing the cardinalities of the 9 sets. In our *base case* (BC-CASE0), we assume that the variables already have one element in common. For S_1 we need $k_1 = c_1 - |L(S_1)| - 1$ additional values (or one more, if the common element was in $L(S_1)$). Similarly, we need k_2 more values for S_2. If we can meet the demand (verified by nonnegativity of slack1, slack2, and slack3 Algorithm 1), there exists a solution, and we update the flags for our 9 sets.

When we are not in the base case, i.e., L1L2 = 0, there are two possibilities. First, there could be a solution in which there is no common element. For this we run the base case, as is. Second, there will be a shared element, originating from P1L2, P2L1, P1P2, or P2P1. For each of these possibilities, we 'remove' the shared element from S_1 and S_2, which brings us in the base case again.

Theorem 1. *Algorithm 1 establishes bounds consistency on the* `pair-atmost1` *constraint.*

Theorem 1 can be proved by a careful case analysis. The time complexity of BC-FILTERPAIRATMOST1 is dominated entirely by the creation of the 9 sets during search, which takes $O(n)$ time where n is the integer domain size. The rest of the algorithm has only a constant number of calls to BC-CASE0 and one call to BC-UPDATEDOMAINS. BC-UPDATEDOMAINS takes time $O(n + k \log n)$, where k is the number of elements removed from an upper bound or added to a lower bound, assuming standard set operations used for maintaining these upper and lower bounds take time $O(\log n)$. We can tighten this analysis by amortizing over an entire path in the search tree from the root to any leaf, such that the total filtering complexity is $O(n \log n)$, while updating the flags takes total time $O(n)$, for the path.

4 Experimental Results

We evaluated the performance of the `pair-atmost1` constraint on the well-known social golfer problem (problem prob010 in CSPLib). The problem `golf-`g`-`s`-`w asks for a partition of n golfers into g groups, each of size s, for w weeks, such that no two golfers are in the same group more than once throughout the whole schedule. We apply the following standard model, using set variables S_{ij} to represent the set of golfers of week i and group j:

$$
\begin{array}{ll}
\texttt{partition}(S_{i1}, \ldots, S_{ig}, \{1, \ldots, n\}), & 1 \le i \le w \\
\texttt{pair-atmost1}(S_{ij}, S_{kl}, s, s), & 1 \le i < k \le w, 1 \le j \le g, 1 \le l \le g \\
|S_{ij}| = s, & 1 \le i \le w, 1 \le j \le g \\
S_{ij} \in [\emptyset, \{1, \ldots, n\}], & 1 \le i \le w, 1 \le j \le g.
\end{array}
$$

To speed up the computation, we also applied a redundant global cardinality constraint [4] on integer variables x_{ij} representing the group in which golfer j

Table 1. Computational results on a number of social golfer instances

Problem	Decomposition (partial filtering)		BC-FilterPairAtmost1 (bounds consistency)	
	time (s)	backtracks	time (s)	backtracks
golf-6-5-5	2106.7	10,986,224	75.5	239,966
golf-6-5-4	1517.7	10,930,370	39.7	197,837
golf-6-5-3	1060.5	10,930,016	29.6	197,607
golf-6-5-2	635.5	10,879,368	17.2	171,664
golf-8-4-4	226.7	1,555,561	157.7	738,393
golf-10-3-10	128.1	150,911	67.2	78,976
golf-10-3-9	86.0	150,452	52.4	78,613
golf-10-3-6	21.3	110,429	17.3	57,364
golf-10-4-5	51.3	310,110	4.5	22,044
golf-10-4-4	42.5	310,109	4.0	22,043
golf-7-4-4	22.5	184,641	4.4	27,877

plays in week i. Our search strategy is a smallest-domain-first on these variables. Finally, to account for some symmetry-breaking, we partly instantiate some of the set variables before starting the search, following Fahle et al. [2]. We note that our filtering algorithm can be applied to any model, including those with more advanced symmetry-breaking techniques.

We implemented our model in ILOG Solver 6.3, and all experiments run on a 3.8 GHz Intel Xeon machine with 2 GB memory running Linux 2.6.9-22.ELsmp. We evaluated the performance of the decomposition implementation of `pair-atmost1` (achieving partial filtering) with our filtering algorithm BC-FilterPairAtmost1 (achieving bounds consistency) on a number of instances, as reported in Table 1. The results demonstrate that using the bounds consistency algorithm, one can solve these instances up to 50 times faster, with a similar reduction in the number of search tree backtracks.

