Willem-Jan van Hoeve
John N. Hooker (Eds.)

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

LNCS 5547

6th International Conference, CPAIOR 2009
Pittsburgh, PA, USA, May 2009
Proceedings

Lecture Notes in Computer Science 5547

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

Willem-Jan van Hoeve John N. Hooker (Eds.)

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

6th International Conference, CPAIOR 2009
Pittsburgh, PA, USA, May 27-31, 2009
Proceedings

Springer

Volume Editors

Willem-Jan van Hoeve
John N. Hooker
Carnegie Mellon University
Tepper School of Business
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
E-mail: vanhoeve@andrew.cmu.edu, john@hooker.tepper.cmu.edu

Library of Congress Control Number: Applied for

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 3-642-01928-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-01928-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12686247 06/3180 5 4 3 2 1 0

Preface

The 6th International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2009) was held in Pittsburgh, USA, May 29–31, 2009. More information about the CPAIOR conference series can be found at `www.cpaior.org`. This volume contains the papers and extended abstracts that were presented during the conference.

In total there were 65 high-quality submissions, including 41 full paper and 24 extended abstract submissions. The full papers reflect original unpublished work, whereas the extended abstracts can be either original unpublished work or a summary of work published elsewhere. Each full paper was reviewed by at least three Program Committee members, and most extended abstracts by two. After general discussion, the Program Committee accepted 20 full papers and 10 extended abstracts for presentation during the conference and publication in this volume. The submissions, reviews, discussion, and the proceedings preparation were all handled by the EasyChair system. We thank the Program Committee, as well as the external reviewers, for their hard work.

In addition to the full paper and extended abstract presentation, the program contained two invited talks, by Eva K. Lee (Georgia Institute of Technology) and Mark Wallace (Monash University). A summary of each invited talk is also included in this volume.

A two-day tutorial on constraint programming was held before the conference, during May 27–28, 2009. There were four parts to the tutorial: "Introduction to CP Concepts" presented by Peter van Beek, "Modeling in CP" presented by Helmut Simonis, "Combining CP and Operations Research" presented by John Hooker, and "CP Languages, Systems, and Examples" presented by Laurent Michel, Pascal Van Hentenryck, and Paul Shaw. We thank all tutorial speakers for their efforts. We also thank the tutorial chair Gilles Pesant for his help in organizing this event.

Two satellite workshops took place on May 28, 2009. The workshop "Bound Reduction Techniques for Constraint Programming and Mixed-Integer Nonlinear Programming" was organized by Pietro Belotti. The workshop "Optimization in Health and Medicine" was organized by Sebastian Brand, Eva K. Lee, and Barry O'Sullivan.

Finally, we want to thank all the sponsors who made this event possible: National Science Foundation, Tepper School of Business, Air Force Office of Scientific Research, Association for Constraint Programming, IBM T.J. Watson Research Center, Jeppesen Technology Services, NICTA, and ILOG.

March 2009

Willem-Jan van Hoeve
John Hooker

Organization

Program Chairs

Willem-Jan van Hoeve	Carnegie Mellon University, USA
John Hooker	Carnegie Mellon University, USA

Tutorial Chair

Gilles Pesant	University of Montreal, Canada

Program Committee

Philippe Baptiste	Ecole Polytechnique France
Roman Barták	Charles University, Czech Republic
Chris Beck	University of Toronto, Canada
Sebastian Brand	University of Melbourne, Australia
John Chinneck	Carleton University, Canada
Emilie Danna	ILOG, USA
Andrew Davenport	IBM, USA
Ismael de Farias	Texas Tech University, USA
Yves Deville	Université Catholique de Louvain, Belgium
Robert Fourer	Northwestern University, USA
Bernard Gendron	University of Montreal, Canada
Carmen Gervet	The German University in Cairo, Egypt
Carla Gomes	Cornell University, USA
Ignacio Grossmann	Carnegie Mellon University, USA
Joerg Hoffmann	SAP, Germany
Narendra Jussien	University of Nantes, France
Thorsten Koch	ZIB, Germany
Olivier Lhomme	ILOG, France
Andrea Lodi	University of Bologna, Italy
Laurent Michel	University of Connecticut, USA
Michela Milano	University of Bologna, Italy
Eric Monfroy	University of Nantes, France
Yehuda Naveh	IBM, Israel
Barry O'Sullivan	University College Cork, Ireland
Laurent Perron	Google, France
Gilles Pesant	University of Montreal, Canada
Jean-Charles Régin	University of Nice-Sophia Antipolis, France
Mauricio Resende	AT&T Labs, USA
Andrea Roli	University of Bologna, Italy
Louis-Martin Rousseau	University of Montreal, Canada

Michel Rueher	University of Nice-Sophia Antipolis, France
Ashish Sabharwal	Cornell University, USA
Nick Sahinidis	Carnegie Mellon University, USA
Matt Saltzman	Clemson University, USA
Tuomas Sandholm	Carnegie Mellon University, USA
Meinolf Sellmann	Brown University, USA
Helmut Simonis	4C, Ireland
Stephen Smith	Carnegie Mellon University, USA
Mohit Tawarmalani	Purdue University, USA
Michael Trick	Carnegie Mellon University, USA
Pascal Van Hentenryck	Brown University, USA
Petr Vilím	ILOG, France
Mark Wallace	Monash University, Australia
Tallys Yunes	University of Miami, USA
Weixiong Zhang	Washington University, USA

External Reviewers

Xiaowei Bao	Bertrand Neveu
Timo Berthold	Fabio Parisini
Marie-Claude Coté	Thomas Stützle
Yahia Lebbah	Guido Tack
Michele Lombardi	Kati Wolter
Arnaud Malapert	Neil Yorke-Smith
Claude Michel	Fengqi You
Sylvain Mouret	Erik Zawadzki
Nina Narodytska	

Table of Contents

Invited Talks

Regular Papers

Extended Abstracts

Machine Learning Framework for Classification in Medicine and Biology

Eva K. Lee

Center for Operations Research in Medicine and HealthCare,
School of Industrial and Systems Engineering,
NSF I/UCRC Center for Health Organization Transformation,
Center for Bioinformatics and Computational Genomics,
Georgia Institute of Technology, Atlanta, Georgia 30332-0205

Abstract. Systems modeling and quantitative analysis of large amounts of complex clinical and biological data may help to identify discriminatory patterns that can uncover health risks, detect early disease formation, monitor treatment and prognosis, and predict treatment outcome. In this talk, we describe a machine-learning framework for classification in medicine and biology. It consists of a pattern recognition module, a feature selection module, and a classification modeler and solver. The pattern recognition module involves automatic image analysis, genomic pattern recognition, and spectrum pattern extractions. The feature selection module consists of a combinatorial selection algorithm where discriminatory patterns are extracted from among a large set of pattern attributes. These modules are wrapped around the classification modeler and solver into a machine learning framework. The classification modeler and solver consist of novel optimization-based predictive models that maximize the correct classification while constraining the inter-group misclassifications. The classification/predictive models 1) have the ability to classify any number of distinct groups; 2) allow incorporation of heterogeneous, and continuous/time-dependent types of attributes as input; 3) utilize a high-dimensional data transformation that minimizes noise and errors in biological and clinical data; 4) incorporate a reserved-judgement region that provides a safeguard against over-training; and 5) have successive multi-stage classification capability. Successful applications of our model to developing rules for gene silencing in cancer cells, predicting the immunity of vaccines, identifying the cognitive status of individuals, and predicting metabolite concentrations in humans will be discussed. We acknowledge our clinical/biological collaborators: Dr. Vertino (Winship Cancer Institute, Emory), Drs. Pulendran and Ahmed (Emory Vaccine Center), Dr. Levey (Neurodegenerative Disease and Alzheimer's Disease), and Dr. Jones (Clinical Biomarkers, Emory).

1 Introduction

Discriminant analysis involves the classification of an entity into one of several a priori, mutually exclusive groups based upon specific measurable characteristics

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 1–7, 2009.

of the entity. A discriminant (predictive) rule is formed from data collected on a sample of entities for which the group classifications are known. New entities, whose classifications are unknown, will be classified based on this rule. Often there is a trade-off between the discriminating ability of the selected attributes and the expense of obtaining measurements on these attributes. Indeed, the measurement of a relatively definitive discriminating feature may be prohibitively expensive to obtain on a routine basis, or perhaps impossible to obtain at the time that classification is needed. Thus, a discriminant rule based on a selected set of feature attributes will typically be an imperfect discriminator, sometimes misclassifying entities. Depending on the application, the consequences of misclassifying an entity may be substantial. In such a case, it may be desirable to form a discrimination rule that allows less specific classification decisions, or even non-classification of some entities to reduce the probability of misclassification.

Many methods have been used to develop classification models, including pattern recognition, artificial intelligence, optimization/mathematical programming-based methods, support vector machines, neural networks, data mining, and statistical analysis. In our computational center, since 1997 (Gallagher et al 1996, 1997, Lee et al 2003), we have been developing a general-purpose discriminant analysis modeling framework and computational engine that is applicable to a wide variety of applications, including biological, biomedical and logistics problems.

2 The Mixed Integer Programming-Based Classification Model

Our work was motivated by the 1969 probability model introduced by Anderson which maximizes the probability of correct allocation subject to misclassification probability constraints. For two groups the optimal solution can be modeled rather straightforward. However, finding an optimal solution (rule) for the general case is a difficult problem, with the difficulty increasing as the number of groups increases. We offer an avenue for modeling and finding the optimal solution in the general case. (Gallagher et al 1996, 1997).

Assume that we have $G = \{1, \ldots, G\}$ groups with a training sample of N entities whose group classifications are known; say n_g entities are in group g, $N_g = \{1, \ldots, n_g\}$, where $\sum_{g=1}^{G} n_g = N$. Let the k dimensional vectors x^{gj}, $g = 1, \ldots, G$, $j = 1, \ldots, n_g$, contain the measurements on k available characteristics of the entities. Let $\hat{f}_h$, $h = 1, \ldots, G$, be the estimated group conditional density functions, let $\hat{\pi}_h$ denote an estimator for the prior probability that a randomly selected entity is from group g, $g = 1, ..., G$, and define $\hat{p}_i(x) = \hat{f}_i(x) / \sum_{t=1}^{G} \hat{f}_t(x)$.

Our objective is to determine a partition $\{R_0, \ldots, R_G\}$ of $\Re^k$ that maximizes the correct classification, while ensuring that the number of group g training entities in region R_h is less than or equal to a pre-specified percentage, α_{hg} $(0 < \alpha_{hg} < 1)$, of the total number, n_g, of group g entities, h, $g \in \{1, \ldots, G\}$, $h \neq g$. Here R_0 is the reserved-judgement region. Mathematically, let u_{hgj} be a binary variable indicating whether or not x^{gj} lies in region R_h; i.e., whether

or not the jth entity from group g is allocated to group h. The MIP-based classification model can be written as

$$\textbf{(DAMIP-Classifier)} \quad \text{Maximize} \sum_{g\in G}\sum_{j\in N_g} u_{ggj}$$

Subject to

$$L_{hgj} = \hat{\pi}_h \hat{p}_h(x^{gj}) - \sum_{i\in G\setminus h} \lambda_{ih}\hat{p}_i(x^{gj}) \quad h,g\in G, j\in N_g \tag{1}$$

$$y_{gj} = \max\{0, L_{hgj} : h = 1,\ldots,G\} \quad g\in G, j\in N_g \tag{2}$$

$$y_{gj} - L_{ggj} \le M(1-u_{ggj}) \quad g\in G, j\in N_g \tag{3}$$

$$y_{gj} - L_{hgj} \ge \varepsilon(1-u_{hgj}) \quad h,g\in G, j\in N_g, h\ne g \tag{4}$$

$$\sum_{j\in N_g} u_{hgj} \le \lfloor \alpha_{hg} n_g \rfloor \quad h,g\in G, h\ne g \tag{5}$$

$$-\infty < L_{hgj} < \infty, y_{gj}\ge 0, \lambda_{ih}\ge 0, u_{hgj}\in\{0,1\}$$

Constraint (1) defines the variable L_{hgj} as the value of the function L_h evaluated at x^{gj}. Therefore, the continuous variable y_{gj}, defined in constraint (2), represents $\max\{L_h(x^{gj}) : h = 0,\ldots,G\}$; and consequently, x^{gj} lies in region R_h if, and only if, $y_{gj} = L_{hgj}$. The binary variable u_{hgj} is used to indicate whether or not x^{gj} lies in region R_h; i.e., whether or not the jth entity from group g is allocated to group h. In particular, constraint (3), together with the objective, force u_{ggj} to be 1 if, and only if, the jth entity from group g is correctly allocated to group g; and constraints (4) and (5) ensure that at most $\lfloor \alpha_{hg} n_g \rfloor$ (i.e., the greatest integer less than or equal to $\alpha_{hg} n_g$) group g entities are allocated to group h, $h \ne g$. One caveat regarding the indicator variables u_{hgj} is that although the condition $u_{hgj} = 0$, $h \ne g$, implies (by constraint (4)) that $x^{gj} \notin R_h$, the converse need not hold. As a consequence, the number of misclassifications may be overcounted. However, in our numerical study we found that the actual amount of overcounting is minimal. One could force the converse (thus, $u_{hgj} = 1$ if and only if $x^{gj} \in R_h$) by adding constraints $y_{gj} - L_{hgj} \le M(1-u_{hgj})$, for example. Finally, we note that the parameters M and ε are extraneous to the discriminant analysis problem itself, but are needed in the model to control the indicator variables u_{hgj}. The intention is for M and ε to be, respectively, large and small positive constants.

3 Complexity and Characteristics of DAMIP-Classifier

Performance of DAMIP classifier and comparison with other methods were reported in Gallagher et al 97, and in Lee et al 2003. We have proved that DAMIP-Classifier is NP-complete for $G > 2$ (Brooks, Lee 2008), further the classification rule resulting from DAMIP-Classifier is universally strongly consistent (Brooks, Lee 2008). Since 1996, Lee and her medical colleagues have explored and demonstrated the capability of DAMIP-Classifier in classifying various types of data arising from real biological and medical problems. In these applications, DAMIP-Classifier has been able to consistently maximize the correct classification rate

(80% - 100% correct rates were obtained) while satisfying pre-set limits on inter-group misclassifications (Gallager et al 1996, 1997, Feltus, Lee et al 2003, Feltus et al 2006, Lee et al 2002, 2003, 2004, Lee 2007, Lee, Wu 2007, Querec et al 2008, McCabe, et al 2009). In each of these studies, beyond reporting the ten-fold cross-validation results, the resulting classification rule was also blind tested against new data of unknown group identity and resulted in remarkable rates of correct prediction. The real applications provide an empirical basis for making some general statements on the characteristics of the DAMIP rules: (1) The predictive power of a DAMIP rule is independent of sample size, the proportions of training observations from each group, and the probability distribution functions of the groups. (2) A DAMIP rule is insensitive to the choice of prior probabilities. (3) A DAMIP rule is capable of maintaining low misclassification rates when the number of training observations from each group varies significantly.

Computationally, the resulting MIP instances are large scale, and ill-conditioned, with the LP relaxation rather dense and difficult to solve. These characteristics are also observed in optimization-based support vector machines. To improve tractability, we developed applicable polyhedral theory and cutting plane algorithms for solving these instances.

4 Classification Results on Real-World Applications

We performed ten-fold cross validation, and designed simulation and comparison studies on our models. The results, reported in Gallagher, et al 1997, Lee, et al 2003, Brooks and Lee, 2008, show the methods are promising, based on applications to both simulated data and real-application datasets from the machine learning database repository. Furthermore, our methods compare well to existing methods, often producing better results than other approaches such as artificial neural networks, quadratic discriminant analysis, tree classification, and other support vector machines.

To illustrate the power and flexibility of the classification model and solution engine, and its multi-group prediction capability, application of the predictive model to a broad class of biological and medical problems is described. Applications include: the differential diagnosis of the type of erythemato-squamous diseases; predicting presence/absence of heart disease; genomic analysis and prediction of aberrant CpG island meythlation in human cancer; discriminant analysis of motility and morphology data in human lung carcinoma; prediction of ultrasonic cell disruption for drug delivery; identification of tumor shape and volume in treatment of sarcoma; multistage discriminant analysis of biomarkers for prediction of early atherosclerois; fingerprinting of native and angiogenic microvascular networks for early diagnosis of diabetes, aging, macular degeneracy and tumor metastasis; prediction of protein localization sites; and pattern recognition of satellite images in classification of soil types. In all these applications, the predictive model yields correct classification rates ranging from 80% to 100%. This provides motivation for pursuing its use as a medical diagnostic, monitoring and decision-making tool.

Strategy For Predicting Immunity Of Vaccines. (Querec et al 2008) The purpose of this study involves the development of methodologies to predict the immunity of a vaccine without exposing individuals to infection. This addresses a long-standing challenge in the development of vaccines—that of only being able to determine immunity or effectiveness long after vaccination and, often, only after being exposed to infection. The study employs the yellow fever vaccine (YF-17D) as a model. Yellow fever vaccine has been administered to nearly half a billion people over the last70 years. A single shot of the vaccine induces immunity in many people for nearly 30 years. Despite the great success of the yellow fever vaccine, little has been known about the immunological mechanisms that make it effective. The team vaccinateda set of healthy individuals with YF-17D and studied the T cell and antibody responses in their blood. Gene expression patterns in white blood cells were collected for a period of time. About 50,000 gene signatures per individual were among the attributes collected. Applying DAMIP-classifier, we were able to identify distinct gene signatures that correlated with the T cell response and the antibody response induced by the vaccine. To determine whether these gene signatures could predict immune response, we vaccinated a second group of individuals and were able to predict with up to 90 percent accuracy which of the vaccinated individuals would develop a strong T or B cell immunity to yellow fever. The ability to successfully predict the immunity and effectiveness of vaccines would facilitate the rapid evaluation/design of new and emerging vaccines, identify individuals who are unlikely to be protected by a vaccine, and answer the fundamental questions that can lead to better vaccinations and prevention of disease.

Identifying Rules for Gene Silencing in Cancer Cells. (Feltus et al 2003, 2006, McCabe et al 2009) CpG islands are the discreet regions of DNA sequence with high concentration of CpG dinucleotides. On their way to becoming tumors, cells have to somehow inactivate several "tumor suppressor" genes that usually prevent cancer formation. Aberrant methylation of normally unmethylated CpG islands occurs frequently in human cancers. Methylation is a subtle punctuation-like modification of the DNA that marks genes for silencing, meaning that they are inactive and do not make RNA or proteins. Using breast cancer cell lines that artificially overproduce an enzyme which adds methylation markers to DNA, we applied a sequence pattern recognition algorithm (Lee, Easton, Kapil, 2006) to identify attributes for each CpG island. Applying the DAMIP-classifier described herein to the patterns found, we were able to derive a classification function based on the frequency of seven novel sequence patterns (PatMAn) that was capable of discriminating methylation-prone from methylation-resistant CpG islands with 90% correctness upon cross-validation, and 85% accuracy when tested against blind CpG islands unknown to us on the methylation status. This predictive rule offers a set of guidelines that allow biologistis to predict which genes have an increased risk of silencing by DNA methylation. That vulnerability could make those genes good markers for diagnosis and risk assessment in patients. In particular, PatMAn, which is based on seven "key words", 8-10 nucleotides long, can

predict which genes become methylated in breast and lung cancers in addition to the artificial cell lines.

If the key words are in the DNA sequence near the promoter of the gene, it is more likely to be methylated. The promoter of a gene is the place where enzymes start making DNA into RNA. Further analysis shows that PatMAn overlaps with the pattern of DNA bound by a set of proteins known as the Polycomb complex in embryonic stem cells. Polycomb appears to keep genes that regulate early development turned off in embryonic stem cells. Combining PatMAn with the Polycomb binding pattern to produce the "super-pattern" SUPER-PatMAn allows one to blind predict methylation-prone genes in cancers with more than 80 percent accuracy. The methylation pattern in cancer cells appears to echo Polycomb's binding in embryonic stem cells. Many of the genes affected play important roles in embryonic development. Many of the genes predicted to be methylation-prone are developmental regulators. The findings could support the idea that methylation-mediated silencing helps to lock the developmental state of tumor cells into being more stem cell-like. Among cancer biologists, hypermethylation is now the most well characterized epigenetic change to occur in tumors. The pattern recognition and classification tools offer the opportunity to classify the more than 29,000 known (but as yet unclassified) CpG islands in human chromosomes. This will provide an important resource for the identification of novel gene targets for further study as potential molecular markers that could have an impact on both cancer prevention and treatment. For aggressive cancers such as pancreatic cancer or some forms of incurable brain tumor, the ability to identify such sites offers potential new therapeutic interventions, leading to improved treatment.

References

1. Brooks, J.P., Lee, E.K.: Solving a Mixed-Integer Programming Formulation of a Multi-Category Constrained Discrimination Model. In: INFORMS Proceedings of Artificial Intelligence and Data Mining, pp. 1–6 (2006)
2. Brooks, J.P., Lee, E.K.: Analysis of the Consistency of a Mixed Integer Programming-based Multi-Category Constrained Discriminant Model. Annals of Operations Research on Data Mining (Early version appeared online) (in press, 2008)
3. Feltus, F.A., Lee, E.K., Costello, J.F., Plass, C., Vertino, P.M.: Predicting Aberrant CpG Island Methylation. Proceedings of the National Academy of Sciences 100(21), 12253–122558 (2003)
4. Feltus, F.A., Lee, E.K., Costello, J.F., Plass, C., Vertino, P.M.: DNA Signatures Associated with CpG island Methylation States. Genomics 87, 572–579 (2006)
5. Gallagher, R.J., Lee, E.K., Patterson, D.: An Optimization Model for Constrained Discriminant Analysis and Numerical Experiments with Iris, Thyroid, and Heart Disease Datasets. In: Cimino, J.J. (ed.) Proceedings of the 1996 American Medical Informatics Association, pp. 209–213 (1996)
6. Gallagher, R.J., Lee, E.K., Patterson, D.A.: Constrained discriminant analysis via 0/1 mixed integer programming. Annals of Operations Research 74, 65–88 (1997) (Special Issue on Non-Traditional Approaches to Statistical Classification and Regression)

7. Lee, E.K., Gallagher, R.J., Patterson, D.: A Linear Programming Approach to Discriminant Analysis with a Reserved Judgment Region. INFORMS Journal on Computing 15(1), 23–41 (2003)
8. Lee, E.K.: Large-scale optimization-based classification models in medicine and biology. Annals of Biomedical Engineering, Systems Biology and Bioinformatics 35(6), 1095–1109 (2007)
9. Lee, E.K., Easton, T., Gupta, K.: Novel evolutionary models and applications to sequence alignment problems. Annals of Operations Research – Computing and Optimization in Medicine and Life Sciences 148, 167–187 (2006)
10. Lee, E.K., Fung, A.Y.C., Brooks, J.P., Zaider, M.: Automated Tumor Volume Contouring in Soft-Tissue Sarcoma Adjuvant Brachytherapy Treatment. International Journal of Radiation Oncology, Biology and Physics 47(11), 1891–1910 (2002)
11. Lee, E.K., Gallagher, R., Campbell, A., Prausnitz, M.: Prediction of ultrasound-mediated disruption of cell membranes using machine learning techniques and statistical analysis of acoustic spectra. IEEE Transactions on Biomedical Engineering 51(1), 1–9 (2004)
12. Lee, E.K., Galis, Z.S.: Fingerprinting Native and Angiogenic Microvascular Networks through Pattern Recognition and Discriminant Analysis of Functional Perfusion Data (submitted, 2008)
13. Lee, E.K., Ashfaq, S., Jones, D.P., Rhodes, S.D., Weintrau, W.S., Hopper, C.H., Vaccarino, V., Harrison, D.G., Quyyumi, A.A.: Prediction of early atherosclerosis in healthy adults via novel markers of oxidative stress and d-ROMs. Working paper (2009)
14. Lee, E.K., Wu, T.L.: Classification and disease prediction via mathematical programming. In: Seref, O., Kundakcioglu, O.E., Pardalos, P. (eds.) Data Mining, Systems Analysis, and Optimization in Biomedicine, AIP Conference Proceedings, vol. 953, pp. 1–42 (2007)
15. McCabe, M., Lee, E.K., Vertino, P.M.: A Multi-Factorial Signature of DNA Sequence and Polycomb Binding Predicts Aberrant CpG Island Methylation. Cancer Research 69(1), 282–291 (2009)
16. Querec, T.D., Akondy, R., Lee, E.K., et al.: Systems biology approaches predict immunogenicity of the yellow fever vaccine in humans. Nature Immunology 10, 116–125 (2008)

G12 - Towards the Separation of Problem Modelling and Problem Solving

Mark Wallace and the G12 team

Monash University, Faculty of Information Technology,
Building H, Caulfield, Vic. 3800, Australia
mark.wallace@infotech.monash.edu.au
www.infotech.monash.edu.au/~wallace

Abstract. This paper presents the G12 large scale optimisation software platform, and discusses aspects of its architecture.

Keywords: constraint programming, modeling, optimization, software platform, search.

G12 is a software platform for solving combinatorial optimisation problems [SGM+05]. It was originally called a "constraint programming" platform, but we rather see it as an agnostic system which equally supports linear and mixed integer programming, constraint propagation and inference and a variety of other search and inference-based approaches for solving complex problems.

Problem modelling is separated in G12 as much as possible from problem solving. It is not (of course) our goal to automatically compile user-oriented problem models to highly efficient algorithms. Our target is to provide a software environment in which a user can initially write down a precise problem specification of his or her problem, without considering issues of computational efficiency. G12 supports the powerful 'Zinc' specification language [MNR+08], and the freely available 'MiniZinc' subset [NSB+07]. Subsequently a possibly different user can then add control information so as to guide the G12 system to exploit particular problem decompositions, inference techniques and search methods [BBB+08].

In recent years three different kinds of language have emerged for specifying and solving combinatorial optimisation problems. The first, most generic, languages are high-level modelling languages whose implementations include a number of solving techniques. These languages include mathematical programming languages, such as AMPL, constraint programming languages, such as CHIP and hybrid languages such as OPL. Such languages can be thought of as "80/20" languages which are designed to be able to handle many problems efficiently, but do not seek to cover all classes of problem.

Many problems require specialised solving techniques, which are not supported by the previous high-level languages. For these problems it is necessary to express specialised constraints and constraint behaviours, as well as specialised problem decompositions, and solver hybrids. Languages for expressing constraint behaviour have been particularly researched in the CP community. In particular

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 8–10, 2009.

we think of languages for writing propagators in Oz; attributed variables, suspensions, delayed goals, demons and action rules in languages such as SICStus Prolog, ECLiPSe and B-Prolog; and novel languages for encoding new global constraints by Beldiceanu, Pesant and others.

The third class of languages for solving complex problems are search languages supporting sophisticated combinations of branching, iteration, and improvement. These languages include Salsa, ToOLS and Comet.

A few years ago it when I was in the ECLiPSE team, we planned that the ECLiPSE language be broken down into three language subsets along these lines: a language for problem specification, a language for specifying constraint behaviour and a language for controlling search.

The G12 approach has deliberately taken quite a different path. Firstly G12 offers no language for specifying constraint behaviour. It supports a range of libraries for constraint propagation and solving, and interfaces which make it relatively straightforward to introduce new libraries at will [BGM+06]. New "global" constraints can also be easily interfaced to the system. However the G12 user cannot easily build new constraints with specialised constraint behaviours.

The G12 view is that efficiency is most effectively and easily achieved by mapping a problem down to an appropriate combination of solvers and search [PSWB08], rather than by adding new constraint implementations. Instead of a language for specifying constraint behaviours, therefore, G12 has a language for mapping problems to a form which can be evaluated efficiently [DDS08]. The problem modelling language is Zinc, and the language for mapping Zinc to a more efficient form is Cadmium. (The language used to run the resulting efficient algorithm is Mercury. The name G12 comes from the group in the periodic table which contains Mercury, Cadmium and Zinc.)

There is no additional language for expressing search in G12. Instead it supports two quite different ways of specifying the search method. The first way is as an extension to the Zinc search language [RMG+08]. A few generic search functions are available in Zinc, parameterised by Zinc functions. The philosophy behind this is that a Zinc problem model has a default behaviour, and the search function naturally belongs to Zinc both syntactically and semantically as a way of overruling its default behaviour. The second way that G12 supports search is that it can be built into certain solvers. G12 solvers have different capabilities: some can simply check for consistency of the current set of constraints, some can infer ("propagate") new constraints, some can optimise and some can even return candidate solutions. Search may be used in support of this last capability.

The G12 experiment is now gradually coming to fruition. G12 suports finite domain constraints, interval constraints over float variables, linear and mixed integer constraint solvers, propositional satisfiability solvers and search methods, BDD, set solvers and more. The Zinc and MiniZinc modelling languages are supported. The Cadmium mapping language is implemented - now in a fully general form so it can be applied to Zinc or any other modelling language for which a syntax and semantics are defined.

The first public release of G12 is due in early 2010, but already there are a number of major application projects in Australia for which G12 is the chosen software platform. We are excited to see G12 emerging at last from the laboratory.

References

[BBB+08] Becket, R., Brand, S., Brown, M., Duck, G., Feydy, T., Fischer, J., Huang, J., Marriott, K., Nethercote, N., Puchinger, J., Rafeh, R., Stuckey, P., Wallace, M.: The many roads leading to Rome: Solving Zinc models by various solvers. In: Proc. ModRef: 7th International Workshop on Constraint Modelling and Reformulation (2008)

[BGM+06] Becket, R., de la Banda, M.G., Marriott, K., Somogyi, Z., Stuckey, P.J., Wallace, M.: Adding constraint solving to Mercury. In: Van Hentenryck, P. (ed.) PADL 2006. LNCS, vol. 3819, pp. 118–133. Springer, Heidelberg (2005)

[DDS08] Duck, G., De Koninck, L., Stuckey, P.: Cadmium: An implementation of ACD term rewriting. In: de la Banda, M.G., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 531–545. Springer, Heidelberg (2008)

[MNR+08] Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P., Garcia de la Banda, M., Wallace, M.: The design of the Zinc modelling language. Constraints 13(3), 229–267 (2008)

[NSB+07] Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, G., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007)

[PSWB08] Puchinger, J., Stuckey, P., Wallace, M., Brand, S.: From high-level model to branch-and-price solution in G12. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 218–232. Springer, Heidelberg (2008)

[RMG+08] Rafeh, R., Marriott, K., de la Banda, M.G., Nethercote, N., Wallace, M.: Adding search to Zinc. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 624–629. Springer, Heidelberg (2008)

[SGM+05] Stuckey, P., de la Banda, M.G., Maher, M., Marriott, K., Slaney, J., 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)

Six Ways of Integrating Symmetries within Non-overlapping Constraints

Magnus Ågren[1], Nicolas Beldiceanu[2], Mats Carlsson[1], Mohamed Sbihi[2], Charlotte Truchet[3], and Stéphane Zampelli[2]

[1] SICS, P.O. Box 1263, SE-164 29 Kista, Sweden
{Magnus.Agren,Mats.Carlsson}@sics.se
[2] École des Mines de Nantes, LINA UMR CNRS 6241, FR-44307 Nantes, France
{Nicolas.Beldiceanu,Mohamed.Sbihi,Stephane.Zampelli}@emn.fr
[3] Université de Nantes, LINA UMR CNRS 6241, FR-44322 Nantes, France
Charlotte.Truchet@univ-nantes.fr

Abstract. This paper introduces six ways for handling a *chain of lexicographic ordering* (*lex-chain*) constraint between the origins of identical orthotopes (e.g., rectangles, boxes, hyper-rectangles) subject to the fact that they should not pairwise overlap. While the first two ways deal with the integration of a *lex-chain* constraint within a generic geometric constraint kernel, the four latter ways deal with the conjunction of a *lex-chain* constraint and a *non-overlapping* or a *cumulative* constraint. Experiments on academic two and three dimensional placement problems as well as on industrial problems show the benefit of such a strong integration of symmetry breaking constraints and non-overlapping ones.

1 Introduction

Symmetry constraints among identical objects are ubiquitous in industrial placement problems that involve packing a restricted number of types of orthotopes (generalized rectangles) subject to *non-overlapping* constraints.

In this context, an orthotope corresponds to the generalization of a rectangle in the k-dimensional case. An *orthotope* is defined by the coordinates of its smallest corner and by its potential orientations. An *orientation* is defined by k integers that give the size of the orthotope in the different dimensions. Two orthotopes are said to be *identical* if and only if their respective orientation sizes form identical multisets. In the rest of this paper, we assume that we pack each orthotope in such a way that its borders are parallel to the boundaries of the placement space.

In the context of Operations Research, breaking symmetries has been handled by characterizing and taking advantage of equivalence and dominance relations between patterns of fixed objects [1]. In the context of Constraint Programming, a natural way to break symmetries is to enforce a lexicographic ordering on the origin coordinates of identical orthotopes. This can be directly done by using a *lex-chain* constraint such as the one introduced in [2]. Even if this drastically reduces the number of solutions, it does not allow much pruning and/or speedup when we are looking for one single solution. This stems from the fact that symmetry is handled independently from non-overlapping.

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 11–25, 2009.

The question addressed by this paper is how to directly integrate a *lex-chain* constraint within a *non-overlapping* constraint and how it pays off in practice.

Section 2 recalls the context of this work, namely the generic geometric constraint kernel and its core algorithm, a multi-dimensional sweep algorithm, which performs filtering introduced in [3]. Since this algorithm will be used in the rest of the paper, Section 2 also recalls the principle of the filtering algorithm behind a *lex-chain* constraint. Section 3 describes two ways of directly handling symmetries in the multi-dimensional sweep algorithm, while Section 4 shows how to derive bounds on the coordinates of an orthotope from the interaction of symmetries and *non-overlapping* constraints. Since the *cumulative* constraint is a necessary condition for the *non-overlapping* constraint [4], Section 5 shows how to directly integrate symmetries within two well known filtering algorithms attached to the *cumulative* constraint. Section 6 evaluates the different proposed methods both on academic and industrial benchmarks and Section 7 concludes the paper.

2 Context

This work is in the context of the global constraint $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{C})$ introduced in [3], which handles the location in space of k-dimensional orthotopes $\mathcal{O}$ ($k \in \mathbb{N}^+$), each of which taking an orientation among a set of possible orientations $\mathcal{S}$, subject to geometrical constraints $\mathcal{C}$.[1] Each possible orientation from $\mathcal{S}$ is defined as a box in a k-dimensional space with the given sizes. More precisely, a *possible orientation* $s \in \mathcal{S}$ is an entity defined by its orientation id $s.sid$, and sizes $s.l[d]$ (where $s.l[d] > 0$ and $0 \leq d < k$). All attributes of a possible orientation are integer values. Each object $o \in \mathcal{O}$ is an entity defined by its unique object id $o.oid$ (an integer), possible orientation id $o.sid$ (an integer for *monomorphic* objects, which have a fixed orientation, or a domain variable[2] for *polymorphic* objects, which have alternative orientations), and origin $o.x[d],\ 0 \leq d < k$ (integers, or domain variables).

Since the most common geometrical constraint is the *non-overlapping* constraint between orthotopes, this paper focuses on breaking symmetries in this context (i.e., each shape is defined by one single box). For this purpose, we impose a *lex-chain* constraint on the origins of identical orthotopes. Given two vectors, x and y of k variables, $\langle x_0, x_1, \ldots, x_{k-1}\rangle \leq_{\text{lex}} \langle y_0, y_1, \ldots, y_{k-1}\rangle$ if and only if $k = 0 \vee (x_0 < y_0) \vee (x_0 = y_0 \wedge \langle x_1, \ldots, x_{k-1}\rangle \leq_{\text{lex}} \langle y_1, \ldots, y_{k-1}\rangle)$. Unless stated otherwise, the constraint is imposed wrt. the k dimensions $0, 1, \ldots, k-1$. The original filtering algorithm of the *lex-chain* constraint described in [2] is a two phase algorithm. In a first phase, it computes for each vector of the chain feasible lexicographic lower and upper bounds. In a second phase, a specific algorithm [5] filters the components of each vector of the chain according to the fact that it has to be located between two fixed vectors.

3 Integrating Symmetries within the Sweep Kernel

This section first recalls the principle of the sweep point algorithm attached to $geost$. It then indicates how to modify it in order to take advantage of the fact that we have

[1] In the context of this paper we have simplified the presentation of $geost$.

[2] A *domain variable* v is a variable ranging over finite set of integers denoted by $\text{dom}(v)$; $\underline{v}$ and $\overline{v}$ denote respectively the minimum and maximum possible values of v.

a restricted number of types of orthotopes. Without loss of generality, it assumes that we have one *non-overlapping* constraint over all orthotopes of $geost$ and one *lex-chain* constraint for each set of identical orthotopes.

3.1 Description of the Original Sweep Algorithm

The use of sweep algorithms in constraint filtering algorithms was introduced in [6] and applied to the non-overlapping 2D rectangles constraints. Let a *forbidden region* f be an orthotope of values for $o.x$ that would falsify the $geost$ constraint, represented as a fixed lower bound vector $f.\min$ and a fixed upper bound vector $f.\max$. Algorithm 1, $\mathrm{PruneMin}(o, d, k)$, searches for the first point c, by lexicographic order wrt. dimensions $d, (d+1) \mod k, \ldots, (d-1) \mod k$, that is inside the domain of $o.x$ but not inside any forbidden region. If such a c exists, the algorithm sets $\underline{o.x[d]}$ to $c[d]$, otherwise it fails. Two state vectors are maintained: the *sweep point* c, which holds a candidate value for $o.x$, and the *jump vector* n, which records knowledge about encountered forbidden regions.

The algorithm starts its recursive traversal of the placement space at point $c = \underline{o.x}$ with $n = \overline{o.x} + 1$ and could in principle explore all points of the domains of $o.x$, one by one, in increasing lexicographic order wrt. dimensions $d, (d+1) \mod k, \ldots, (d-1) \mod k$, until the first desired point is found. To make the search efficient, it skips points that are known to be inside some forbidden region. This knowledge is encoded in n, which is updated for every new f (see line 5) recording the fact that new candidate points can be found beyond that value. Whenever we skip to the next candidate point, we reset the elements of n that were used to their original values (see lines 6–15).

3.2 Enhancing the Original Sweep Kernel wrt. Identical Shapes

In the context of multiple occurrences of identical orthotopes, we can enhance the sweep algorithm attached to $geost$ by trying to reuse the information computed so far from one orthotope to another orthotope. For this purpose we introduce the notion of *domination* of an orthotope by another orthotope.

Given a $geost(k, \mathcal{O}, \mathcal{S}, \mathcal{C})$ constraint where $\mathcal{C}$ consists of one *non-overlapping* constraint between all orthotopes of $\mathcal{O}$ and a *lex-chain* constraint between each set of identical orthotopes, an orthotope $o_j \in \mathcal{O}$ *is dominated by* another orthotope $o_i \in \mathcal{O}$ if and only if the following conditions hold:

1. $\mathrm{dom}(o_j.x[p]) \subseteq \mathrm{dom}(o_i.x[p]), \forall p \in [0, k-1]$,
2. $\mathrm{dom}(o_j.sid) \subseteq \mathrm{dom}(o_i.sid)$,
3. the origin of o_j should be lexicographically greater than or equal to the origin of o_i.

Now, for one invocation of the sweep algorithm, which performs a recursive traversal of the placement space, we can make the following observation. If an orthotope o_j is dominated by another orthotope o_i and if we have already called the sweep algorithm for updating the minimum value of $o_i.x[p]$ ($p \in [0, k-1]$), we can take advantage of the information obtained while computing the minimum of $o_i.x[p]$. Let c_{ip} and n_{ip} respectively denote the final values of vectors c and n after running $\mathrm{PruneMin}(o_i, p, k)$. Note that while computing the minimum of $o_j.x[p]$, instead of starting the recursive

PROCEDURE PruneMin(o, d, k) : bool
1: $c \leftarrow \underline{o.x}$ // initial position of the point
2: $n \leftarrow \overline{o.x} + 1$ // upper limits+1 in the different dimensions
3: $f \leftarrow \mathrm{GetFR}(o, c, k)$ // check if c is infeasible
4: **while** $f \neq \bot$ **do**
5: $n \leftarrow \min(n, f.\max + 1)$ // maintain n as min of u.b. of forbidden regions
6: **for** $j \leftarrow k - 1$ **downto** 0 **do**
7: $j' \leftarrow (j + d) \mod k$ // least significant dimension first
8: $c[j'] \leftarrow n[j']$ // use $n[j']$ to jump
9: $n[j'] \leftarrow \overline{o.x}[j'] + 1$ // reset $n[j']$ to max
10: **if** $c[j'] \leq \overline{o.x}[j']$ **then**
11: **goto** next // candidate point found
12: **else**
13: $c[j'] \leftarrow \underline{o.x}[j']$ // exhausted a dimension, reset $c[j']$
14: **end if**
15: **end for**
16: **return false** // no next candidate point
17: next: $f \leftarrow \mathrm{GetFR}(o, c, k)$ // check again if c is infeasible
18: **end while**
19: $\underline{o.x}[d] \leftarrow c[d]$ // adjust earliest start in dim. d
20: **return** `true`

Algorithm 1. Adjusting the lower bound $o.x[d]$. $\mathrm{GetFR}(o, c, k)$ scans a list of forbidden regions, starting at the latest encountered one, returns $\bot$ if c is in the domain of $o.x$ and not inside any forbidden region f, and $f \neq \bot$ otherwise.

traversal of the placement space from $c = \underline{o_j.x}$ with $n = \overline{o_j.x} + 1$, we can start from the position c_{ip} and with the jump vector $\overline{n_{ip}}$. By using this observation, we decrease the number of jumps needed for filtering the bounds of the coordinates of n identical orthotopes from $k{\cdot}n^2$ to $k{\cdot}n$. Finally note that for one invocation of the sweep algorithm, forbidden regions for the origins of identical orthotopes need only be computed once. This observation is valid even if we don't have any lexicographic ordering constraints and is crucial for scalability in the context of identical orthotopes.