References

[1] Bessiere, C., Hebrard, E., Hnich, B., Walsh, T.: Disjoint, partition and intersection constraints for set and multiset variables. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 138–152. Springer, Heidelberg (2004)
[2] Fahle, T., Schamberger, S., Sellmann, M.: Symmetry breaking. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 93–107. Springer, Heidelberg (2001)
[3] Gervet, C.: Constraints over structured domains. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, Elsevier, Amsterdam (2006)
[4] Régin, J.-C.: Generalized arc consistency for global cardinality constraint. In: AAAI 1996, vol. 1, pp. 209–215 (1996)
[5] Sadler, A., Gervet, C.: Global reasoning on sets. In: Proc. of Workshop on Modelling and Problem Formulation (FORMUL 2001) (2001)

Mobility Allowance Shuttle Transit (MAST) Services: MIP Formulation and Strengthening with Logic Constraints

Luca Quadrifoglio[1], Maged M. Dessouky[2], and Fernando Ordóñez[2]

[1] Zachry Department of Civil Engineering
Texas A&M University
lquadrifoglio@civil.tamu.edu
[2] Daniel J. Epstein Department of Industrial and Systems Engineering
University of Southern California

We study a hybrid transportation system referred to as Mobility Allowance Shuttle Transit (MAST) where vehicles may deviate from a fixed path consisting of a few mandatory checkpoints to serve demand distributed within a proper service area. In this paper we propose a Mixed Integer Programming (MIP) formulation for the static scheduling problem of a MAST type system. Since the problem is NP Hard, we develop sets of logic cuts, by using reasonable assumptions on passengers' behavior. The purpose of these constraints is to speed up the search for optimality by removing inefficient solutions from the original feasible region. Experiments show the effectiveness of the developed inequalities, achieving a reduction up to 90% of the CPU solving time for some of the instances.

Summary[1]

We study a hybrid transportation system referred to as Mobility Allowance Shuttle Transit (MAST) where vehicles may deviate from a fixed path consisting of a few mandatory checkpoints to serve demand distributed within a proper service area. A MAST system is described by a set of vehicles driving along a base fixed-route and serving a specific geographic area. The base route can be laid out around a loop or between two terminals. Vehicles must stop at a set of checkpoints along the main path. The checkpoints are conveniently located at major transfer points or high density demand zones, are relatively far from each other and have fixed departure times. Given a proper amount of slack time, vehicles are allowed to deviate from the fixed path to serve (pick-up and/or drop-off) customers at their desired locations, as long as they are within a service area.

The MAST system considered consists of a single vehicle, associated with a predefined schedule along a fixed-route consisting of C checkpoints. A trip *r* is defined as a portion of the schedule beginning at one of the terminals and ending at the other one after visiting all the intermediate checkpoints. The service area is represented by a

[1] This is a summary of the following paper: Quadrifoglio L., Dessouky M., Ordóñez F., "Mobility Allowance Shuttle Transit services: MIP formulation and strengthening with logic constraints", *European Journal of Operational Research*, 2008, 185, 481–494.

L. Perron and M. Trick (Eds.): CPAIOR 2008, LNCS 5015, pp. 387–391, 2008.

rectangular region defined by L×W, where L (on the x axis) is the distance between terminals 1 and C and W/2 (on the y axis) is the maximum allowable deviation from the main route in either side (see Figure 1).

The demand is defined by a set of requests. Each request is defined by pick up/drop off service stops and a ready time for pick up. The MAST service can respond to four different types of requests: pick up (P) and drop off (D) at the checkpoints; non checkpoint pick up (NP) and drop off (ND), representing customers picked up/dropped off at any location within the service area. A certain amount of slack time between any consecutive pair of checkpoints is needed in order to allow deviations to serve NP or ND requests. There are consequently four different possible types of customers' requests: PD ("Regular"), pick up and drop off at the checkpoints; PND ("Hybrid"), pick up at the checkpoint, drop off not at the checkpoint; NPD ("Hybrid"), pick up not at the checkpoint, drop off at the checkpoint; NPND ("Random"), pick up and drop off not at the checkpoints. In this paper we consider a static scenario in which all the demand is known in advance. We also assume one customer per request, no vehicle capacity constraint and a deterministic environment.