3.3 Integrating a Chain of Lexicographic Ordering Constraint within the Sweep Kernel

The main interest of the sweep algorithm attached to *geost* is to aggregate the set of forbidden points coming from different geometric constraints. In our context, these are the *non-overlapping* and *lex-chain* constraints. As a concrete example, consider the following problem:

Example 1. We have to place within a placement space of size 6×5 three squares s_1, s_2, s_3 of size 2×2 so that their respective origin coordinates (x_1, y_1), (x_2, y_2), (x_3, y_3) are lexicographically ordered in increasing order. Moreover, assume that the first and third squares are fixed so that $(x_1, y_1) = (2, 3)$ and $(x_3, y_3) = (5, 2)$, and

that $(x_2, y_2) \in ([1,5],[1,4])$. If we don't consider together the *non-overlapping* and the *lex-chain* constraint we can only restrict the domain of x_2 to interval $[2,5]$. But, as shown in Figure 1, if we aggregate the forbidden points coming from the *lex-chain* and *non-overlapping* constraints, we can further restrict the domain of x_2 to interval $[3,4]$. □

So the question is how to generate forbidden regions for a *lex-chain* constraint of the form $\langle l_0, l_1, \ldots, l_{k-1}\rangle \leq_{\text{lex}} \langle x_0, x_1, \ldots, x_{k-1}\rangle \leq_{\text{lex}} \langle u_0, u_1, \ldots, u_{k-1}\rangle$ where l_i, x_i and u_i respectively correspond to integers, domain variables and integers.[3] Let us first illustrate what forbidden regions we want to obtain in the context of Example 1.

Continuation of Example 1. Consider the constraint $\langle 2,4\rangle \leq_{\text{lex}} \langle x_2, y_2\rangle \leq_{\text{lex}} \langle 5,1\rangle$. We can associate to this *lex-chain* constraint the following forbidden regions; see the crosses in Part (B) of Figure 1:

- Since $x_2 < 2$ is not possible, we have $f.\min = [1,1], f.\max = [1,5]$ (column 1);
- Since $x_2 = 2 \wedge\ y_2 < 4$ is not possible, we have $f.\min = [2,1], f.\max = [2,3]$ (column 2);
- Since $x_2 > 5$ is not possible, we have $f.\min = [6,1], f.\max = [6,5]$ (column 6);
- Since $x_2 = 5 \wedge\ y_2 > 1$ is not possible, we have $f.\min = [5,2], f.\max = [5,5]$ (column 5). □

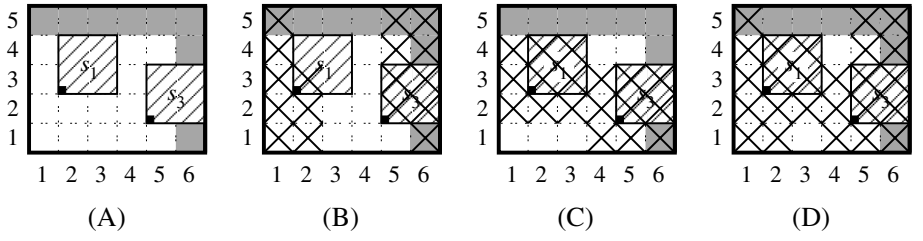


Fig. 1. (A) The two fixed squares s_1 and s_3 (gray squares are not possible for the origin of s_2 since it has to be included within the placement space depicted by a thick line); (B) Forbidden points (a cross) wrt. the *lex-chain* constraint; (C) Forbidden points (a cross) wrt. the *non-overlapping* constraint; (D) Aggregating all forbidden points: $(3,1)$ and $(4,4)$ are the only feasible points for the origin of s_2, which leads to restricting x_2 to interval $[3,4]$.

We show in Algorithm 2 how to generate such forbidden regions in a systematic way. As in Example 1, lines 1–6 generate for the lower bound constraint a forbidden region according to the fact that the most significant components $x_0, x_1, \ldots, x_{i-1}$ of vector x are respectively fixed to $l_0, l_1, \ldots, l_{i-1}$ ($i \in [0, k-1]$). Similarly, lines 7–12 generate k forbidden regions wrt. the upper bound u.

[3] As mentioned in Section 2, propagating a *lex-chain* constraint leads to generating such sub-problems.

PROCEDURE LexBetweenGenForbiddenReg$(k, x, l, u) : f[0..2 \cdot k-1]$
1: // GENERATE FORBIDDEN REGIONS WITH RESPECT TO LOWER BOUND l
2: **for** $i \leftarrow 0$ **to** $k-1$ **do**
3: $\forall j \in [0,i) : f[i].\min[j] \leftarrow l_j$; $f[i].\max[j] \leftarrow l_j$
4: $f[i].\min[i] \leftarrow \underline{x_i}$; $f[i].\max[i] \leftarrow l_i - 1$
5: $\forall j \in [i+1,k) : f[i].\min[j] \leftarrow \underline{x_j}$; $f[i].\max[j] \leftarrow \overline{x_j}$
6: **end for**
7: // GENERATE FORBIDDEN REGIONS WITH RESPECT TO UPPER BOUND u
8: **for** $i \leftarrow 0$ **to** $k-1$ **do**
9: $\forall j \in [0,i) : f[k+i].\min[j] \leftarrow u_j$; $f[k+i].\max[j] \leftarrow u_j$
10: $f[k+i].\min[i] \leftarrow u_i + 1$; $f[k+i].\max[i] \leftarrow \overline{x_i}$
11: $\forall j \in [i+1,k) : f[k+i].\min[j] \leftarrow \underline{x_j}$; $f[k+i].\max[j] \leftarrow \overline{x_j}$
12: **end for**
13: **return** f

Algorithm 2. Generates the $2 \cdot k$ forbidden regions wrt. variables $x_0, x_1, \ldots, x_{k-1}$ associated with the constraint $\langle l_0, l_1, \ldots, l_{k-1}\rangle \leq_{\text{lex}} \langle x_0, x_1, \ldots, x_{k-1}\rangle \leq_{\text{lex}} \langle u_0, u_1, \ldots, u_{k-1}\rangle$.

4 Integrating Symmetries within the Non-overlapping Constraint

We just saw how to aggregate forbidden regions coming from a *lex-chain* and a set of *non-overlapping* constraints. This section shows how to combine these two types of constraints more intimately in order to perform more deduction.

4.1 Deriving Bounds from the Interaction of the Chain of Lexicographic Ordering and Non-overlapping Constraints: The Monomorphic Case

We first consider the case of n orthotopes $\{o_j \mid 0 \leq j < n\}$ corresponding to a given fixed orientation s subject to *non-overlapping* as well as *lex-chain*.[4] In this context, we provide a lower bound low_j and an upper bound up_j for the origin of each orthotope, wrt. both constraints. Let $S[i]$ denote the size of the placement space in dimension i $(0 \leq i < k)$. Furthermore, let us denote by $O[0..k-1]$ and $P[0..k-1]$ the points respectively defined by $O[i] = \min(\underline{o_0.x[i]}, \underline{o_1.x[i]}, \ldots, \underline{o_{n-1}.x[i]})$ and by $P[i] = \max(\overline{o_0.x[i]}, \overline{o_1.x[i]}, \ldots, \overline{o_{n-1}.x[i]}) + s.l[i]$. We have $low_j \leq_{\text{lex}} o_j.x \leq_{\text{lex}} up_j$, $0 \leq j < n$, where:

$$low_j[i] = O[i] + \left\lfloor \frac{j \mod (\prod_{p=i}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor)}{\prod_{p=i+1}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor} \right\rfloor \cdot s.l[i] \ (0 \leq i < k) \tag{1}$$

$$up_j[i] = P[i] - \left\lfloor \frac{(n-1-j) \mod (\prod_{p=i}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor)}{\prod_{p=i+1}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor} \right\rfloor \cdot s.l[i] - s.l[i] \ (0 \leq i < k) \tag{2}$$

[4] In practice, this occurs in placement problems involving *several* occurrences of a given orthotope with the *same fixed orientation*.

The intuition behind formula (1)[5] in order to find the lower bound of the j^{th} object in dimension i is:

- First, fill complete slices wrt. dimensions $i, i+1, \ldots, k-1$ (such a complete slice involves $\prod_{p=i}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor$ objects),
- Then, with the remaining objects to place (i.e., $j \mod (\prod_{p=i}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor)$ objects), compute the number of complete slices wrt. dimensions $i+1, i+2, \ldots, k-1$ (i.e., $\left\lfloor \frac{j \mod (\prod_{p=i}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor)}{\prod_{p=i+1}^{p=k-1} \lfloor \frac{S[p]}{s.l[p]} \rfloor} \right\rfloor$ slices) and multiply this number by the length of a slice (i.e., $s.l[i]$).

4.2 Deriving Bounds from the Interaction of the Chain of Lexicographic Ordering and Non-overlapping Constraints: The Polymorphic Case

We now consider the case of n identical orthotopes $\{o_j \mid 0 \leq j < n\}$, again subject to *non-overlapping* as well as *lex-chain*. In this context, we provide three incomparable lower and upper bounds for the origin of each object. The first bound is based on the bound previously introduced. It simply consists in reducing the box sizes to their smallest values. For the second and third bounds, instead of reducing the sizes of a box to its smallest size, we decompose a box into n_ℓ smaller identical boxes that all have the same size ℓ in the different dimensions.[6] Assume that we want to find the lower bound for box o_j. The idea is to saturate the placement space with $n_\ell \cdot (j+1)$ boxes by considering the least significant dimension first and by starting at the lower left corner of the placement space. Then we subtract from the last end corner the different sizes of o_j in decreasing order (i.e., for the most significant dimension we subtract the largest size).[7] In the context of an upper bound, the idea is to saturate the placement space with $n_\ell \cdot (n-j) - 1$ small boxes by considering the least significant dimension first and by starting at the upper right corner of the placement space. Then we subtract ℓ from the last end corner of the $(n_\ell \cdot (n-j))^{th}$ smallest box. Based on the preceding formulas we obtain the following bounds. Without loss of generality, we assume that $s.l$ are sorted in decreasing order. A box can be decomposed into $n_\ell = \prod_{d=0}^{k-1} \lfloor \frac{s.l[d]}{\ell} \rfloor$ cubes of size ℓ with possibly some loss. We have[8]

$$low_j[i] = O[i] + \left\lfloor \frac{((j+1) \cdot n_\ell - 1) \mod (\prod_{p=i}^{p=k-1} \lfloor \frac{S[p]}{\ell} \rfloor)}{\prod_{p=i+1}^{p=k-1} \lfloor \frac{S[p]}{\ell} \rfloor} \right\rfloor \cdot \ell + \ell - s.l[i] \quad (3)$$

$$up_j[i] = P[i] - \left\lfloor \frac{((n-j) \cdot n_\ell - 1) \mod (\prod_{p=i}^{p=k-1} \lfloor \frac{S[p]}{\ell} \rfloor)}{\prod_{p=i+1}^{p=k-1} \lfloor \frac{S[p]}{\ell} \rfloor} \right\rfloor \cdot \ell - \ell \quad (4)$$

[5] Formula (2) is obtained in a similar way. The proof is available in [7].

[6] ℓ takes its value between 1 and the smallest size of the box we consider (i.e., $1 \leq \ell \leq \min\{s.l[i] \mid 0 \leq i < k\}$).

[7] In Figures 2 and 3, diagonal lines depict this subtraction.

[8] The proof is available in [7].

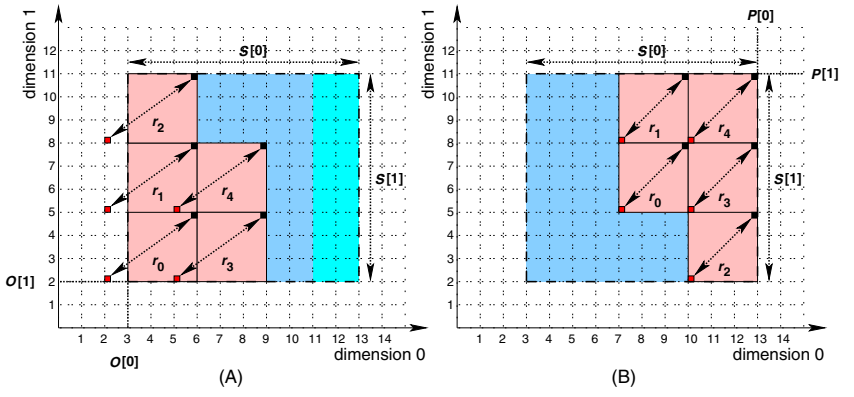


Fig. 2. Computing the lower (A) and upper (B) bounds of a set of rectangles for the second bound with $\ell = \min(4, 3)$ for the polymorphic case

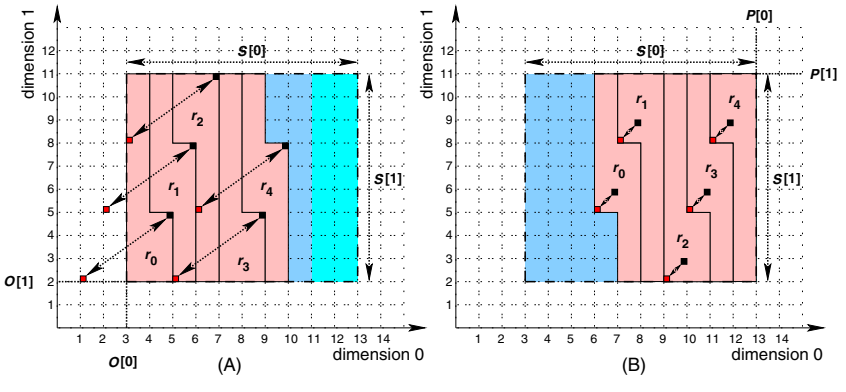


Fig. 3. Computing the lower (A) and upper (B) bounds of a set of rectangles for the second bound with $\ell = \gcd(4, 3)$ for the polymorphic case

In practice it is not clear which value of ℓ provides the best bound. Therefore, we currently restrict ourselves to the values $s.minl$ and $\gcd(s.l[0], s.l[1], \ldots, s.l[k-1])$. The bounds obtained with these two values are incomparable. Figures 2 and 3 respectively illustrate this second bound for placing a set of 5 rectangles for which the orientation sizes form the multiset $\{\{3, 4\}\}$ within a big rectangle of size 10×9 with $\ell = \min(4, 3)$ and $\ell = \gcd(4, 3)$.

5 Integrating Symmetries within the Cumulative Constraint

We have already shown how to combine a *lex-chain* and a *non-overlapping* constraint. But, in the context of a *non-overlapping* constraint, the *cumulative* constraint is a well known necessary condition [4]. This section shows how to directly integrate the fact that we have a *lex-chain* constraint within two well known filtering algorithms of the

cumulative constraint: filtering wrt. the *compulsory part profile* [8] and filtering wrt. *task intervals* [9].

5.1 Handling Symmetries in the Context of the Compulsory Part Profile

Let us first recall the notion of compulsory part profile, which will be used throughout this section. In the context of the *cumulative* constraint, the *compulsory part* of a task t corresponds to the intersection of all feasible schedules of t. As the domain of the start of task t gets more and more restricted the compulsory part of t will increase until becoming a schedule of task t. The compulsory part of a task t can be directly computed by making the intersection between the earliest start and the latest end of task t. The *compulsory part profile* associated with the tasks $\mathcal{T}$ of a *cumulative* constraint is the cumulated profile of all compulsory parts of tasks of $\mathcal{T}$.

In the context of *non-overlapping* constraints, many search strategies [10] try to first fix the coordinates of all objects in a given dimension d before fixing all the coordinates in the other dimensions.[9] But now, if we don't take care of the interaction between the *cumulative* and *lex-chain* constraints, we can have a huge compulsory part profile which will be totally ignored by the *lex-chain* constraint. The following illustrative example will make things clear.

Example 2. Assume that we have to place 8 squares of size 2×2 within the bounding box $[0, 9] \times [0, 3]$ (i.e., in the context of *cumulative*, 0 and $9+1$ respectively correspond to the earliest start and the latest end, while 4 is the resource limit). In addition, assume that the compulsory part profile in the most significant (wrt. $\leq_{\text{lex}}$) dimension of the placement space corresponds to the following 3 consecutive intervals $[0, 3]$, $[4, 5]$ and $[6, 9]$ of respective heights 0, 2 and 0.[10] If there is no interaction between this *cumulative* constraint and the lexicographic ordering constraint that states that the eight 2×2 squares should be lexicographically ordered, then we get the following domain reductions: The earliest start of the first two squares of the lexicographic ordering is 0, the earliest start of the third and fourth squares is 2, the earliest start of the fifth and sixth squares is 4, and the earliest start of the last two squares is 6. This is obviously an underestimation since, because of the compulsory part profile of the *cumulative* constraint, we can start at most one single square at instant 4. □

In the context of a *cumulative* constraint, we now show how to estimate the earliest start in the most significant dimension (msd) of each orthotope of a *lex-chain* constraint according to an existing compulsory part profile.[11] To each orthotope o corresponds a task t for which the origin, the duration and the height are respectively the coordinate of o in the msd, the size of o in the msd, and the product of the sizes of o in the dimensions different from the msd. Now, the idea is to simply consider the orthotopes in increasing lexicographic order and to find out for each corresponding task its earliest possible start

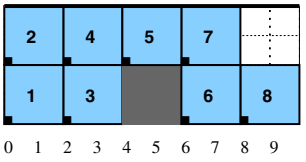
[9] In the benchmarks presented in Section 6, this is the case e.g. for the heuristic used for the monomorphic Partridge problem.

[10] The compulsory part corresponding to interval $[4, 5]$ does not correspond to the 8 squares to place, for it comes from another fixed object.

[11] The same idea can be used for estimating the latest end in the msd.

on the msd. The following condition is checked for testing whether a start is feasible or not: When added to the cumulative profile, the maximum height should not exceed the resource limit.[12]

By reconsidering Example 2, this idea is illustrated on the right hand side, estimating the minimum value of the coordinates in the msd of eight squares of size 2. The squares are successively placed at their earliest possible start according to the compulsory part profile. Consequently, the minimum values of the coordinates in the most significant dimension of squares $1, 2, \ldots, 8$ equal respectively 0, 0, 2, 2, 4, 6, 6 and 8 (and not to 0, 0, 2, 2, 4, 4, 6 and 6 as before).



5.2 Handling Symmetries in the Context of Task Intervals

In the context of the *cumulative* constraint, *task interval methods* prevent the overuse as well as the underuse of intervals derived from the earliest start and the latest end of the tasks to schedule. This section focuses on the problem of pruning the origin of the tasks of the *cumulative* constraint so that we don't lose too much space within a given fixed interval according to the fact that we have an ordering on the origin of identical tasks.[13] For this purpose, consider the set of all identical tasks T of duration d and height h, an interval $[\inf, \sup)$ and the height *gap* of free space on top of the interval, and the slack σ of the interval (i.e., the maximum allowed unused space of the interval). For a given set of tasks S, let $overlap(S)$ denote the sum of the maximum overlap of the tasks in S. To find out whether or not $t \in T$ must intersect $[\inf, \sup)$, the task intervals pruning rule makes the test:

$$(\sup - \inf) \cdot gap - (overlap(T) - overlap(\{t\})) > \sigma \tag{5}$$

If this test succeeds, we know that t must overlap the free space of $[\inf, \sup)$ to some extent. Specifically, t must then overlap the free space of $[\inf, \sup)$ at least by

$$(\sup - \inf) \cdot gap - (overlap(T) - overlap(\{t\})) - \sigma$$

which means that t must intersect in time $[\inf, \sup)$ at least by:

$$\left\lceil \frac{(\sup - \inf) \cdot gap - (overlap(T) - overlap(\{t\})) - \sigma}{\mathrm{d}} \right\rceil$$

This can be strengthened in the presence of symmetries. Assume a partial order $\preceq$ over the start times of the tasks T implied by a *lex-chain* constraint. Assume moreover that $t_i \neq t_j \in T$ are tasks such that $t_i \preceq t_j$. Then the positionings of t_i and t_j wrt. interval $[\inf, \sup)$ are in fact not independent:

[12] The resource limit equals the product of the sizes of the placement space in the dimensions different from the msd.

[13] Such an ordering exists for the *cumulative* constraint associated with the msd of the lexicographic ordering constraint.

- if t_j is assumed to end strictly before the interval $[\inf, \sup)$, then t_i must also be assumed to end strictly before $[\inf, \sup)$; and
- if t_i is assumed to start strictly after the interval $[\inf, \sup)$, then t_j must also be assumed to start strictly after $[\inf, \sup)$.

Considering now the chain $t_1 \preceq \cdots \preceq t_n$ and assuming that t is the i^{th} task t_i of this chain, we split the pruning rule above into two cases: the first case corresponding to the tasks $t_1, \ldots, t_{i-1}$ not succeeding t_i; and the second case corresponding to the tasks $t_{i+1}, \ldots, t_n$ not preceding t_i.

For the first case, since each of the tasks $t_1, \ldots, t_{i-1}$ must not succeed t_i, assuming that t_i ends before $[\inf, \sup)$ implies that the tasks $t_1, \ldots, t_{i-1}$ must also end before $[\inf, \sup)$. Hence, the test (5) can be strengthened to:

$$(\sup - \inf) \cdot \mathit{gap} - (\mathit{overlap}(T) - \mathit{overlap}(\{t_1, \ldots, t_i\})) > \sigma \tag{6}$$

If this test succeeds, we know that all the tasks $t_1, \ldots, t_i$ must overlap the free space of $[\inf, \sup)$ at least by:

$$(\sup - \inf) \cdot \mathit{gap} - (\mathit{overlap}(T) - \mathit{overlap}(\{t_1, \ldots, t_i\})) - \sigma \tag{7}$$

Now, since we wish to prune t_i, this must be translated into how far into $[\inf, \sup)$ we must force t_i so that the remaining tasks may overlap the free space of $[\inf, \sup)$ enough. This can be calculated in two steps as follows:

- STEP 1: Calculate the largest number *dfill* of columns of maximum height and width d, covering part of but not more than the free space of $[\inf, \sup)$.
- STEP 2: Calculate the smallest number *unitfill* of columns of maximum height and width 1, covering the remaining free space of $[\inf, \sup)$.

We use *tofill* to denote the value (7). STEP 1 can be calculated by:

$$\alpha \leftarrow \min\left(\left\lfloor \tfrac{\mathit{gap}}{\mathrm{h}} \right\rfloor, i\right) \quad \text{[largest number of stacked tasks]}$$

$$\beta \leftarrow \left\lfloor \tfrac{\mathit{tofill}}{\alpha \cdot \mathrm{h}} \right\rfloor \quad \text{[largest number of unit-size columns]}$$

$$\mathit{dfill} \leftarrow \left\lfloor \tfrac{\beta}{\mathrm{d}} \right\rfloor \quad \text{[largest number of d-size columns]}$$

Given this, the remaining free space of $[\inf, \sup)$ is:

$$\mathit{restfill} = \mathit{tofill} - \mathit{dfill} \cdot \alpha \cdot \mathrm{d} \cdot \mathrm{h}$$

When $\mathit{restfill} > 0$, STEP 2 can then be calculated by:

$$\gamma \leftarrow \min(i - \mathit{dfill} \cdot \alpha, \alpha) \quad \text{[largest number of stacked tasks still available]}$$

$$\mathit{unitfill} \leftarrow \left\lceil \tfrac{\mathit{restfill}}{\mathrm{h} \cdot \gamma} \right\rceil \quad \text{[smallest number of unit-size columns]}$$

Now, given the values *dfill* and *unitfill*, to overlap the free space of $[\inf, \sup)$ by at least the value (7), the start time of t_i must be at least $\inf + (\mathit{dfill} - 1) \cdot t_i.\mathrm{d} + \mathit{unitfill}$. An example of this method can be found in [7].

6 Performance Evaluation

All the new filtering methods described in this paper were integrated into our *geost* kernel [3] in order to strengthen the sweep-based filtering for non-overlapping constraints. The experiments were run in SICStus Prolog 4 compiled with gcc version 4.1.0 on a 3GHz Pentium IV with 1MB of cache.

We ran two benchmarks, *Scale* and *KLS*, seeking to evaluate the performance gain of domination in *greedy* execution mode, where the constraint tries to assign all variables in a single run, and simply fails if it cannot. Note that this greedy mode fits well inside a tree search based procedure: at every node of the search tree, a greedy step can be attempted in order to solve the problem in one shot, and if it fails, a normal propagation and branching step can be done. Three benchmarks, *Conway*, *Partridge* and *Pallet* were run in normal propagation mode, under tree search. The symmetry that stems from multiple pieces of the same shape is broken by imposing a lexicographic order on their origins. The purpose here was to compare the performance of treating these lexicographic ordering constraints inside *non-overlapping* and *cumulative* as opposed to posting them separately. Since this is not a paper on heuristics, the exact models and search procedures are probably of little interest, and are only given in the corresponding code of the benchmarks in Appendix B of [7]. We now describe the five benchmarks and the results, which are shown in Table 1.

Scale. As in [3], we constructed a set of loosely constrained placement problems (i.e., 20% spare space), generating one set of random problem instances of $m \in \{2^{10}, 2^{11}, \ldots, 2^{22}\}$ 2D items involving $t \in \{1, 16, 256, 1024\}$ distinct shapes. The results indicate that domination brings the time complexity down from roughly $O(m^2)$ to virtually $O(m)$. The results also show that the speedup gained by domination goes down as the number of distinct shapes goes up. In the larger instances, the total number of items vastly outnumbers the number of distinct shapes. With domination, we could now pack 2^{22} 2D items of 1024 distinct shapes (over 8 million domain variables) in four CPU minutes, an improvement by more than two orders of magnitude over [3].

KLS. To evaluate the greedy mode in a setting involving side-constraints in addition to non-overlapping, we studied the problem of packing a given number of 3D items into containers, with the objective to minimize the number of containers required. The containers all have the same size and weight capacity, whereas the items come in 59 different shapes and weights. The items cannot overlap and must be fully inside some container. The total weight of the items inside a given container must not exceed the weight capacity. Also, some items must be placed on the container floor, whereas other items cannot be placed underneath any other item. The whole problem can be modeled as a single 6D *geost* constraint. We ran 25 instances of different size; see the figure on the right hand side. The largest instance, with 16486 items, was solved in 35 seconds with domination and 1284 seconds without.

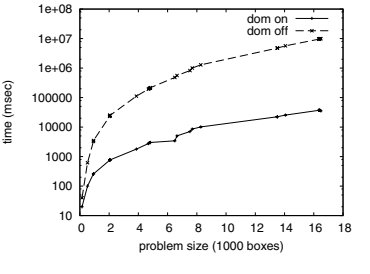
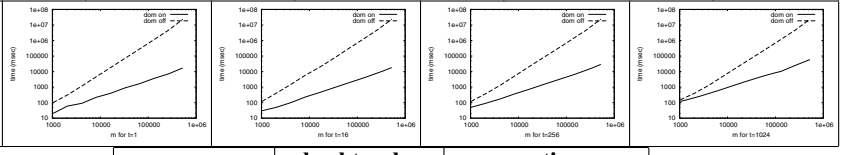


Table 1. Top: *Scale* for 2D items, with domination on and off. **Center:** Results for *Conway* and *Partridge*. **Bottom left:** An instance $\mathbf{pallet}(x, y, a, b, n, h)$ denotes the task of packing h pieces of shape $a \times b$ and $n - h$ pieces of shape $b \times a$ into a placement space of shape $x \times y$. **Bottom right:** Polymorphic variants of the same instances, where the parameter h has been left free. *lex-chain* constraints are treated inside *geost* in columns marked **lex in** and posted separately in columns marked **lex out**. All runtimes (ms) and backtrack counts are for finding the first solution.

m	$t=1$		$t=16$		$t=256$		$t=1024$	
	dom on	**dom off**	**dom on**	**dom off**	**dom on**	**dom off**	**dom on**	**dom off**
1024	20	100	30	120	50	120	120	150
2048	60	310	50	410	90	370	210	400
4096	90	1160	100	1480	170	1270	380	1320
8192	220	4640	230	5780	360	5030	780	5170
16384	400	18060	450	19010	710	19990	1550	20270
32768	890	71210	910	73230	1410	77340	3050	77200
65536	1650	279480	1880	300540	2920	296650	6100	299510
131072	3590	1118410	3760	1177900	5910	1188740	10280	1186030
262144	7020	4488510	7980	4812300	12020	4758390	25280	4746410
524288	17100	22671540	18000	23210070	29210	23553550	58910	23512450



	backtracks		**runtime**	
	lex in	**lex out**	**lex in**	**lex out**
conway(5,5,5)	6658	10192	11890	12850
partridge(8,1)	565	853	6400	3460
partridge(9,1)	27714	63429	347100	367050
partridge(10,1)	683643	1265284	15160080	9154320
partridge(11,1)	80832	189797	2009150	1964130
partridge(12,1)	790109	1676827	37850240	24203920
partridge(6,3)	7122	20459	13680	29610

	monomorphic				**polymorphic**			
	backtracks		**runtime**		**backtracks**		**runtime**	
	lex in	**lex out**	**lex in**	**lex out**	**lex in**	**lex out**	**lex in**	**lex out**
pallet(26,19,5,2,49,30)	0	0	130	110	8	8	180	90
pallet(28,17,5,2,47,25)	184	325	570	320	398	433	660	360
pallet(29,20,4,3,48,28)	664	1419	1890	1300	9767	14457	22500	14870
pallet(30,17,4,3,42,18)	778	1580	2380	1290	19807	28015	28190	20130
pallet(30,19,7,2,40,24)	74	115	190	140	19	81	150	90
pallet(31,19,7,2,41,24)	20544	73695	34190	57840	728743	932846	666010	506730
pallet(32,17,7,2,38,20)	491	850	630	660	159	172	310	140
pallet(33,17,7,2,39,20)	8129	26644	13300	26030	390539	567304	366320	286930
pallet(33,19,7,2,44,30)	3556	34778	9690	23450	789894	1460451	689080	743530
pallet(33,22,5,3,48,24)	41	54	220	160	65	73	290	140
pallet(34,17,5,3,38,24)	0	268	90	170	425	900	390	380
pallet(36,34,7,4,43,25)	14030	28855	25830	16800	33874	41648	66520	42220
pallet(37,19,7,2,49,33)	96	136	240	160	113	215	260	170
pallet(38,26,5,4,49,29)	6141	12830	14880	10910	39486	52787	75450	46530

Conway. The problem consists in placing 6 pieces of shape $4 \times 2 \times 1$, 6 pieces of shape $3 \times 2 \times 2$ and 5 unit cubes within a $5 \times 5 \times 5$ cube. All pieces can be rotated freely.

Partridge. The problem consists in tiling a square of size $\frac{n \cdot (n+1)}{2}$ by 1 square of size 1, 2 squares of size $2, \ldots, n$ squares of size n. It was initially proposed by R. Wainwright.[14] We tried the instances $n = 8, \ldots, n = 12$. Note that, to our best knowledge, this is the first reported solution for $n = 12$. We also tried a polymorphic variant of the problem: tile a rectangle of size 21×63 by 1 rectangle of size 1×3, 2 rectangles of size 2×6, $\ldots$, 6 rectangles of size 6×18, where all rectangles can be rotated.

Pallet. The problem consists in placing a given number of identical, non-overlapping, rectangular pieces of a given size onto a rectangular pallet, also of a given size. We selected several instances from D. Lobato's data sets[15] and ran two variants of each instance: (i) a polymorphic variant, with 90 degrees rotation allowed, and (ii) a monomorphic variant with the number of horizontal vs. vertical pieces fixed.

Evaluation of methods. We ran the last three benchmarks in versions where only one given method at a time was switched on. For reasons of space we cannot present the full results; instead, we summarize the findings. First, we found that for monomorphic benchmarks, integration of symmetries into *cumulative* was more effective than integration into *non-overlapping*, but for polymorphic benchmarks, it was the other way around. Finally, integration into *non-overlapping* had the highest runtime overhead among the methods. Integration into task intervals was the least effective among the methods, but had a very low overhead.

7 Conclusion

For the first time, symmetry breaking has been fully integrated into the filtering algorithms of global constraints. This was done in two contexts:

(a) Real-life placement problems tend to involve many more objects to place than distinct shapes. They can be too large to solve solely with constructive search. The ability to perform a greedy assignment, possibly with a limited amount of search, staying inside a constraint programming framework, can be crucial to solving such problems. By using the fact that many objects are of the same shape, we showed that the complexity of such a greedy assignment in the context of a sweep algorithm can go down from $O(n^2)$ to virtually $O(n)$ for n objects.

(b) We identified and exploited four ways of handling symmetry breaking *lex-chain* constraints inside a *non-overlapping* or *cumulative* constraint. Our results show that the tight integration saves search effort but not necessarily CPU time: slowdown up to 2 times, but also sometimes speedup up to 2.5 times, was observed.

Finally, we found the first reported solution to **partridge(12,1)**.

[14] See `http://mathpuzzle.com/partridge.html`

[15] See `http://lagrange.ime.usp.br/~lobato/packing/`

Acknowledgements

This research was conducted under European Union Sixth Framework Programme Contract FP6-034691 “Net-WMS”. In this context thanks to A. Aggoun from KLS OPTIM (http://www.klsoptim.com/) for providing relevant industrial benchmarks.

References

1. Scheithauer, G.: Equivalence and dominance for problems of optimal packing of rectangles. Ricerca Operativa 27(83), 3–34 (1998)
2. Carlsson, M., Beldiceanu, N.: Arc-consistency for a chain of lexicographic ordering constraints. Technical Report T2002-18, Swedish Institute of Computer Science (2002)
3. Beldiceanu, N., Carlsson, M., Poder, E., Sadek, R., Truchet, C.: A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 180–194. Springer, Heidelberg (2007)
4. Aggoun, A., Beldiceanu, N.: Extending CHIP in order to solve complex scheduling and placement problems. Mathl. Comput. Modelling 17(7), 57–73 (1993)
5. Beldiceanu, N., Carlsson, M., Rampon, J.-X.: Global constraint catalog. Technical Report T2005-08, Swedish Institute of Computer Science (2005), http://www.emn.fr/x-info/sdemasse/gccat/Clex_between.html
6. Beldiceanu, N., Carlsson, M.: Sweep as a generic pruning technique applied to the non-overlapping rectangles constraints. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 377–391. Springer, Heidelberg (2001)
7. Ågren, M., Beldiceanu, N., Carlsson, M., Sbihi, M., Truchet, C., Zampelli, S.: Six ways of integrating symmetries within non-overlapping constraints. SICS Technical Report T2009:01, Swedish Institute of Computer Science (2009)
8. Lahrichi, A.: Scheduling: the notions of hump, compulsory parts and their use in cumulative problems. C.R. Acad. Sci., Paris 294, 209–211 (1982)
9. Caseau, Y., Laburthe, F.: Cumulative scheduling with task intervals. In: Joint International Conference and Symposium on Logic Programming (JICSLP 1996). MIT Press, Cambridge (1996)
10. Simonis, H., O’Sullivan, B.: Search strategies for rectangle packing. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 52–66. Springer, Heidelberg (2008)

Throughput Constraint for Synchronous Data Flow Graphs

Alessio Bonfietti, Michele Lombardi, Michela Milano, and Luca Benini

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

Abstract. Stream (data-flow) computing is considered an effective paradigm for parallel programming of high-end multi-core architectures for embedded applications (networking, multimedia, wireless communication). Our work addresses a key step in stream programming for embedded multicores, namely, the efficient mapping of a synchronous data-flow graph (SDFG) onto a multi-core platform subject to a minimum throughput requirement. This problem has been extensively studied in the past, and its complexity has lead researches to develop incomplete algorithms which cannot exclude false negatives. We developed a CP-based complete algorithm based on a new throughput-bounding constraint. The algorithm has been tested on a number of non-trivial SDFG mapping problems with promising results.

1 Introduction

The transition in high-performance embedded computing from single CPU platforms with custom application-specific accelerators to programmable multi processor systems-on-chip (MPSoCs) is now a widely acknowledged fact [3,4]. All leading hardware platform providers in high-volume applications areas such as networking, multimedia, high-definition digital TV and wireless base stations are now marketing MPSoC platforms with ten or more cores and are rapidly moving towards the hundred-cores landmark [5,6,7]. Large-scale parallel programming has therefore become a pivotal challenge well beyond the small-volume market of high-performance scientific computing. Virtually all key markets in data-intensive embedded computing are in desperate need of expressive programming abstractions and tools enabling programmers to take advantage of MPSoC architectures, while at the same time boosting productivity.

Stream computing based on a data-flow model of computation [8,9] is viewed by many as one of the most promising programming paradigms for embedded multi-core computing. It matches well the data-processing dominated nature of many algorithms in the embedded computing domains of interest. It also offers convenient abstractions (synchronous data-flow graphs) that are at the same time understandable and manageable by programmers and amenable to automatic translation into efficient parallel executions on MPSoC target platforms. Our work addresses one of the key challenges in the development of programming tool-flow for stream computing, namely, the efficient mapping of synchronous

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 26–40, 2009.

data-flow graphs (SDFG) onto multi-core platforms. More in detail, our objective is to find allocations and schedules of SDFG nodes (also called actors or tasks) onto processors that meet throughput constraints or *maximize throughput*, which can be informally defined as the number of executions of a SDFG in a time unit. In particular, an allocation is an unique association between actors and processors and a schedule is a static order between actors running on the same processor. Meeting a throughput constraint is often the key requirement in many embedded application domains, such as digital television, multimedia streaming, etc.

The problem of SDFG mapping onto multiple processors has been studied extensively in the past. However, the complex execution semantic of SDFGs on multiple processors has lead researchers to focus only on incomplete mapping algorithms based on decomposition [12] [14]. Allocation of actors onto processors is first obtained, using approximate cost functions such as workload balancing [12], and incomplete search algorithms. Then the throughput-feasible or throughput-maximal scheduling of actors on single processors is computed, using incomplete search techniques such as list scheduling [8]. The reason for the use of incomplete approaches is that both computing an optimal allocation and an optimal schedule is NP-hard.

Our approach is based on Constraint Programming and tackles the overall problem of allocating actors to processors and schedule their order such that a throughput constraint is satisfied; hence we avoid the intrinsic loss of optimality due to the decomposition of the problem into two separated stages. In fact, our method is complete, meaning that it is guaranteed to find a feasible or an optimal solution in case it exists. The core of the approach is a novel throughput constraint, based on the computation of the maximum cycle mean over a graph that is modified during search according to allocation and scheduling decisions. To our knowledge this is the first time a throughput constraint is implemented into a CP language. We have evaluated the scalability of our code on three sets of realistic instances: cyclic, acyclic and strongly connected graphs. The second set of instances is quite difficult and scales poorly, while the first and third sets scale well. We can solve instances up to 20-30 nodes in the order of seconds for finding a feasible solution and in the order of few minutes for proving throughput optimality.

2 Preliminaries on SDFG and HSDFG

Synchronous Dataflow Graphs (SDFGs) [1] are used to model multimedia applications with timing constraints that must be bound to a Multi Processor System on Chip. They allow modeling of both pipelined streaming and cyclic dependencies between tasks. To test the performances of an application on a platform, one important parameter is the throughput. In the following we provide some preliminary notions on synchronous data flow graphs used in this paper.

Definition 1. *An SDFG is a tuple (A,D) consisting of a finite set A of actors and a finite set D of* dependency edges. *A dependency edge d = (a,b, p,q,tok)*

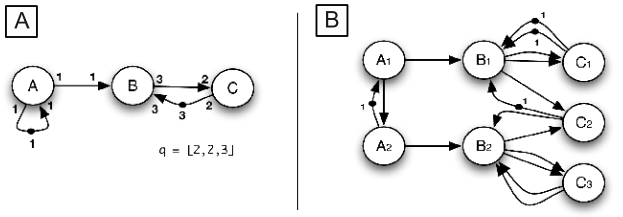


Fig. 1. (A) An example of SDFG; (B) the corresponding equivalent HSDFG

denotes a dependency of actor b on a. When a executes it produces p tokens on d and when b executes it removes q tokens from d. Edge d may also contain initial tokens. This number is notated by tok.

Actor execution is defined in terms of firings. An essential property of SDFGs is that every time an actor fires it consumes a given and fixed amount of tokens from its input edges and produces a known and fixed amount of tokens on its output edges. These amounts are called *rates*. The SDFG illustrated in figure 1A presents initial token on edges (A,A) and (C,B) with *tok* values respectively 1 and 3. The rates on the edges determine how often actors have to fire w.r.t. each other such that the distribution of tokens over all edges is not changed. This property is captured in the repetition vector.

Definition 2. *A repetition vector of an SDFG=(A,D) is a function* $\gamma : A \rightarrow N$ *such that for every edge* $(a, b, p, q, tok) \in D$ *from* $a \in A$ *to* $b \in A$, $p\gamma(a) = q\gamma(b)$. *A repetition vector q is called non-trivial if* $\forall a \in A$, $\gamma(a) > 0$.

The SDFG reported in figure 1A has three actors. Actor A has a dependency edge to itself with one token on it. It means that the two firings of A cannot be executed in parallel because the token on the edge (A,A) forces the sequential execution of the actor A. Also, each time A executes it produces one token that can be consumed by B. Each time B executes it produces 3 tokens while C consumes 2 tokens. Also when C executes it produces 2 tokens while C requires 3 tokens to fire. Thus, every 2 executions of B correspond to 3 executions of C. This is captured in the repetition vector reported in figure 1A. Concerning initial tokens, they both define the order of actor firings and the number of instances of a single actor simultaneously running. For example, at the beginning of the application, A only can start (as there are tokens enough on each ingoing arc). A will be then followed by B and then C.