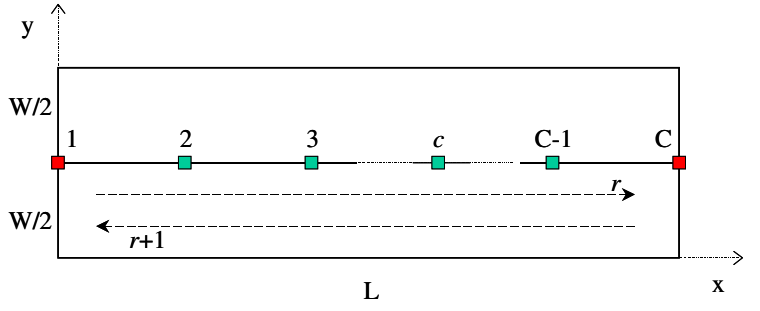


Fig. 1. MAST system

The MAST scheduling problem can be formulated as a mixed integer linear program:

$$\min \quad \omega_1\left(\sum_{(i,j)\in A}\delta_{i,j}x_{i,j}\right)\Big/v+\omega_2\left(\sum_{k\in K}(d_k-p_k)\right)+\omega_3\left(\sum_{k\in K}(p_k-\tau_k)\right) \qquad \text{Objective Function}$$

Subject to:

$$\sum_i x_{i,j}=1 \qquad \forall j\in N/\{1\}$$

$$\sum_j x_{i,j}=1 \qquad \forall i\in N/\{TC\}$$

$$t_i=\theta_i \qquad \forall i\in N_0$$

$$p_k=t_{pu(k)} \qquad \forall k\in K/K_{PND}$$

$$d_k=\bar{t}_{do(k)} \qquad \forall k\in K/K_{NPD}$$

$$\sum_{r\in HRD(k)} z_{k,r}=1 \qquad \forall k\in K_{HYB}$$

$$p_k\geq t_{pu(k,r)}-M(1-z_{k,r}) \qquad \forall k\in K_{PND},\ \forall r\in HRD(k)$$

$$p_k\leq t_{pu(k,r)}+M(1-z_{k,r}) \qquad \forall k\in K_{PND},\ \forall r\in HRD(k)$$

$$d_k\geq \bar{t}_{do(k,r)}-M(1-z_{k,r}) \qquad \forall k\in K_{NPD},\ \forall r\in HRD(k)$$

$$d_k\leq \bar{t}_{do(k,r)}+M(1-z_{k,r}) \qquad \forall k\in K_{NPD},\ \forall r\in HRD(k)$$

$$p_k\geq \tau_k \qquad \forall k\in K$$

$$d_k>p_k \qquad \forall k\in K$$

$$\bar{t}_j\geq t_i+x_{i,j}\delta_{i,j}/v-M(1-x_{i,j}) \qquad \forall (i,j)\in A$$

SETS of NODES
N_0 = checkpoints
N_n = non-checkpoints
$N = N_0 \cup N_n$

SETS of ARCS
A = all arcs

SETS of CUSTOMERS
K_{PD} = PD requests
K_{PND} = PND requests
K_{NPD} = NPD requests
K_{NPND} = NPND requests
$K_{HYB} = K_{PND} \cup K_{NPD}$
$K = K_{PD} \cup K_{HYB} \cup K_{NPND}$
$pu(k) \in N$, $\forall k \in K/K_{PND}$ = pick-up of k
$do(k) \in N$, $\forall k \in K/K_{NPD}$ = drop-off of k
$pu(k,r) \in N$, $\forall k \in K_{PND}$ = pick-ups of k
$do(k,r) \in N$, $\forall k \in K_{NPD}$ = drop-offs of k

SETS of TRIPS
RD = all trips
$HRD(k) \subset RD$, $\forall k \in K_{HYB}$ = feasible trips of k

VARIABLES
$\boldsymbol{x_{i,j} = \{0, 1\}, \forall (i,j) \in A}$
t_i, $\forall i \in N$ = departure time from i
, $\forall i \in N/\{1\}$ = arrival time at i
p_k, $\forall k \in K$ = pick-up time of request k
d_k, $\forall k \in K$ = drop-off time of request k
$\boldsymbol{z_{k,r} = \{0,1\}, \forall k \in K_{HYB}, \forall r \in HRD(k)}$