An SDFG is called consistent if it has a non-trivial repetition vector. The smallest non-trivial repetition vector of a consistent SDFG is called *the* repetition vector. Consistency and absence of deadlock are two important properties for SDFGs which can be verified efficiently [2], [11]. Any SDFG which is not consistent requires unbounded memory to execute or deadlocks, meaning that no actor is able to fire. Such SDFGs are not useful in practice. Therefore, we focus on consistent and deadlock free SDFGs.

Throughput is an important design constraint for embedded multimedia systems. The throughput of an SDFG refers to how often an actor produces an output token. To compute throughput, a notion of time must be associated with the firing of each actor (i.e., each actor has a duration also called *response time*) and an execution scheme must be defined. We consider as execution scheme the self timed execution of actors: each actor fires as soon as all of its input data are available (see [11] for details). In a real platform self timed execution is implemented by assigning to each processor a sequence of actors to be fired in fixed order: the exact firing times are determined by synchronizing with other processors at run time.

SDFGs in which all rates associated to ports equal 1 are called Homogeneous Synchronous Data Flow Graphs (HSDFGs, [1]). As all rates are 1, the repetition vector for an HSDFG associates 1 to all actors. Every SDFG G = (A,D) can be converted to an equivalent HSDFG GH = (AH,DH), by using the conversion algorithm in [2], sec. 3.8. In figure 1B we report the HSDFG corresponding to the SDFG in figure 1A. For each node in the SDFG we have a number of nodes in the HSDFG equal to the corresponding number in the repetition vector. Equivalence means that there exists a one-to-one mapping between the SDFG and HSDFG actor firings, therefore the two graphs have the same throughput. The fastest method to compute the throughput of an HSDFG is the use of the maximum cycle mean (MCM) algorithm [2], as the throughput is $1/MCM$. In the context of SDFGs, the cycle mean of a cycle C is the total computation time of actors in C divided by the number of tokens on the edges in C; the maximum cycle mean for an SDFG is also known as *iteration period*. Clearly longer cycles influence the throughput more than shorter ones.

The problem we face in this paper is the following: given a multiprocessor platform with homogeneous processors we have to allocate each actor to a processor and to order actors on each processors such as all precedence constraints and the throughput constraints are met. Both the allocation and the schedule are static, meaning that they remain the same over all the iterations. In this paper we assume negligible delay associated to inter-processor communication and a uniform memory model for the processors. This models fits well the behavior of a cache-coherent, shared memory single-chip multiprocessor, such as the ARM MPCore [18].

3 Related Work

The body of work on SDFG mapping is extensive and covers more than two decades, starting from the seminal work by Lee and Messerschmitt [10]. Hence, a complete account of all related contributions is not possible in this paper. The interested reader is referred to [2] for an excellent, in-depth survey of the topic. Here we focus on categorizing the two main classes of approaches taken in the past, summarizing state-of-the-art and putting them in perspective with our work.

The first class of approaches, pioneered by the group lead by E. Lee [11] and extensively explored by many other researchers [2], can be summarized

as follows. A SDFG specification is first checked for consistency, and its non-null repetition vector is computed. The SDFG is then transformed, using the algorithm described in [2] into a HSDFG. The HSDFG is then mapped onto the target platform in two phases. First, an allocation of HSDFG nodes onto processors is computed, then a static-order schedule is found for each processor. The overall goal is to maximize throughput, given platform constraints. Unfortunately, throughput depends on *both* allocation and scheduling, but in this approach scheduling decisions are postponed to a second phase. Hence, an approximate cost function is used to drive allocation decisions: for instance, a bin-packing heuristic aiming at balancing processor workload is proposed in [12]. After allocation, scheduling is reduced to a set of single-processor actor ordering decisions. Scheduler implementation issues mandate for static orders, which are decided off-line. This is however not an easy problem to solve either, as throughput depends on the order of execution of actors and we have an exponential blowup of the solution space. Incomplete search strategies are used, such as list scheduling, driven by a priority function related to throughput. For instance, high priority can be given to actors which belong to long cycles in the HSDFG and therefore have most probably more impact on the throughput [2].

The second class of approaches computes mapping and scheduling directly on the SDFG, without a preliminary HSDFG transformation [13]. This has the main advantage to avoid potential blow-up in the number of actors, which can happen for SDFGs with highly un-balanced rates with a large minimum common multiple. Unfortunately, there is no known way to analytically compute throughput for a generic SDFG with scheduling and allocation, hence these approaches resort to heuristic cost functions to generate promising solutions, and then compute the actual throughput by performing state-space exploration on the SDFG with allocation end scheduling information until a fixed point is reached [14]. This process is quite time consuming. Furthermore, even though the throughput computation via state-space exploration is accurate, there is no guarantee that the solutions generated by the heuristic search are optimal.

The incomplete approaches summarized above obviously cannot give any proof of optimality. Our work aims at addressing this limitation, and it proposes a complete search strategy which can compute max-throughput mappings for realistic-size instances. Our starting point is a HSDFG, which can be obtained from a SDFG by a pseudo-polynomial transformation. The analysis of complete search strategies starting directly from a generic SDFG, without HSDFG transformation, will be subject of future work.

4 Scheduling and Allocation as a Constraint Problem

Due to the cyclic nature of the problem, a scheduling approach deciding the starting time of optional activities running on alternative unary resources has to cope with the transition phase which always appears at execution time before the application becomes periodic (enabling throughput computation). A classical solution is to schedule over time several iterations of the HSDFG until

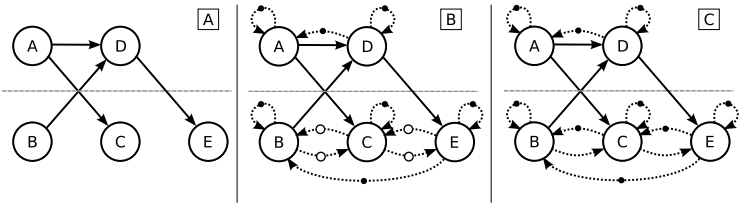


Fig. 2. The HSDFG corresponding to the SDFG of figure

the periodic phase is reached, with a possibly drastical increase of the number of actors. We therefore opted for an alternative approach, somehow similar to Precedence Constraint Posting techniques [20], and devised an order-based, rather than time-based model. Our approach relates to [13] in that the basic idea is to model the effects of design choices by means of modifications to the graph. Consider for example the HSDFG in figure 2A, where all rates are assumed to be one and actors A,D are mapped on one processor, while actors B, C and E on another. From the fact that no more than one task can execute on a processor at the same time, it also follows that two instances of the same task cannot run simultaneously: this can be modeled by adding an auto-cycle to each actor, as depicted in figure 2B where the added arcs are dotted.

Moreover, actors mapped on the same processor must be mutually exclusive: this is captured by adding arcs to create a cycle between each pair of non dependent tasks with the same mapping; note this requires adding one arc for the (A, D) and (B, E) pairs in figure 2B and two arcs for the (B, C) and(C, E) pairs. In order to avoid deadlocks a token must be placed for each of these cycles: while the choice is forced for the (A, D) and (B, E) pair, both arcs are suitable for (B, C), (C, E), as hinted by the empty circles in figure 2B. Choosing the placement of those tokens amounts to take scheduling decisions: for example in figure 2C the order B, C, E was chosen. Note the presence of the edge (B,E) is not necessary, since the path from B to E (B,D,E) implies that the execution of actor E depends on the execution of B.

The main advantage with this approach is that a standard throughput computation algorithm can be used (almost) off the shelf to evaluate the effect of each design decision. We therefore devised a two layer CP model; on one level the model features two sets of decision variables, respectively representing the processor each actor is mapped to and the scheduling/ordering decisions, and on another level a set of graph description variables. Let n be the number of actors in the input HSDFG and let p be the number of processor in the platform, then the decision variables are:

$$\forall i = 0 \ldots n-1 : \texttt{PROC}_\texttt{i} \in [0..p-1] \qquad \forall i = 0 \ldots n-1 : \texttt{NEXT}_\texttt{i} \in [-1..n-1]$$

whereas the (dynamically changing) graph structure is described via a matrix of binary variables $\texttt{Y}_{\texttt{ij}} \in [0,1]$ such that $\texttt{Y}_{\texttt{ij}} = 1$ iff an arc from a_i to a_j exists. Note that the token positioning is implicitly defined by the $\texttt{NEXT}_\texttt{i}$ variables and is built on-line only at throughput computation time. The value of $\texttt{NEXT}_\texttt{i}$

defines the successor of the actor i; the negative value means the lack of a successor. These variables are connected to allocation decisions and to dependencies between actors.

Existing arcs in the input HSDFG result in some pre-filling of the $\mathtt{Y}$ matrix, such that $\mathtt{Y_{ij}} = 1$ for each arc $(a_i, a_j, 1, 1, tok)$ in the original graph. Channeling constraints link decision and graph description variables; in particular, as for the $\mathtt{PROC_i}$ variables, the relation depends on whether a path with no tokens exists in the original graph between two nodes a_i, a_j. Let us write $a_i \prec a_j$ if such path exists; then, if $i \neq j$ and neither $a_i \prec a_j$ nor $a_j \prec a_i$:

$$\mathtt{PROC_i} = \mathtt{PROC_j} \Rightarrow \mathtt{Y_{ij}} + \mathtt{Y_{ji}} = 2 \tag{1}$$

Constraint (1) forces two arcs to be added, if two independent nodes are mapped to the same processor (e.g. nodes B and C in figure 2B). If instead there is a path from a_i to a_j ($a_i \prec a_j$), then the following constraint is posted:

$$\left[(\mathtt{PROC_i} = \mathtt{PROC_j}) \wedge \left(\sum_{a_k \prec a_i} (\mathtt{PROC_k} = \mathtt{PROC_i}) = 0 \right) \wedge \left(\sum_{a_j \prec a_k} (\mathtt{PROC_k} = \mathtt{PROC_j}) = 0 \right) \right] \Rightarrow \mathtt{Y_{ji}} = 1$$

The above constraint completes dependency cycles: considering only tasks on the same processor (first element in the constraint condition), if there is no task before a_i in the original graph (second element) and there is no task after a_j in the original graph (third element), then close the loop, by adding an arc from a_j to a_i. Finally, auto-cycles can be added to each node in a pre-processing step and are not considered here.

The $\mathtt{NEXT}$ variables do not affect the graph description matrix; a number of constraints, however, are used to ensure their semantic consistence. In first place, dependencies in the input SDFG cannot be violated: thus $a_i \prec a_j \Rightarrow \mathtt{NEXT_j} \neq i$. Less intuitively, the presence of an arc $(a_i, a_j, 1, 1, tok)$ with $tok = 1$ in an HSDFG implies a_i to fire always *after* a_j (e.g. A_2 and A_1 in fig. 2) , and therefore, $\mathtt{NEXT_i} \neq j$.

No two nodes can have the same $\mathtt{NEXT}$ on the same processor: $\mathtt{PROC_i} = \mathtt{PROC_j} \Rightarrow \mathtt{NEXT_i} \neq \mathtt{NEXT_j}$. Then, a node a_j can be next of a_i only if they are on the same processor: $\mathtt{PROC_i} \neq \mathtt{PROC_j} \Rightarrow \mathtt{NEXT_i} \neq j$. The -1 value is given to the last node of each (non empty) processor:

$$\forall proc : \sum_{i=0}^{n-1} (\mathtt{PROC_i} = proc) > 0 \Rightarrow \sum_{i=0}^{n-1} [(\mathtt{PROC_i} = proc) \times (\mathtt{NEXT_i} = -1)] = 1$$

Finally, transitive closure on the actors running on a single processor is kept by posting a `nocycle` constraint [19] on the related $\mathtt{NEXT}$ variables. Standard tree search is used with minimum size domain as variable selection heuristics. Symmetry due to homogeneous processors are broken at search time.

4.1 Throughput Constraint

The relation between decision variables and the throughput value is captured in the proposed model by means of a novel throughput constraint, with signature:

$$th\_cst(\mathtt{TPUT}, [\mathtt{PROC_i}], [\mathtt{NEXT_i}], [\mathtt{Y_{ij}}], \mathcal{W}, \mathcal{T})$$

where TPUT is a real valued variable representing the throughput, $[\texttt{PROC}_\texttt{i}]$, $[\texttt{NEXT}_\texttt{i}]$ and $[\texttt{Y}_\texttt{ij}]$ are defined as above, $\mathcal{W}$ is a vector of real values such that $\mathcal{W}_i$ is the computation time of actor a_i and $\mathcal{T}$ is a matrix such that $\mathcal{T}_{ij}$ is the number of initial tokens *tok* on the arc from a_i to a_j. Clearly $\mathcal{T}_{ij} > 0$ implies $\texttt{Y}_\texttt{ij} = \texttt{1}$; note that in this paper we assume $\mathcal{T}_{ij} = [0, 1]$: this is usually true for HSDFG resulting from conversion of an original SDFG.

We devised a filtering algorithm consistently updating an upper bound on TPUT (this is sufficient for a throughput maximization problem) and performing what-if propagation on the NEXT and PROC variables. The filtering algorithm relies on throughput computation inspired from the algorithms described in [15] and [16], which in turn are based on Karp's algorithm (1978). The description is organized in three steps: steps 1 and 2 describe how to build a HSDFG based on current search state to obtain a throughput bound, while step 3 focuses on the computation algorithm.

Step 1 - building the input graph: The input for the throughput computation is a "minimal" graph built by adding arcs to the original HSDFG based on current state of the model. More precisely, an arc is assumed to exist between actors a_i and a_j iff $\texttt{Y}_\texttt{ij} = \texttt{1}$; unbound Y variables are therefore treated as if they were set to 0. Note that the computation of a lower bound for the throughput would require arcs for unbound Y variables as well, thus providing a "maximal" graph.

Step 2 - Token positioning: Next we construct a dependency graph DG with the same nodes as the original HSDF graph G, and such that an arc (a_i, a_j) exists in DG iff either an arc $(a_i, a_j, 1, 1, 0)$ exists in G (detected since $\texttt{Y}_\texttt{ij} = 1$ and $\mathcal{T}_{ij} = 0$) or $\texttt{NEXT}_\texttt{i} = j$. A token matrix TK is then built, according to the following rules:

$$\texttt{Y}_{ij} = 0 \Rightarrow TK_{ij} = 0 \qquad\qquad \texttt{Y}_{ij} = 1 \Rightarrow \begin{cases} TK_{ij} = 0 \text{ if } a_i \prec^{DG} a_j \\ TK_{ij} = 1 \text{ otherwise} \end{cases}$$

where we write $a_i \prec^{DG} a_j$ if there is path from a_i to a_j in DG. This matrix describes the position of the tokens on the new graph G used for the bound computation. The above rules ensure the number of tokens is over-estimated, until all NEXT and PROC are fixed. In the actual implementation, the dependency check is performed without building any graph, while the token matrix TK is actually

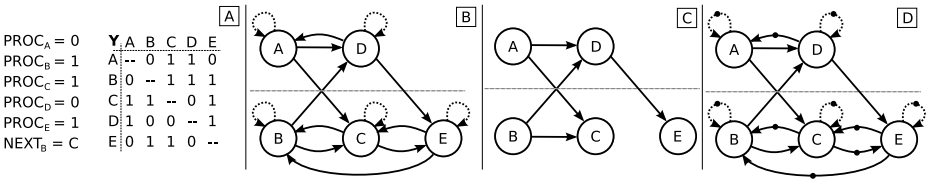


Fig. 3. HSDF graph structure and token positioning for throughput computation during the search; note the number of tokens between actors C and E is overestimated

stored in the constraint. Note that considering a suitable under-estimation of the number of token would be required to compute a throughput lower bound. Figure 3A shows some assignments of decision variables for the HSDFG in figure 2A (remaining variables are considered unbound); figure 3B shows the corresponding arc structure (without autocycles), figure 3C the extracted dependency graph DG and figure 3D the derived token positioning.

Step 2: Throughput computation: For a HSDFG, the throughput equals the inverse of a quantity known as the iteration period of the graph and denoted as $\lambda(HSDFG)$; formally:

$$\frac{1}{th} = \lambda(HSDFG) = \max_{C \in HSDFG} \frac{W(C)}{T(C)}$$

where C is a cycle in the HSDFG, $W(C) = \sum_{a_i \in C} \mathcal{W}_i$ is the sum of the execution time of all actors in C and $T(C) = \sum_{(a_i,a_j) \in C} TK_{ij}$ is the total number of tokens on the arcs of C. Intuitively, $T(C)$ can be thought of as the amount of concurrency which can be exploited on the cycle. In [15] it is shown how to compute the iteration period as the *maximum cycle mean* of an opportunely derived *delay graph*; Karp's algorithm is used for the computation. With some tricks, we show that a maximum cycle mean algorithm can be used to compute the iteration period directly on a HSDFG.

The basic idea is that, according to Karp's theorem, the critical loop constraining the iteration period can be found by analyzing cycles on the worst case k-arcs paths (e.g. the longest ones) starting from an arbitrary source. Since no cycle can involve more than n nodes, considering k-arcs paths with k up to n is sufficient. Our algorithm shares most of its structure with that proposed in [16]. Worst case paths are stored in a $(n+1) \times n$ table D; the element on level (row) k and related to node a_i is referred to as $D_k(a_i)$ and contains the length of the k-arcs path P from the source to a_i maximizing the quantity $W(P)/T(P)$ (if any such path exists). Other than D, the algorithm also employs two equally

Algorithm 1. Throughput computation - build D table

```
1: set all D_k(a_i) = −∞, π_k(a_i) = NIL, τ_k(a_i) = 0
2: V_c = {a_0}
3: V_n = ∅
4: D_0(a_0) = 0
5: for k = 1 to n do
6:     for a_i ∈ V do
7:         for a_j ∈ A+(a_i) do
8:             V_n = V_n ∪ {a_j}
9:             define TP = max(1, τ_k(a_i) + TK_ij)
10:            define WP = D_k(a_i) + W_j
11:            if W(P)/T(P) > D_k(a_j)/τ_k(a_j) then
12:                D_k(a_j) = WP, τ_k(a_j) = TP, π_k(a_j) = i
13:            end if
14:        end for
15:    end for
16:    find loops on level k
17:    V_c = V_n, V_n = ∅
18: end for
```

sized tables π and τ, respectively storing the predecessor $\pi_k(a_i)$ of node a_i and the number of tokens $\tau_k(a_i)$ on the paths.

Pseudo code for the throughput computation is reported in Algorithm 1, where $A^+(a_i)$ denotes the set of direct successors of a_i. Once the table is initialized (line 1), an arbitrary source node is chosen; in the current implementation this is the node with minimum index (a_0). Note that the choice has no influence on the method correctness, but a strong impact on its performance, hence the introduction of a suitable heuristic will be thoroughly considered in the future.

Next, the procedure is primed by setting $D_0(a_0)$ to 0 (line 4) and adding a_0 to the list of nodes to visit V (line 3); then we start to fill levels 1 to n one by one (lines 6 to 18). For each node in V each successor a_j is considered (lines 6, 7), and, if necessary, the corresponding cells in D, π, τ are updated to store the k-arcs path from a_0 to a_j (lines 9 to 12); in case of an equality at line 11 the number of tokens is considered. Once a level is filled, loops are detected as described in Algorithm 2 and then we move to the next k value (line 17). A single

Algorithm 2. Throughput computation - finding loops

1: let k be the starting level on the D table
2: let Vbe the set of nodes to visit on level k
3: **for** $a_i \in V$ **do**
4: define $k' = k, a' = a_i$
5: define $WC = \mathcal{W}_i, TC = 0$
6: **repeat**
7: define $h =$ index of a'
8: $TC = \max(1, TC + TK_{\pi_{k'}(a'),h})$
9: $WC = WC + \mathcal{W}_{\pi_{k'}(a')}$
10: **if** $\frac{WC}{TC} > \frac{WC^*}{TC^*}$ **then**
11: $WC = WC^*,\ TC = TC^*$
12: TPUT $\leq \frac{WC^*}{TC^*}$
13: **end if**
14: $a' = a_{\pi_{k'}(a')},\ k = k - 1$
15: **until** $a' = a_i \vee a' = NIL$
16: **end for**

STEP 0: D,π,τ

	A	B	C	D	E
0	0/-/0	-/-/0	-/-/0	-/-/0	-/-/0
1	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0
2	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0
3	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0
4	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0
5	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0

STEP 1: D,π,τ

	A	B	C	D	E
0	0/-/0	-/-/0	-/-/0	-/-/0	-/-/0
1	-/-/0	-/-/0	1/A/0	1/A/0	-/0/0
2	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0
3	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0
4	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0
5	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0

STEP 2: D,π,τ

	A	B	C	D	E
0	0/-/0	-/-/0	-/-/0	-/-/0	-/-/0
1	-/-/0	-/-/0	1/A/0	1/A/0	-/0/0
2	2/D/1	2/C/1	-/-/0	-/-/0	2/D/0
3	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0
4	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0
5	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0

STEP 3: D,π,τ

	A	B	C	D	E
0	0/-/0	-/-/0	-/-/0	-/-/0	-/-/0
1	-/-/0	-/-/0	1/A/0	1/A/0	-/0/0
2	2/D/1	2/C/1	-/-/0	-/-/0	2/D/0
3	-/-/0	3/E/1	3/A/1	3/A/1	-/-/0
4	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0
5	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0

STEP 4: D,π,τ

	A	B	C	D	E
0	0/-/0	-/-/0	-/-/0	-/-/0	-/-/0
1	-/-/0	-/-/0	1/A/0	1/A/0	-/0/0
2	2/D/1	2/C/1	-/-/0	-/-/0	2/D/0
3	-/-/0	3/E/1	3/A/1	3/A/1	-/-/0
4	4/D/2	4/C/2	4/B/1	4/B/1	4/C/2
5	-/-/0	-/-/0	-/-/0	-/-/0	-/-/0

STEP 5: D,π,τ

	A	B	C	D	E
0	0/-/0	-/-/0	-/-/0	-/-/0	-/-/0
1	-/-/0	-/-/0	1/A/0	1/A/0	-/0/0
2	2/D/1	2/C/1	-/-/0	-/-/0	2/D/0
3	-/-/0	3/E/1	3/A/1	3/A/1	-/-/0
4	4/D/2	4/C/2	4/B/1	4/B/1	4/C/2
5	5/D/2	5/C/2	5/A/2	5/A/2	5/D/1

Fig. 4. Example of table filling (Algorithm 1)

iteration of the algorithm is sufficient to compute the throughput of a strictly connected graph; otherwise, the process is repeated starting from the first never touched node, until no such node exists.

Figure 4 shows the value of all tables at each step when Algorithm 1 is executed on the graph of figure 3; all execution times are assumed to be 1. As an example, consider the transition between step 1 and 2 in the figure: at level 1 the set V_c of the nodes to be visited contains actors C and D. When C is visited all its direct successors are processed: when moving from C to B we traverse one more node ($WP = D_1(C) + \mathcal{W}_B = 1 + 1$) and collect one more token ($TP = \tau_1(C) + 1 = 0 + 1$), hence we set $D_2(B) = WP = 2$, $\tau_2(B) = TP = 1$ and the predecessor $\pi_2(B)$ becomes C; similarly, when processing the arc from C to E, we set $D_2(E) = 2$, $\tau_2(E) = 1$, $\pi_2(E) = C$. Next, actor D is visited; when processing the arc from D to A we set $D_2(A) = 2$, $\tau_2(A) = 1$, $\pi_2(A) = D$; the second outgoing arc of D ends in the already visited node E: for this path $WP = 2$ and $TP = \max(1, \tau_1(D) + 0) = 1$, but no token is collected; for this reason, even if the ratio WP/TP has the same value as $D_2(E)/\tau_2(E)$, we set D as the predecessor of E and $D_2(E) = 2$, $\tau_2(E) = 0$.

The loop finding procedure (Algorithm 2) is started at a specific level (let this be k). From each actor a_i on level k to be visited, the algorithm moves backward along the predecessor chain ($\pi_{k'}(a')$ is the predecessor of current node) collecting execution time of every node met along the path (lines 8, 9). When a second occurrence of the starting node a_i is detected ($a' = a_i$ in line 14) a cycle is found. If this loop constrains the iteration period more the last one found so far (line 10), this is set as critical cycle (WC^*, TC^* — initially $WC^* = 0, TC^* = 1$) and pruning is performed (line 12). The algorithm also stops when the start of D is reached ($a' = NIL$ in line 14).

Algorithm 2 is executed for each value of k during the throughput computation. For instance, with regard to figure 4, at step 5 the procedure finds the loops (stated backward): "$A \leftarrow D \leftarrow B \leftarrow E \leftarrow D/2$ tokens", "$B \leftarrow C/1$ token", "$D \leftarrow A/1$ token", "$E \leftarrow D \leftarrow B/1$ token". No loop is found at this level starting from C. Loop $E \leftarrow D \leftarrow B$ is critical and sets the iteration bound to 3; the computed throughput is therefore 1/3, which is a valid upper bound for the throughput of the original HSDFG. Note that by finding new loops the upper bound always decreases; hence, if at any step a cycle is found such that the resulting throughput is lower the than the minimum value of the `TPUT` variable, then the constraint fails. Moreover, it could be proven that no more than 1 token can be collected by traversing a sequence of nodes on a single processor: the filtering algorithm exploits this property to improve the computed bound at early stages of the search, where the number of tokens is strongly overestimated.

5 Experimental Results

The system described so far was implemented on top of ILOG Solver 6.3 and tested on graphs belonging to three distinct classes of HSDFGs, built by means of the generator provided in the SDF3 framework [17]. Graph classes include

in first place cyclic, connected graphs, which commonly arise when modeling streaming applications. Those are probably the most interesting instances, as they also reflect the typical structure of homogeneous graphs resulting from the conversion of a SDFG. Note the throughput for a cyclic graph is intrinsically bounded by the heaviest cycle, no matter how many processor it is mapped to.

Furthermore, acyclic and strictly connected HSDFGs were considered. Acyclic graphs expose the highest parallelism: this makes them the most challenging instances, but also lets the solver achieve the best throughput values, as no intrinsic bound is present (beside the computation time of the heaviest actor). The class of strictly connected graphs is interesting as this is the type of graph the Maximum Cycle Mean computation algorithms (such as the one we use) were originally designed for; for this reason, the solver is expected to have the best run time on this class of instances.

For each class, groups of 6 graphs with 10, 15, 20, 25 nodes were generated and tested on platforms with different number of processors (2, 4, 8). An iterative testing process was adopted: first the mapper and scheduler is run with a very loose throughput requirement; whenever a solution is returned, this is stored and the throughput value is used as a lower bound for the next iteration. When the problem becomes infeasible, the last solution found is the optimal one. A time limit of 20 minutes is set for each of the iterations; all tests were run on a Core2Duo machine with 1GB of RAM.

Table 1 reports results for the first class of graphs on all types of platform; the number of processors is reported as the first column ('proc'). For each group of 6 instances with the same number of nodes the table shows in column 'T > TL' the number of timed out instances (those for which no solution at all could be found within the time limit), and statistics about the solution time. In detail, the average running time and number of fails for an iteration are reported ('T(all)' and 'F(all)'), together with the average time to solve the iteration when the throughput constraint is the tightest one ('T(opt)'), to show how the method performs in a very constrained condition, when a heuristic approach is likely to fail; the number of instances not considered in the averages is always shown between round brackets. Finally, the average time for the fastest ('T(best)') and the slowest ('T(worst)') iteration give an intuition of the variability on the running time, which is indeed considerably high. A triple dash "---" is shown

Table 1. Results for cyclic, connected graphs - times are in seconds

proc	nodes	T(opt)	T(all)	F(all)	T(best)	T(worst)	T > TL
2	10	0.03	0.04	53	0.02	0.05	0
	15	0.30(1)	5.27	3658	0.17	27.97	1
	20	2.52(2)	28.37(2)	4374(2)	0.63(2)	86.44(2)	2
	25	---	---	---	8.33	78.65	3
4	10	0.03	0.05	46	0.03	0.07	0
	15	1.13(1)	4.45(1)	3303	0.21	25.03	1
	20	1.94(2)	21.60(2)	1808(2)	0.64(2)	32.17(2)	2
	25	---	---	---	8.62	228.86	3
8	10	0.04	0.06	47	0.03	0.09	0
	15	0.25(1)	4.91(1)	3195	0.20	29.38	1
	20	6.65(2)	26.13(2)	2222(2)	0.72(2)	42.12(2)	2
	25	---	---	---	9.83	264.98	3

Table 2. Results for acyclic graphs - times are in seconds

proc	nodes	T(opt)	T(all)	F(all)	T(best)	T(worst)	T > TL
2	10	0.05	0.13	234	0.06	0.58	0
	15	0.11	6.75	9263	0.09	22.52	3
	20	---	---	---	---	---	6
	25	---	---	---	---	---	6
4	10	0.03	0.11	187	0.02	0.58	0
	15	0.09	4.94	6450	0.07	24.17	3
	20	---	---	---	---	---	6
	25	---	---	---	---	---	6
8	10	0.09	0.11	161	0.03	0.64	0
	15	0.11	4.63	5712	0.09	24.60	3
	20	---	---	---	---	---	6
	25	---	---	---	---	---	6

Table 3. Results for strictly connected graphs - times are in seconds

proc	nodes	T(opt)	T(all)	F(all)	T(best)	T(worst)	T > TL
2	10	0.03	0.04	12	0.03	0.05	0
	15	1.38	0.82	430	0.10	1.58	0
	20	1.49(4)	35.17(4)	4474(4)	0.48(1)	112.47(1)	0
	25	---	---	348(4)	1.17(4)	---	1
4	10	0.03	0.04	29	0.03	0.04	0
	15	0.11	0.14	40	0.10	0.23	0
	20	0.89(4)	30.25(4)	3578(4)	0.43(1)	122.88(1)	0
	25	---	---	349(4)	1.27(1)	---	1
8	10	0.14	0.16	177	0.03	0.30	0
	15	0.13	0.17	42	0.12	0.28	0
	20	---	---	---	0.50(1)	303.87(1)	0
	25	---	---	352(4)	1.22(1)	---	1

when the available data are not sufficient for the average to be meaningful. All time values are in seconds.

As expected, the time to solve the most constrained iteration and the average running time grows exponentially with the size of the instances. However, the solution time is reasonable for realistic size graphs, counting 10 to 20 nodes. Propagation for the throughput constraint often takes around 50% of the overall process time, pushing for the introduction of caching and incremental computation in the filtering algorithm. The best case and worst case behavior is quite erratic, as it is quite common for CP/pure tree search based approaches. Finally, it is worth to point how, unlike in usual allocation and scheduling problems, the number of processors appears to have no strong impact on the problem hardness.

Tables 2 and 3 show the same data respectively for acyclic ad strictly connected graphs. Acyclic graphs are indeed very hard to cope with, yielding timeouts for all the 20 and 25 nodes instances (regardless of the number of processors): this will require further investigation. Conversely, strictly connected graphs are the most easily tackled; despite the average run time is often a little higher than table 1, the number of timed out instances is the lowest among the three classes. As previously pointed out, this was somehow expected.

Finally, figure 5 shows the run time (in logarithmic scale) of all iterations of the testing process for a sample instance; as the iteration number grows, the throughput constraint becomes tighter and tighter. The observed trend is

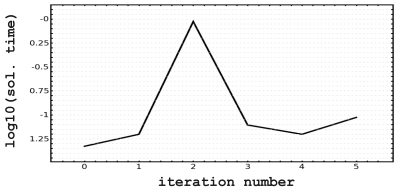


Fig. 5. Log. runtime/iteration number of the testing process for a sample instance

common to many of the instances used for the tests and features a sharp complexity peak in the transition phase between a loosely and a tightly constrained problem. This is an appealing peculiarity, since the complexity peak is located where a heuristic approach would likely perform quite well and could be used instead of the complete method, which on the other hand becomes more effective right in the (very constrained) region where it is also more useful.

6 Conclusions

We presented a CP based method for allocating and scheduling HSDFGs on multiprocessor platforms; to the best of our knowledge this is the first complete approach for the target problem. The core of the system is a novel throughput constraint embedding a maximum cycle mean computation procedure, which proved to be crucial for the performance, but very time consuming as well. This sets the need for strong optimization of the filtering algorithm and for the introduction of caching and incremental computation. Also, a revision of the throughput constraint aimed to improve its usability is planned, in order to make it more easily applicable to other problems as well.

The approach has interesting run time for classes of realistic size instances. Sometimes however the conversion of a SDFG into a HSDFG leads to a blow up of the number of nodes; if the graph becomes too large the approach is not likely to work out. A method to partly overcome the problem is to avoid the conversion and compute throughput directly on the SDFG; for example in [13] this is done by simulation. Integration of such a technique in the throughput constraint is another interesting topic for future research.

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

References

1. Lee, E., Messerschmitt, D.: Synchronous dataflow. Proceedings of the IEEE 75(9), 1235–1245 (1987)
2. Sriram, S., Bhattacharyya, S.: Embedded Multiprocessors Scheduling and Synchronization. Marcel Dekker, Inc., New York (2000)

3. Muller, M.: Embedded Processing at the Heart of Life and Style. In: IEEE ISSCC 2008, pp. 32–37 (2008)
4. Jerraya, A., et al.: Roundtable: Envisioning the Future for Multiprocessor SoC. IEEE Design & Test of Computers 24(2), 174–183 (2007)
5. Pham, D., et al.: Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. IEEE Journal of Solid-State Circuits 41(1), 179–196 (1996)
6. Paganini, M.: Nomadik: A Mobile Multimedia Application Processor Platform. In: IEEE ASP-DAC 2007, pp. 749–750 (2007)
7. Bell, S., et al.: TILE64 Processor: A 64-Core SoC with Mesh Interconnect. In: ISSCC 2008, pp. 88–598 (2008)
8. Lee, A., Bhattacharyya, S.S., Murthy, K.: Software synthesis from data flow graphs. Kluwer Academic Publishers, Dordrecht (1996)
9. Moreira, O., Poplavko, P., Pastrnak, M., Mesman, B., Mol, J.D., Stuijk, S., Gheoghita, V., Bekooij, M., Hoes, R., van Meerbergen, J.: Data Flow analysis for real-time embedded multiprocessor system design. In: Dynamic and robust streaming in and between connected consumer-electronic devices. Springer, Heidelberg (2005)
10. Lee, E.A., Messerschmitt, D.C.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. on Computers (Feburary 1987)
11. Sriram, S., Lee, E.: Determining the order of processor transactions in statically scheduled multiprocessors. Journal of VLSI Signal Processing 15, 207–220 (1996)
12. Moreira, O., Mol, J.D., Bekooij, M., van Meerbergen, J.: Multiprocessor Resource Allocation for Hard-Real Time Streaming with a Dynamic Job-Mix. In: IEEE RTAS 2005, pp. 332–341 (2005)
13. Geilen, M., Stuijk, S., Basten, T., Corporaal, H.: Multiprocessor resource allocation for throughput-constrained synchronous data flow graphs. In: DAC 2007 (2007)
14. Stuijk, S., Basten, T., Moonen, A., Bekooij, M., Theelen, B., Mousavi, M., Ghamarian, A., Geilen, M.: Throughput analysis of synchronous data flow graphs. Application of Concurrency to System Design (2006)
15. Ito, K., Parhi, K.K.: Determining the Minimum Iteration Period of an Algorithm. VLSI Signal Processing 11(3), 229–244 (1995)
16. Dasdan, A., Gupta, R.K.: Faster maximum and minimum mean cycle algorithms for system-performance analysis. IEEE Trans. on CAD of Integrated Circuits and Systems 17(10), 889–899 (1998)
17. Stuijk, S., Geilen, M., Basten, T.: Sdf^3: sdf for free. In: Acsd 2006, pp. 276–278 (2006)
18. Goodacre, J., Sloss, A.N.: Parallelism and the ARM instruction set architecture. Computer 38(7), 42–50 (2005)
19. Pesant, G., Gendreau, M., Potvin, J.-Y., Rousseau, J.M.: An exact constraint logic programming algorithm for the travelling salesman problem with time windows. Transportation Science 32, 12–29 (1998)
20. Policella, N., Cesta, A., Oddi, A., Smith, S.F.: From precedence constraint posting to partial order schedules: a csp approach to robust scheduling. AI Commun. 20(3), 163–180 (2007)

A Shortest Path-Based Approach to the Multileaf Collimator Sequencing Problem

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

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

Abstract. The multileaf collimator sequencing problem is an important component in effective cancer treatment delivery. The problem can be formulated as finding a decomposition of an integer matrix into a weighted sequence of binary matrices whose rows satisfy a consecutive ones property. Minimising the cardinality of the decomposition is an important objective and has been shown to be strongly NP-Hard, even for a matrix restricted to a single row. We show that in this latter case it can be solved efficiently as a shortest path problem, giving a simple proof that the one line problem is fixed-parameter tractable in the maximum intensity. This result was obtained recently by [9] with a complex construction. We develop new linear and constraint programming models exploiting this idea. Our approaches significantly improve the best known for the problem, bringing real-world sized problem instances within reach of complete methods.

1 Introduction

Radiation therapy is a treatment modality that uses ionising radiation in the treatment of patients diagnosed with cancer (and occasionally benign disease). Radiation therapy represents one of the main treatments against cancer, with an estimated 60% of cancer patients requiring radiation therapy as a component of their treatment. The aim of radiation therapy is to deliver a precisely measured dose of radiation to a well-defined tumour volume whilst sparing the surrounding normal tissue, achieving an optimum therapeutic ratio. Recent progress in technology and computing science have allowed significant improvement in the planning and delivery of all radiation therapy techniques.

Our primary *objective* is to apply recent advances in constraint programming to multileaf collimator sequencing in intensity-modulated radiotherapy (IMRT). At the core of advanced radiotherapy treatments are hard combinatorial optimisation problems, which are typically computationally intractable (Section 2 and 3). The *contributions* of this paper rely on the insight that the multileaf collimator sequencing problem restricted to a single row can be solved as a shortest path problem. A similar but more general result was obtained recently by [9]. We give a simple proof that the single row problem is fixed-parameter tractable in the maximum intensity of the row (Section 4) and exploit this insight to develop novel linear and constraint programming models (Section 5). These approaches significantly out-perform the best known for the problem, and bring real-world sized instances within reach of complete methods (Section 6).

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 41–55, 2009.

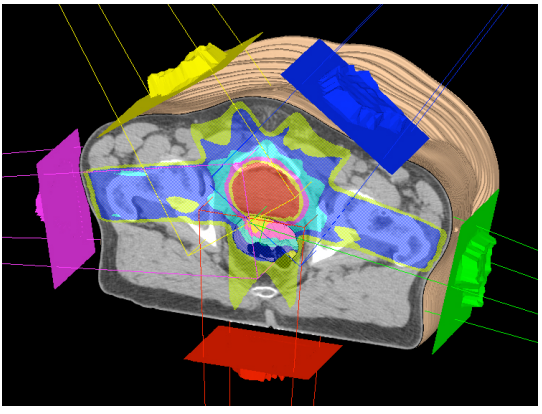
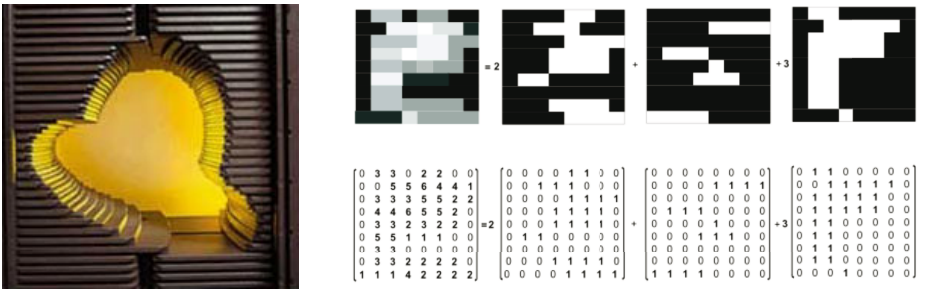


Fig. 1. An example IMRT treatment plan (Courtesy of the Advanced Oncology Center, Inc.)

2 Intensity-Modulated Radiotherapy

IMRT is an advanced mode of high-precision radiotherapy that utilises computer controlled x-ray accelerators to deliver precise radiation doses to a malignant tumour, or specific areas within the tumour. A treatment plan is devised for an individual patient based on the three-dimensional (3D) shape of the patient's tumour. Figure 1 presents an example IMRT treatment plan, clearly showing the location of the tumour in the centre of the image, the positions from which the tumour will be irradiated, and the dosage to be delivered from each position. The treatment plan is carefully developed based on 3D computed tomography images of the patient, in conjunction with computerised dose calculations to determine the dose intensity pattern that will best conform to the tumour shape. There are three optimisation problems relevant to this treatment. Firstly, the *geometry problem* considers the best positions for the beam head from which to irradiate. Secondly, the *intensity problem* is concerned with computing the exact levels of radiation to use in each area of the tumour. Thirdly, the *realisation problem*, tackled in this paper, deals with the delivery of the intensities computed in the intensity problem.