PARAMETERS
$\delta_{i,j}$, $\forall (i, j) \in A$ = distance from i to j
v, vehicle speed
b_i, boarding/disembarking time at i
$\boldsymbol{\theta_i, \forall i \in N_0}$ **= departure times from checkpoints**
$\boldsymbol{\tau_k, \forall k \in K}$ **= ready time of request** $\boldsymbol{k}$
$\boldsymbol{\omega_1/\omega_2/\omega_3}$ **= objective function weights**

Where $x_{i,j}$ indicates whether an arc (i,j) is used ($x_{i,j} = 1$) or not ($x_{i,j} = 0$) and $z_{k,r}$ indicates whether the checkpoint stop of the hybrid request k (a pick-up if $k \in K_{PND}$ or a drop-off if $k \in K_{NPD}$) is scheduled in trip r, $\forall r \in RD$.

The above formulation is sufficient to find the optimal solution of the problem, but it is ineffective in the sense that it includes many feasible inefficient solutions and thus has a weak LP relaxation.

A way to speed up the search for optimality is the development of constraints and their addition to the math program formulation. These constraints are called valid if they reduce the dimensions of the relaxed feasible region, but all integer feasible solutions of the original model are not touched. The ideal purpose of these constraints is to produce the convex hull of the integer feasible solutions which would allow LP algorithms to solve the problem much faster. Another category of constraints, the so called "logic cuts", have the purpose to eliminate some integer feasible solutions that are provably suboptimal. Thus, they can not be considered valid, but they can be indeed very effective. They may significantly shrink the feasible region, even by some orders of magnitude, and they allow improving the quality of the LP relaxation bound, considerably speeding up the reduction of the optimality gap throughout the iterations of the solver. As a result, they can be extremely beneficial in reducing the CPU time in the search for optimality.

In this paper we develop and add "logic cuts" to strengthen the above MAST formulation. The underlying concept behind all the developed inequalities is that hybrid customers will be choosing their P or D checkpoints as close as possible to their corresponding ND or NP stop, once these are placed in the schedule. More formally, we can state (proofs in the full paper) the following Propositions 1 and 2:

Proposition 1. A necessary condition for optimality is that NPD customers must disembark the vehicle at the first occurrence of their D checkpoint following their scheduled NP pick-up stop.

Proposition 2. If $\omega_2>\omega_3$, a necessary condition for optimality is that PND customers must board the vehicle at the last occurrence of their P checkpoint prior to their scheduled ND drop-off stop.

Although the logic behind the above Propositions may seem obvious to a human mind, it is not explicitly stated in the formulation and the solver would still consider several feasible but inefficient solutions (violating the above Propositions) as possible candidates while searching for optimality. Therefore, based on the above Propositions, we develop three different groups of valid inequalities to add to the formulation.

Group #1: The first group of inequalities is developed by directly applying Propositions 1 and 2. They include constraints linking the z variables to the t variables (departure times) of non checkpoint stops of hybrid requests and constraints linking the z variables to some of the x variables. An example is

$$t_{do(k)} < z_{k,r}\theta_j + M(1-z_{k,r}), \text{ with } j = pu(k,r+1), \forall k \in K_{PND}, \forall r \in RD/\{R\}$$

Group #2: A second group of inequalities includes constraints linking z and x variables by making use of Propositions 1 and 2 along with the ready times τ of the requests. An example is

$$\tau_{q(i)} + \delta_{i,j} + b_j \le z_{k,r}\theta_j + M(2-z_{k,r}-x_{do(k),i}),$$
$$\text{with } i = pu(q(i)), j = pu(k,r+1), \forall k \in K_{PND}, \forall r \in RD/\{R\}, \forall (do(k),i) \in A_n$$

Group #3: A third group of inequalities links z and x variables by applying the results from the Propositions to pairs of hybrid requests. An example is

$$z_{h,s}\theta_i - z_{k,r}\theta_j < M(3-z_{h,s}-z_{k,r}-x_{do(k),do(h)}),$$
$$\text{with } i = pu(h,s), j = pu(k,r+1), \forall k,h \in K_{PND}, \forall r \in RD/\{R\}, \forall s \in RD$$