Combinatorial optimisation methods in cancer treatment planning have been reported as early as the 1960s [3]. There is a large literature on the optimisation of IMRT, which has tended to focus on the realisation problem [8]. Most researchers consider the sequencing of multileaf collimators (Figure 2(a)). The typical formulation of this problem considers the dosage plan from a particular position as an integer matrix, in which each integer corresponds to the amount of radiation that must be delivered to a particular region of the tumour. The requisite dosage is built up by focusing the radiation beam using a multileaf collimator, which comprises a double set of metal leaves that close from the outside inwards. Therefore, the collimator constrains the possible set of shapes that can be treated at a particular time. To achieve a desired dosage, a sequence of settings of the multileaf collimator must be used. One such sequence is presented in Figure 2(b). The desired dosage is presented on the left, and it is delivered through a sequence of three settings of the multileaf collimator, which are represented by three



(a) A multileaf collimator. (b) A multileaf sequencing problem.

Fig. 2. A simplified view of the optimisation problem associated with sequencing multileaf collimators in IMRT, Figure 2(b) has been adapted from [1]

matrices. Each matrix is exposed for a specific amount of time, corresponding to the weight associated with the matrix, thus delivering the requisite dosage.

Formally, this problem can be formulated as the decomposition of an integer matrix into a weighted sum of 0/1 matrices, in which each row has the "consecutive ones property" [2,6]. The state-of-the-art approach is based on constraint programming [1].

3 Formulation of the Multileaf Collimator Sequencing Problem

We present a direct formulation of the multileaf collimator sequencing problem. Let I represent the dosage intensity matrix to be delivered. We represent this as an $m \times n$ (rows $\times$ columns) matrix of non-negative integers. We assume that the maximum dosage that is delivered to any region of the tumour is M units of radiation. Therefore, we set $I_{ij} \leq M, 1 \leq i \leq m, 1 \leq j \leq n$.

To ensure that each step in the treatment sequence corresponds to a valid setting of the multileaf collimator, we represent each step using a 0/1 matrix over which we specify a row-wise *consecutive ones* property (C1). Informally, the property requires that if any ones appear in a row, they appear together in a single block. A C1 matrix is a binary matrix in which every row satisfies the consecutive ones property. Formally, x is an $m \times n$ C1 matrix if and only if for any line i, $1 \leq a < b < c \leq n$,

$$x_{ia} = 1 \wedge x_{ic} = 1 \rightarrow x_{ib} = 1. \tag{1}$$

A solution to the problem is a sequence of C1 matrices, Ω, in which each x_k is associated with a positive integer b_k such that: $I = \sum_{k \in \Omega}(b_k \cdot x_k)$. Let B and K be the sum of coefficients b_k and the number of matrices x_k used in the decomposition of I, respectively. Then $B = \sum_{k \in \Omega} b_k$ and $K = |\Omega|$. B is referred to as the total *beam-on time* of the plan and K is its *cardinality*. The typical problem is to minimise B or K independently (known as the decomposition time and decomposition cardinality problem, respectively) or a combination of both. The minimisation of B alone is known to be linear [2,6], while minimizing K alone is strongly NP-Hard [2]. We will tackle the

formulation preferred by practitioners, and mostly used in the literature so far, which is to minimise B first and then K (see Figure 2(b)). The problem is the following: given the optimal value B^* of B, find a treatment plan that minimises K, i.e.

$$\begin{array}{l} Minimise(K) \quad \text{such that} \\ \sum_{k\in\Omega} b_k = B^* \\ I = \sum_{k\in\Omega} b_k x_k \\ \forall k \in \Omega, x_k \text{ is a C1 matrix.} \end{array}$$

We now briefly explain how B^* can be found. The minimum sum of weights needed to have a C1 decomposition of a (single) row matrix $[I_1, \ldots, I_n]$ can be computed as:

$$\sum_{i=0}^{n-1} max(I_{i+1} - I_i, 0) \tag{2}$$

assuming $I_0 = 0$ [2,6]. The expression $I_{i+1} - I_i$ represents the supplementary sum of weights needed for I_{i+1}, the remainder being reused without breaking the consecutive ones property. We provide a small example to help understand Equation 2.

Example 1 (Computing the Minimum Sum of Weights). Consider a dosage plan $I = [3, 2, 0, 3]$. The minimum sum of weights computed according to the previous formula would be $3 + 0 + 0 + 3 = 6$. The weights used to achieve the first value 3 could be reused for the following 2, but all weights already used before the 0 cannot be re-used for the last 3, since all the corresponding row matrices must have a 0 in this position to satisfy the C1 property. Then, three more unit of weights are needed to make the last 3, giving a decomposition $I = (1, 0, 0, 0) + 2 \cdot (1, 1, 0, 0) + 3 \cdot (0, 0, 0, 1)$. ▲

The B^* corresponding to the whole matrix is the maximum of the B^* values amongst the m rows of the matrix. A number of CP models for the multileaf collimator sequencing problem have been proposed in the literature. We briefly present these below.

3.1 The Direct Model

For a fixed K, the problem specification given above can be almost directly encoded with a variable per coefficient of the decomposition and a variable per cell of the matrices in the decomposition, as follows:

$$\begin{array}{lll} Variables: & \forall k \leq K & b_k \in \{1, \ldots, M\} \\ & \forall k \leq K, i \leq m, j \leq n, & x^k_{ij} \in \{0, 1\} \\ \\ DM_1: & & \sum_{k\leq K} b_k = B^* \\ DM_2: & & b_1 \geq b_2, \ldots \geq b_k \\ DM_3: & \forall k \leq K, i \leq m, & \textsc{ConsecutiveOnes}(\{x^k_{i1}, \ldots, x^k_{in}\}) \\ DM_4: & \forall i \leq m, j \leq n & \sum_{k\leq K} b_k \times x^k_{ij} = I_{ij} \end{array}$$

The CONSECUTIVEONES constraint (DM_3) can be implemented using a contiguity constraint [10] or the REGULAR global constraint [12] with a straightforward deterministic finite automaton. Constraint DM_2 eliminates symmetries amongst the weights. A number of symmetries amongst the x^k variables remain. We quote here the example given in [1] to highlight this important drawback of the Direct Model.

Example 2 (Symmetries of the Direct Model). Consider part of a decomposition with two identical weights of value 2. The two following decompositions are symmetrical.

$$2\begin{pmatrix}1&1&1\\1&1&\mathbf{1}\end{pmatrix}+2\begin{pmatrix}0&1&0\\0&1&\mathbf{0}\end{pmatrix}=2\begin{pmatrix}1&1&1\\1&1&\mathbf{0}\end{pmatrix}+2\begin{pmatrix}0&1&0\\0&1&\mathbf{1}\end{pmatrix}$$

The values in the right bottom corners of the matrices can be swapped without changing the solution since the weights are identical. ▲

Symmetries due to identical weights can be partially avoided by dynamically adding lexicographic constraints on the rows and columns (once the weights are known in the search) but that would not be enough. Rows and columns remain lexicographically ordered in this example. The next model was proposed in [1] to address this issue.

3.2 The Counter Model

The Counter Model is based on N_b variables representing the number occurrences of weight b in the decomposition. Q^b_{ij} variables refine this information by counting the number of times a weight b contributes to the sum of I_{ij}. The model is stated as follows:

$$\begin{array}{lll}
Objective: & Minimise(K) \quad \text{such that:} & \\
Variables: & \forall b \leq M & N_b \in \{0,\ldots,B^*\} \\
 & \forall i \leq m, j \leq n, b \leq M & Q^b_{ij} \in \{0,\ldots,M\} \\
 & & \\
CM_1: & \forall i \leq m, j \leq n & \sum_{b=1}^{M} b \times Q^b_{ij} = I_{ij} \\
CM_2: & & \sum_{b=1}^{M} b \times N_b = B^* \\
CM_3: & & \sum_{b=1}^{M} N_b = K \\
CM_4: & \forall b \leq M, i \leq m, & \textsc{SumOfIncrements}(\{Q^b_{i1},\ldots,Q^b_{in}\}, N_b)
\end{array}$$

Constraint $CM1$ ensures that each element of I is properly decomposed. $CM2$ and $CM3$ relate the N_b variables to B^* and K. The model makes use of an ad-hoc global constraint to enforce the C1 property of this decomposition expressed in term of occurrences of each weight. It is the SUMOFINCREMENTS [4], defined by:

$$\textsc{SumOfIncrements}(\{V_1,\ldots,V_n\},U) \equiv \sum_{i=0}^{n-1} max(V_{i+1}-V_i,0) \leq U \text{ with } V_0 = 0 \quad (3)$$

A C1 decomposition of I can be derived from a C1 decomposition of Q^b for each b. The key intuition behind this model is that it is sufficient to find an unweighted decomposition (only with weights of 1) of each Q^b instead of looking for a decomposition of I (see [1]). Keep in mind that Q^b is the matrix defining the number of times a weight equal to b is used to decompose each element of I, thus a weighted decomposition of Q^b would result in identical matrices. The cardinality would, therefore, be reduced by merging the corresponding matrices and increasing the weight. Thus all matrices have to be different and the decomposition of Q^b is necessarily unweighted in an optimal solution.

As explained earlier in Section 3, the expression $\sum_{i=1}^{n} max(V_{i+1}-V_i,0)$ is a convenient way to compute the minimum sum of weights needed for a C1 decomposition of a row. Obviously, if we seek an unweighted decomposition, the formula returns the minimum *number* of weights needed. Therefore, this formula can be used as a lower bound for N_b to ensure the C1 property (constraint CM_4).

4 The Single Row Problem as a Shortest Path

As mentioned previously, finding the minimum total beam-on time, B, for a given intensity matrix can be solved in linear time. However, minimising the cardinality of the multileaf collimator sequence is NP-hard [5]. More recently, it has been shown that even when restricting the problem to a single row of the intensity matrix, minimising the cardinality is strongly NP-Hard [2]. This result was refined by [9] who showed that not only is the single row problem polynomial when the maximum intensity is bounded, but also the complete problem. In this section we show a simple construction representing the single row problem as a shortest path. This gives a simple proof that the single row problem is fixed-parameter tractable (FPT) in the maximum element in the row I. Although [9] achieves a better complexity, we will develop very efficient algorithms based on our construction outperforming the results of [9] in practice. In general, a problem is FPT with respect to a parameter k if there exists an algorithm for it that has running time $\mathcal{O}(f(k) \cdot n^{\mathcal{O}(1)})$, where n is the size of the problem and $f(k)$ is an arbitrary function depending only on k. This result will be the keystone to designing very efficient linear and CP models in the remainder of this paper.

C1 Decomposition Cardinality Problem (DC)
Instance: A row matrix of n integers, $I = \langle I_1, \ldots, I_n \rangle$, a positive integer K.
Question: Find a decomposition of I into at most K C1 row matrices.

In any solution of the DC problem, there must be a subset of the weights of the decomposition that sum to every element I_i of the row. In other words, the decomposition must contain an integer partition of every intensity. To represent these integer partitions the following notation will be used: $P(a)$ is the set of partitions of integer a, $p \in P(a)$ is a particular partition of a, and $|p|$ its number of integer summands in p. We denote by $occ(p, v)$, the number of occurrences of value v in p. For example, $P(5) = \{\langle 5\rangle, \langle 4,1\rangle, \langle 3,2\rangle, \langle 3,1,1\rangle, \langle 2,2,1\rangle, \langle 2,1,1,1\rangle, \langle 1,1,1,1,1\rangle\}$, and if $p = \langle 3,1,1\rangle$ then $|p| = 3$ and $occ(p,1) = 2$.

Observe that the DC problem can be formulated as a shortest path problem in a weighted directed acyclic graph, G, which we refer to a *partition graph*. A partition graph G of a row matrix $I = \langle I_1, \ldots, I_n \rangle$ is a layered graph with $n+2$ layers, the nodes of each layer i corresponding to the set of integer partitions of the row matrix element I_i. A source and sink nodes, denoted p_0 and p_{n+1} respectively, are associated with the empty partition $\emptyset$ for sake of simplicity. Two adjacent layers form a complete bipartite graph and the cost added to an edge, $p_i \rightarrow p_j$, between two partitions, p_i and p_j represents the number of additional weights that need to be added to the decomposition to satisfy the C1 property when decomposing the two consecutive elements with the corresponding partitions. The cost of each edge $p_i \rightarrow p_j$ in the partition graph is:

$$c(p_i, p_j) = \sum_{b=1}^{M} c(b, p_i, p_j) \tag{4}$$

where $c(b, p_i, p_j) = max(occ(b, p_j) - occ(b, p_i), 0)$. A shortest path in the partition graph answers DC.

Example 3 (A Partition Graph). Consider a single row intensity matrix $I = [3, 2, 3, 1]$. The partition graph for this row problem is presented in Figure 3. Excluding the source

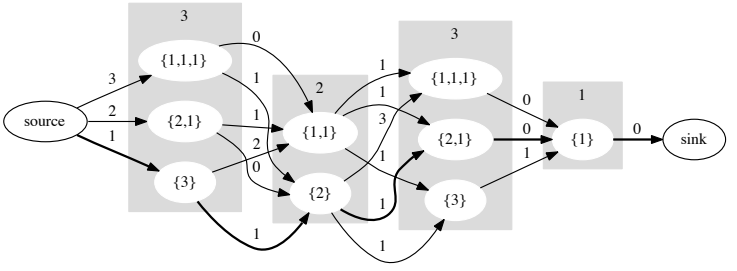


Fig. 3. A partition graph, showing transition weights, for the single row intensity matrix $I = [3, 2, 3, 1]$. A shortest path is indicated in bold.

and sink, there are four levels, one corresponding to each element of I. The costs associated with the edges are computing using Equation 4. For example, the cost associated with the edge between the partition $\langle 1, 1, 1\rangle$ of element 3 of layer 1 and partition $\langle 2\rangle$ of element 2 represents the extra weight that must be added to decompose element 2 if $\langle 1, 1, 1\rangle$ is used for element 3. In other words, as one moves along a path in this graph the partition chosen to decompose the element at layer i contains the only weights that can be reused to decompose the element at layer $i + 1$ because of the C1 property. ▲

This formulation is only a refinement over the Counter Model which gives an easy way to show that this algorithm is correct. Consider a row I of n elements and its partition graph G. A path $\Pi = \langle p_0, \ldots, p_{n+1}\rangle$ in G defines a decomposition of I whose cardinality is the length of the path: $K = \sum_{i=0}^{n} c(p_i, p_{i+1})$. From this path Π, we can build a solution to the Counter Model (without the beam-on time constraint CM_2) by setting $N_b = \sum_{i=0}^{n} c(b, p_i, p_{i+1})$ and $Q_i^b = occ(b, p_i)$. Constraint CM_1 is satisfied as p_i is an integer partition of I_i. Constraint CM_2 is ignored as we are in the case of the unconstrained cardinality. Constraint CM_4 is satisfied because $N_b = \sum_{i=0}^{n} c(b, p_i, p_{i+1}) = \sum_{i=1}^{n} max(Q_{i+1}^b - Q_i^b, 0)$ which is the SUMOFINCREMENTS constraint. Finally, one can check the cardinality of the path (constraint CM_3) by computing the sum of the N_b variables: $\sum_{b=1}^{M} N_b = \sum_{b=1}^{M} \sum_{i=0}^{n} c(b, p_i, p_{i+1}) = \sum_{i=0}^{n} \sum_{b=1}^{M} c(b, p_i, p_{i+1}) = \sum_{i=1}^{n} c(p_i, p_{i+1}) = K$. As a path encodes a solution to the Counter Model, and the length of the path is exactly the cardinality, the shortest path gives the optimal K and an answer to DC. We now consider the single row problem when constraining the beam-on time.

C1 DECOMPOSITION-CARDINALITY WITH TIME CONSTRAINT (DCT)
Instance: A row matrix of n integers, $I = [I_1, \ldots, I_n]$, positive integers K and B.
Question: Find a decomposition of I into at most K C1 row matrices such that the sum of its weights is at most B.

To deal with this problem we extend the previous graph with a resource for every edge:

$$r(p_i, p_j) = \sum_{b=1}^{M} b \times c(b, p_i, p_j). \tag{5}$$

Finding a shortest path $\Pi = \langle p_0, \ldots, p_{n+1}\rangle$ in the partition graph whose sum of weights, $\sum_{i=0}^{n} r(p_i, p_{i+1})$, is at most B is a shortest path problem with resource

constraints (SPPRC). The two-resource SPPRC is better known as the shortest path problem with time windows (SPPTW), which was studied initially by [11]. A single time window $[0, B]$ can be added to the sink node, capturing constraint CM_2 of the Counter Model. The problem is NP-Hard, but pseudo-polynomial algorithms do exist based on dynamic programming. An algorithm of complexity $O(n^2B)$ is given in [11], where n is the number of nodes of the graph.

Example 4 (Encoding DCT as a SPPTW). The new partition graph of $I = [3, 2, 3, 1]$, with a cost and resource consumption per edge is given Figure 4. ▲

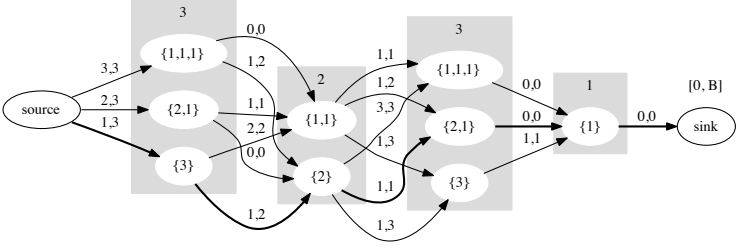


Fig. 4. Encoding an example DCT problem

This formulation of the single row problem corresponds to a simple FPT result.

Theorem 1 (Fixed-Parameter Tractibility of the Single Row Problem). *Finding an optimal solution to the DC and DCT problems is fixed-parameter tractable in the size of the maximum element of the single-row intensity matrix, I.*

Proof. Let k be the maximum element of the row matrix I. The number of edges in the partition graph is bounded by $|P(k)|^2 \times n$ because the number of nodes of a layer i is the number of integer partitions of the corresponding integer value I_i. In this acyclic graph, solving a simple shortest path problem can be done in $\mathcal{O}(|P(k)|^2 \times n)$. The time complexity can be written as $\mathcal{O}(nf(k))$, where $f(k) = |P(k)|^2$, showing that DC is fixed-parameter tractable in k. Similarly for DCT, using the pseudo-polynomial algorithm of [11] we get a complexity of $\mathcal{O}(n^2|P(k)|^2B)$. B can be bounded by nk giving a time complexity of $\mathcal{O}(n^3f(k))$, with $f(k) = k|P(k)|^2$. ■

In [9] a more sophisticated construction for the shortest path is presented giving an $\mathcal{O}(n)$ complexity for DCT whereas our representation as a SPPTW gives a complexity in $\mathcal{O}(n^3)$. In [4] a global constraint, called the SUMOFINCREMENTS was proposed for maintaining the C1 property and a bounds consistency algorithm in $\mathcal{O}(n)$ was given. An $\mathcal{O}(nd)$ arc-consistency algorithm can be obtained based on finding shortest paths.

Corollary 1. *Generalised Arc Consistency on the* SUMOFINCREMENTS *constraint can be done in $\mathcal{O}(nd)$ where n is the number of variables and d the maximum domain size.*

Proof. We recall that SUMOFINCREMENTS($\{V_1, \ldots, V_n\}, U$) is equivalent to the expression $\sum_{i=0}^{n-1} max(V_{i+1} - V_i, 0) \leq U$ with $V_0 = 0$. Consider a layered graph in which each layer corresponds to a variable V_i and each node of layer i corresponds to the values of $D(V_i)$. The cost associated with two values $a \in D(V_i)$ and $b \in D(V_{i+1})$ is simply $max(b - a, 0)$. Consider an instantiation of all the V_i variables. The value of the expression $\sum_{i=0}^{n-1} max(V_{i+1} - V_i, 0)$ is obviously given by the cost of the corresponding path. Ensuring that the SUMOFINCREMENTS is GAC can be easily done by checking that the value of the shortest path in the layered graph is less than the upper bound of U. The shortest path from the source to all nodes and from all nodes to the sink can be obtained in $\mathcal{O}(e)$ where e is the number of edges of the layered graph. Thus, the filtering process can be done in $\mathcal{O}(nd)$ where d is the maximum domain size. ■

5 Shortest Path-Based Models

5.1 A Shortest Path Constraint Programming Model

We index, in lexicographic order, the integer partitions of each element I_{ij} of the intensity matrix, and use an integer variable P_{ij} to denote which partition is used to decompose element I_{ij}. For example, $P(5) = \{\langle 5\rangle, \langle 4,1\rangle, \langle 3,2\rangle, \langle 3,1,1\rangle, \langle 2,2,1\rangle, \langle 2,1,1,1\rangle, \langle 1,1,1,1,1\rangle\}$, so $P_{ij} = 4$ means that the weights $3, 1$ and 1 are used to sum to this element in the decomposition. The domain of P_{ij}, denoted $D(P_{ij})$ thus ranges from 1 to $|P(I_{ij})|$. We also have a variable N_b giving the number of occurrences of weight b in the decomposition, similar to the Counter Model presented earlier.

Our CP model makes use of the SHORTESTPATH($G, \{P_1, \ldots, P_n\}, U$) constraint, which enforces U to be greater than the shortest path in a graph G. Our CP model posts the SHORTESTPATH constraint over three different graphs $G_1(i)$, $G_2(i, b)$, $G_3(i)$, which although topologically identical, are weighted using three different costs:

$$\begin{array}{ll} c_1(p_i, p_j) & = \sum_{b=1}^{M} c_2(b, p_i, p_j) \\ c_2(b, p_i, p_j) & = max(occ(b, p_j) - occ(b, p_i), 0) \\ c_3(p_i, p_j) & = \sum_{b=1}^{M} b \times c_2(b, p_i, p_j) \end{array} \tag{6}$$

Example 5 (Example of the Costs G_1, G_2, G_3). Consider $I = [3, 2, 3, 1]$. The three partition graphs are identical in structure, only the costs vary. G_1, G_2 for value $b = 1$ and G_3 are shown in Figure 5, giving the three costs c_1, c_2, c_3, respectively. ▲

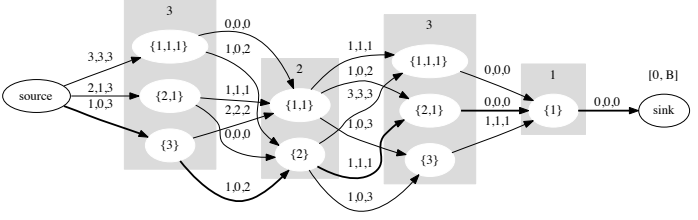


Fig. 5. Example of the three graph costs used in our CP model

Therefore, our CP model is summarised as follows:

$$
\begin{array}{lll}
Objective: Minimise(K) \text{ with } K \in \{0,\dots,B^*\} & & \\
 & \forall b \leq M & N_b \in \{0,\dots,B^*\} \\
 & \forall i \leq m, j \leq n, & P_{ij} \in \{1,\dots,|P(I_{ij})|\} \\
 & & \\
CP_1: & & \sum_{b=1}^{M} b \times N_b = B^* \\
CP_2: & & \sum_{b=1}^{M} N_b = K \\
CP_3: & \forall i \leq m, & \textsc{ShortestPath}(G_1(i), \{P_{i1},\dots,P_{in}\}, K) \\
CP_4: & \forall i \leq m, b \leq M & \textsc{ShortestPath}(G_2(i,b), \{P_{i1},\dots,P_{in}\}, N_b) \\
CP_5: & \forall i \leq m, & \textsc{ShortestPath}(G_3(i), \{P_{i1},\dots,P_{in}\}, B^*) \\
CP_6: & \forall i \leq m, \forall j < m \text{ s.t } I_{ij} = I_{i,j+1} & P_{ij} = P_{i,j+1}
\end{array}
$$

The C1 property of the decomposition is enforced by constraints CP_4. The number of weights of each kind, b, needed so that a C1 decomposition exists for each line i is maintained as a shortest path in $G_2(b,i)$. As those shortest paths are computed independently, maintaining a shortest path in $G_1(i)$ provides a lower bound on the cardinality needed for the decomposition of each line i. This is the purpose of CP_3, which acts as a redundant constraint. Finally CP_5 is a useful redundant shortest path constraint that maintains the minimum value of B associated with each line, which can provide valuable pruning by strengthening CP_1. CP_6 breaks some symmetries by stating that the same partition can be used for two consecutive identical elements in the same row. If the two partitions were different, a solution could be obtained by using any of the two partitions for the two elements. This could not violate the C1 property as the elements are consecutive and any of those two partitions was also satisfying the C1 property.

Filtering the SHORTESTPATH Constraint. The shortest path constraint has already been studied in Constraint Programming [7]. Here, the SHORTESTPATH constraint is simple as the graph is layered and contains only non-negative costs. The constraint $\textsc{ShortestPath}(G, \{P_1,\dots,P_n\}, U)$ states that U is greater than the shortest path in the partition graph defined by the domains of $\{P_1,\dots,P_n\}$ and the cost information G. A layer i of the graph corresponds to variable P_i and the nodes of each layer to the domain values of P_i. Our implementation of the constraint maintains for every node α of layer i, the value of the current shortest path from the source, $S^{\alpha}_{\leftarrow}$, and to the sink, $S^{\alpha}_{\rightarrow}$. These two integers are restorable upon backtracking.

If a value is pruned from a layer we proceed with forward (resp. backward) phases to update the $S_{\leftarrow}$ (resp. $S_{\rightarrow}$) values maintaining the simple following equations :

$$
\begin{array}{l}
S^{\alpha}_{\leftarrow} = \min_{\beta \in D(P_{i-1})}(S^{\beta}_{\leftarrow} + c(\beta,\alpha)) \\
S^{\alpha}_{\rightarrow} = \min_{\beta \in D(P_{i+1})}(S^{\beta}_{\rightarrow} + c(\alpha,\beta))
\end{array}
\tag{7}
$$

The constraint is partially incremental, so if none of the $S_{\leftarrow}$ (resp. $S_{\rightarrow}$) values of the nodes on layer i have been updated, the process stops and does not examine layer $i+1$ (resp. $i-1$). At each update of a $S_{\leftarrow}$ or $S_{\rightarrow}$, we prune the corresponding value if $S_{\leftarrow} + S_{\rightarrow}$ is greater than the upper bound of U. The time complexity of the forward and backward step including the pruning is $\mathcal{O}(e)$ where e is the number of edges in the graph. $S^{sink}_{\leftarrow}$ (or $S^{source}_{\rightarrow}$) is used to update the lower bound of U. As the upper bound of U is not updated, there is no need to reach a fixed point and arc-consistency is achieved

in $\mathcal{O}(e)$. Notice that this constraint could also be decomposed by introducing $S_{\leftarrow}$ and $S_{\rightarrow}$ as variables, stating Equations 7 as constraints as well as $S^{\alpha}_{\leftarrow} + S^{\alpha}_{\rightarrow} > U \implies P_i \neq \alpha$.

*Example 6 (*SHORTESTPATH *using G_1).* Consider $I = [3, 2, 3, 1]$ and $U = 3$. The graph underlying SHORTESTPATH$(G_1, \{P_1, \ldots, P_4\}, U)$ is shown in Figure 6. The two restorable integers $S_{\leftarrow}$ and $S_{\rightarrow}$ are given for each node in brackets. A node filled in grey has been pruned because the sum of its two shortest paths is greater than 3. Values $\{1, 1, 1\}, \{1, 1\}$ and $\{1, 1, 1\}$ are pruned respectively from P_1, P_2 and P_3. ▲

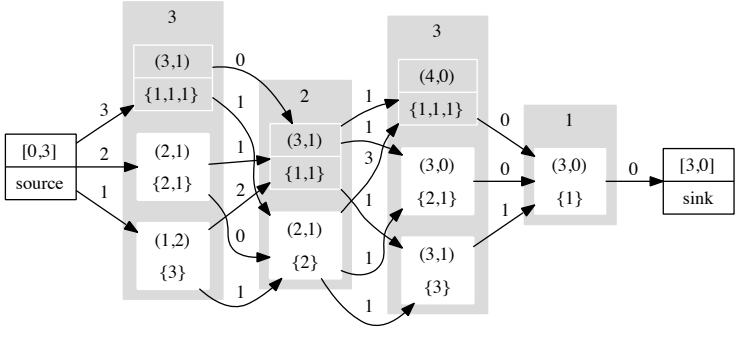


Fig. 6. The SHORTESTPATH using Cost G_1

Note that our CP model is exponential in space as the implementation of the SHORTESTPATH constraint maintains information for each integer partition of the elements of the matrix. Therefore, the model strongly relies on the fact that the maximum intensity in the matrix is bounded in practice and instances with small intensities remain open.

Search. The branching strategy first assign the K variable in a bottom-up fashion (from its lower bound to its upper bound) until a feasible solution is found (the first feasible solution found is thus an optimal one). The branching then considers the N_b variables and proceeds with 'minimum domain first' variable ordering and lexicographic value ordering (from the lower bound to the upper bound of each N_b). Once the N_b variables are known, the problem is split into m independent sub-problems (one per row). Those problems are solved independently by branching on the P variables, again using minimum domain variable ordering and lexicographic value ordering. The rows are examined in decreasing value of their beam on-time, similar to [1]. Branching on P is mandatory, since the shortest paths on $G_2(i, b)$ are maintained independently for each b. At this stage we are facing a multi-resource constrained shortest path problem as we have a limit N_b of each resource b as well as a limit K on the shortest path in G_1.

5.2 A Shortest Path Linear Programming Model

A simple shortest path formulation in Linear Programming (LP) is unimodular, guaranteeing that the continuous relaxation provides an integral solution. We investigated if

encoding the previous model based on shortest path in LP could lead to a strong lower bound for the whole problem. The linear model simply introduces a boolean variable for each possible integer partition of each element I_{ij} of the intensity matrix.

$$\begin{array}{ll} \forall b \leq M & N_b \in \{0, \ldots, B^*\} \\ \forall i \leq m, j \leq n, p \leq |P(I_{ij})| & x_{ijp} \in \{0,1\} \end{array}$$

Again, N_b denotes the number of occurrences of value b in the decomposition of B^*, whereas x_{ijp} indicates whether partition p is used or not to sum to I_{ij}. The consecutive ones property is enforced as a shortest path problem on each line in the partition graphs G_1, G_2 and G_3. The nodes of those graphs are mapped to the x_{ijp} variables and the costs are computed using Equations 6. $x_{i,0,0}$ and $x_{i,n+1,0}$ are two nodes acting as the source and sink of the graph of line i, respectively. A linear model called $SP(i)$ encoding the shortest path problem for each line i uses one variable per edge:

$$\forall j \leq n, p_\alpha \leq |P(I_{ij})|, p_\beta \leq |P(I_{i,j+1})| \;\; y_{i,j,p_\alpha,p_\beta} \in \{0,1\}.$$

The variables y_{i,j,p_α,p_β} indicate whether or not the edge between partition p_α of layer j and partition p_β of layer $j+1$ is used in the solution of line i. The three shortest path constraints introduced in the CP model can be encoded using a simple linear model for shortest path by stating the flow conservation at each node. The following constraints encode the flow conservation, the three costs of the paths and channels the edge variables to the nodes variables, respectively.

$$\begin{array}{ll} \forall j \leq n, p_\alpha \leq |P(I_{ij})|, & \sum_{p_\beta \leq |P(I_{i,j-1})|} y_{i,j-1,p_\beta,p_\alpha} = \sum_{p_\beta \leq |P(I_{i,j+1})|} y_{i,j,p_\alpha,p_\beta} \\ & \sum_{p_\alpha \leq |P(I_{i1})|} y_{i,0,0,p_\alpha} = 1 \\ & \sum_{p_\alpha \leq |P(I_{in})|} y_{i,n,n+1,p_\alpha} = 1 \\ & \sum_{b \leq M} N_b \geq \sum_{j,p_\alpha,p_\beta} c_1(p_\alpha, p_\beta) \times y_{i,j,p_\alpha,p_\beta} \\ \forall b \in [1, \ldots, M] & N_b \geq \sum_{j,p_\alpha,p_\beta} c_2(b, p_\alpha, p_\beta) \times y_{i,j,p_\alpha,p_\beta} \\ & B^* \geq \sum_{j,p_\alpha,p_\beta} c_3(p_\alpha, p_\beta) \times y_{i,j,p_\alpha,p_\beta} \\ \forall j \leq n, p_\alpha \leq |P(I_{ij})|, & x_{ijp_\alpha} = \sum_{p_\beta \leq |P(c_{i,j+1})|} y_{i,j,p_\alpha,p_\beta} \\ \forall j \leq n, p_\alpha \leq |P(I_{ij})|, & x_{ijp_\alpha} = \sum_{p_\beta \leq |P(c_{i,j-1})|} y_{i,j-1,p_\beta,p_\alpha} \end{array}$$

The overall model is written in the following way:

$$\begin{array}{ll} minimise \sum_{b \leq M} N_b & \\ \forall i \leq m & SP(i) \\ & \sum_{b \leq M} b \times N_b = B^* \end{array}$$

The number of variables for this model is exponential as it depends on the number of integer partitions of the maximum element of the matrix, but this is bounded in practice.

6 Experimental Results

We performed a direct comparison between our CP model and the current state-of-the-art [4,1] which showed our approach to be the fastest by more than two orders-of-magnitude, as well as the most scalable. It solves all 340 instances in the benchmark

Table 1. Comparing the Shortest Path Model CPSP with the Counter Model

Inst	CPSP Time (seconds)					Counter model Time (seconds)				
	NS	min	med	avg	max	NS	min	med	avg	max
12-12-10	20	0.11	0.18	0.25	0.66	20	0.32	0.83	1.00	3.62
12-12-11	20	0.14	0.22	0.71	3.32	20	0.67	2.27	2.63	6.23
12-12-12	20	0.23	0.50	0.94	6.94	20	1.04	3.91	4.76	12.30
12-12-13	20	0.28	1.63	1.93	4.77	20	2.26	7.13	8.57	30.50
12-12-14	20	0.35	1.59	3.28	26.36	20	1.19	9.63	11.58	49.76
12-12-15	20	0.61	5.76	12.70	74.59	20	4.37	23.00	40.68	156.23
15-15-10	20	0.13	0.31	0.73	5.67	20	2.51	13.16	14.18	46.14
15-15-12	20	0.41	1.29	3.86	18.20	20	9.02	53.41	105.22	475.95
15-15-15	**20**	1.55	15.45	28.98	102.92	16	111.01	587.73	790.11	3409.69
18-18-10	20	0.24	0.46	1.01	5.88	20	26.22	135.03	183.91	851.34
18-18-12	**20**	0.47	3.07	6.00	19.36	18	121.84	1131.85	1371.88	4534.41
18-18-15	**20**	2.35	20.47	64.85	571.19	6	2553.80	3927.24	3830.12	4776.23
20-20-10	**20**	0.23	0.52	3.99	43.11	19	81.63	660.03	1190.01	3318.89
20-20-12	**20**	0.72	5.10	15.34	83.03	10	666.42	2533.41	3105.34	6139.53
20-20-15	**20**	3.15	61.73	180.70	697.51	0	-	-	-	-
30-30-10	**20**	0.88	2.97	76.77	474.26	0	-	-	-	-
40-40-10	**20**	1.42	11.33	468.19	3533.22	0	-	-	-	-

suite whereas the best known approach can solve only 259 of them[1]. Secondly, we evaluated the quality of the continuous relaxation of our LP model, showing they typically were extremely close to optimal, and demonstrated that it could sometimes be useful for giving lower bounds to the CP model, providing significant speed-up over the CP model alone on large instances.

Evaluation of the CP Model. We compared our CP Shortest Path model (CPSP), from Section 5.1, against the Counter model of [4,1], which is the best known approach to this problem. The same 340 problem instances and an executable binary from [1] were kindly provided by the authors, facilitating a direct and fair comparison. The benchmarks comprised 17 categories of 20 instances ranging in size from 12×12 to 40×40 with maximum elements between 10 and 15, denoted m-n-M in our results tables. Table 1 reports the number of instances solved in each category (column NS), along with the minimum, median, average and maximum time for each category using a time limit of 2 hours on an iMac[2]. Our CPSP approach clearly outperforms the Counter Model as the size grows. On 20-20-10 instances where the Counter Model fails to solve one instance within two hours, the speed-up is almost two orders-of-magnitude. Our CP model is implemented in Choco[3] and Java, whereas the Counter Model is implemented in Mercury[4] and compiled to C. Results in [9] use 15×15 intensity matrices with a maximum element of 10, requiring up to 10 hours to solve using a 2GHz workstation.

Evaluation of the LP Model. Although the IP shortest-path model is not able to compete with CPSP, the continuous relaxation (LP) is very tight and leads to excellent lower bounds, which are often optimal for large instances. Table 2 reports, for each category, the percentage of instances for which the optimal value of the relaxation matches the

[1] Benchmark suite available from http://www.4c.ucc.ie/datasets/imrt

[2] Mac OS X 10.4.11, 2.33 GHz Intel Core 2 Duo, 3 GB 667MHz DDR2 SDRAM.

[3] http://choco.sourceforge.net

[4] http://nicta.com.au/research/projects/constraint_programming_platform

Table 2. The quality and time taken to compute the linear programming relaxation

Inst	%Opt	Avg time	Inst	%Opt	Avg time
12-12-10	95	1.76	18-18-10	100	3.96
12-12-11	85	2.52	18-18-12	95	16.91
12-12-12	95	5.00	18-18-15	100	93.97
12-12-13	95	7.91	20-20-10	100	4.69
12-12-14	95	13.79	20-20-12	90	18.41
12-12-15	60	26.91	20-20-15	95	136.97
15-15-10	95	2.61	30-30-10	95	13.40
15-15-12	85	9.86	40-40-10	100	24.86
15-15-15	85	50.04			

Table 3. Comparing the CP model with and without initial lower bounds from the LP relaxation

Inst	CPSP Time (seconds)					Nodes	Hybrid = LP + CPSP Time (seconds)					Nodes
	NS	min	med	avg	max	avg	NS	min	med	avg	max	avg
12-12-10	20	0.05	0.10	0.14	0.60	125.55	20	1.25	1.79	1.86	2.78	108.10
12-12-11	20	0.07	0.12	0.43	2.25	259.25	20	1.82	2.56	2.80	5.26	201.20
12-12-12	20	0.14	0.32	0.66	5.47	194.65	20	3.24	5.11	5.38	8.94	156.35
12-12-13	20	0.18	1.24	1.46	3.70	250.00	20	4.16	8.02	8.64	15.95	171.80
12-12-14	20	0.23	1.17	2.61	21.16	373.25	20	5.39	14.63	15.40	26.07	298.00
12-12-15	20	0.40	4.64	10.65	63.98	611.85	20	16.39	30.36	35.27	85.61	518.05
15-15-10	20	0.07	0.20	0.51	4.07	301.15	20	1.70	2.54	2.76	5.04	177.45
15-15-12	20	0.25	1.05	3.24	15.75	389.15	20	7.61	10.22	11.66	25.50	289.50
15-15-15	20	1.13	13.08	25.37	89.55	938.35	20	31.13	56.63	62.75	138.34	613.65
18-18-10	20	0.16	0.34	0.82	5.30	367.05	20	2.50	3.87	4.24	6.31	296.30
18-18-12	20	0.25	2.66	5.31	18.10	598.95	20	11.73	18.75	18.83	31.84	409.50
18-18-15	20	1.81	17.28	56.03	494.07	1366.20	20	70.40	96.85	105.86	**169.47**	622.35
20-20-10	20	0.14	0.41	3.56	39.83	1313.40	20	2.52	5.31	5.20	9.84	564.15
20-20-12	20	0.45	4.59	13.98	73.91	1329.30	20	12.51	19.25	24.01	81.89	836.75
20-20-15	20	2.44	58.04	159.50	612.43	4435.65	20	92.43	155.19	207.95	635.92	2295.70
30-30-10	20	0.52	2.60	75.52	472.25	15771.05	20	10.73	14.87	**26.55**	**161.09**	7144.85
40-40-10	20	0.91	6.89	466.68	3631.50	130308.80	20	24.27	28.98	**49.07**	**209.02**	23769.35

real optimal value, as well as the average time of LP. Table 3 compares the CP model (CPSP) against a hybrid approach in which lower bounds are first computed based on the LP to start the bottom-up approach of CPSP[5]. The LP was solved using the barrier algorithm with CPLEX (version 10.0.0). Although the hybrid model is often slowed down by the continuous relaxation (the minimum times of CPSP are far better than the minimum times of the hybrid), it scales better on the 40-40-10 instances. On 40-40-10, the hybrid approach is on average 9 times faster than CPSP.

7 Conclusion

We have provided a new approach to solving the Multileaf Collimator Sequencing Problem. Although the complexity of the resulting algorithm depends on the number of integer partitions of the maximum intensity, which is exponential, it can be used to design very efficient approaches in practice. We proposed a new CP and Linear models encoding each line as a set of shortest path problems and obtained two orders-of-magnitude improvements compared to the best known method for this problem. The linear model is a very tight formulation giving excellent lower bounds for the cardinality. A simple

[5] These experiments ran as a single thread on a Dual Quad Core Xeon CPU, 2.66GHz with 12MB of L2 cache per processor and 16GB of RAM overall, running Linux 2.6.25 x64.

hybrid approach, using the continuous relaxation at the root node before starting the search with CP, outperforms the CP model alone on large instances.

The resulting approaches strongly rely on the fact that the maximum radiation intensity is often small compared to the size of the matrix. It is, therefore, interesting to determine the complexity of the algorithm by the maximum intensity. [1] explains that in the instances available to them, the maximum intensity does not exceed 20 whereas the collimators can reach 40 rows. This limitation might, thus, not be critical in practice. However many possibilities remain to be investigated to allow better scaling in terms of the maximum element of the intensity matrix. The LP model typically has an exponential number of variables and could certainly be solved more efficiently using column generation techniques. Reasoning on the CP models could be strengthened by solving resource constrained shortest path for each row using dynamic programming and avoiding any branching on the partition variables. Finally, there are other objective functions to consider in this problem, which we will study in the future.

Acknowledgements. This work was supported by Science Foundation Ireland under Grant Number 05/IN/I886. We are indebted to Sebastian Brand for providing his benchmark instances and an executable version of the solver presented in [1].

References

1. Baatar, D., Boland, N., Brand, S., Stuckey, P.J.: Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 1–15. Springer, Heidelberg (2007)
2. Baatar, D., Hamacher, H.W., Ehrgott, M., Woeginger, G.J.: Decomposition of integer matrices and multileaf collimator sequencing. Discrete Applied Mathematics 152(1-3), 6–34 (2005)
3. Bahr, G.K., Kereiakes, J.G., Horwitz, H., Finney, R., Galvin, J., Goode, K.: The method of linear programming applied to radiation therapy planning. Radiology 91, 686–693 (1968)
4. Brand, S.: The sum-of-increments constraints in the consecutive-ones matrix decomposition problem. In: SAC 2009: 24th Annual ACM Symposium on Applied Computing (2009)
5. Burkard, R.E.: Open problem session. In: Oberwolfach Conference on Combinatorial Optimization (November 2002)
6. Engel, K.: A new algorithm for optimal multileaf collimator field segmentation. Discrete Applied Mathematics 152(1-3), 35–51 (2005)
7. Gellermann, T., Sellmann, M., Wright, R.: Shorter path constraints for the resource constrained shortest path problem. In: Barták, R., Milano, M. (eds.) CPAIOR 2005. LNCS, vol. 3524, pp. 201–216. Springer, Heidelberg (2005)
8. Hamacher, H.W., Ehrgott, M.: Special section: Using discrete mathematics to model multileaf collimators in radiation therapy. Discrete Applied Mathematics 152(1-3), 4–5 (2005)
9. Kalinowski, T.: The complexity of minimizing the number of shape matrices subject to minimal beam-on time in multileaf collimator field decomposition with bounded fluence. Discrete Applied Mathematics (in press)
10. Maher, M.J.: Analysis of a global contiguity constraint. In: Workshop on Rule-Based Constraint Reasoning and Programming (2002)
11. Martin, D., Francois, S.: A generalized permanent labelling algorithm for the shortest path problem with time windows. INFOR 26(3), 191–212 (1988)
12. 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)

Backdoors to Combinatorial Optimization: Feasibility and Optimality

Bistra Dilkina[1], Carla P. Gomes[1], Yuri Malitsky[2], Ashish Sabharwal[1], and Meinolf Sellmann[2]

[1] Department of Computer Science, Cornell University, Ithaca, NY 14853, U.S.A.
{bistra,gomes,sabhar}@cs.cornell.edu
[2] Department of Computer Science, Brown University, Providence, RI 02912, U.S.A.
{ynm,sello}@cs.brown.edu

Abstract. There has been considerable interest in the identification of structural properties of combinatorial problems that lead, directly or indirectly, to the development of efficient algorithms for solving them. One such concept is that of a backdoor set—a set of variables such that once they are instantiated, the remaining problem simplifies to a tractable form. While backdoor sets were originally defined to capture structure in decision problems with discrete variables, here we introduce a notion of backdoors that captures structure in optimization problems, which often have both discrete and continuous variables. We show that finding a feasible solution and proving optimality are characterized by backdoors of different kinds and size. Surprisingly, in certain mixed integer programming problems, proving optimality involves a smaller backdoor set than finding the optimal solution. We also show extensive results on the number of backdoors of various sizes in optimization problems. Overall, this work demonstrates that backdoors, appropriately generalized, are also effective in capturing problem structure in optimization problems.

Keywords: search, variable selection, backdoor sets.

1 Introduction

Research in constraint satisfaction problems, in particular Boolean satisfiability (SAT), and in combinatorial optimization problems, in particular mixed integer programming (MIP), has had many historic similarities (see, e.g., [2]). For example, the earliest solution methods for both started out as processes that non-deterministically or heuristically chose new inferred information to add repeatedly until the problem was fully solved. In SAT, this took the form of adding "resolvents" of two clauses and formed the original Davis-Putnam procedure. In MIP, this took the form of repeatedly adding cuts. In both cases, it was soon observed that the vast array of possibilities for such resolvents and cuts to add can easily turn into a process without much focus, and thus with limited success. The remedy seemed to be to apply a different, top-down technique instead of deriving and adding new information bottom-up. The top-down process took the form of DPLL search for SAT and of branch-and-bound for MIP. Again,

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 56–70, 2009.

it was realized that such branch-and-bound style systematic search has its own drawbacks, one of them being not *learning* anything as the search progresses. The fix—a relatively recent development in the long history of SAT and MIP methods—was to combine the two approaches. In SAT, this took the form of "clause learning" during the branch-and bound process, where new derived constraints are added to the problem upon backtracking. In MIP, this took the form of adding "cuts" and "tightening bounds" when exploring various branches during the branch-and-bound search.

This similarity between SAT and MIP research suggests that concepts that have been used successfully in one realm can perhaps also be extended to the other realm and lead to new insights. We investigate this from the angle of applying ideas from SAT to MIP. In particular, we consider heavy-tailed behavior of runtime distribution and the related concept of backdoor sets. It has been observed that (randomized) SAT solvers often exhibit a large variation in runtimes even when using randomization only for tie-breaking. At the same time, one often sees a SAT solver solve a hard real-world problem very quickly when in fact the problem should have been completely out of the reach of the solver by standard complexity arguments. Backdoor sets provide a way to understand such extremely short runs often seen on structured real-world instances and rarely seen on randomly generated instances.

We remark that the study of backdoors in constraint satisfaction problems was motivated by the observation that the performance of backtrack-style search methods can vary dramatically depending on the order of variable and value selection during the search. In particular, backtrack search methods exhibit large variance in runtime for different heuristics, different problem instances, and, for randomized methods, for different random seeds even on the same instance. The discovery of the "heavy-tailed" nature of the runtime distributions in the context of SAT [4, 5, 7, 10] has resulted in the effective introduction of randomization and restart techniques [6] and has been related to the presence of small backdoors [12]. A question, then, naturally arises: *do the runtime distributions of combinatorial optimization problems also exhibit a similar behavior? In particular, are these distributions heavy-tailed?*

Formally, heavy-tail distributions exhibit power-law decay near the tail end of the distribution and are characterized by infinite moments. The distribution tails are asymptotically of the Pareto-Levy form. Most importantly for us, the log-log plot of the tail of the survival function (i.e., how many instances are *not* solved in a given runtime) of a heavy-tailed distribution exhibits *linear behavior*. We considered the runtime distributions of MIP instances from the MIPLIB library [1], using CPLEX's [8] branch-and-bound search with a randomized branching heuristic. While heavy-tailed behavior has been reported mostly in the context of constraint satisfaction, some of the MIP optimization instances in our experiments did show heavy-tailed behavior.

These observations for MIP optimization problems motivate an in-depth study of the concept of backdoors for these problems, which is the main focus of this paper. Informally, backdoors for constraint satisfaction are sets of variables the

systematic search can, at least in principle, be limited to when finding a solution or proving infeasibility. We extend the concept of backdoor sets to optimization problems, which raises interesting new issues not addressed by earlier work on backdoor sets for constraint satisfaction. We introduce "weak optimality backdoors" for finding optimal solutions and "optimality-proof backdoors" for proving optimality. The nature of optimization algorithms, often involving adding new information such as cuts and tightened bounds as the search progresses, naturally leads to the concept of "order-sensitive" backdoors, where information learned from previous search branches is allowed to be used by the sub-solver underlying the backdoor. This often leads to much smaller backdoors than the "traditional" ones.

We investigate whether significantly small backdoors also exist for standard benchmark instances of mixed integer programming optimization problems, and find that such instances often have backdoors involving under 5% of the discrete variables. Interestingly, sometimes the optimality-proof backdoors can in fact be smaller than the weak optimality backdoors, and this aligns with the relative runtime distributions for these problems when finding an optimal solution vs. when proving optimality. A large part of our experimental work involves the problem of determining how many backdoors of various kinds and sizes exist in such optimization problems, and whether information provided by linear programming relaxations (e.g., the "fractionality" of the variables in the root LP relaxation) can be used effectively when searching for small backdoors. Our results provide positive answers to these questions.

2 Background: Backdoors for Constraint Satisfaction

We begin by recalling the concept of weak and strong backdoor sets for constraint satisfaction problems. For simplicity of exposition, we will work with the Boolean satisfiability (SAT) problem in this section, although the concepts discussed apply equally well to any discrete constraint satisfaction problem.

Backdoor sets are defined with respect to efficient sub-algorithms, called *sub-solvers*, employed within the systematic search framework of SAT solvers. In practice, these sub-solvers often take the form of efficient procedures such as unit propagation, pure literal elimination, and failed-literal probing. In some theoretical studies, solution methods for structural sub-classes of SAT such as 2-SAT, Horn-SAT, and RenamableHorn-SAT have also been studied as sub-solvers. Formally [11], a *sub-solver $\mathcal{A}$ for SAT* is any poly-time algorithm satisfying certain natural properties on every input formula F: (1) Trichotomy: $\mathcal{A}$ either determines F correctly (as satisfiable or unsatisfiable) or fails; (2) $\mathcal{A}$ determines F for sure if F has no constraints or an already violated constraint; and (3) if $\mathcal{A}$ determines F, then $\mathcal{A}$ also determines $F|_{x=0}$ and $F|_{x=1}$ for any variable x.

In the definitions of backdoor sets that follow, the sub-solver $\mathcal{A}$ will be implicit. For a formula F and a truth assignment τ to a subset of the variables of F, we will use $F[\tau]$ to denote the simplified formula obtained after applying the (partial) truth assignment to the affected variables.

Definition 1 (Weak and Strong Backdoors for SAT [11]). *Given a Boolean formula F on variables X, a subset of variables $B \subseteq X$ is a* weak backdoor *(w.r.t. a specified sub-solver $\mathcal{A}$) if for* some *truth assignment $\tau : B \to \{0,1\}$, $\mathcal{A}$ returns a satisfying assignment for $F[\tau]$. Such a subset B is a* strong backdoor *if for* every *truth assignment $\tau : B \to \{0,1\}$, $\mathcal{A}$ returns a satisfying assignment for $F[\tau]$ or concludes that $F[\tau]$ is unsatisfiable.*

Weak backdoor sets capture the fact that a well-designed heuristic can get "lucky" and find the solution to a hard satisfiable instance if the heuristic guidance is correct even on the small fraction of variables that constitute the backdoor set. Similarly, strong backdoor sets B capture the fact that a systematic tree search procedure (such as DPLL) restricted to branching only on variables in B will successfully solve the problem, whether satisfiable or unsatisfiable. Furthermore, in this case, the tree search procedure restricted to B will succeed independently of the order in which it explores the search tree.

3 Backdoor Sets for Optimization Problems

This section extends the notion of backdoor sets from constraint satisfaction problems to combinatorial optimization problems. We begin by formally defining optimization problems and discussing desirable properties of sub-solvers for such problems. Without loss of generality, we will assume throughout this text that the optimization to be performed is minimization. For simplicity of notation, we will also assume that all variables involved have the same value domain, D.

Definition 2 (Combinatorial Optimization Problem). *A combinatorial optimization problem is a four-tuple (X, D, C, z) where $X = \{x_i\}$ is a set of variables with domain D, C is a set of constraints defined over subsets of X, and $z : D^{|X|} \to \mathbb{Q}$ is an objective function to be minimized.*

A constraint $c \in C$ over variables $var(c)$ is simply a subset of all possible value assignments to the variables involved in c, i.e., $c \subseteq D^{|var(c)|}$. A value assignment v is said to *satisfy* c if the restriction of v to the variables $var(c)$ belongs to the set of value tuples constituting c.

Definition 3 (Sub-Solver for Optimization). *A* sub-solver $\mathcal{A}$ *for optimization is an algorithm that given as input a combinatorial optimization problem (X, D, C, z) satisfies the following four conditions:*
[(a)]

1. Trichotomy: *$\mathcal{A}$ either infers a lower bound on the optimal objective value z or correctly determines (X, D, C, z) (as either unsatisfiable or as optimized providing a feasible solution that is locally optimal),*
2. Efficiency: *$\mathcal{A}$ runs in polynomial time,*
3. Trivial solvability: *$\mathcal{A}$ can determine whether (X, D, C, z) is trivially satisfied (has no constraints) or trivially unsatisfiable (has an empty constraint), and*

4. Self-reducibility: *If* $\mathcal{A}$ *determines* (X, D, C, z)*, then for any variable* x_i *and value* $v \in D$*,* $\mathcal{A}$ *also determines* $(X, D, C \cup \{x_i = v\}, z)$.

For some partial assignments, the sub-solver might learn a new lower bound on the objective value. For some partial assignments, the solver may find a feasible solution that is a locally optimal solution. Any feasible solution provides an upper bound on the optimal objective value. Hence, for some partial assignments, the sub-solver might learn a new upper bound on the objective value.

In extending the notion of backdoor sets to optimization problems, we need to take into account that we face two tasks in constrained optimization: first, we need to find a feasible and optimal solution, and second, we need to prove its optimality which essentially involves proving infeasibility of the problem when the objective bound is reduced beyond the optimal value. This naturally leads to three kinds of backdoors: *weak optimality backdoors* will capture the task of finding optimal solutions, *optimality-proof backdoors* will capture the task of proving optimality given the optimal objective value, and *strong optimality backdoors* will capture the full optimization task, i.e., finding an optimal solution and proving its optimality.

Weak backdoors for optimization are the most straightforward generalization from the constraint satisfaction realm. One notable difference, however, is that since we are trying to decouple the solution-finding task from the optimality-proof task, we assume that the solution-finding task is, in a sense, somehow aware of the optimal objective value and can stop when it hits an optimal solution. In our experiments designed to identify backdoor sets, we achieve this by precomputing the optimal objective value and forcing the search to stop when a feasible solution achieving this objective value is encountered.

We use the word "traditional" in the next few definitions to distinguish them from the concept of order-sensitive backdoors to be discussed in Section 3.1. In the following, $C \cup \tau$ denotes adding to C the constraint $\{(v_1, \ldots, v_n) \mid \forall x_i \in B,\ v_i = \tau(x_i)\}$ imposing the partial assignment τ on the variables in B.

Definition 4 ((Traditional) Weak Optimality Backdoor). *Given a combinatorial optimization problem* (X, D, C, z)*, a subset of the variables* $B \subseteq X$ *is a* (traditional) weak optimality backdoor *(w.r.t. a specified sub-solver* $\mathcal{A}$*) if there exists an assignment* $\tau : B \to D$ *such that* $\mathcal{A}$ *returns a feasible solution for* $(X, D, C \cup \tau, z)$ *which is of optimal quality for* (X, D, C, z).

In contrast to decision problems, solving an optimization instance also requires proving that no better feasible solution exists. Therefore, we define the notion of backdoor sets for the optimality proof itself. Once we have found an optimal feasible solution x^*, this immediately also provides the optimal upper bound to the objective, making the new problem of seeking a better objective value infeasible. Optimality-proof backdoors are sets of variables that help one deduce this infeasibility efficiently.

Definition 5 ((Traditional) Optimality-Proof Backdoor). *Given a combinatorial optimization problem* (X, D, C, z) *and an upper bound* z^* *on the objective value, a subset of the variables* $B \subseteq X$ *is a* (traditional) optimality-proof

backdoor *(w.r.t. a specified sub-solver $\mathcal{A}$) if for every assignment $\tau : B \to D$, $\mathcal{A}$ correctly decides $(X, D, C \cup \tau \cup \{z < z^*\}, z)$ to be infeasible.*

The notion of optimality-proof backdoor allows us to decouple the process of finding feasible solutions from proving the optimality of a bound. An optimality-proof backdoor is particularly relevant when there is an external procedure that finds good feasible solutions (e.g., heuristic greedy search). Given the best solution quality found by the greedy search, we can use an optimality-proof backdoor to confirm that no better solution exists or perhaps to disprove the bound by finding a better feasible solution.

Both the definition of weak optimality backdoor and of optimality-proof backdoor implicitly or explicitly rely on the knowledge of an upper bound z^* on the objective value, i.e., they do not capture solving the original optimization problem for which the optimal value is unknown. Recall that strong backdoors for constraint satisfaction problems capture the set of variables that are enough to fully solve the problem—either prove its infeasibility or find a solution. We would like to define a similar notion for optimization problems as well. To this end, we define strong backdoors for optimization, which are enough to both find an optimal solution and prove its optimality, or to show that the problem is infeasible altogether. When the problem is feasible, a strong backdoor set is both a weak optimality backdoor and an optimality-proof backdoor.

Definition 6 ((Traditional) Strong Optimality Backdoor). *Given a combinatorial optimization problem (X, D, C, z), a subset of the variables $B \subseteq X$ is a* (traditional) strong optimality backdoor *(w.r.t. a specified sub-solver $\mathcal{A}$) if it satisfies the following conditions. For every assignment $\tau : B \to D$, $\mathcal{A}$ infers a lower bound $lb(\tau)$ on the optimal objective value for $(X, D, C \cup \tau, z)$; $lb(\tau) = +\inf$ if infeasible. If, for τ, the sub-solver also finds an optimal solution $\hat{x}(\tau)$ for $(X, D, C \cup \tau, z)$, then let $\hat{z}(\tau) = z(\hat{x}(\tau))$, else let $\hat{z}(\tau) = +\inf$. We must have: $\min_\tau lb(\tau) = \min_\tau \hat{z}(\tau)$.*

3.1 Order-Sensitive Backdoors

We now discuss an issue that arises naturally when we work with backdoor sets for state-of-the-art optimization algorithms, such as CPLEX for mixed integer programming (MIP) problems: *order-sensitivity* of backdoor sets. Order-sensitivity plays an increasingly important role as we extend the notion of backdoors to constraint optimization problems.

The standard requirement implicit in the notion of backdoor sets in constraint satisfaction problems is that the underlying systematic search procedure restricted to backdoor variables should succeed independently of the order in which it explores various truth valuations of the variables; in fact, for strong backdoors, the sub-solver must succeed on every single search branch based solely on the value assignment to the backdoor variables. This condition, however, ignores an important fact: a crucial feature of most branch-and-bound algorithms for constrained optimization problems is that they *learn* information

about the search space as they explore the search tree. For example, they learn new bounds on the objective value and the variables, and they might add various kind of "cuts" that reduce the search space without removing any optimal solution. These tightened bounds and cuts potentially allow the sub-solver to later make stronger inferences from the same partial assignment which would have normally not lead to any strong conclusions. Indeed, in our experiments designed to identify weak optimality backdoor sets for MIP problems, it was often found that variable-value assignments at the time CPLEX finds an optimal solution during search do not necessarily act as traditional weak backdoors, i.e., feeding back the specific variable-value assignment doesn't necessarily make the underlying sub-solver find an optimal solution. This leads to a natural distinction between "traditional" (as defined above) and "order-sensitive" weak optimality backdoors. In the following definitions, *search order* refers to the sequence of branching decisions that a search method uses in exploring the search space and possibly transferring any available learned information (such as cuts or tigher bounds) from previously explored branches to subsequent branches.

Definition 7 (Order-Sensitive Weak Optimality Backdoor). *Given a combinatorial optimization problem (X, D, C, z), a subset of the variables $B \subseteq X$ is an* order-sensitive weak optimality backdoor *(w.r.t. a specified sub-solver $\mathcal{A}$) if there exists some search order involving only the variables in B that leads to an assignment $\tau : B \rightarrow D$ such that $\mathcal{A}$ returns a feasible solution for $(X, D, C \cup \tau, z)$ which is of optimal quality for (X, D, C, z).*

In fact, added cuts and tightened bounds form an integral part of solving a MIP optimization problem and can critically help even when "only" detecting a feasible solution of optimal quality. The same distinction also applies to optimality-proof backdoors and to strong backdoors, simplifying the rather cumbersome definition in the latter case.

Definition 8 (Order-Sensitive Optimality-Proof Backdoor). *Given a combinatorial optimization problem (X, D, C, z) and an upper bound z^* on the objective value, a subset of the variables $B \subseteq X$ is an* order-sensitive optimality-proof backdoor *(w.r.t. a specified sub-solver $\mathcal{A}$) if there exists some search order involving only the variables in B such that $\mathcal{A}$ correctly decides $(X, D, C \cup \{z < z^*\}, z)$ to be infeasible.*

Definition 9 (Order-Sensitive Strong Optimality Backdoor). *Given a combinatorial optimization problem (X, D, C, z), a subset of the variables $B \subseteq X$ is an* order-sensitive strong backdoor *(w.r.t. a specified sub-solver $\mathcal{A}$) if there exists some search order involving only the variables in B such that $\mathcal{A}$ either correctly decides that the problem is infeasible, or finds an optimal solution and proves its optimality.*

4 Experimental Evaluation

To investigate the distribution of backdoor sizes in optimization problems, we consider the domain of Mixed Integer Programming. In our empirical study, we

use instances from the MIPLIB library [1], and employ the branch-and-bound search framework provided by CPLEX [8]. Due to the computationally intensive analysis performed in this study, we only evaluate MIPLIB instances that could be solved reasonably fast with CPLEX.

The sub-solver applied by CPLEX at each search node of the branch-and-bound routine uses a dual simplex LP algorithm in conjunction with a variety of cuts. In our previous study [3] of backdoors in Satisfiability problems, we investigated the sub-solver routine used in *Satz* [9] which applied probing to each search node. Similarly here, we set CPLEX to use strong branching, adding a lot of additional inference at each node. In summary, the sub-solver is *dual simplex+CUTS+probing*.[1]

We investigate the probability that a randomly selected subset of the variables of a given cardinality k is a backdoor. To approximate this probability, we sample many sets (500) of each given size, and for each evaluate whether the chosen set is a traditional weak optimality backdoor, order-sensitive weak optimality backdoor, and/or optimality-proof backdoor.

In our experiments we consider order-sensitive optimality-proof backdoors (and not traditional optimality-proof backdoors). For brevity, we will refer to them simply as optimality-proof backdoors. To decide whether a given set B of variables is an optimality-proof backdoor, we initialize CPLEX with the optimal solution and allow branching only on the set B. As soon as we reach a search node at which all variables of B are fixed but the infeasibility of the sub-problem at the node cannot be concluded, we reject B. Note that with a different search order, CPLEX could have succeeded in proving infeasibility if the alternative order provided stronger cuts earlier. Hence our results provide a lower bound on the probability that a set of a certain size is an optimality-proof backdoor.

To decide whether a given set B of variables is an order-sensitive weak optimality backdoor, we allow branching only on the chosen set. As soon as we find an incumbent which has optimal objective value, (precomputed ahead of time), we accept the set. That is, we stop the search when an optimal solution is found, but the optimal value is not given explicitly to the CPLEX search procedure to avoid that the subsolver can infer information from the lower bound on the objective. If we reach a search node in which all variables in the set B are fixed but the sub-solver cannot conclude the infeasibility of the sub-problem or infer a integer feasible solution, we prune the search node and continue searching. If we explore the full partial tree on the set B without finding an optimal solution, we reject B. Again, there could have been an alternative search order in which succeeded with B. Our results are again lower bounds on the true probability of order-sensitive weak optimality backdoors.

If a set was rejected as an order-sensitive weak backdoor with this procedure, then we indeed explored the full partial tree over B. Hence, we know that for

[1] For practical reasons, we consider the dual-simplex algorithm as an efficient subsolver despite its exponential worst-case complexity; after all, the problem it solves lies in the complexity class P and dual-simplex is one of the most efficient practical procedures for this problem.

sure that B is not a traditional weak optimality backdoor. In addition, for every set that was accepted as order-sensitive weak backdoor, we record the values of the variables in B at the incumbent node with optimal value. We test whether assigning B to these values without prior search results in inferring an integer feasible solution of optimal quality. If yes, then the set is accepted as traditional weak optimality backdoor. If not it is rejected. Again, there can be false negatives due the fact that some other assignment different than the incumbent found could have sufficed. Therefore, our results on the probability that a set is a traditional weak optimality backdoor are lower bounds.

4.1 Smallest Backdoors

The size of the smallest traditional weak optimality and order-sensitive optimality-proof backdoor that we have found is presented in Table 1, representing an upper bound on the true smallest size. The values for the traditional weak optimality backdoor sizes are also an upper bound on the smallest order-sensitive weak optimality backdoor size. Note that the instance *10teams* does not have an optimality-proof backdoor because its objective value is already fixed in the problem specification. Overall, we find that the vast majority of instances have small or very small (traditional) weak optimality backdoors of less than 6% of the variables. For *air04* and *air05* we even find that setting less than one thousandth of the variables is already enough to enable the sub-solver to compute an overall optimal integer feasible solution! However, as the exceptions *pk1*, *pp08a* and *pp08aCUTS* show, some real-world MIPs might not exhibit small backdoors, even for very strong sub-solvers.

Table 1. Upper bounds on the smallest size of (traditional) weak optimality backdoors and of optimality-proof backdoors in absolute value and as percentage of the number of discrete variables in the problem instance

instance	variables	discrete variables	weak backdoors size	weak backdoors %	orderOpt backdoors size	orderOpt backdoors %
10teams	2025	1800	10	0.56%	NA	NA
aflow30a	842	421	11	2.61%	85	20.19%
air04	8904	8904	3	0.03%	14	0.16%
air05	7195	7195	3	0.04%	29	0.40%
fiber	1298	1254	7	0.56%	5	0.40%
fixnet6	878	378	6	1.59%	5	1.32%
rout	556	315	8	2.54%	172	54.60%
set1ch	712	240	14	5.83%	28	11.67%
vmp2	378	168	11	6.55%	19	11.31%
pk1	86	55	20	36.36%	55	100.00%
pp08a	240	64	11	17.19%	47	73.44%
pp08aCUTS	240	64	11	17.19%	32	50.00%

4.2 Probability of Finding Small Backdoors

In addition to the smallest size of a backdoor, one is interested in knowing how hard it is to find small backdoor sets. One way to assess this difficulty is to estimate how many backdoor sets of a particular size exist for a given problem.

We want to approximate the probability that a set of variables of a given cardinality k is a backdoor. For each given backdoor size k, we sampled, uniformly at random, subsets of cardinality k from the discrete variables of the problem. For each set we evaluated whether it is a backdoor or not with the setup described in the beginning of this section.

We conducted this experiment many thousands of times for various cardinalities k. Figure 1 presents results for the instances *fiber* and *vpm2*. The curves labeled *orderOpt* refer to order-sensitive optimality-proof backdoors. The curves labeled *tradWeak* refer to weak optimality backdoors that are not order-sensitive. The curves labeled *orderWeak* refer to weak optimality backdoors that are order-sensitive. The curves labeled *tradWeak+orderOpt* refer to sets that are both (traditional) weak optimality backdoors and optimality-proof backdoors. Finally, the curves labeled *orderStrong* refer to sets that are both order-sensitive weak optimality backdoors and order-sensitive optimality-proof backdoors.

For the instance *fiber*, we observe that the probability that a set of a given size is an optimality-proof backdoor is much higher than the probability that a set of this size is a weak optimality backdoor. This evidence suggests that there are many more small optimality-proof backdoors than weak optimality backdoors. In addition, the probability that a set is both an optimality-proof backdoor and a weak backdoor is almost equal to the probability that it is a weak optimality backdoor. Our data shows that almost every set that was a weak optimality backdoor was also an optimality-proof backdoor. This suggests that for *fiber* the difficulty of the problem might lie in finding the optimal solution as opposed to proving its optimality.

Our study suggests that solving problems with a hardness profile similar to *fiber* can be significantly boosted by the availability of good initial solutions found by some heuristic search. This aligns well with the recent development of

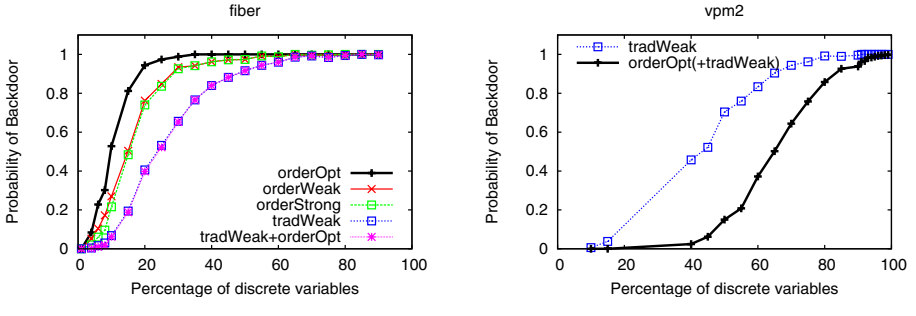


Fig. 1. Probability that a subset of variables of a given size is a backdoor when sampling uniformly

state-of-the-art MIP solvers for which it has been found that primal heuristics, so-called "feasibility pumps", can significantly boost performance.

For the instance *vpm2*, we have avoided displaying the order-sensitive weak backdoors because they fully overlap with the curve for the traditional weak backdoors. Contrary to *fiber*, for *vpm2* the probability that a set of a given size is a weak optimality backdoor is considerably higher than the probability that it is an optimality-proof backdoor. In addition, every set that was found to be an order-sensitive optimality-proof backdoor was also weak optimality backdoor. In other words, the curve for optimality-proof backdoors perfectly overlaps with the curve for sets that are both weak optimality and optimality-proof backdoors, including the curve for order-sensitive strong backdoors. We label the curve *orderOpt(+tradWeak)*. The results for *vpm2* give the intuition that the difficulty of the problem lies in proving optimality as opposed to finding an optimal solution.

To confirm the intuitions about the hardness profiles for solving *fiber* and *vpm2*, in Figure 2 we present the runtime distributions for *fiber* and *vpm2* in terms of the probability that a run is completed in a given number of search nodes. Three curves are presented for each instance. The curves labeled 'full run' represent the number of search nodes that it took to solve the problem fully - both find an optimal solution and prove its optimality. The curves 'opt soln run' represent the number of search nodes that were explored before the incumbent solution had the known optimal value. The curves 'proof run' capture the number of search nodes that were explored to prove that a solution of a better quality does not exist, i.e., proving infeasibility once an optimal solution is provided. This comparison allows us to estimate the relative effort spent on each task and the effort overall. We see that the intuition from the distribution of backdoor sizes was indeed correct. For *fiber*, the effort spent of finding the optimal solution explains almost all of the full runtime, while the effort that is needed when only proving infeasibility is considerably less. On the other hand, for *vpm2* the gap between the effort on finding an optimal solution and the full effort is substantial, especially in the beginning. The full runs are clearly taking much longer than the

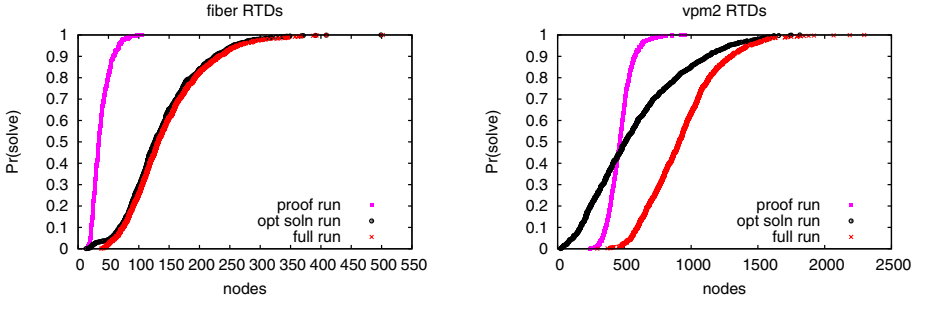


Fig. 2. Runtime distributions for finding an optimal solution, for proving optimality, and for fully solving the problem

fastest solution-finding runs, but about the same as the slowest solution-finding runs. Here, proving optimality takes longer than the fastest solution-finding runs but shorter than the slowest solution-finding runs.

4.3 LP Relaxations as Primal Heuristics

We saw that MIPs, even when they have small backdoors, may only have very few weak backdoor sets of a particular (small) size. The question arises of how a MIP solver could exploit its sub-solver to find small backdoors. To see whether LP relaxations can provide guidance about which variables may belong to a small backdoor set, we slightly modified the experiment from the previous section. Rather than sampling sets of desired cardinality by selecting variables uniformly at random, we biased the selection based on the "fractionality" of variables in the root relaxation. The fractionality of a variable measures how far it is from the nearest integer value. E.g., the fractionality of a variable X with domain 0,1 is simply $f(X) = min\{|x|, |1-x|\}$. More formally, if the root LP value of variable X_i is $\bar{x_i}$, then its fractional part is $f_i = \bar{x_i} - \lfloor \bar{x_i} \rfloor$. We assign to each variable a weight $f(X_i) \leftarrow 0.5 - |0.5 - f_i|$. Note that the quantity $f(X_i)$ captures the "infeasibility" of a variable which is a well-known measure for picking branching variables in mixed integer programming. Some discrete variables could be integral in the root LP. For such variables X_i, we assign a small non-zero weight $f(X_i) = \varepsilon$. After we normalize the variable weights, we choose a subset of size k where each variable is selected with probability proportional to its normalized weight.

For each desired size k, we sampled many sets of variables again and tested which ones were backdoors. The result of these experiments is summarized in Figure 3 for *fiber* and in Figure 4 for *vpm2*. The effect of sampling sets in a biased fashion is clearly visible (curves resulting from biased selection are marked with a "b-"). For the instance *fiber*, choosing sets biased by the root LP clearly increases the probability of selection a set which is an optimality-proof backdoor or a set which is a weak optimality backdoor. Surprisingly, selecting 6% of the variables

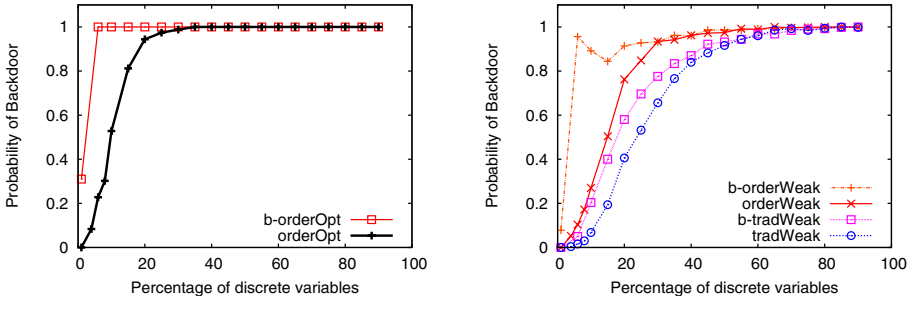


Fig. 3. FIBER: Comparing the probability that a subset of variables of a given size is a backdoor when sampling uniformly versus when sampling based on the fractionality of variables at the root

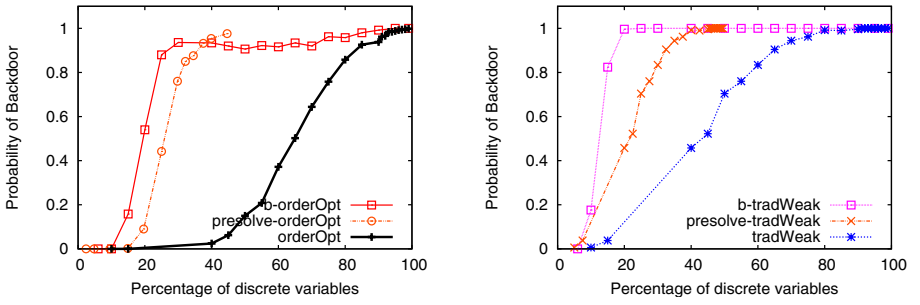


Fig. 4. VPM2: Comparing the probability that a subset of variables of a given size is a backdoor when sampling uniformly versus when sampling based on the fractionality of variables at the root

in this fashion is enough to guarantee that the set is an optimality-proof backdoor (100%), and give a 95% chance that the selected set is a weak backdoor.

The improvement effect is even more dramatic for the instance *vpm2*. Here, with 20% of the variables selected in the biased way we are guaranteed to select a weak backdoor, compared to a less than 2% chance when selected uniformly. Also, while with 30% of the variables selected in the biased way we have a 93% chance of selecting an optimality-proof backdoor set, we have less than 0.02% chance of such event when selecting uniformly. This shows clearly that an LP sub-solver can be exploited effectively to find small backdoors.

One thing to note is that before solving the root LP, CPLEX applies a *preprocessing* procedure which simplifies the problem and removes some variables whose values can be trivially inferred or can be expressed as an aggregation of other variables[2]. This procedure can sometimes result in dramatic reduction in the effective problem size. In *fiber*, the discrete variables removed by preprocessing are less than 17%. However, for *vpm2* the preprocessing removes 50% of the discrete variables.

One advantage of biasing the set selection by the root LP is that the variables trivially inferred by the preprocessing will have integral values, and will be selected only with some very small probability. To evaluate whether the biased selection draws its advantage over the uniform selection solely on avoiding pre-processed variables, we evaluated the probability of selecting a backdoor set when sampling uniformly among only the discrete variables remaining after preprocessing for *vpm2*. The results for this experiment are presented in the curves *presolve-orderOpt* and *presolve-tradWeak* in Figure 4. These curves show that choosing uniformly among the remaining variables is more effective for finding backdoors than choosing uniformly among all discrete variables, but it is not as good as the biased selection based on the root LP relaxation. Hence biasing the selection by the fractionality of the variables of the root LP has additional merit for discovering small backdoor sets.

[2] However, the user-defined branching procedure of CPLEX still works on the original set of variables.

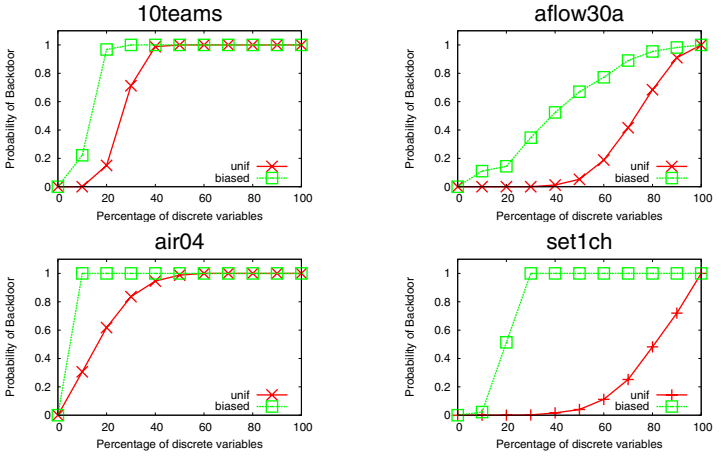


Fig. 5. Probability that a subset of variables of a given size is a traditional weak optimality backdoor backdoor when sampling uniformly (crosses) versus when sampling based on the fractionality of variables at the root (biased)

Other MIPLIB instances for which we have found that the biased selection has a substantial effect are *10teams*, *aflow30a*, *air04*, and *set1ch*. We present the results in Figure 5. For these instances, we only performed a quick evaluation, where we tested whether a set of variables B is a traditional weak optimality backdoor by setting their values to the values in the optimal solution found by default by CPLEX. Hence, the results are loose lower bounds on the actual probabilities.

5 Conclusion

In this work, we extended the concept of backdoor sets from constraint satisfaction problems to combinatorial optimization problems. This extension also involved incorporating learning into the notion of backdoors by introducing order-sensitive backdoors. While it has been previously shown that real-world SAT instances have very small backdoors, here we showed that small backdoors also exists to standard benchmark instances in mixed integer programming. In particular, optimization instances can have very small weak optimality backdoors and often also small optimality-proof backdoors. Surprisingly, sometimes the optimization-proof backdoors can in fact be smaller than the weak optimality backdoors.

We also considered the question of how hard it is to find small backdoor sets and provided extensive numerical results. We studied the probability that a set of a given size is an order-sensitive optimality-proof backdoor and the probability that it is an order-sensitive or traditional weak optimality backdoor. In general, we have shown that the difference in the distributions of weak optimality backdoors and of optimality-proof backdoors for a particular instance is well

aligned with the difference in the runtime distributions for the tasks of finding an optimal solution and proving optimality, respectively. Finally, we have also demonstrated that the fractionality of variables in the root LP relaxation is a very good heuristic for uncovering small backdoors for both solution finding and for proof of optimality.

Acknowledgments

This research was supported by IISI, Cornell University (AFOSR grant FA9550-04-1-0151), NSF Expeditions in Computing award for Computational Sustainability (Grant 0832782), NSF IIS award (Grant 0514429), and the Cornflower Project (NSF Career award 0644113). The third author was partly supported by NSERC Postgraduate Fellowship. Part of this work was done while the forth author was visiting McGill University.