Experimental results on several instances (which we are omitting in this summary, but are explained in details in the full paper) show the effectiveness of the developed inequalities, which are able to reduce the CPU solution time by up to more than **90%** for some cases. Specifically, Group "#1" provide the best overall results that always effective, followed in general by Group "#2" and Group "#3", which are not always effective. The synergistic effect of including all the cuts together further reduces the CPU solution time in many cases. We provide the result for one case in the following Table 1.:

Table 1. Experimental results

Case: B1a	**TS=10: R=2; $\|K_{PD}\|$=1; $\|K_{PND}\|$=2; $\|K_{NPD}\|$=1; $\|K_{NPND}\|$=1**											
cuts	*var*	*bin*	*lin*	*con*	*sec*	*n*	*i*	*rel*	*opt*	*ub*	*lb*	*gap*
none	67	43	24	85	0.04	64	403	81.2	114.7	/	/	0.0%
#1	67	43	24	91	0.03	27	221	81.8	114.7	/	/	0.0%
#2	67	43	24	87	0.04	50	324	81.2	114.7	/	/	0.0%
#3	67	43	24	85	0.04	64	403	81.2	114.7	/	/	0.0%
all	67	43	24	93	0.03	25	217	81.8	114.7	/	/	0.0%
Case: B1b	**TS=15: R=4; $\|K_{PD}\|$=1; $\|K_{PND}\|$=2; $\|K_{NPD}\|$=2; $\|K_{NPND}\|$=1**											
cuts	*var*	*bin*	*lin*	*con*	*sec*	*n*	10^3 *i*	*rel*	*opt*	*ub*	*lb*	*gap*
none	124	89	35	156	0.56	695	7.91	105.8	164.9	/	/	0.0%
#1	123	88	35	199	0.19	126	1.39	105.8	164.9	/	/	0.0%
#2	124	89	35	188	0.50	643	5.46	105.8	164.9	/	/	0.0%
#3	124	89	35	256	0.62	815	7.25	105.8	164.9	/	/	0.0%
all	123	88	35	309	0.25	89	1.55	105.8	164.9	/	/	0.0%
Case: B1c	**TS=20: R=4; $\|K_{PD}\|$=1; $\|K_{PND}\|$=5; $\|K_{NPD}\|$=4; $\|K_{NPND}\|$=1**											
cuts	*var*	*bin*	*lin*	*con*	*sec*	10^3 *n*	10^6 *i*	*rel*	*opt*	*ub*	*lb*	*gap*
none	247	197	50	299	619.0	723.3	5.58	132.8	217.8	/	/	0.0%
#1	244	195	49	351	49.0	60.7	0.47	132.8	217.8	/	/	0.0%
#2	247	197	50	400	355.7	319.9	3.33	132.8	217.8	/	/	0.0%
#3	247	197	50	639	508.1	460.2	4.03	132.8	217.8	/	/	0.0%
all	244	195	49	742	32.0	27.2	0.31	132.8	217.8	/	/	0.0%
Case: B1d	**TS=25: R=4; $\|K_{PD}\|$=2; $\|K_{PND}\|$=6; $\|K_{NPD}\|$=6; $\|K_{NPND}\|$=2**											
cuts	*var*	*bin*	*lin*	*con*	*sec*	10^6 *n*	10^6 *i*	*rel*	*opt*	*ub*	*lb*	*gap*
none	398	336	62	452	36,000	20.2	249	193.0	?	312.8	293.0	6.3%
#1	398	336	62	506	36,000	17.5	235	193.0	?	312.8	304.4	2.7%
#2	398	336	62	552	36,000	17.0	246	193.0	?	312.8	293.4	6.2%
#3	397	335	62	590	36,000	14.4	215	193.0	?	312.8	295.6	5.5%
all	397	335	62	744	36,000	15.3	219	193.0	?	312.8	299.8	4.1%

In the table, TS is the total number of stops in the network, R is the number of trips. We solved the same instance without adding any groups of inequalities (“none”), adding only one group at a time (“#1”, “#2” or “#3”) or adding all the groups together (“all”). For each run we show the size of the problem solved: total variables (“var”), divided into binary (“bin”) and linear (“lin”) and total number of constraints (“con”). The following columns show the time to reach optimality in seconds (“sec”), the number of nodes visited in the branch and bound tree (“n”), the number of simplex iterations performed (“i”), the relaxed optimal value (“rel”) and the real optimum (“opt”).

Author Index