References

[1] Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Operations Research Letters 34(4), 1–12 (2006), http://miplib.zib.de
[2] Bixby, R.E.: Solving real-world linear programs: A decade and more of progress. Oper. Res. 50(1), 3–15 (2002)
[3] Dilkina, B., Gomes, C.P., Sabharwal, A.: Tradeoffs in the complexity of backdoor detection. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 256–270. Springer, Heidelberg (2007)
[4] Gent, I.P., Walsh, T.: Easy problems are sometimes hard. AI J. 70, 335–345 (1994)
[5] Gomes, C.P., Selman, B., Crato, N.: Heavy-tailed distributions in combinatorial search. In: Smolka, G. (ed.) CP 1997. LNCS, vol. 1330, pp. 121–135. Springer, Heidelberg (1997)
[6] Gomes, C.P., Selman, B., McAloon, K., Tretkoff, C.: Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. In: 4th Int. Conf. Art. Intel. Planning Syst. (1998)
[7] Hogg, T., Williams, C.: Expected gains from parallelizing constraint solving for hard problems. In: Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI 1994), Seattle, WA, pp. 1310–1315. AAAI Press, Menlo Park (1994)
[8] ILOG, SA. CPLEX 10.1 Reference Manual (2006)
[9] Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: 15th IJCAI, Nagoya, Japan, pp. 366–371 (August 1997)
[10] Smith, B.M., Grant, S.A.: Sparse constraint graphs and exceptionally hard problems. In: 14th IJCAI, Montreal, Canada, vol. 1, pp. 646–654 (August 1995)
[11] Williams, R., Gomes, C., Selman, B.: Backdoors to typical case complexity. In: 18th IJCAI, Acapulco, Mexico, pp. 1173–1178 (August 2003)
[12] Williams, R., Gomes, C., Selman, B.: On the connections between heavy-tails, backdoors, and restarts in combinatorial search. In: 6th SAT, Santa Margherita Ligure, Italy, pp. 222–230 (May 2003)

Solution Enumeration for Projected Boolean Search Problems

Martin Gebser, Benjamin Kaufmann, and Torsten Schaub*

Institut für Informatik, Universität Potsdam, August-Bebel-Str. 89, D-14482 Potsdam, Germany

Abstract. Many real-world problems require the enumeration of all solutions of combinatorial search problems, even though this is often infeasible in practice. However, not always all parts of a solution are needed. We are thus interested in projecting solutions to a restricted vocabulary. Yet, the adaption of Boolean constraint solving algorithms turns out to be non-obvious provided one wants a repetition-free enumeration in polynomial space. We address this problem and propose a new algorithm computing projective solutions. Although we have implemented our approach in the context of Answer Set Programming, it is readily applicable to any solver based on modern Boolean constraint technology.

1 Introduction

Modern Boolean constraint technology has led to a tremendous boost in solver performance in various areas dealing with combinatorial search problems. Pioneered in the area of Satisfiability checking (SAT; [1,2,3]) where it has demonstrated its maturity for real-world applications, its usage is meanwhile also advancing in neighboring areas, like Answer Set Programming (ASP; [4]) and even classical Constraint Processing. Although traditionally problems are expressed in terms of satisfiability or unsatisfiability, many real-world applications require surveying all solutions of a problem. For instance, inference in Bayes Nets can be reduced to #SAT (cf. [5]) by counting the number of models. However, the exhaustive enumeration of all solutions is often infeasible. Yet not always all parts of a solution are needed. Restrictions may lead to a significant decrease of computational efforts; in particular, whenever the discarded variables have their proper combinatorics and thus induce a multitude of redundant solutions.

We are thus interested in projecting solutions to a restricted vocabulary. However, the adaption of Boolean constraint solving algorithms turns out to be non-obvious, if one wants a repetition-free enumeration in polynomial space. We address this by proposing a new algorithm for solution projection. Given a problem Δ having solutions $\mathcal{S}(\Delta)$ and a set P of variables to project on, we are interested in computing the set $\{S \cap P \mid S \in \mathcal{S}(\Delta)\}$. We refer to its elements as the *projective solutions* for Δ wrt P. To compute all such projections, we first provide a direct extension of a conflict-driven learning algorithm by means of solution recording. Although this approach is satisfactory when the number of projective solutions is limited, it does not scale since it is exponential in space. After analyzing the particularities of the search problem, we propose a new conflict-driven learning algorithm that uses an elaborated backtracking scheme and only

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

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 71–86, 2009.

a linear number of solution-excluding constraints. Although we have implemented our approach in the context of ASP, it is readily applicable to any solving approach based on modern Boolean constraint technology. Lastly, we provide an empirical analysis demonstrating the computational impact of our approach.

2 Background

The idea of ASP is to encode a problem as a logic program such that its answer sets represent solutions to the original problem. More formally, a *logic program* Π is a finite set of *rules* of the form $a \leftarrow b_1, \ldots, b_m, \sim c_{m+1}, \ldots, \sim c_n$, where a, b_i, c_j are atoms for $0 < i \leq m, m < j \leq n$ and $\sim$ is (default) negation. The *answer sets* of Π are particular models of Π satisfying an additional stability criterion. For brevity, we refer the reader to [6] for a formal introduction to ASP.

As a running example, consider the program composed of the following rules:

$$
\begin{array}{llllll}
x \leftarrow q, r & (1) & y \leftarrow x, \sim q & (5) & z \leftarrow x, \sim r & (9) \\
x \leftarrow \sim y, \sim z & (2) & y \leftarrow \sim x, \sim z & (6) & z \leftarrow \sim x, \sim y & (10) \\
p \leftarrow x & (3) & q \leftarrow x & (7) & r \leftarrow x & (11) \\
p \leftarrow \sim x & (4) & q \leftarrow \sim r & (8) & r \leftarrow \sim q & (12)\ .
\end{array}
$$

Among the ten (classical) models of this program, we find five answer sets: $\{p, q, y\}$, $\{p, q, z\}$, $\{p, q, r, x\}$, $\{p, r, y\}$, and $\{p, r, z\}$. Projecting them onto the atoms $\{p, q, r\}$ results in only three distinct solutions: $\{p, q\}$, $\{p, q, r\}$, and $\{p, r\}$.

An *assignment* $\mathbf{A}$ is a sequence $(\sigma_1, \ldots, \sigma_n)$ of *literals* σ_i of the form $\mathbf{T}v_i$ or $\mathbf{F}v_i$ where v_i is a (Boolean) variable for $1 \leq i \leq n$; $\mathbf{T}v_i$ expresses that v_i is *true* and $\mathbf{F}v_i$ that it is *false*. We denote the complement of a literal σ by $\overline{\sigma}$, that is, $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$. Also, we let $var(\mathbf{T}v) = var(\mathbf{F}v) = v$. We sometimes abuse notation and identify an assignment with the set of its contained literals. Given this, we access the true and false variables in $\mathbf{A}$ via $\mathbf{A}^{\mathbf{T}} = \{v \mid \mathbf{T}v \in \mathbf{A}\}$ and $\mathbf{A}^{\mathbf{F}} = \{v \mid \mathbf{F}v \in \mathbf{A}\}$. For a canonical representation of (Boolean) constraints, we make use of nogoods [7]. In our setting, a *nogood* is a finite set $\{\sigma_1, \ldots, \sigma_m\}$ of literals, expressing a constraint violated by any assignment $\mathbf{A}$ containing $\sigma_1, \ldots, \sigma_m$. For a set Δ of nogoods, define $var(\Delta) = \bigcup_{\delta \in \Delta} \{var(\sigma) \mid \sigma \in \delta\}$. An assignment $\mathbf{A}$ such that $\mathbf{A}^{\mathbf{T}} \cap \mathbf{A}^{\mathbf{F}} = \emptyset$ and $\{\delta \in \Delta \mid \exists \sigma \in \delta : \overline{\sigma} \in \mathbf{A}\} = \Delta$ is a *solution* for Δ. For a given set P of variables, we call a set $\mathbf{P}$ of literals such that $\mathbf{P}^{\mathbf{T}} \cup \mathbf{P}^{\mathbf{F}} = P$ a *projective solution* for Δ wrt P, if there is some solution $\mathbf{A}$ for Δ such that $\mathbf{P} \subseteq \mathbf{A}$.

A translation of logic programs in ASP into nogoods is developed in [4]. For brevity, we illustrate it by two examples. First, consider the nogoods induced by atom y in the above program. Atom y depends on two bodies: $\{x, \sim q\}$ and $\{\sim x, \sim z\}$ in (5) and (6). We get the nogoods $\{\mathbf{T}y, \mathbf{F}\{x, \sim q\}, \mathbf{F}\{\sim x, \sim z\}\}$, $\{\mathbf{F}y, \mathbf{T}\{x, \sim q\}\}$, and $\{\mathbf{F}y, \mathbf{T}\{\sim x, \sim z\}\}$ by taking for convenience the actual bodies rather than introducing new variables. For instance, the first nogood eliminates solutions where y is true although neither the rule in (5) nor (6) are applicable. In turn, body $\{x, \sim q\}$ induces nogoods $\{\mathbf{F}\{x, \sim q\}, \mathbf{T}x, \mathbf{F}q\}$, $\{\mathbf{T}\{x, \sim q\}, \mathbf{F}x\}$, and $\{\mathbf{T}\{x, \sim q\}, \mathbf{T}q\}$. The last two nogoods deny solutions where the body is true although one of its conjuncts is false.

3 Algorithms for Solution Projection

When enumerating solutions, standard backtracking algorithms like that of Davis, Putnam, Logemann, and Loveland (DPLL; [8,9]) usually encounter multiple solutions being identical on a projected vocabulary. Such redundancy could easily be avoided by branching on projected before any other variables. However, the limitation of branching can cause an exponential degradation of performance (see below).

Also our enumeration algorithms for projective solutions make use of a decision heuristic: $\textsc{Select}(\Delta, \nabla, \mathbf{A}, P)$ takes a set Δ of (input) nogoods, a set ∇ of (recorded) nogoods, an assignment $\mathbf{A}$, and a set P of variables as arguments. Dynamic heuristics devised for DPLL typically consider Δ and $\mathbf{A}$ for their decisions. In contrast, heuristics devised for Conflict-Driven Clause Learning (CDCL; [1,2,3]) are far more interested in ∇, containing nogoods derived from conflicts. Finally, as speculated above, a heuristic tailored for the enumeration of projective solutions could pay particular attention to the set P of variables to project on. For instance, OPTSAT [10] makes use of a decision heuristic preferring minimal literals in a partially ordered set. Although OPTSAT does not aim at enumeration, a similar intervention could be used in our setting for canceling redundancies. However, we argue below that constraining the heuristic in such a way can have a drastic negative impact. Hence, we refrain from devising any ad hoc heuristic and leave the internals of $\textsc{Select}(\Delta, \nabla, \mathbf{A}, P)$ unspecified. As a matter of fact, the formal properties of our algorithms are largely independent of heuristics.

Projective Solution Recording. Our goal is the repetition-free enumeration of all projective solutions for a given set Δ of nogoods wrt a set P of variables. To illustrate the peculiarities, we start with a straightforward approach recording all projective solutions in order to avoid recomputation. Our enumeration algorithm is based on CDCL, but presented in terms of nogoods and thus called Conflict-Driven Nogood Learning (CDNL). It deviates from the corresponding decision algorithm, which halts at the first solution found, merely by recording computed projective solutions as nogoods and then searching for alternative solutions.

Algorithm 1 shows our first main procedure for enumerating projective solutions. Its input consists of a set Δ of nogoods, a set P of variables to project on, and a number s of projective solutions for Δ wrt P to compute. Projective solutions are obtained from assignments $\mathbf{A}$ (initialized in Line 1) that are solutions for Δ. The dynamic nogoods in ∇ (initialized in Line 2) are derived from conflicts (cf. Line 9–10). In general, nogoods in ∇ are consequences of those in Δ and may thus be deleted at any time in order to achieve polynomial space complexity. Only such nogoods that are asserting (explained below) must not be deleted from ∇, but their number is bound by the cardinality of $var(\Delta)$. Finally, the decision level dl (initialized in Line 3) counts the number of heuristically selected decision literals in $\mathbf{A}$. The global structure of Algorithm 1 is similar to the one of the decision version of CDNL (or CDCL) by iterating propagation (Line 5) and distinguishing three resulting cases: a conflict (Line 6–11), a solution (Line 12–20), or a heuristic decision (Line 22–24). Function $\textsc{BooleanConstraintPropagation}(\Delta \cup \nabla, \mathbf{A})$ first augments $\mathbf{A}$ with implied literals, that is, literals necessarily contained in any solution for $\Delta \cup \nabla$ that extends $\mathbf{A}$. A well-known technique to identify such literals is *unit propagation* (cf. [2,3]); it

Algorithm 1. CDNL-RECORDING

Input : A set Δ of nogoods, a set P of variables, and a number s of requested solutions.

```
1  A ← ∅                                                          // assignment
2  ∇ ← ∅                                                          // set of (dynamic) nogoods
3  dl ← 0                                                         // decision level
4  loop
5      A ← BooleanConstraintPropagation(Δ ∪ ∇, A)
6      if ε ⊆ A for some ε ∈ Δ ∪ ∇ then                            // conflict
7          if dl = 0 then exit
8          else
9              (δ, dl) ← ConflictResolution(ε, Δ ∪ ∇, A)
10             ∇ ← ∇ ∪ {δ}
11             A ← A \ {σ ∈ A | dlevel(σ) > dl}
12     else if {δ ∈ Δ | ∃σ ∈ δ : σ̄ ∈ A} = Δ then                  // solution
13         S ← {σ_p ∈ A | var(σ_p) ∈ P}
14         print S
15         s ← s − 2^(|P|−|S|)
16         if s ≤ 0 or max{dlevel(σ_p) | σ_p ∈ S} = 0 then exit
17         else
18             Δ ← Δ ∪ {S}                                        // record solution (persistently)
19             dl ← max{dlevel(σ_p) | σ_p ∈ S} − 1
20             A ← A \ {σ ∈ A | dlevel(σ) > dl}
21     else
22         σ_d ← Select(Δ, ∇, A, P)                               // decision
23         dlevel(σ_d) ← dl ← (dl + 1)
24         A ← A ∘ σ_d
```

iteratively adds complements $\overline{\sigma}$ to $\mathbf{A}$, if $\delta \setminus \mathbf{A} = \{\sigma\}$ for some $\delta \in \Delta \cup \nabla$, until reaching a conflict or a fixpoint. In the context of ASP, propagation also includes unfounded set checks (cf. [4,11]). In principle, other techniques, such as failed literal detection, could be applied in addition, but they are less common in CDNL (or CDCL). We next detail the cases encountered after propagation, starting with the simplest one of a decision.

Decision. As mentioned above, we do not assume any particular heuristic but stipulate for any literal σ_d returned by $\textsc{Select}(\Delta, \nabla, \mathbf{A}, P)$ that $\{\sigma_d, \overline{\sigma_d}\} \cap \mathbf{A} = \emptyset$ and $var(\sigma_d) \in var(\Delta \cup \nabla)$. That is, σ_d must be undecided and occur in the input. For every literal $\sigma \in \mathbf{A}$, $dlevel(\sigma)$ provides its decision level. Based on this, operation $\mathbf{A} \circ \sigma'$ inserts σ' as the last literal of $dlevel(\sigma')$ into $\mathbf{A}$, before any $\sigma \in \mathbf{A}$ such that $dlevel(\sigma) > dlevel(\sigma')$. A decision literal σ_d is always appended to $\mathbf{A}$ (in Line 24).

Conflict. A conflict is encountered whenever some nogood ε is violated by $\mathbf{A}$ (cf. Line 6). If no decision has been made, there is no (further) solution for Δ, and enumeration terminates (Line 7). Otherwise, a reason δ for the conflict is calculated (Line 9) and recorded as a dynamic nogood (Line 10). We assume that the nogood δ returned by $\textsc{ConflictResolution}(\varepsilon, \Delta \cup \nabla, \mathbf{A})$ is violated by $\mathbf{A}$ and contains a

Unique Implication Point (UIP; [1,12]), viz., there is some literal $\sigma \in \delta$ such that $dlevel(\sigma) > \max\{dlevel(\sigma') \mid \sigma' \in \delta \setminus \{\sigma\}\}$. We assume conflict resolution to work according to the *First-UIP* scheme [3,12], resolving ε against nogoods used to derive implied literals in ε (this is why $\mathbf{A}$ is a sequence; cf. [4,11]) until reaching the first UIP, which is not necessarily a decision literal. Backjumping (Line 11) then returns to decision level $dl = \max\{dlevel(\sigma') \mid \sigma' \in \delta \setminus \{\sigma\}\}$, where δ implies $\overline{\sigma}$ by unit propagation. Note that δ is the single nogood in $\Delta \cup \nabla$ justifying the inclusion of $\overline{\sigma}$ in $\mathbf{A}$ at decision level dl; such a dynamic nogood is called *asserting*. Even though Algorithm 1 does not mention deletion, dynamic nogoods that are not asserting may be deleted at any time. Since there cannot be more asserting nogoods than literals in $\mathbf{A}$, this permits running the decision version of CDNL in polynomial space. Finally, by altering conflict resolution to simply return all decision literals in $\mathbf{A}$, we can mimic DPLL with Algorithm 1 (rather than explicitly flipping a decision literal, its complement is asserted). Thus, the considerations below apply also to DPLL variants for enumerating projective solutions.

Solution. The last case is that of a solution, viz., an assignment $\mathbf{A}$ containing the complement of at least one literal from each nogood (cf. Line 12). The corresponding projective solutions for Δ wrt P are represented by $\mathbf{S}$, the set of literals in $\mathbf{A}$ over variables in P (cf. Line 13). After printing $\mathbf{S}$ (Line 14), we calculate the number of projective solutions still requested (Line 15). Note that, for $P \setminus (\mathbf{A^T} \cup \mathbf{A^F}) = \{p_1, \ldots, p_k\}$, each of the 2^k sets $\mathbf{S} \cup \{\mathbf{X}_i p_i \mid 1 \leq i \leq k\}$ such that $\mathbf{X}_i \in \{\mathbf{T}, \mathbf{F}\}$ for $1 \leq i \leq k$ is a projective solution for Δ wrt P, so that $\mathbf{A}$ represents $2^{|P|-|\mathbf{S}|}$ of them. If the number of requested projective solutions have been enumerated or if all literals in $\mathbf{S}$ are implied at decision level 0 (independent of decisions), we are done with enumeration (Line 16). Otherwise, our first procedure records $\mathbf{S}$ persistently in Δ (Line 18). In fact, unlike dynamic nogoods in ∇, $\mathbf{S}$ is not a consequence of Δ because its literals belong to a solution for Δ. Hence, we must exclude the deletion of $\mathbf{S}$, and so cannot store it as a dynamic nogood in ∇. Finally, at least one literal of $\mathbf{S}$ has to be unassigned in order to enumerate any further projective solutions. This is accomplished by retracting the maximum decision level of literals in $\mathbf{S}$ as well as all greater decision levels (Line 19–20). In principle, it is also possible to backtrack further or even to restart search from scratch by retracting all decision levels except for 0. The strategy of leaving as many decision levels as possible assigned is guided by the goal of facilitating the discovery of projective solutions nearby $\mathbf{S}$. However, as with nogood deletion, restarts can optionally be included, permitting the customization of backtracking from a solution.

We proceed by stating formal properties of Algorithm 1. The first one, *termination*, follows from the termination of CDNL on unsatisfiable sets of nogoods (cf. [13] for a proof) and the fact that solutions are excluded by strengthening the original problem.

Theorem 1. *Let Δ be a finite set of nogoods, P a set of variables, and s a number. Then, we have that* CDNL-RECORDING(Δ, P, s) *terminates.*

The second property, *soundness*, is due to the condition in Line 12 of Algorithm 1.

Theorem 2. *Let Δ be a finite set of nogoods, P a set of variables, and s a number. For every $\mathbf{S}$ printed by* CDNL-RECORDING(Δ, P, s) *and every $Q \subseteq P$, we have that $\mathbf{S} \cup \{\mathbf{T}q \mid q \in Q \setminus \mathbf{S^F}\} \cup \{\mathbf{F}r \mid r \in P \setminus (Q \cup \mathbf{S^T})\}$ is a projective solution for Δ wrt P.*

The third property, *completeness*, follows from the prerequisite that any nogood in ∇ is a consequence of those in Δ. Hence, no projective solution for Δ wrt P is ever excluded by $\Delta \cup \nabla$ before it was enumerated.[1]

Theorem 3. *Let Δ be a finite set of nogoods, P a set of variables, and $P_\Delta = var(\Delta) \cap P$. For every projective solution $\mathbf{P}$ for Δ wrt P, we have that* CDNL-RECORDING$(\Delta, P_\Delta, 2^{|P_\Delta|})$ *prints some $\mathbf{S} \subseteq \mathbf{P}$.*

Finally, *redundancy-freeness* is obtained from the fact that each already enumerated projective solution is represented by a nogood $\delta \in \Delta$, so that all further solutions for Δ must contain the complement $\overline{\sigma_p}$ of at least one literal $\sigma_p \in \delta$.

Theorem 4. *Let Δ be a finite set of nogoods, P a set of variables, and s a number. For every projective solution $\mathbf{P}$ for Δ wrt P, we have that* CDNL-RECORDING(Δ, P, s) *prints some $\mathbf{S} \subseteq \mathbf{P}$ at most once.*

In the worst case, there are exponentially many (representative literal sets of) projective solutions for Δ wrt P, each of which must be recorded in some way by Algorithm 1. Thus, our next goal is revising Algorithm 1 to work in polynomial space under maintaining its properties, in particular, redundancy-freeness. The peculiarities of this task are listed next. For brevity, we refrain from giving exemplary inputs Δ and P exhibiting the listed possibilities, but it is not difficult to come up with them.

First, for a solution $\mathbf{A}$ for Δ, there can be another solution $\mathbf{B}$ for Δ differing from $\mathbf{A}$ only on variables outside P (requiring a different decision on some variable outside P).

Fact 1. *Let $\mathbf{A}$ be a solution for a set Δ of nogoods containing decision literals $\{\sigma_1, \ldots, \sigma_j\}$. It is possible that there is some solution $\mathbf{B}$ for Δ such that $\{\sigma_p \in \mathbf{A} \mid var(\sigma_p) \in P\} \subseteq \mathbf{B}$, but $\{\overline{\sigma_1}, \ldots, \overline{\sigma_j}\} \cap \mathbf{B} \neq \emptyset$. Then, if $\overline{\sigma_i} \in \mathbf{B}$ for $1 \leq i \leq j$, we have $var(\sigma_i) \notin P$. We conclude that flipping some literal(s) in $\{\sigma_i \mid 1 \leq i \leq j, var(\sigma_i) \notin P\}$ may not exclude repetitions of projective solutions for Δ wrt P.*

Second, for a solution $\mathbf{A}$ for Δ, there can be another solution $\mathbf{B}$ for Δ differing from $\mathbf{A}$ on some variable in P, but not on any decision literal in $\mathbf{A}$ over P.

Fact 2. *Let $\mathbf{A}$ be a solution for a set Δ of nogoods containing decision literals $\{\sigma_1, \ldots, \sigma_j\}$. It is possible that there is some solution $\mathbf{B}$ for Δ such that $\{\sigma_p \in \mathbf{A} \mid var(\sigma_p) \in P\} \not\subseteq \mathbf{B}$, but $\{\sigma_i \mid 1 \leq i \leq j, var(\sigma_i) \in P\} \subseteq \mathbf{B}$. Then, $\mathbf{B}$ includes the decision literals over P from $\mathbf{A}$, still covering different projective solutions for Δ wrt P. We conclude that flipping some literal(s) in $\{\sigma_i \mid 1 \leq i \leq j, var(\sigma_i) \in P\}$ may eliminate non-redundant projective solutions for Δ wrt P.*

Combining Fact 1 and 2, we observe that flipping decision literals over variables outside P does not guarantee redundancy-freeness, while flipping decision literals over P might sacrifice completeness. Hence, with a heuristic free to return an arbitrary decision literal, we do not know which literal of a solution $\mathbf{A}$ for Δ should be flipped. This obscurity could be avoided by deciding variables in P before those outside P. However, such an approach suffers from a negative proof complexity result on unsatisfiable inputs, and for hard satisfiable problems, similar declines are not unlikely.

Fact 3. *Any restricted decision heuristic that returns a literal σ_d such that $var(\sigma_d) \notin P$ only wrt assignments $\mathbf{A}$ such that $var(\Delta \setminus \{\delta \in \Delta \mid \exists \sigma \in \delta : \overline{\sigma} \in \mathbf{A}\}) \cap$*

[1] It is sufficient to consider the set P_Δ of variables occurring in both Δ and P, along with the size $2^{|P_\Delta|}$ of the power set of P_Δ.

$P \subseteq \mathbf{A}^{\mathbf{T}} \cup \mathbf{A}^{\mathbf{F}}$ *(that is,* $var(\sigma) \notin P$ *holds for all undecided literals* σ *in not yet satisfied nogoods of* Δ*) incurs super-polynomially longer optimal computations than can be obtained with an unrestricted decision heuristic on certain inputs. This handicap follows from Lemma 3 in [14], showing that CDCL with decisions restricted to variables* P *acting as input gates of Boolean circuits has super-polynomially longer minimum-length proofs of unsatisfiability than DPLL on infinite family* $\{\mathrm{EPHP}_n^{n+1}\}$ *of Boolean circuits. The circuits in this family can be translated into a set* Δ *of nogoods [14] such that every assignment* $\mathbf{A}$ *satisfying* $var(\Delta \setminus \{\delta \in \Delta \mid \exists \sigma \in \delta : \overline{\sigma} \in \mathbf{A}\}) \cap P \subseteq \mathbf{A}^{\mathbf{T}} \cup \mathbf{A}^{\mathbf{F}}$ *yields an immediate conflict. We conclude that any restricted decision heuristic is doomed to return only literals* σ_d *such that* $var(\sigma_d) \in P$*; hence, it handicaps CDNL computations in the sense of Lemma 3 in [14].*

The last fact tells us that any heuristic guaranteeing redundancy-freeness (and completeness) right away must fail on certain inputs. To avoid this, we need to devise a procedure that adaptively excludes redundancies.

Projective Solution Enumeration. Our second procedure for the enumeration of projective solutions for Δ wrt P is shown in Algorithm 2. Its overall structure, iterating propagation before distinguishing the cases of conflict (Line 6–18), solution (Line 19–38), and decision (Line 40–42), is similar to our first algorithm. We thus focus on the differences between both procedures. In this regard, the progress information of Algorithm 2 involves an additional systematic backtracking level bl (initialized in Line 3). The basic idea is to gather decision literals over P at decision levels 1 to bl that are to be backtracked systematically for the sake of enumerating further non-redundant projective solutions. In this way, Algorithm 2 establishes an enumeration scheme that can be maintained in polynomial space, abolishing the need of persistent solution recording. But as mentioned above, an important objective is to avoid interference with the actual search. In particular, before any projective solutions have been found, there is no cause for enforcing systematic backtracking. Hence, systematic backtracking levels are introduced only after finding some projective solutions, but not a priori. The case of a solution is explained next.

Solution. Projective solutions for Δ wrt P are extracted from a solution $\mathbf{A}$ for Δ and counted like in the first algorithm (cf. Line 19–22). As before, enumeration terminates if enough projective solutions have been computed or if the search space has been exhausted (Line 23). If neither is the case, the treatment of the discovered projective solutions in $\mathbf{S}$ distinguishes Algorithm 2 from its predecessor that simply records $\mathbf{S}$. Let us assume that $\mathbf{S}$ has been constructed from at least one heuristically selected literal (Line 31–38), so that alternative decisions may lead to distinct projective solutions. In order to enumerate them, we must certainly flip some decision literal(s) in $\mathbf{A}$, but Fact 1 and 2 tell us that we cannot be sure about which one(s). This obscurity is now dealt with via systematic backtracking, and thus we increment bl (Line 31) in order to introduce a new systematic backtracking level. The introduction involves storing $\mathbf{S}$ in Δ, but now as a nogood $\delta(bl)$ associated with bl (Line 32–33). The other cases of Algorithm 2 are such that $\delta(bl)$ is removed from Δ as soon as bl is retracted, which establishes polynomial space complexity. Until then, $\delta(bl)$ guarantees redundancy-freeness. The next step consists of retracting all literals of decision levels not smaller than bl from $\mathbf{A}$ (Line 34)

Algorithm 2. CDNL-PROJECTION

Input : A set Δ of nogoods, a set P of variables, and a number s of requested solutions.

```
1  A ← ∅                                                          // assignment
2  ∇ ← ∅                                                          // set of (dynamic) nogoods
3  dl ← bl ← 0                                                    // decision and (systematic) backtracking level
4  loop
5      A ← BOOLEANCONSTRAINTPROPAGATION(Δ ∪ ∇, A)
6      if ε ⊆ A for some ε ∈ Δ ∪ ∇ then                            // conflict
7          if dl = 0 then exit
8          else if dl = bl then
9              Δ ← Δ \ {δ(bl)}                                    // remove for polynomial space complexity
10             σ_d ← dliteral(bl)
11             A ← A \ {σ ∈ A | dlevel(σ) = bl}
12             dlevel(σ̄_d) ← dl ← bl ← (bl − 1)
13             A ← A ∘ σ̄_d
14         else
15             (δ, k) ← CONFLICTRESOLUTION(ε, Δ ∪ ∇, A)
16             ∇ ← ∇ ∪ {δ}
17             dl ← max{k, bl}
18             A ← A \ {σ ∈ A | dlevel(σ) > dl}
19     else if {δ ∈ Δ | ∃σ ∈ δ : σ̄ ∈ A} = Δ then                  // solution
20         S ← {σ_p ∈ A | var(σ_p) ∈ P}
21         print S
22         s ← s − 2^(|P|−|S|)
23         if s ≤ 0 or max{dlevel(σ_p) | σ_p ∈ S} = 0 then exit
24         else if max{dlevel(σ_p) | σ_p ∈ S} = bl then
25             Δ ← Δ \ {δ(bl)}                                    // remove for polynomial space complexity
26             σ_d ← dliteral(bl)
27             A ← A \ {σ ∈ A | dlevel(σ) ≥ bl}
28             dlevel(σ̄_d) ← dl ← bl ← (bl − 1)
29             A ← A ∘ σ̄_d
30         else
31             bl ← bl + 1
32             δ(bl) ← S
33             Δ ← Δ ∪ {δ(bl)}                                    // record solution (temporarily)
34             A ← A \ {σ ∈ A | dlevel(σ) ≥ bl}
35             let σ_d ∈ δ(bl) \ A in
36                 dliteral(bl) ← σ_d
37                 dlevel(σ_d) ← dl ← bl
38                 A ← A ∘ σ_d
39     else
40         σ_d ← SELECT(Δ, ∇, A, P)                               // decision
41         dlevel(σ_d) ← dl ← (dl + 1)
42         A ← A ∘ σ_d
```

to make a clean cut on some unassigned literal σ_d (selected in Line 35) from $\delta(bl)$. Recall Fact 2 telling us that flipping σ_d may eliminate non-redundant projective solutions, hence, it is taken unflipped as decision literal of bl (Line 36–38). In summary, the reassignment of a literal σ_d from $\mathbf{S}$ makes sure that not yet enumerated projective solutions are not excluded by $\mathbf{A}$ (for completeness), while the temporary inclusion of $\delta(bl)$ in Δ prohibits a recomputation of $\mathbf{S}$ (for redundancy-freeness and termination). In the subsequent iterations, Algorithm 2 first exhausts the search space for further projective solutions including σ_d, and afterwards flips σ_d to $\overline{\sigma_d}$ along with removing the then satisfied nogood $\delta(bl)$ from Δ (for polynomial space complexity). In fact, such a systematic backtracking step is performed (in Line 25–29) if the maximum decision level of literals in $\mathbf{S}$ is bl (tested in Line 24), which means that the decision literal σ_d of bl (marked before in Line 36 and recalled in Line 26) must now be flipped for enumerating any further projective solutions. Finally, note that complement $\overline{\sigma_d}$ is assigned (in Line 29) at decision level $(bl - 1)$ or the new systematic backtracking level (cf. Line 28), respectively. As a matter of fact, there is no nogood in $\Delta \cup \nabla$ that implies $\overline{\sigma_d}$, so that conflict resolution (as in Line 15) cannot be applied at the new systematic backtracking level.

Conflict. As before, a conflict at decision level 0 means that there are no (further) projective solutions (Line 7). Otherwise, we now distinguish two cases: a conflict at systematic backtracking level bl (Line 9–13) or beyond bl (Line 15–18). As mentioned above, a conflict at bl cannot be analyzed because of literals in $\mathbf{A}$ lacking a reason in $\Delta \cup \nabla$. In fact, any conflict at bl is caused by $\delta(bl)$ or flipped decision literal(s) $\overline{\sigma_d}$ such that σ_d belongs to previously computed projective solutions. Unlike in Algorithm 1, such projective solutions are no longer available in Δ, and the mere reason for the presence of $\overline{\sigma_d}$ in $\mathbf{A}$ is that the search space of σ_d has been exhausted. Thus, a conflict at bl is not analyzed, and systematic backtracking proceeds as with projective solutions located at bl (compare Line 9–13 with Line 25–29). On a conflict beyond bl, conflict resolution (Line 15) returns a dynamic nogood δ (recorded in Line 16), as with Algorithm 1. The modification consists of restricting backjumping (in Line 18) to neither retracting bl nor any smaller decision level (Line 17), even if δ is asserting at a decision level $k < bl$. Note that such an assertion reassigns some literal of previously computed solutions. If this leads to a conflict, $\delta(bl)$ or some flipped literal $\overline{\sigma_d}$ at bl is involved. Then, both are retracted by a systematic backtracking step in the next iteration.

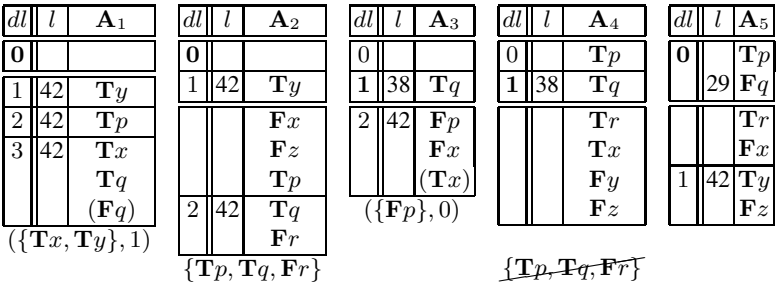


Fig. 1. Trace of Algorithm 2

For illustration, consider Figure 1 giving a trace of Algorithm 2 on (nogoods resulting from) the rules in (1)–(12) and $P = \{p, q, r\}$. We give all five assignments $\mathbf{A}_i$ yielding either a conflict or a solution; a resulting nogood is shown below. Column dl provides the decision levels of literals in $\mathbf{A}_i$, where the systematic backtracking level bl is given in $\mathbf{bold}$. For (flipped) decision literals, column l provides the line of Algorithm 2 in which the literal has been assigned; all other literals are inferred by propagation (in Line 5). For simplicity, we do not include variables for bodies of the rules in (1)–(12) in $\mathbf{A}_i$, but note that such variables are functionally dependent. Tracing Algorithm 2, successive decisions on $\mathbf{T}y$, $\mathbf{T}p$, and $\mathbf{T}x$ give rise to the conflicting assignment $\mathbf{A}_1$ by propagation. While $\mathbf{T}q$ is needed for deriving x from the rule in (1), complementary literal $\mathbf{F}q$ is mandatory for deriving y from the rule in (5). This makes us enter conflict resolution (in Line 15), yielding nogood $\{\mathbf{T}x, \mathbf{T}y\}$ and decision level 1 to jump back to. Hence, assignment $(\mathbf{T}y)$ constitutes the basis of $\mathbf{A}_2$. Propagating with $\{\mathbf{T}x, \mathbf{T}y\}$ gives $\mathbf{F}x$; further propagation and decision literal $\mathbf{T}q$ lead to solution $\mathbf{A}_2 = (\mathbf{T}y, \mathbf{F}x, \mathbf{F}z, \mathbf{T}p, \mathbf{T}q, \mathbf{F}r)$, whose projective solution, $\{\mathbf{T}p, \mathbf{T}q, \mathbf{F}r\}$, is printed (in Line 21). At this point, our proceeding starts to deviate from standard CDNL. Given that the maximum decision level 2 of literals $\mathbf{T}p$, $\mathbf{T}q$, and $\mathbf{F}r$ lies above $\mathbf{0}$, we store $\{\mathbf{T}p, \mathbf{T}q, \mathbf{F}r\}$ (in Line 33) to avoid computing answer sets comprising the same projective solution. Afterwards, selecting $\mathbf{T}q$ at the new systematic backtracking level $\mathbf{1}$ makes us first explore further projective solutions containing $\mathbf{T}q$. Assignment $(\mathbf{T}q)$ is then extended to conflicting assignment $\mathbf{A}_3$, and conflict resolution results in the addition of nogood $\{\mathbf{F}p\}$, effective at decision level 0. Nonetheless, the systematic backtracking and decision level remain at $\mathbf{1}$, and further propagation yields solution $\mathbf{A}_4$, comprising projective solution $\{\mathbf{T}p, \mathbf{T}q, \mathbf{T}r\}$. The fact that all its literals are established at $\mathbf{1}$ indicates the exhaustion of the search space for $\mathbf{T}q$. Hence, the projective solution at hand is not recorded, while $\{\mathbf{T}p, \mathbf{T}q, \mathbf{F}r\}$, associated with systematic backtracking level $\mathbf{1}$, is removed to stay in polynomial space. All literals assigned at $\mathbf{1}$ are then retracted from $\mathbf{A}_4$ (in Line 27). Finally, the systematic backtracking level is decremented and the search directed to projective solutions with $\mathbf{F}q$. The construction of solution $\mathbf{A}_5$ thus starts with $\mathbf{T}p$ and $\mathbf{F}q$ at the new systematic backtracking level $\mathbf{0}$ and ends after printing the corresponding projective solution $\{\mathbf{T}p, \mathbf{F}q, \mathbf{T}r\}$. Notably, $\mathbf{A}_5$ still contains a decision literal, $\mathbf{T}y$, but flipping it cannot lead to any further projective solutions.

We conclude this section by providing formal properties of Algorithm 2. As with Algorithm 1, soundness is clear due to verifying solutions (in Line 19) before printing anything. Termination and redundancy-freeness are obtained from the fact that enumerated projective solutions are excluded either by temporarily storing them (in Line 33) or by flipping some of their literals (in Line 13 or 29) upon systematic backtracking. Finally, completeness is guaranteed because temporarily stored projective solutions do not exclude not yet enumerated ones, while a systematic backtracking step is applied only if no further projective solutions are left beyond bl. Notably, any dynamic nogood derived by resolving with temporarily stored projective solutions $\mathbf{S}$ (in Line 15) is universally valid: since the literals of $\mathbf{S}$ are not to be reestablished in the future, $\mathbf{S}$ is indeed a nogood. Given the above considerations, we conclude that Theorem 1, 2, 3, and 4 remain valid if replacing CDNL-RECORDING in their statements with CDNL-PROJECTION. Beyond this, Algorithm 2 runs in polynomial space, in view of the fact that there cannot

be more temporarily stored projective solutions and asserting dynamic nogoods than literals in $\mathbf{A}$, while all other dynamic nogoods can be deleted at any time. However, it would be unfair to claim that the exponential savings in space complexity come without a cost: an introduced systematic backtracking level can only be retracted by a systematic backtracking step (in either Line 11 or 27), while backjumping (cf. Line 17–18) and optional restarts must leave all decision levels up to bl intact for not losing progress information.[2] However, systematic backtracking levels are introduced only after finding projective solutions, so that negative proof complexity results for procedures restricting decisions a priori [14] do not apply to Algorithm 2.

4 Experiments

We implemented our approach to solution projection within the ASP solver *clasp* (1.2.0-RC3; [4]). Our experiments consider *clasp* using four different types of enumeration: (a) its standard solution enumeration mode [11]; (b) enumeration by appeal to standard solution recording; (c) projective solution recording; (d) projective solution enumeration. Moreover, we implemented and evaluated two refinements of Algorithm 2 differing in the way selections are made in Line 35 and 40, respectively. Variant (d[h]) uses *clasp*'s BerkMin-like decision heuristic to select σ_d in Line 35 (without sign selection); otherwise, simply the first unassigned literal in $\delta(bl)$ is selected. Variant (d[p]) makes use of *clasp*'s progress saving option to direct the choice of σ_d in Line 40. Progress saving enforces sign selection according to the previously assigned truth value and thus directs search into similar search spaces as visited before (cf. [15]). Variant (d[hp]) combines both features, while (d[]) uses none of them. We refrained from testing further solvers because, to the best of our knowledge, no ASP nor SAT solver features the redundancy-free computation of projective solutions. Furthermore, ASP solvers enumerate standard solutions either via systematic backtracking, e.g., *smodels*, or like SAT solvers via solution recording, e.g., *cmodels*. The latter strategy is subsumed by *clasp* variant (b), while the former has in [11] been shown to have no edge over variant (a). Also note that we did not implement any decision heuristic specialized to preferring projected variables, as it had required another customization of *clasp*. All experiments were run on a 3.4GHz PC under Linux, each individual run restricted to 1000s time and 1GB RAM.[3]

In Table 1 and 2, we investigate the relative performance of the different enumeration approaches in terms of the proportion of projected variables. To this end, we consider two highly combinatorial benchmarks, the 11/11-Pigeons "problem" and the 15-Queens puzzle. For both of them, we gradually increase the number of projected variables (in columns #var), viz., the number of monitored pigeons or queens, respectively. The number of obtained projective solutions is given in columns #sol; the two last ones give the number of standard solutions. Columns (a)–(d[hp]) provide the runtimes of the different *clasp* variants in seconds; "$>$1000" stands for timeout. Note that #var and #sol do not affect (a) and (b), which always (attempt to) enumerate all standard solutions. At the bottom of Table 1 and 2, row $\varnothing$ provides the average runtime of each *clasp* variant.

[2] This is similar to the enumeration algorithm for non-projected solutions in [11].

[3] The benchmarks are available at: `http://www.cs.uni-potsdam.de/clasp/`

Table 1. Benchmark Results: 11/11-Pigeons

#var	#sol	(a)	(b)	(c)	(d[])	(d[h])	(d[p])	(d[hp])
1	11	100.38	>1000	0.01	0.01	0.01	0.01	0.01
2	110	100.38	>1000	0.01	0.01	0.01	0.01	0.01
3	990	100.38	>1000	0.05	0.07	0.06	0.07	0.07
4	7920	100.38	>1000	0.60	0.35	0.34	0.35	0.35
5	55440	100.38	>1000	9.08	1.67	1.68	1.61	1.67
6	332640	100.38	>1000	281.05	6.34	6.32	6.50	6.34
7	1663200	100.38	>1000	>1000	20.63	20.17	21.04	20.39
8	6652800	100.38	>1000	>1000	49.97	51.20	50.10	49.18
9	19958400	100.38	>1000	>1000	88.77	88.73	89.63	91.18
10	39916800	100.38	>1000	>1000	114.17	119.36	119.12	114.82
11	39916800	100.38	>1000	>1000	114.30	113.92	116.80	118.83
∅		100.38	>1000	480.98	36.03	36.53	36.84	36.62

Table 2. Benchmark Results: 15-Queens

#var	#sol	(a)	(b)	(c)	(d[])	(d[h])	(d[p])	(d[hp])
1	15	243.14	773.57	0.01	0.02	0.01	0.02	0.01
2	182	243.14	773.57	0.08	0.08	0.08	0.14	0.12
3	1764	243.14	773.57	0.79	0.63	0.66	1.47	1.37
4	13958	243.14	773.57	11.69	5.79	6.08	10.91	11.51
5	86360	243.14	773.57	158.40	40.71	43.71	63.76	69.88
6	369280	243.14	773.57	454.33	153.49	168.46	219.87	226.75
7	916096	243.14	773.57	>1000	331.42	357.31	444.69	437.23
8	1444304	243.14	773.57	>1000	463.46	461.78	584.59	542.46
9	1795094	243.14	773.57	>1000	512.19	523.86	652.37	577.66
10	2006186	243.14	773.57	>1000	528.36	436.70	647.49	478.34
11	2133060	243.14	773.57	>1000	525.23	407.40	616.43	450.80
12	2210862	243.14	773.57	>1000	516.56	357.22	552.67	384.30
13	2254854	243.14	773.57	>1000	462.83	322.50	496.17	356.18
14	2279184	243.14	773.57	>1000	413.72	283.82	432.62	327.35
15	2279184	243.14	773.57	>1000	250.13	250.06	245.97	249.11
∅		243.14	773.57	641.69	280.31	241.31	331.28	274.20

Looking into Table 1, it is apparent that variant (b) and (c), persistently recording either standard or projective solutions, do not scale. For the last problem solved by (c), projecting to 6 out of 11 pigeons, the ratio of standard to projective solutions is 120. Furthermore, all variants of (d) are faster than standard solution enumeration (a) up to 9 out of 11 pigeons, at which point there are twice as many standard as projective solutions. For 10 and 11 pigeons, variant (a) is a bit faster than (d). In fact, (a) saves some overhead by not distinguishing projected variables within solutions. Finally, there are no significant differences between the variants of (d), given that the underlying problem is fully symmetric.

With the 15-Queens puzzle in Table 2, search becomes more important than with 11/11-Pigeons. Due to the reduced number of solutions, standard solution recording (b)

now completes in less than 1000s, even though it is still slower than all enumeration schemes without persistent recording. We also see that projective solution recording (c) is the worst approach. In fact, its recorded projective solutions consist of #var literals each, while (b) stores decision literals whose number decreases the more solutions have been enumerated. For the variants of (d), we see that the number of projective solutions does not matter that much beyond 7 queens. Rather, heuristic aspects of the search start to gain importance, and variant (d[h]), which aims at placing the most critical queen first, has an edge. In contrast, progress saving alone here tends to misdirect search, as witnessed by (d[p]). Finally, (a) enumerating standard solutions becomes more efficient than (d) from 7 queens on, where the ratio of standard to projective solutions is about 2.5. As with 11/11-Pigeons, the reason is less overhead; in particular, (a) does not even temporarily store any nogoods for excluding enumerated solutions. The reconvergence between (a) and variants of (d) at 15 queens is by virtue of an implementation trick: if the decision literal at level $(bl + 1)$ in a solution (cf. Line 31–38 in Algorithm 2) is over a variable in P, then *clasp* simply increments bl and backtracks like in Algorithm 1 (Line 19). This shortcut permits unassigning fewer variables.

The benchmarks in Table 3 belong to three different classes. The first one deals with finding Hamiltonian cycles in clumpy graphs containing n clumps of n vertices each. For each value of n, we average over 11 randomly generated instances. Note that, due to high connectivity within clumps, clumpy graphs typically allow for a vast number of Hamiltonian cycles, but finding one is still difficult for systematic backtracking methods. In our experiments, we project Hamiltonian cycles to the edges connecting different clumps, thus, reducing the number of distinct solutions by several orders of magnitude. Second, we study benchmarks stemming from consistency checks of biological networks [16]. The five categories, each containing 30 randomly generated yet biologically meaningful instances, are distinguished by the number n of vertices in a network. The task is to reestablish consistency by flipping observed variations (increase or decrease) of vertices. Solutions are then projected to the vertices whose variations have been flipped, while discarding witnesses for the consistency of the repaired network. After a repair, there are typically plenty of witnesses, so that the number of projective solutions is several orders of magnitude smaller than that of standard ones. The third class considers a variation of Ravensburger's Labyrinth game on quadratic boards with n rows and n columns, each size comprising 20 randomly generated configurations. The idea is that an avatar is guided from a starting to a goal position by moving the rows and columns of the board as well as the avatar itself, and projection consists of disregarding the moves of the avatar. It turns out that Labyrinth instances are pretty difficult to solve, and usually there are not many more standard than projective solutions.

Table 3 shows average runtimes and numbers of timeouts per benchmark category; timeouts are taken as maximum time, viz., 1000s. The rows with $\varnothing/\Sigma$ provide the average runtime and sum of timeouts for each *clasp* variant over all instances of a benchmark class and in total, respectively. For the clumpy graphs and biological networks, denoted by Clumpy and Repair in Table 3, there are far too many standard solutions to enumerate them all with either (a) or (b). Even on the smallest category of Clumpy, (a) and (b) already produce timeouts, while enumerating projective solutions with (c) or (d) is unproblematic. On the larger Clumpy categories, there is no clear winner among (c) and

Table 3. Benchmark Results: Clumpy,Repair, and Labyrinth

Benchmark	n	(a)	(b)	(c)	(d[])	(d[h])	(d[p])	(d[hp])
Clumpy	08	204.50/02	468.48/05	0.02/0	0.02/0	0.02/0	0.02/0	0.02/0
	18	>1000/11	>1000/11	99.65/1	104.43/1	105.18/1	81.31/0	79.72/0
	20	>1000/11	>1000/11	255.04/2	254.80/2	313.22/1	219.05/1	118.95/0
	21	>1000/11	>1000/11	603.74/6	612.33/6	619.37/6	396.47/4	318.04/3
	22	>1000/11	>1000/11	144.64/1	266.72/2	275.54/2	410.98/4	321.07/3
$\varnothing/\Sigma$		840.90/46	893.70/49	220.62/10	247.66/11	262.67/10	221.57/9	167.56/6
Repair	2000	>1000/30	>1000/30	126.81/0	118.43/0	118.69/0	113.04/0	112.79/0
	2500	>1000/30	>1000/30	232.57/2	223.07/2	223.37/2	217.17/2	216.22/2
	3000	>1000/30	>1000/30	404.75/6	386.70/5	387.39/5	377.74/5	378.18/5
	3500	>1000/30	>1000/30	322.10/6	312.76/6	312.72/6	306.93/6	306.67/6
	4000	>1000/30	>1000/30	424.23/7	409.50/7	409.84/7	400.44/7	399.78/7
$\varnothing/\Sigma$		>1000/150	>1000/150	302.09/21	290.09/20	290.40/20	283.06/20	282.73/20
Labyrinth	16	52.49/0	58.46/1	59.69/1	61.72/1	59.03/1	61.54/1	59.11/1
	17	165.15/2	162.60/2	198.32/2	220.13/2	196.83/3	220.26/3	198.25/3
	18	212.59/2	218.90/2	289.84/4	298.56/3	253.06/3	286.05/3	257.38/3
	19	238.24/4	241.26/4	260.63/4	266.96/5	245.83/4	264.68/5	250.90/4
	20	319.67/5	324.43/5	355.48/6	359.51/7	343.47/6	360.33/7	346.13/6
$\varnothing/\Sigma$		197.63/13	201.13/14	232.79/17	241.38/18	219.64/17	238.57/19	222.35/17
Total $\varnothing/\Sigma$		708.24/209	718.91/213	264.68/48	266.47/49	262.20/47	257.39/48	242.17/43

the variants of (d), taking also into account that difficulty and number of projective solutions vary significantly over instances. However, it appears that progress saving (d[p]) and its combination with heuristic (d[hp]) tend to help. In the Repair categories, there are hardly any differences between the variants of (d), and projective solution recording (c) is competitive too. Finally, on Labyrinth, non-projecting enumeration approaches (a) and (b) have an edge on projecting ones. This is not a surprise because there not many more standard than projective solutions here. The disadvantages of projective solution enumeration are still not as drastic as their advantages are on other benchmarks. Among the different (d) variants, the use of a heuristic slightly promotes (d[h]), while progress saving alone (d[p]) is not very helpful. Finally, the last row in Table 3 shows that, over all instances, projective solution enumeration variants are not far away from each other, even though (d[hp]) has a slight advance. In fact, enumeration can benefit from the incorporation of search techniques, such as a heuristic or progress saving. Their usefulness, however, depends on the particular benchmark class, so that fine-tuning is needed. Importantly, the enumeration of all projective solutions may still be possible when there are far too many standard solutions, which can be crucial for the feasibility of applications.

5 Discussion

Answer set projection is already supported by almost all ASP systems, given that `hide` and `show` directives are available in the input language. However, up to now, no ASP system was able to enumerate projective solutions without duplicates. Rather, the

existing solvers exhaustively enumerate the entire set of solutions and merely restrict the output to visible atoms. This is accomplished either via systematic backtracking or via solution recording; the latter is also done by SAT solvers. To the best of our knowledge, the first dedicated solution enumeration algorithm that integrates with CDNL in polynomial space was proposed in [11] in the context of ASP; cf. variant (a) in Section 4. This algorithm turned out to be competitive for exhaustive solution enumeration, but it cannot be used for redundancy-free solution projection in view of the arguments given in Section 3. Although our new technique has also been implemented for ASP, it is readily applicable in neighboring areas dealing with Boolean or (with the necessary adaptions) even general constraints.

From a user's perspective, the sometimes intolerable redundancy of exhaustive solution enumeration necessitates the development of wrappers feeding projections of computed solutions as constraints back into a solver. For instance, such a workaround was originally used for the diagnosis task in [16] where certificates are required for solutions. These certificates, however, do neither belong to a projective solution nor can the resulting symmetries be broken by hand. The sketched approach boils down to projective solution recording, which does not scale because of exponential space complexity. If there are too many (projective) solutions to store them all, it is of course also impossible for a user to inspect each of them individually. However, if one is interested in counting occurrences of (combinations of) literals within solutions, enumerating more solutions than can be stored explicitly is tolerable. To enable it, the duplicate-free enumeration of solutions projected to relevant parts is crucial. Finally, abolishing the need of developing wrappers to cut redundancies is already something that should help users to concentrate on the interesting aspects of their applications.

Acknowledgements. This work was funded by DFG under grant SCHA 550/8-1.

References

1. Marques-Silva, J., Sakallah, K.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
2. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. DAC 2001, pp. 530–535. ACM Press, New York (2001)
3. Mitchell, D.: A SAT solver primer. Bulletin of the EATCS 85, 112–133 (2005)
4. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In: Proc. IJCAI 2007, pp. 386–392. AAAI Press, Menlo Park (2007)
5. Davies, J., Bacchus, F.: Using more reasoning to improve #SAT solving. In: Proc. AAAI 2007, pp. 185–190. AAAI Press, Menlo Park (2007)
6. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Cambridge (2003)
7. Dechter, R.: Constraint Processing. Morgan Kaufmann, San Francisco (2003)
8. Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM 7, 201–215 (1960)
9. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM 5, 394–397 (1962)
10. Giunchiglia, E., Maratea, M.: Solving optimization problems with DLL. In: Proc. ECAI 2006, pp. 377–381. IOS Press, Amsterdam (2006)

11. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set enumeration. In: Baral, C., Brewka, G., Schlipf, J. (eds.) LPNMR 2007. LNCS, vol. 4483, pp. 136–148. Springer, Heidelberg (2007)
12. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: Proc. ICCAD 2001, pp. 279–285. IEEE Press, Los Alamitos (2001)
13. Ryan, L.: Efficient algorithms for clause-learning SAT solvers. MSc's thesis, SFU (2004)
14. Järvisalo, M., Junttila, T.: Limitations of restricted branching in clause learning. Constraints (to appear), http://www.tcs.hut.fi/~mjj/
15. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisability solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 294–299. Springer, Heidelberg (2007)
16. Gebser, M., Schaub, T., Thiele, S., Usadel, B., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. In: Garcia De La Banda, M., Pontelli, E. (eds.) ICLP 2008. LNCS, vol. 5366, pp. 130–144. Springer, Heidelberg (2008)

k-Clustering Minimum Biclique Completion via a Hybrid CP and SDP Approach

Stefano Gualandi

Politecnico di Milano, Dipartimento di Elettronica e Informazione
stefano.gualandi@polimi.it

Abstract. This paper presents a hybrid Constraint Programming (CP) and Semidefinite Programming (SDP) approach to the k-clustering minimum biclique completion problem on bipartite graphs. The problem consists in partitioning a bipartite undirected graph into k clusters such that the sum of the edges that complete each cluster into a biclique, *i.e.*, a complete bipartite subgraph, is minimum. The problem arises in telecommunications, in particular in bundling channels in multicast transmissions. In literature, the problem has been tackled with an Integer Bilinear Programming approach. We introduce two quasi-biclique constraints and we propose a SDP relaxation of the problem that provides much stronger lower bounds than the Bilinear Programming relaxation. The quasi-biclique constraints and the SDP relaxation are integrated into a hybrid CP and SDP approach. Computational results on a set of random instances provide further evidence about the potential of CP and SDP hybridizations.

1 Introduction

The k-Clustering Minimum Biclique Completion (k-CmBC) problem consists in partitioning an undirected bipartite graph into k clusters such that the sum of the edges that complete each cluster into a biclique, *i.e.*, a complete bipartite subgraph, is minimum. While the problem of covering undirected graphs by bicliques has been widely studied in the literature for its connections to factorization problems of 0/1 matrices (e.g, for a survey see [1]), the k-CmBC problem has received little attention. This combinatorial optimization problem is NP-hard even for $k = 2$, as proven in [2]. In literature, it has been tackled only by an Integer Bilinear Programming approach.

Constraint Programming has proved to be a successful programming paradigm to solve pure combinatorial optimization problems, such as, for instance, the maximum clique problem [3],[4], and the minimum graph coloring problem [5]. The success of applying Constraint Programming to pure combinatorial optimization problems relies on the design of cost-based filtering algorithms, introduced in [6]. The integration of Constraint Programming and Semidefinite Programming is pioneered in [7], where a Semidefinite Programming relaxation is exploited within a Constraint Programming solver for solving the maximum clique problem. The Semidefinite Programming relaxation is used to derive a

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 87–101, 2009.

tight lower bound at the root node of the CP search tree, and to design an effective labeling heuristic by interpreting the solution of the Semidefinite Programming relaxation as the likelihood that a variable get assigned a given value in the optimal solution. Semidefinite Programming relaxations for the MAX-SAT problem are compared to Integer Linear Programming relaxations in [8], where the computational results show that the Semidefinite Programming relaxation provides stronger bounds and is very effective when used as guidance in the labeling heuristic.

The main contributions of this paper are to introduce two optimization constraints, the *quasi-biclique* and the *one-shore-induced quasi-biclique* constraints, and to present a new Semidefinite Programming relaxation of the k-CmBC problem. The *quasi-biclique* constraint forces the subgraph induced by a subset of vertices to be a bipartite graph completed into a biclique by a number of additional edges. The *one-shore-induced quasi-biclique* has the additional requirement that the second shore is equal to the union of the neighborhoods of the vertices of the first shore. These constraints are closely related to the maximum clique constraint introduced in [3] and improved in [4]. The proposed SDP relaxation is based on an interpretation of the problem as a Max-Cut problem on the complementary bipartite graph, and is closely related to the relaxations of the Max-Cut problem proposed in [9] and of the Max-k-Cut problem proposed in [10]. The motivation for developing the SDP relaxation is that it provides much stronger lower bounds than the bilinear programming relaxation.

The new optimization constraints and the SDP relaxation are integrated into a hybrid CP and SDP approach that extends to clustering problems the hybrid method introduced in [7]. The values of the SDP relaxation are interpreted as the likelihood that two vertices belong to the same cluster, yielding a labeling heuristic that selects a pair of values and tries to assign the two values to the same set variable. In addition, we propose a method to solve the SDP relaxation at every node of the CP search tree that produces solution having a small gap with the optimum. Computational results on a set of random instances generated as in [2] show that our hybrid approach is competitive with the existing Integer Bilinear Programming approach, and provide further evidence about the potential of CP and SDP hybridizations.

The outline of this paper is as follows. Section 2 defines the problem, and presents an Integer Bilinear Programming formulation. Section 3 and Section 4 introduce respectively the CP formulation and the SDP relaxation of the k-CmBC problem. Section 5 presents the integration of the CP model and the SDP relaxation. Section 6 discusses the computational results, and Section 7 concludes the paper presenting future works.

1.1 Notation

Let $G = (S, T, E)$ be an undirected bipartite graph. The **complementary graph** of a bipartite graph is $\bar{G} = (S, T, \bar{E})$, with $\bar{E} = \{(i, j) \mid i \in S, j \in T, (i, j) \notin E\}$. The bipartite subgraph induced by two subsets $S' \subseteq S$ and

$T' \subseteq T$ is denoted by $G[S', T'] = (S', T', E')$, where $E' = \{(i, j) \mid i \in S', j \in T', (i, j) \in E\}$. The neighborhood of a vertex i is denoted by $N(i)$, and the degree by $\delta(i) = |N(i)|$. The **one-shore-induced** bipartite subgraph induced by a subset $S'' \subseteq S$ is denoted by $G[S''] = (S'', T'', E'')$, where $T'' = \bigcup_{i \in S''} N(i)$, and $E'' = \{(i, j) \mid i \in S'', j \in T'', (i, j) \in E\}$. The degree of vertex i in the complementary graph $\bar{G}$ is denoted by $\delta_{\bar{G}}(i)$.

A **biclique** is a complete bipartite graph, that is $E = S \times T$. A **c-quasi-biclique**, or a quasi-biclique of cost c, is a bipartite graph that is completed into a biclique by c additional edges. The cost of the c-quasi-biclique $G = (S, T, E)$ is equal to $c = |S| \cdot |T| - |E| = |\bar{E}| = \sum_{i \in S} \delta_{\bar{G}}(i)$.

2 Problem Description

Let $G = (S, T, E)$ be a bipartite undirected graph, and k be the number of desired clusters. A cluster is defined as a bipartite subgraph of G, or equivalently as a c-quasi-biclique of G. The k-CmBC problem consists of partitioning the set of vertices S into k subsets S_i, with $i = 1, \ldots, k$, such that the sum of the cost c_i of each quasi-bicliques $G[S_i]$ is minimum.

Example 1. Figure 1.a shows a bipartite graph G with $S = \{1, \ldots, 4\}$ and $T = \{5, \ldots, 9\}$. Figure 1.b represents a possible 2-clustering induced by $S_1 = \{1, 2\}$ and $S_2 = \{3, 4\}$. The dashed edges belong to the one-shore-induced subgraph $G[S_1]$ and those in bold to $G[S_2]$. Note that vertex 9 belongs to both quasi-bicliques. The complementary graph $\bar{G}$ is shown in Fig. 1.c. The cost of this 2-clustering is equal to four, given by three edges of $\bar{G}[S_1]$ and one of $\bar{G}[S_2]$, that are respectively $(1, 9)$,$(2, 5)$, and $(2, 7)$ for the first biclique, and $(3, 9)$ for the second biclique.

The k-CmBC problem has a significant application in telecommunications, as shown in [2], for bundling channels in multicast transmissions. Given a set of

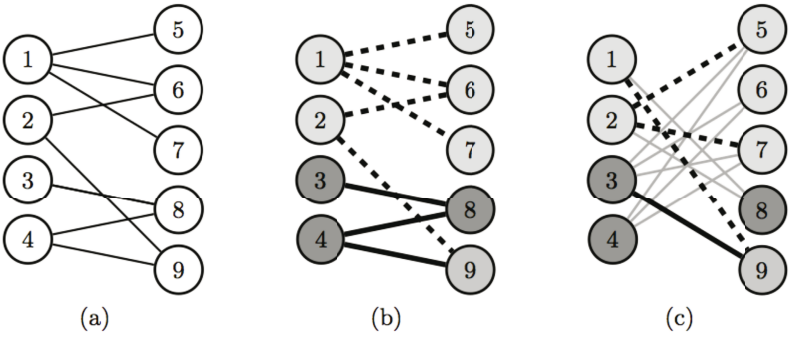


Fig. 1. An example of the k-CmBC problem for $k = 2$

demands of services from clients, the application consists of finding k multicast sessions that partition the set of demands. Each service has to belong to a single multicast session, while each client can appear in more sessions. This problem is represented on a bipartite graph $G = (S, T, E)$ as follows: every service i is represented by a vertex in S, and every client j by a vertex in T. The demand of a service i from a client j is represented by the edge $(i, j) \in E$. A c-quasi-biclique of G represents a multicast session that transmits c times an unrequested service. The cost c gives a measure of the waste of bandwidth of the corresponding multicast session. Solving the k-CmBC problem on this bipartite subgraph, is equivalent to finding k multicast sessions that minimizes the overall waste of bandwidth.

Integer Bilinear Programming formulation. Let x_{ip} and y_{jp} be 0/1 variables indicating whether the vertex $i \in S$ or the vertex $j \in T$ are in the cluster p. Let $K = \{1, \ldots, k\}$ be the set of the clusters. The Integer Bilinear Programming formulation of the k-CmBC problem is as follows:

$$w^* = \min \quad \sum_{p \in K} \sum_{(i,j) \in \bar{E}} x_{ip}\, y_{jp} \tag{1}$$

$$\text{s.t.} \quad \sum_{p \in K} x_{ip} = 1, \qquad \forall i \in S, \tag{2}$$

$$\sum_{p \in K} x_{ip}\, y_{jp} = 1, \qquad \forall (i,j) \in E, \tag{3}$$

$$x_{ip}, y_{jp} \in \{0,1\}, \qquad \forall i \in S, \forall j \in T, \forall p \in K. \tag{4}$$

The objective function (1) minimizes the number of edges that completes each induced bipartite subgraph into a biclique. Constraints (2) assign each vertex of the shore S to a single cluster. Constraints (3) force each neighbor j of a vertex i to be (also) in the same cluster of i.

The model (1)–(4) is linearized by introducing 0/1 variables z_{ijp} equal to 1 if the edge $(i, j) \in \bar{E}$ completes the p-th cluster into a biclique. The resulting Integer Linear Programming model is as follows:

$$w^* = \min \quad \sum_{p \in K} \sum_{(i,j) \in \bar{E}} z_{ijp} \tag{5}$$

$$\text{s.t.} \quad \sum_{p \in K} x_{ip} = 1, \qquad \forall i \in V, \tag{6}$$

$$x_{ip} + y_{jp} \leq 1 + z_{ijp}, \qquad \forall (i,j) \in \bar{E}, \forall p \in K, \tag{7}$$

$$x_{ip} - y_{jp} \leq 0, \qquad \forall (i,j) \in E, \forall p \in K, \tag{8}$$

$$x_{ip}, y_{jp} \in \{0,1\}, \qquad \forall i \in V, \forall j \in W, \forall p \in K, \tag{9}$$

$$z_{ijp} \in \{0,1\}, \qquad \forall (i,j) \in \bar{E}, \forall p \in K. \tag{10}$$

The new variable z_{ijp} and the constraints (7) are used to linearize the bilinear term $x_{ip}y_{jp}$ appearing in the objective function (1). Constraints (8) linearize the constraints (3).

The main limit of the formulation (5)–(10) is that any permutation of the indices p gives the same optimal solution. This issue is tackled in [2] by introducing symmetry-breaking constraints.

3 Constraint Programming Formulation

The CP formulation of the k-CmBC problem is based on two new optimization constraints: the **quasi-biclique** constraint and the **one-shore-induced quasi-biclique** constraint. The definitions of these two constraints are based on the notation used in the book [11]. A constraint $\mathbb{C}$ on the ordered set of variables $\mathcal{X}(\mathbb{C})$ is a subset $\mathcal{T}(\mathbb{C})$ of the Cartesian product between the domain of each variable that specifies the allowed combinations of values for the variables in $\mathcal{X}(\mathbb{C})$.

Definition 1. *A* **quasi-biclique constraint** *is a constraint $\mathbb{C}$ defined on a bipartite graph $G = (S, T, E)$, on two set variables $X \subseteq S$ and $Y \subseteq T$ and a finite domain integer variable c, with $0 \leq c \leq |E|$, and*

$$\begin{aligned}\mathcal{T}(\mathbb{C}) = \{&\tau \text{ s.t. } \tau \text{ is a tuple of } \mathcal{X}(\mathbb{C}) = [X, Y, c],\\ &\text{and } G[X, Y] = (X, Y, F),\\ &\text{and } F = \{(i, j) \mid i \in X, j \in Y, (i, j) \in E\},\\ &\text{and } c = |X| \cdot |Y| - |F|\}.\end{aligned}$$

It is denoted by $\texttt{qbiclique}(X, Y, c, G)$.

Definition 2. *A* **one-shore induced quasi-biclique constraint** *is a constraint $\mathbb{C}$ defined on a bipartite graph $G = (S, T, E)$, on two set variables $X \subseteq S$ and $Y \subseteq T$ and a finite domain integer variable c, with $0 \leq c \leq |E|$, and*

$$\begin{aligned}\mathcal{T}(\mathbb{C}) = \{&\tau \text{ s.t. } \tau \text{ is a tuple of } \mathcal{X}(\mathbb{C}) = [X, Y, c],\\ &\text{and } Y = \bigcup_{i \in X} N(i),\\ &\text{and } \texttt{qbiclique}(X, Y, c, G)\}.\end{aligned}$$

It is denoted by $\texttt{osi-qbiclique}(X, Y, c, G)$.

3.1 The CP Model

Let (X_p, Y_p) be a pair of finite domain integer set variables that represents the vertices of the p-th quasi-biclique. Let c_p be an integer variable representing the cost of the p-th quasi-biclique, *i.e.*, the number of added edges, and let d be an integer variable for the sum of the cost of each cluster. The CP formulation of the k-CmBC problem is as follows:

$$\textit{variables/domains:}\quad X_p \subseteq S, \qquad \forall p \in K, \tag{11}$$
$$Y_p \subseteq T, \qquad \forall p \in K, \tag{12}$$
$$0 \leq c_p \leq |\bar{E}|, \quad \forall p \in K, \tag{13}$$
$$0 \leq d \leq |\bar{E}|, \tag{14}$$
$$\textit{constraints:}\quad \texttt{partition}([X_1, \ldots, X_p], S), \tag{15}$$
$$\texttt{osi-qbiclique}(X_p, Y_p, c_p, G), \quad \forall p \in K, \tag{16}$$
$$d = \sum_{p \in K} c_p. \tag{17}$$

Constraints (15) force each vertex of the shore S to appear in a single cluster. Constraints (16) constrain each subgraph induced by a pair (X_p, Y_p) to be a quasi-biclique of cost equal to c_p. Constraint (17) sums up the cost of every cluster. The CP model (11)–(17) relies on the filtering algorithm used for the `osi-qbiclique` constraint, which is described next.

3.2 A Filtering Algorithm for the `osi-qbiclique` Constraint

Similarly to the maximum clique constraint introduced in [3], the `qbiclique` and `osi-qbiclique` constraints use a pair of *current sets* (C_1, C_2), with $C_1 \subseteq S$ and $C_2 \subseteq T$, for the vertices that belong to the current quasi-biclique $G[C_1, C_2]$, and a pair of *candidate sets* (P_1, P_2), with $P_1 \subseteq S \setminus C_1$ and $P_2 \subseteq T \setminus C_2$, for the vertices that could extend the current quasi-biclique. The filtering algorithms of these constraints remove from the pair of candidate sets (P_1, P_2) the vertices that cannot extend the pair (C_1, C_2) to a quasi-biclique of cost equal to c.

Lemma 1. *Let $G = (S, T, E)$ be a bipartite graph and let $G[C_1, C_2]$ be a quasi-biclique of G induced by $C_1 \subseteq S$ and $C_2 \subseteq T$, with cost $\bar{c}$. Let $P_1 \subseteq S$ and $P_2 \subseteq T$ be the candidate sets for C_1 and C_2. Let c be the required cost for the quasi-biclique. Then,*

1. *$\forall i \in P_1$ such that $\bar{c} + |C_2 \setminus N(i)| > c$, the vertex i cannot extend C_1 to obtain a c-quasi-biclique;*
2. *$\forall j \in P_2$ such that $\bar{c} + |C_1 \setminus N(j)| > c$, the vertex j cannot extend C_2 to obtain a c-quasi-biclique.*

Proof. (sketch for point 1.) If we added a vertex $i \in P_1$ to C_1, the lower bound of the cost of the biclique would increase of a factor $|C_2 \setminus N(i)|$, equal to the number of vertices of C_2 that are not adjacent to vertex i. □

Lemma 2. *Let $G = (S, T, E)$ be a bipartite graph and let $G[C_1]$ be the one-shore-induced quasi-biclique of G induced by $C_1 \subseteq S$, with cost $\bar{c}$. Let $P_1 \subseteq S$ be the candidate set for C_1, and let c be the required cost for the one-shore-induced quasi-biclique. Then, $\forall i \in P_1$ such that:*

$$\bar{c} \quad + \quad (|C_1| + 1) \cdot |N(i) \setminus C_2| + |C_2| - \delta(i) \quad > \quad c, \tag{18}$$

the vertex i cannot extend C_1 to a one-shore-induced quasi-biclique of cost c.

Algorithm 1. Sketch of the `osi-biclique` filtering algorithm.

Var (C_1, C_2) : pair of current sets
Var (P_1, P_2) : pair of candidate sets
Var $\bar{c}$: cost of the one-shore-induced quasi-biclique $G[C_1]$
Var c : desired cost
In G : bipartite graph

1: $\bar{c} \leftarrow |C_1| \cdot |C_2| - \sum_{i \in C_1} \delta(i)$
2: $P_2 \leftarrow \bigcup_{i \in C_1} N(i)$
3: **for all** $i \in P_1$ **do**
4: **if** $\bar{c} + (|C_1| + 1) \cdot |N(i) \setminus C_2| + |C_2| - \delta(i) > c$ **then** ▷ Apply Lemma 2
5: $P_1 \leftarrow P_1 \setminus \{i\}$
6: $P_2 \leftarrow P_2 \setminus \{N(i) \setminus P_2\}$
7: **end if**
8: **end for**

Proof. If the vertex i is added to C_1, then $N(i)$ is added to C_2, and the new quasi-biclique constraint is $\widehat{G} = (C_1 \cup \{i\}, C_2 \cup N(i), \widehat{E})$. Note that $|\widehat{E}| = \sum_{j \in C_1 \cup \{i\}} \delta(j)$. By definition, the cost $\hat{c}$ of the extended biclique is equal to:

$$\hat{c} = |C_1 \cup \{i\}| \cdot |C_2 \cup N(i)| - |\widehat{E}| = |C_1 \cup \{i\}| \cdot |C_2 \cup N(i)| - \sum_{j \in C_1 \cup \{i\}} \delta(j) =$$

$$= (|C_1| + 1) \cdot (|C_2| + |N(i) \setminus C_2|) - \left(\delta(i) + \sum_{j \in C_1} \delta(j) \right) =$$

$$= \underbrace{|C_1| \cdot |C_2| - \sum_{j \in C_1} \delta(j)}_{\text{this is equal to } \bar{c}} + (|C_1| + 1) \cdot |N(i) \setminus C_2| + |C_2| - \delta(i),$$

that is equal to the first term of (18). □

Algorithm 1 sketches the filtering algorithm of the `osi-qbiclique` constraint. For the sake of clarity, the filtering algorithm is described using the pairs of current sets (C_1, C_2) and of candidate sets (P_1, P_2), while the `osi-qbiclique` constraint is defined using two finite set variables X and Y. However, they are strictly related, since the greatest lower bound of the set variable X is equal to C_1, and the lowest upper bound is equal to $C_1 \cup P_1$, *i.e.*, $C_1 \subseteq X \subseteq C_1 \cup P_1$, and similarly, $C_2 \subseteq Y \subseteq C_2 \cup P_2$. The worst-case complexity of the filtering algorithm is $O(|S| \cdot |T|)$.

4 Semidefinite Programming Formulations

This section formulates the SDP relaxation of the k-CmBC problem. The proposed formulation is related to the SDP relaxations of the Max-Cut problem

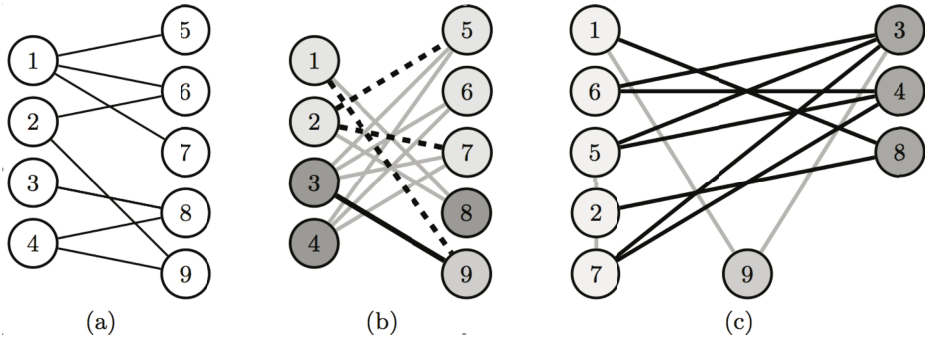


Fig. 2. (a) The k-CmBC problem with $k = 2$. (b) The complementary graph of an optimal solution. (c) The extended Max-Cut interpretation of the complementary graph: the edges in bold belong to the cut edges set. Vertex 9 is considered as it were in both subsets, since it belongs to two clusters.

proposed in [9] and of the Max-k-Cut problem proposed in [10]. Let us consider first the case $k = 2$, and then to extend the formulation to the case $k > 2$.

The k-CmBC problem minimizes the number of edges of the complementary graph that appear within the same cluster. This corresponds to maximize the number of edges of the complementary graph that are in the cut given by the two clusters $G[S_1]$ and $G[S_2]$. Therefore, for $k = 2$, the idea is to formulate a relaxation of the k-CmBC problem as an extended Max-Cut problem. Figure 2.b shows the optimal solution of the Example 1 and Fig. 2.c an extended Max-Cut interpretation. The two sets of vertices that belong to a single cluster represent the two shores of a solution of the Max-Cut problem. The vertices appearing in both clusters behave as they were in both shores, and their incident edges do not belong to the cut edge set. In Fig. 2.c the two shores of the cut are the sets $\{1, 2, 5, 6, 7\}$ and $\{3, 4, 8\}$, while vertex 9 behaves as it were in both shores.

4.1 Extended Max-Cut

Let x_i be a $\{-1, 1\}$ variable that indicates whether the vertex $i \in S$ is in the cluster S_1 if $x_i = 1$, or in the cluster S_2 if $x_i = -1$. If two vertices i and j are in the same shore, the product $x_i x_j$ of the corresponding variables is equal to 1, otherwise is equal to -1. Similarly to the Max-Cut model presented in [9], let z_{ij} be a $\{-1, 1\}$ variable equal to 1 if the complementary edge $(i, j) \in \bar{E}$ is not in the cut. An edge $(i, j) \in \bar{E}$ does not belong to the cut if another vertex l exists such that $l \in N(j)$ and l is in the same cluster of i, that is $x_i x_l = 1$. This is equivalent to set $z_{ij} = \max_{l \in N(j)}\{x_i x_l\}$, which is linearized by the following inequalities: $z_{ij} \geq x_i x_l, \forall l \in N(j)$.

Using the variables x_i and z_{ij}, the 2-CmBC problem is formulated as the following integer quadratic problem:

$$w_{mc} = \max \quad \frac{1}{2} \sum_{(i,j)\in\bar{E}} (1 - z_{ij}) \tag{19}$$

$$\text{s.t.} \quad z_{ij} \geq x_i x_l, \qquad \forall (i,j) \in \bar{E}, l \in N(j), \tag{20}$$

$$x_i \in \{-1, 1\}, \qquad \forall i \in S, \tag{21}$$

$$z_{ij} \in \{-1, 1\}, \qquad \forall (i,j) \in \bar{E}. \tag{22}$$

Property 1. The following relation holds: $w^* = |\bar{E}| - w_{mc}$.

We used the labeling technique proposed in [9] to derive an SDP relaxation of the problem (19)–(22). Every vertex i of $S \cup T$ is labeled with a unit vector $\mathbf{v}_i \in \mathbb{R}^{|S|+|T|}$. The geometric interpretation is that two vertices i and j are in the same cluster if the angle between them is small enough, that is, if $\mathbf{v}_i^t \mathbf{v}_j = 1$. Let V be a matrix such that column i is given by vector $\mathbf{v}_i$, and let $Z = V^t V$. Let $\mathbf{e}$ be the vector of all ones. The SDP relaxation of the 2-CmBC problem is as follows:

$$\max \quad \frac{1}{2} \sum_{(i,j)\in\bar{E}} (1 - Z_{ij}) \tag{23}$$

$$Z_{ij} \geq Z_{il}, \qquad \forall (i,j) \in \bar{E}, l \in N(j), \tag{24}$$

$$\text{diag}(Z) = \mathbf{e}, \tag{25}$$

$$Z \succeq 0. \tag{26}$$

Constraints (24) are equivalent to constraints (20), where each entry $(i,j) \in \bar{E}$ of Z_{ij} is used for variable z_{ij}, and each entry $(i,l) \in S \times S$ is used for the product $x_i x_l$. Together constraints (25)–(26) correspond to relax constraints (21) and (22) in $-1 \leq Z_{ij} \leq 1$.

4.2 Extended Max-k-Cut

We extend the model (23)–(26) to the case $k > 2$ by using a formulation similar to the Max-k-Cut formulation given in [10]. Let us consider k unit vectors $\mathbf{a}_1, \ldots, \mathbf{a}_k \in \mathbb{R}^{k-1}$ satisfying $\mathbf{a}_i^t \mathbf{a}_j = -\frac{1}{k-1}$, for $1 \leq i \neq j \leq k$. Let $\mathbf{x}_i$ be a real vector variable for each vertex $i \in S$, such that $\mathbf{x}_i \in \{\mathbf{a}_1, \ldots, \mathbf{a}_k\}$, *i.e.*, the variable $\mathbf{x}_i$ is equal to one of the $\mathbf{a}_i$ vectors. Note that in the case $k = 2$, we get $a_1 = -1$, $a_2 = 1$, and $x_i \in \{-1, 1\}$, as in 2-CmBC . Let two vertices i and j in S be in the same cluster if $\mathbf{x}_i^t \mathbf{x}_j = 1$, and be in different clusters if $\mathbf{x}_i^t \mathbf{x}_j = -\frac{1}{k-1}$. As for the case $k = 2$, the variable z_{ij} can take only two values: if the complementary edge (i,j) is not in the k-cut then $z_{ij} = 1$, otherwise $z_{ij} = -\frac{k}{k-1}$. The formulation of the k-CmBC problem is as follows:

$$w_{mkc} = \max \quad \frac{k-1}{k} \sum_{(i,j)\in\bar{E}} (1 - z_{ij}) \tag{27}$$

$$\text{s.t.} \quad z_{ij} \geq \mathbf{x}_i^t \mathbf{x}_l, \qquad \forall (i,j) \in \bar{E}, l \in N(j), \tag{28}$$

$$\mathbf{x}_i \in \{\mathbf{a}_1, \ldots, \mathbf{a}_k\}, \qquad \forall i \in S, \tag{29}$$

$$z_{ij} \in \{-\frac{k}{k-1}, 1\}, \qquad \forall (i,j) \in \bar{E}. \tag{30}$$

Constraints (28) are equivalent to constraints (20), and force to consider two no-adjacent vertices $i \in S$ and $j \in T$ as they were in the same cluster, if at least one neighbor $l \in N(j)$ of vertex j is in the same cluster as vertex i. The objective function has the correction factor $\frac{k-1}{k}$ that compensates the case in which the edge (i,j) is in the k-cut, since in this case $z_{ij} = \frac{-1}{k-1}$ and $(1 - \frac{-1}{k-1}) = \frac{k}{k-1}$.

The SDP relaxation of the k-CmBC problem for $k > 2$ is obtained using the same labeling technique as for $k = 2$, and it is as follows:

$$w_{sdp} = \max \quad \frac{k-1}{k} \sum_{(i,j)\in\bar{E}} (1 - Z_{ij}) \tag{31}$$

$$Z_{ij} \geq Z_{il}, \qquad \forall (i,j) \in \bar{E}, l \in N(j), \tag{32}$$

$$\text{diag}(Z) = \mathbf{e}, \tag{33}$$

$$Z_{ij} \geq -\frac{1}{k-1}, \qquad \forall i,j \in S \cup T, i \neq j, \tag{34}$$

$$Z \succeq 0. \tag{35}$$

Together constraints (33)–(35) relax constraints (29),(30) into $-\frac{1}{k-1} \leq Z_{ij} \leq 1$. This formulation is used to compute a lower bound of k-CmBC problem.

Property 2. Since $w_{sdp} \geq w_{mkc}$, the following relation holds: $w^* \geq |\bar{E}| - \lfloor w_{sdp} \rfloor$.

5 CP and SDP Integration

The integration of SDP relaxations within CP solvers was pioneered in [7] for tackling the maximum clique problem. The idea for exploiting the SDP relaxation was to solve the SDP relaxation once, and then to use the optimal solution in the CP solver for deriving a labeling heuristic and for bounding the cost-variable.

We extend the idea by proposing a different labeling heuristic, and by providing two graph transformations that allow to recompute the SDP relaxation within the CP search tree.

5.1 SDP-Based Labeling Heuristic

The SDP-based labeling heuristic given in [7] assumes implicitly that given an integer variable i, for each value v of its domain there is an entry in the solution matrix Z^*_{iv} of the SDP relaxation representing the likelihood that the variable i

Algorithm 2. SDP-based labeling heuristic for clustering problems.

In Z^* : optimal solution of an SDP relaxation
In $[X_1, \ldots, X_k]$: vector of set variables; each variable is a pair $X_p = (C_p, P_p)$
Out : a labeling basic constraint

1: $P \leftarrow \bigcup_{i \in K} P_i$
2: $(v, w) \leftarrow \text{argmax}_{i \in P \vee j \in P}\{|Z^*_{ij}| - \frac{k}{2(k-1)}\}$
3: $i \leftarrow \text{selectVariable}(X, v, w)$ ▷ v and/or w are in the candidate set P_i
4: **if** $Z^*_{vw} \geq \frac{k}{2(k-1)}$ **then**
5: * try to assign v **and** w to X_i *
6: **else**
7: * try to assign **either** v **or** w to X_i *
8: **end if**

is equal to v in an optimal solution. The higher is the value of Z^*_{iv}, the sooner the variable i gets labeled with value v in the CP search tree.

The main idea for extending the SDP-based labeling heuristic is to interpret the entries of Z^* as the likelihood that two vertices belong to the same cluster. The entries Z^*_{vw} range in the real interval $[-\frac{1}{k-1}, 1]$, which has the midpoint at $\frac{k}{2(k-1)}$. Given a set variable X_i and a pair of values v and w in its domain, the higher is the value of Z^*_{vw}, the higher is the likelihood that $\{v, w\} \in X_i$. The smaller is the value of Z^*_{vw}, the higher is the probability either $v \in X_i$ or $w \in X_i$. The closer the value of Z^*_{vw} is to $\frac{k}{2(k-1)}$, the more the membership of v and w is uncertain.

Algorithm 2 shows the labeling heuristic based on this interpretation of the SDP relaxation. First, the heuristic takes every value in the candidate sets. Second, it selects a pair of values (v, w) such that at least one of the two values is in P, and the distance of Z^*_{vw} to $\frac{k}{2(k-1)}$ is maximum. Third, it selects a set variable X_i having v and/or w in its domain. Finally, if $Z^*_{vw} - \frac{k}{2(k-1)}$ were positive, it tries to put v and w in the same set, otherwise in different sets.

5.2 SDP-Based Cost Pruning

In our approach, the SDP relaxation computes bounds at each node of the CP search tree. Two graph transformations map a partial solution to a weighted version of the k-CmBC problem. The first graph transformation is used when in the partial solution two vertices i and j are in the same cluster, while the second transformation is used when the two vertices are in different clusters. Let $G = (S, T, E)$ be a bipartite graph, and let $\widehat{X}$ be a partial solution of the k-CmBC problem. Recall that each element of $\widehat{X}$ has a current set and a candidate set, *i.e.*, $\widehat{X}_i = (C_i, P_i)$. Let $f(G, \widehat{X})$ denote the cost of the k-CmBC problem given the graph G and the partial assignment in $\widehat{X}$.

Proposition 1. *Given a candidate set C_i in $\widehat{X}$, the graph G is augmented with the edges of the induced complementary subgraph $\bar{G}[C_i]$, obtaining the new graph $G^{M(C_i)}$. The graph $G^{M(C_i)}$ is used to compute a lower bound of $\widehat{X}$ as follows:*

$$f(G,\widehat{X}) \geq w_{sdp}\left(G^{M(C_i)}\right) + \sum_{j\in C_i} \delta_{\bar{G}}(i). \tag{36}$$

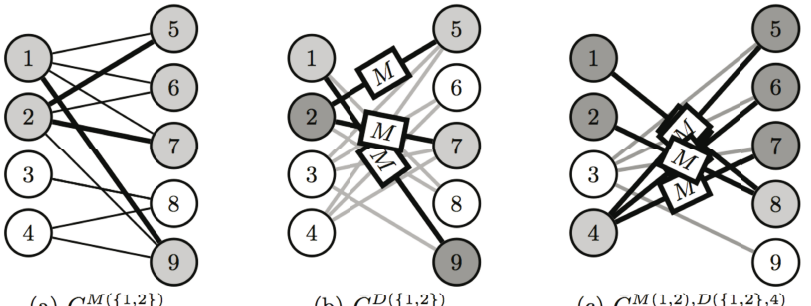
Proposition 2. *Given a pair of candidate sets C_i and C_j in $\widehat{X}$, the complementary graph $\bar{G}$ is transformed into a weighted graph $\bar{G}^{D(C_i,C_j)}$ by giving a weight equal to $M = |\bar{E}|+1$ to the edge set $\bar{E}^{D(C_i,C_j)} = C_i \times \{\bigcup_{l\in C_j} N(l) \setminus \bigcup_{l\in C_i} N(l)\}$ of the complementary graph. The graph $\bar{G}^{D(C_i,C_j)}$ is used to compute a lower bound of $\widehat{X}$ as follows:*

$$f(G,\widehat{X}) \geq w_{sdp}\left(G^{D(C_i,C_j)}\right) + (1-M) \sum_{i,j\in K: j>i} \left|\bar{E}^{D(C_i,C_j)}\right|. \tag{37}$$

The lower bound of $f(G,\widehat{X})$ is computed by performing two series of transformations: first the graph G is modified into G^M by applying Proposition 1 to every candidate set C_i, then G^M is transformed into $G^{M,D}$ by applying Proposition 2 to every pair C_i and C_j, with $i,j = 1,\ldots,k$ and $i<j$. If we combine the two equations (36) and (37), we get the following lower bound:

$$f(G,X) \geq w_{sdp}(G^{M,D}) + \sum_{i\in K}\sum_{j\in C_i} \delta_{\bar{G}}(j) + (1-M) \sum_{i,j\in K: j>i} \left|\bar{E}^{M,D(C_i,C_j)}\right|.$$

Example 2. Let us consider the problem of Example 1. Figure 3.a shows in bold the edges added to G for considering the vertices i and j in the same cluster. Figure 3.b shows in bold the edges of the complementary graph $\bar{G}$ that have been weighted since the vertices 1 and 2 are in different clusters.



(a) $G^{M(\{1,2\})}$ (b) $G^{D(\{1,2\})}$ (c) $G^{M(1,2),D(\{1,2\},4)}$

Fig. 3. The graph of Example 1 modified by considering: (a) the vertices (1,2) in the same cluster; (b) in different cluster; (c) the vertices (1,2) in the same cluster, but (1,4), (2,4) in different clusters

Example 3. Let $\widehat{X}$ be such that $C_1 = \{1, 2\}$ and $C_2 = \{4\}$ are the candidate sets of X_1 and X_2. This is equivalent to merge the vertices $(1, 2)$ in the same cluster, and to force $(1, 4)$ and $(2, 4)$ to be in different clusters. Figure 3.c shows the complementary graph of $G^{M,D}$ resulting after the corresponding transformations.

6 Preliminary Results

The `osi-qbiclique` constraint has been implemented in C++ using the Gecode constraint development environment [12]. The SDP relaxation is solved using the DSDP solver [13] that implements a dual scaling interior point algorithm. The ILP (5)–(10) is solved using CPLEX 11.0 with the default settings. As benchmarks, we used a set of random instances corresponding to the most difficult instances reported in [2], and available at [14].

We performed a first set of experiments for assessing the quality of the SDP-based labeling heuristic proposed in Sect. 5. The idea is to compare the gap from the optimum w^* given by the cost $\bar{w}$ of the first solution found by a CP solver that implements a given labeling heuristics. A *perfect* labeling heuristic would yield a gap equal to one, *i.e.*, $\bar{w} = w^*$. The first labeling heuristic considered, called MaxDegree, selects an unassigned set variable and the value of its domain corresponding to the vertex having maximum degree. The other heuristic are based on the SDP relaxation: the SDP-once heuristic solves the relaxation at the root node of the CP search tree, and then it follows Algorithm 2. The SDP-all heuristic recomputes the relaxation at every node of the CP search tree using both graph transformations given in Sect. 5.2. The SDP-pos heuristic recomputes the relaxation only when two nodes are merged in the same cluster, corresponding to use the graph transformation given in Proposition 1.

Table 1 reports the results of the first set of experiments, giving in each row the averages over 10 instances of bipartite graphs with $|S| = |T| = 12$. The first two columns of the table give the density of the graph d and the number of clusters k. Then, for each labeling heuristic, the table reports the averages and the standard deviations (between brackets) of the gap $\frac{\bar{w}}{w^*}$, and the average computation time in seconds. The MaxDegree heuristic is very fast, but it produces solutions with loose gaps. Solving the SDP relaxation within the search tree helps in decreasing the gap, but at the cost of higher computational times. The SDP-pos heuristics produces very good solutions, but is too expensive to be embedded in a branch-and-bound search. However, the SDP-once heuristic offers a good trade-off between the quality of the gap and the computational time.

A second set of experiments aimed at comparing a pure CP approach with the MaxDegree labeling, a hybrid CP-SDP approach with the SDP-once labeling, and the Integer Bilinear Programming approach, for solving to optimality the k-CmBC problem. Table 2 reports for each approach the averages of branch-and-bound nodes and CPU times, using the same set of instances of Table 1. For the two CP approaches, the table reports also the time t_{w^*} at which the optimal solution was found. The pure CP approach is very fast for $k = 2$, even if it enumerates more search nodes. For $k = 4$, the hybrid SDP-once approach is

Table 1. Comparing the quality of the first solution found with different labeling heuristics. The gap with the optimum w^* is measured as $\frac{\bar{w}}{w^*}$. The time is in seconds. Each row gives the averages over 10 instances of bipartite graphs with $|S| = |T| = 12$. The standard deviations of the gaps are given between brackets.

		MaxDegree		SDP-once		SDP-all		SDP-pos	
d	k	gap	time	gap	time	gap	time	gap	time
0.3	2	1.24 (0.14)	0.00	1.23 (0.14)	0.29	1.28 (0.10)	38.1	**1.13** (0.10)	14.4
	4	1.91 (0.24)	0.00	1.59 (0.22)	0.43	1.59 (0.21)	37.1	**1.21** (0.11)	15.0
	6	2.59 (0.43)	0.00	1.48 (0.38)	0.69	1.20 (0.14)	41.8	**1.07** (0.05)	16.7
0.5	2	**1.04** (0.05)	0.43	1.16 (0.08)	0.24	1.18 (0.10)	39.3	1.18 (0.09)	13.4
	4	1.41 (0.13)	0.81	1.30 (0.15)	0.30	1.32 (0.14)	33.3	**1.15** (0.06)	16.7
	6	1.60 (0.23)	1.02	1.28 (0.23)	0.40	1.25 (0.18)	38.1	**1.08** (0.06)	17.9
0.7	2	1.19 (0.06)	0.00	1.18 (0.11)	0.24	**1.15** (0.08)	14.7	1.22 (0.08)	7.2
	4	1.26 (0.10)	0.00	1.30 (0.17)	0.30	**1.20** (0.15)	13.7	1.25 (0.09)	8.5
	6	1.39 (0.17)	0.00	1.29 (0.21)	0.40	1.20 (0.13)	14.2	**1.17** (0.11)	8.2
Mean:		**1.51 (0.17)**		**1.31 (0.19)**		**1.26 (0.14)**		**1.16 (0.08)**	

Table 2. Comparison between the number of branch&bound nodes and the time (in seconds) for proving optimality; t_{w^*} denotes the time the optimal solution was found. Each row gives the averages over 10 instances of bipartite graphs with $|S| = |T| = 12$.

		Max-degree			SDP-once			ILP (5)–(10)	
k	d	nodes	t_{w^*}	time	nodes	t_{w^*}	time	nodes	time
2	0.3	502	0.10	**0.17**	360	0.35	0.44	24	2.33
	0.5	601	0.06	**0.29**	459	0.33	0.47	24	1.11
	0.7	507	0.04	**0.22**	376	0.48	0.61	92	2.01
4	0.3	186,004	24	70	30,133	3.9	**17**	2,939	19
	0.5	218,674	35	69	44,456	4.3	**21**	2,987	**21**
	0.7	207,658	14	88	29,314	2.5	17	2,987	**10**
6	0.3	3,939,535	640	1152	330,434	1.4	**151**	154,200	341
	0.5	6,582,521	933	2273	513,511	1.2	**276**	670,865	1104
	0.7	3,670,015	269	1223	420,450	5.9	**238**	154,200	356

competitive with the Integer Bilinear Approach, while for $k = 6$ is slightly faster. However, the hybrid approach is very fast in finding the optimum solution, *i.e.*, t_{w^*} is short, but it needs to explore many search nodes to prove optimality.

7 Conclusions

We have presented a hybrid CP and SDP approach to the k-CmBC problem. The CP model is based on two new optimization constraints, the *quasi-biclique* and the *one-shore-induced quasi-biclique* constraints. A SDP relaxation was developed, since the Bilinear Programming relaxation provides weak lower bounds. The proposed SDP relaxation differs from SDP relaxations used in the literature,

because it provides the likelihood that two values belong to the same set variable. Computational results provide further evidence that hybrid CP and SDP approaches are a promising approach to tackle combinatorial optimization problems. In particular, the proposed SDP labeling heuristics produces very good initial solutions, when at every node of the CP search tree partial solutions are mapped to new SDP relaxations. However, the limit of this approach is that current SDP solvers provide little support for re-optimization. Any improvement in the development of SDP solvers with this respect will enhance hybrid CP and SDP approaches as well.

References

1. Monson, S., Pullman, N., Rees, R.: A survey of clique and biclique coverings and factorizations of (0,1)-matrices. Bull. of the Combin. and its Appl. 14, 17–86 (1992)
2. Faure, N., Chrétienne, P., Gourdin, E., Sourd, F.: Biclique completion problems for multicast network design. Discrete Optim. 4(3), 360–377 (2007)
3. Fahle, T.: Simple and fast: Improving a branch-and-bound algorithm for maximum clique. In: Möhring, R.H., Raman, R. (eds.) ESA 2002. LNCS, vol. 2461, pp. 485–498. Springer, Heidelberg (2002)
4. Régin, J.C.: Using constraint programming to solve the maximum clique problem. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 634–648. Springer, Heidelberg (2003)
5. Barnier, N., Brisset, P.: Graph coloring for air traffic flow management. Ann. of Oper. Res. 130, 163–178 (2004)
6. Focacci, F., Lodi, A., Milano, M.: Cost-based domain filtering. In: Jaffar, J. (ed.) CP 1999. LNCS, vol. 1713, pp. 189–203. Springer, Heidelberg (1999)
7. van Hoeve, W.: Exploiting semidefinite relaxations in constraint programming. Computers & OR 33, 2787–2804 (2006)
8. Gomes, C., van Hoeve, W.J., Leahu, L.: The power of semidefinite programming relaxations for MAXSAT. In: Beck, J.C., Smith, B.M. (eds.) CPAIOR 2006. LNCS, vol. 3990, pp. 104–118. Springer, Heidelberg (2006)
9. Goemans, M., Williamson, D.: Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. J. of the ACM 42, 1115–1145 (1995)
10. Frieze, A., Jerrum, M.: Improved approximation algorithms for max k-cut and max-bisection. Algorithmica 18, 67–81 (1997)
11. Régin, J.C.: Global Constraints and Filtering Algorithms. In: Milano, M. (ed.) Constraint and Integer Programming-Toward a Unified Methodology. Kluwer, Dordrecht (2004)
12. Gecode: Generic constraint development environment, http://www.gecode.org
13. Benson, S.J., Ye, Y.: Algorithm 875: DSDP5—software for semidefinite programming. ACM Trans. Math. Softw. 34(3), 1–20 (2008)
14. k-CmBC Web Resources, http://home.dei.polimi.it/gualandi/bicliques

Optimal Interdiction of Unreactive Markovian Evaders

Alexander Gutfraind[1], Aric Hagberg[2], and Feng Pan[3]

[1] Center for Applied Mathematics, Cornell University, Ithaca, New York 14853
ag362@cornell.edu
[2] Theoretical Division, Los Alamos National Laboratory, Los Alamos, New Mexico 87545
hagberg@lanl.gov
[3] Risk Analysis and Decision Support Systems, Los Alamos National Laboratory, Los Alamos, New Mexico 87545
fpan@lanl.gov

Abstract. The interdiction problem arises in a variety of areas including military logistics, infectious disease control, and counter-terrorism. In the typical formulation of *network* interdiction, the task of the interdictor is to find a set of edges in a weighted network such that the removal of those edges would maximally increase the cost to an evader of traveling on a path through the network.

Our work is motivated by cases in which the evader has incomplete information about the network or lacks planning time or computational power, *e.g.* when authorities set up roadblocks to catch bank robbers, the criminals do not know all the roadblock locations or the best path to use for their escape.

We introduce a model of network interdiction in which the motion of one or more evaders is described by Markov processes and the evaders are assumed not to react to interdiction decisions. The interdiction objective is to find an edge set of size B, that maximizes the probability of capturing the evaders.

We prove that similar to the standard least-cost formulation for deterministic motion this interdiction problem is also NP-hard. But unlike that problem our interdiction problem is submodular and the optimal solution can be approximated within $1-1/e$ using a greedy algorithm. Additionally, we exploit submodularity through a priority evaluation strategy that eliminates the linear complexity scaling in the number of network edges and speeds up the solution by orders of magnitude. Taken together the results bring closer the goal of finding realistic solutions to the interdiction problem on global-scale networks.

1 Introduction

Network interdiction problems have two opposing actors: an "evader" (*e.g.* smuggler) and an "interdictor" (*e.g.* border agent.) The evader attempts to minimize some objective function in the network, *e.g.* the probability of capture while traveling from network location s to location t, while the interdictor attempts to limit the evader's success by removing network nodes or edges. Most often the interdictor has limited resources and can thus only remove a very small fraction of the nodes or edges. The standard formulation is the max-min problem where the interdictor plays first and chooses at most B edges to remove, while the evader finds the least-cost path on the remaining network. This is known as the B most vital arcs problem [1].

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 102–116, 2009.

This least-cost-path formulation is not suitable for some interesting interdiction scenarios. Specifically in many practical problems there is a fog of uncertainty about the underlying properties of the network such as the cost to the evader in traversing an edge (arc, or link) in terms of either resource consumption or detection probability. In addition there are mismatches in the cost and risk computations between the interdictor and the evaders (as well as between different evaders), and all agents have an interest in hiding their actions. For evaders, most least-cost-path interdiction models make optimal assumptions about the evader's knowledge of the interdictor's strategy, namely, the choice of interdiction set. In many real-world situations evaders likely fall far short of the optimum. This paper, therefore, considers the other limit case, which for many problems is more applicable, when the evaders do not respond to interdictor's decisions. This case is particularly useful for problems where the evader is a process on the network rather than a rational agent.

Various formulations of the network interdiction problem have existed for many decades now. The problem likely originated in the study of military supply chains and interdiction of transportation networks [2,3]. But in general, the network interdiction problem applies to wide variety of areas including control of infectious disease [4], and disruption of terrorist networks [5]. Recent interest in the problem has been revived due to the threat of smuggling of nuclear materials [6]. In this context interdiction of edges might consist of the placement of special radiation-sensitive detectors across transportation links. For the most-studied formulation, that of max-min interdiction described above [1], it is known that the problem is NP-hard [7,8] and hard to approximate [9].

2 Unreactive Markovian Evader

The formulation of a stochastic model where the evader has limited or no information about interdiction can be motivated by the following interdiction situation. Suppose bank robbers (evaders) want to escape from the bank at node s to their safe haven at node t_1 or node t_2. The authorities (interdictors) are able to position roadblocks at a few of the roads on the network between s, t_1 and t_2. The robbers might not be aware of the interdiction efforts, or believe that they will be able to move faster than the authorities can set up roadblocks. They certainly do not have the time or the computational resources to identify the global minimum of the least-cost-path problem.

Similar examples are found in cases where the interdictor is able to clandestinely remove edges or nodes (*e.g.* place hidden electronic detectors), or the evader has bounded rationality or is constrained in strategic choices. An evader may even have no intelligence of any kind and represent a process such as Internet packet traffic that the interdictor wants to monitor. Therefore, our fundamental assumption is that the evader does not respond to interdiction decisions. This transforms the interdiction problem from the problem of increasing the evader's cost or distance of travel, as in the standard formulation, into a problem of directly capturing the evader as explicitly defined below. Additionally, the objective function acquires certain useful computational properties discussed later.

2.1 Evaders

In examples discussed above, much of the challenge in interdiction stems from the unpredictability of evader motion. Our approach is to use a stochastic evader model to capture this unpredictability [6,10]. We assume that an evader is traveling from a source node s to a target node t on a graph $G(N,E)$ according to a guided random walk defined by the Markovian transition matrix $\mathbf{M}$; from node i the evader travels on edge (i,j) with probability M_{ij}. The transition probabilities can be derived, for example, from the cost and risk of traversing an edge [10].

Uncertainty in the evader's source location s is captured through a probability vector $\mathbf{a}$. For the simplest case of an evader starting known location s, $a_s = 1$ and the rest of the a_i's are 0. In general the probabilities can be distributed arbitrarily to all of the nodes as long as $\sum_{i \in N} a_i = 1$. Given $\mathbf{a}$, the probability that the evader is at location i after n steps is the i'th entry in the vector $\pi^{(n)} = \mathbf{a}\mathbf{M}^n$.

When the target is reached the evader exits the network and therefore, $M_{tj} = 0$ for all outgoing edges from t and also $M_{tt} = 0$. The matrix $\mathbf{M}$ is assumed to satisfy the following condition: for every node i in the network either there is a positive probability of reaching the target after a sufficiently large number of transitions, or the node is a dead end, namely $M_{ij} = 0$ for all j. With these assumptions the Markov chain is absorbing and the probability that the evader will eventually reach the target is ≤ 1. For equality to hold it is sufficient to have the extra conditions that the network is connected and that for all nodes $i \neq t$, $\sum_j M_{ij} = 1$ (see [11].)

A more general formulation allows multiple evaders to traverse the network, where each evader represents a threat scenario or a particular adversarial group. Each evader k is realized with probability $w^{(k)}$ ($\sum_k w^{(k)} = 1$) and is described by a possibly distinct source distribution $\mathbf{a}^{(k)}$, transition matrix $\mathbf{M}^{(k)}$, and target node $t^{(k)}$. This generalization makes it possible to represent any joint probability distribution $f(s,t)$ of source-target pairs, where each evader is a slice of f at a specific value of t: $\mathbf{a}^{(k)}|_s = f(s,t^{(k)})/\sum_s f(s,t^{(k)})$ and $w^{(k)} = \sum_s f(s,t^{(k)})$. In this high-level view, the evaders collectively represent a stochastic process connecting pairs of nodes on the network. This generalization has practical applications to problems of monitoring traffic between any set of nodes when there is a limit on the number of "sensors". The underlying network could be *e.g.* a transportation system, the Internet, or water distribution pipelines.

2.2 Interdictor

The interdictor, similar to the typical formulation, possesses complete knowledge about the network and evader parameters $\mathbf{a}$ and $\mathbf{M}$. Interdiction of an edge at index i,j is represented by setting $r_{ij} = 1$ and $r_{ij} = 0$ if the edge is not interdicted. In general some edges are more suitable for interdiction than others. To represent this, we let d_{ij} be the interdiction efficiency, which is the probability that interdiction of the edge would remove an evader who traverses it.

So far we have focused on the interdiction of edges, but interdiction of nodes can be treated similarly as a special case of edge interdiction in which all the edges leading to an interdicted node are interdicted simultaneously. For brevity, we will not discuss node interdiction further except in the proofs of Sec. 3 where we consider both cases.

2.3 Objective Function

Interdiction of an unreactive evader is the problem of maximizing the probability of stopping the evader before it reaches the target. Note that the fundamental matrix for $\mathbf{M}$, using $\mathbf{I}$ to denote the identity matrix is

$$\mathbf{N} = \mathbf{I} + \mathbf{M} + \mathbf{M}^2 + \cdots = (\mathbf{I} - \mathbf{M})^{-1}, \tag{1}$$

and $\mathbf{N}$ gives all of the possible transition sequences between pairs of nodes before the target is reached. Therefore given the starting probability $\mathbf{a}$, the expected number of times the evader reaches each node is (using (1) and linearity of expectation)

$$\mathbf{aN} = \mathbf{a}(\mathbf{I} - \mathbf{M})^{-1}. \tag{2}$$

If edge (i, j) has been interdicted ($r_{ij} = 1$) and the evader traverses it then the evader will not reach j with probability d_{ij}. The probability of the evader reaching j from i becomes

$$\hat{M}_{ij} = M_{ij} - M_{ij} r_{ij} d_{ij}. \tag{3}$$

This defines an interdicted version of the $\mathbf{M}$ matrix, the matrix $\hat{\mathbf{M}}$.

The probability that a single evader does not reach the target is found by considering the t'th entry in the vector E after substituting $\hat{\mathbf{M}}$ for $\mathbf{M}$ in Eq. (2),

$$J(\mathbf{a}, \mathbf{M}, \mathbf{r}, \mathbf{d}) = 1 - \left(\mathbf{a} \left[\mathbf{I} - (\mathbf{M} - \mathbf{M} \odot \mathbf{r} \odot \mathbf{d}) \right]^{-1} \right)_t, \tag{4}$$

where the symbol $\odot$ means element-wise (Hadamard) multiplication. In the case of multiple evaders, the objective J is a weighted sum,

$$J = \sum_k w^{(k)} J^{(k)}, \tag{5}$$

where, for evader k,

$$J^{(k)}(\mathbf{a}^{(k)}, \mathbf{M}^{(k)}, \mathbf{r}, \mathbf{d}) = 1 - \left(\mathbf{a}^{(k)} \left[\mathbf{I} - \left(\mathbf{M}^{(k)} - \mathbf{M}^{(k)} \odot \mathbf{r} \odot \mathbf{d} \right) \right]^{-1} \right)_{t^{(k)}}. \tag{6}$$

Equations (4) and (5) define the *interdiction probability*. Hence the *Unreactive Markovian Evader* interdiction problem (UME) is

$$\underset{\mathbf{r} \in F}{\operatorname{argmax}} \; J(\mathbf{a}, \mathbf{M}, \mathbf{r}, \mathbf{d}), \tag{7}$$

where r_{ij} represents an interdicted edge chosen from a set $F \subseteq 2^E$ of feasible interdiction strategies. The simplest formulation is the case when interdicting an edge has a unit cost with a fixed budget B and F are all subsets of the edge set E of size at most B. This problem can also be written as a mixed integer program as shown in the Appendix.

Computation of the objective function can be achieved with $\sim \frac{2}{3} |N|^3$ operations for each evader, where $|N|$ is the number of nodes, because it is dominated by the cost of Gaussian elimination solve in Eq. (4). If the matrix $\mathbf{M}$ has special structure then it could be reduced to $O(|N|^2)$ [10] or even faster. We will use this evader model in the

simulations, but in general the methods of Secs. 3 and 4 would work for any model that satisfies the hypotheses on $\mathbf{M}$ and even for non-Markovian evaders as long as it is possible to compute the equivalent of the objective function in Eq. (4).

Thus far interdiction was described as the removal of the evader from the network, and the creation of a sub-stochastic process $\hat{\mathbf{M}}$. However, the mathematical formalism is open to several alternative interpretations. For example interdiction could be viewed as redirection of the evader into a special absorbing state - a "jail node". In this larger state space the evader even remains Markovian. Since $\hat{\mathbf{M}}$ is just a mathematical device it is not even necessary for "interdiction" to change the physical traffic on the network. In particular, in monitoring problems "interdiction" corresponds to labeling of intercepted traffic as "inspected" - a process that involves no removal or redirection.

3 Complexity

This section proves technical results about the interdiction problem (7) including the equivalence in complexity of node and edge interdiction and the NP-hardness of node interdiction (and therefore of edge interdiction). Practical algorithms are found in the next section.

We first state the decision problem for (7).

Definition 1. UME-Decision
Instance*: A graph $G(N,E)$, interdiction efficiencies $0 \leq d_i \leq 1$ for each $i \in N$, budget $B \geq 0$, and real $\rho \geq 0$; a set K of evaders, such that for each $k \in K$ there is a matrix $\mathbf{M}^{(k)}$ on G, a sources-target pair $(\mathbf{a}^{(k)}, t^{(k)})$ and a weight $w^{(k)}$.*

Question*: Is there a set of (interdicted) nodes Y of size B such that*

$$\sum_{k \in K} w^{(k)} \left(\mathbf{a}^{(k)} \left(\mathbf{I} - \hat{\mathbf{M}}^{(k)} \right)^{-1} \right)_{t^{(k)}} \leq \rho ? \tag{8}$$

The matrix $\hat{\mathbf{M}}^{(k)}$ is constructed from $\mathbf{M}^{(k)}$ by replacing element $M_{ij}^{(k)}$ by $M_{ij}^{(k)}(1-d_i)$ for $i \in Y$ and each (i,j) corresponding to edges $\in E$ leaving i. This sum is the weighted probability of the evaders reaching their targets. □

The decision problem is stated for node interdiction but the complexity is the same for edge interdiction, as proved next.

Lemma 1. *Edge interdiction is polynomially equivalent to node interdiction.*

Proof. To reduce edge interdiction to node interdiction, take the graph $G(N,E)$ and construct G' by splitting the edges. On each edge $(i,j) \in E$ insert a node v to create the edges $(i,v),(v,j)$ and set the node interdiction efficiency $d_v = d_{ij}, d_i = d_j = 0$, where d_{ij} is the interdiction efficiency of (i,j) in E.

Conversely, to reduce node interdiction to edge interdiction, construct from $G(N,E)$ a graph G' by representing each node v with interdiction efficiency d_v by nodes i,j, joining them with an edge (i,j), and setting $d_{ij} = d_v$. Next, change the transition matrix $\mathbf{M}$ of each evader such that all transitions into v now move into i while all departures from v now occur from j, and $M_{ij} = 1$. In particular, if v was an evader's target node in G, then j is its target node in G'. □

Consider now the complexity of node interdiction. One source of hardness in the UME problem stems from the difficulty of avoiding the case where multiple edges or nodes are interdicted on the same evader path - a source of inefficiency. This resembles the *Set Cover* problem [12], where including an element in two sets is redundant in a similar way, and this insight motivates the proof.

First we give the definition of the set cover decision problem.

Definition 2. Set Cover. *For a collection C of subsets of a finite set X, and a positive integer β, does C contain a cover of size $\leq \beta$ for X?* □

Since *Set Cover* is NP-complete, the idea of the proof is to construct a network $G(N,E)$ where each subset $c \in C$ is represented by a node of G, and each element $x_i \in X$ is represented by an evader. The evader x_i is then made to traverse all nodes $\{c \in C | x_i \in c\}$. The set cover problem is exactly problem of finding B nodes that would interdict all of the evaders (see Fig. 1)

Theorem 2. *The UME problem is NP-hard even if $d_i = h$ (constant) $\forall$ nodes $i \in N$.*

Proof. First we note that for a given a subset $Y \subseteq N$ with $|Y| \leq B$, we can update $\mathbf{M}^{(k)}$ and compute (8) to verify *UME-Decision* as a yes-instance. The number of steps is bounded by $O(|K||N|^3)$. Therefore, *UME-Decision* is in NP.

To show *UME-Decision* is NP-complete, reduce *Set Cover* with X, C to *UME -Decision* on a suitable graph $G(N,E)$. It is sufficient to consider just the special case where all interdiction efficiencies are equal, $d_i = 1$. For each $c \in C$, create a node c in N.

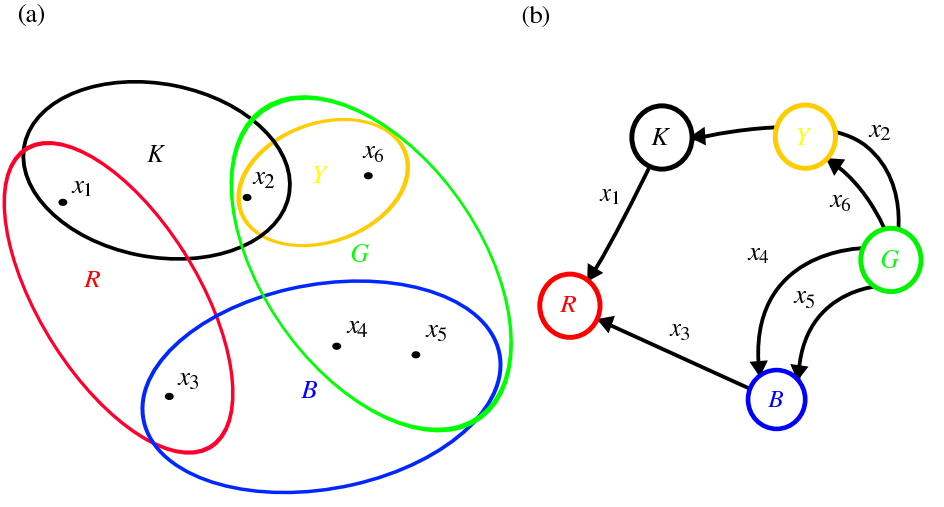


Fig. 1. Illustration of the reduction of Set Cover to UME-Decision. (a) A set cover problem on elements $x_1 \ldots x_6 \in X$ with subsets $K = \{x_1, x_2\}, R = \{x_1, x_3\}, B = \{x_3, x_4, x_5\}, G = \{x_2, x_4, x_5, x_6\}, Y = \{x_2, x_6\}$ contained in X. (b) The induced interdiction problem with each subset represented by a node and each element by an evader. Each arrow indicates the path of a single evader.

We consider three cases for elements $x \in X$; elements that have no covering sets, elements that have one covering set, and elements that have at least two covering sets.

Consider first all $x \in X$ which have at least two covering sets. For each such x create an evader as follows. Let O be any ordering of the collection of subsets covering x. Create in E a Hamiltonian path of $|O|-1$ edges to join sequentially all the elements of O, assigning the start, a and end t nodes in agreement with the ordering of O. Construct an evader transition matrix of size $|C| \times |C|$ and give the evader transitions probability $M_{ij} = 1$ iff $i, j \in C$ and $i < j$, and $= 0$ otherwise.

For the case of zero covering sets, that is, where $\exists x \in X$ such that $x \notin S$ for all $S \in C$, represent x by an evader whose source and target are identical: no edges are added to E and the transition matrix is $\mathbf{M} = 0$. Thus, J in Eq. (4) is non-zero regardless of interdiction strategy.

For the case when x has just one covering set, that is, when $\exists x \in X$ such that there is a unique $c \in C$ with $x \in c$, represent c as two nodes i and j connected by an edge exactly as in the case of more than one cover above. After introducing j, add it to the middle of the path of each evader x if i is in the path of x, that is, if $c \in C$. It is equivalent to supposing that C contains another subset exactly like c. This supposition does not change the answer or the polynomial complexity of the given instance of *Set Cover*. To complete the reduction, set $B = \beta$, $\rho = 0$, $X = K$, $w^{(k)} = 1/|X|$ and $d_i = 1, \forall i \in N$.

Now assume *Set Cover* is a yes-instance with a cover $\hat{C} \subseteq C$. We set the interdicted transition matrix $\hat{M}_{ij}^{(k)} = 0$ for all $(i, j) \in E$ corresponding to $c \in \hat{C}$, and all $k \in K$. Since $\hat{C}$ is a cover for X, all the created paths are disconnected, $\sum_{k \in K} \left(\mathbf{a}^{(k)} (\mathbf{I} - \hat{\mathbf{M}}^{(k)})^{-1}\right)_{t^{(k)}} = 0$ and *UME-Decision* is an yes-instance.

Conversely, assume that *UME-Decision* is a yes-instance. Let Y be the set of interdicted nodes. For $y \in Y$, there is element y of C. Since all the evaders are disconnected from their target and each evader represents a element in X, $Y \subseteq C$ covers X and $|Y| \leq \beta$. Hence, *Set Cover* is a yes-instance. Therefore, *UME-Decision* is NP-complete. □

This proof relies on multiple evaders and it remains an open problem to show that UME is NP-hard with just a single evader. We conjecture that the answer is positive because the more general problem of interdicting a single unreactive evader having an arbitrary (non-Markovian) path is NP-hard. This could be proved by creating from a single such evader several Markovian evaders such that the evader has an equal probability of following the path of each of the Markovian evaders in the proof above.

Thus far no consideration was given to the problem where the cost c_{ij} of interdicting an edge (i, j) is not fixed but rather is a function of the edge. This could be termed the "budgeted" case as opposed to the "unit cost" case discussed so far. However, the budgeted case is NP-hard as could be proved through reduction from the knapsack problem to a star network with "spokes" corresponding to items.

4 An Efficient Interdiction Algorithm

The solution to the UME problem can be efficiently approximated using a greedy algorithm by exploiting submodularity. In this section we prove that the UME problem is submodular, construct a greedy algorithm, and examine the algorithm's performance.

We then show how to improve the algorithm's speed by further exploiting the submodular structure using a "priority" evaluation scheme and "fast initialization".

4.1 Submodularity of the Interdiction Problem

In general, a function is called submodular if the rate of increase decreases monotonically, which is akin to concavity.

Definition 3. *A real-valued function on a space S, $f : S \to \mathbb{R}$ is* submodular *[13, Prop. 2.1iii] if for any subsets $S_1 \subseteq S_2 \subset S$ and any $x \in S \setminus S_2$ it satisfies*

$$f(S_1 \cup \{x\}) - f(S_1) \geq f(S_2 \cup \{x\}) - f(S_2)\,. \tag{9}$$

Lemma 3. *$J(\mathbf{r})$ is submodular on the set of interdicted edges.*

Proof. First, note that it is sufficient to consider a single evader because in Eq. (5), $J(\mathbf{r})$ is a convex combination of k evaders [13, Prop. 2.7]. For simplicity of notation, we drop the superscript k in the rest of the proof.

Let $S = \{(i,j) \in E | r_{ij} = 1\}$ be the interdiction set and let $J(S)$ be the probability of interdicting the evader using S, and let $Q(p)$ be the probability of the evader taking a path p to the target. On path p, the probability of interdicting the evader with an interdiction set S is

$$P(p|S) = Q(p)\left(1 - \prod_{(i,j)\in p\cap S}(1 - d_{ij})\right). \tag{10}$$

Moreover,

$$J(S) = \sum_p P(p|S)\,. \tag{11}$$

If an edge $(u,v) \notin S$ is added to the interdiction set S (assuming $(u,v) \in p$), the probability of interdicting the evader in path p increases by

$$P(p|S \cup \{(u,v)\}) - P(p|S) = Q(p)d_{uv} \prod_{(i,j)\in p\cap S}(1 - d_{ij})\,,$$

which can be viewed as the probability of taking the path p times the probability of being interdicted at (u,v) but not being interdicted elsewhere along p. If $(u,v) \in S$ or $(u,v) \notin p$ then adding (u,v) has, of course, no effect: $P(p|S \cup \{(u,v)\}) - P(p|S) = 0$.

Consider now two interdiction sets S_1 and S_2 such that $S_1 \subset S_2$. In the case where $(u,v) \notin S_1$ and $(u,v) \in p$, we have

$$\begin{aligned} P(p|S_1 \cup \{(u,v)\}) - P(p|S_1) &= Q(p)d_{uv} \prod_{(i,j)\in p\cap S_1}(1 - d_{ij})\,, && (12)\\ &\geq Q(p)d_{uv} \prod_{(i,j)\in p\cap S_2}(1 - d_{ij})\,, && (13)\\ &\geq P(p|S_2 \cup \{(u,v)\}) - P(p|S_2)\,. && (14) \end{aligned}$$

In the above (13) holds because an edge $(u',v') \in (S_2 \setminus S_1) \cap p$ would contribute a factor of $(1 - d_{u'v'}) \leq 1$. The inequality (14) becomes an equality iff $(u,v) \notin S_2$. Overall (14)

holds true for any path and becomes an equality when $(u,v) \in S_1$. Applying the sum of Eq. (11) gives

$$J(p|S_1 \cup \{(u,v)\}) - J(p|S_1) \geq J(p|S_2 \cup \{(u,v)\}) - J(p|S_2), \tag{15}$$

and therefore $J(S)$ is submodular. □

Note that the proof relies on the fact that the evader does not react to interdiction. If the evader did react then it would no longer be true in general that $P(p|S) = Q(p)\left(1 - \prod_{(i,j)\in p\cap S}(1-d_{ij})\right)$ above. Instead, the product may show explicit dependence on paths other than p, or interdicted edges that are not on p. Also, when the evaders are not Markovian the proof is still valid because specifics of evader motion are contained in the function $Q(p)$.

4.2 Greedy Algorithm

Submodularity has a number of important theoretical and algorithmic consequences. Suppose (as is likely in practice) that the edges are interdicted incrementally such that the interdiction set $S_l \supseteq S_{l-1}$ at every step l. Moreover, suppose at each step, the interdiction set S_l is grown by adding the one edge that gives the greatest increase in J. This defines a greedy algorithm, Alg. 1.

Algorithm 1. Greedy construction of the interdiction set S with budget B for a graph $G(N,E)$.

```
S ← ∅
while B > 0 do
    x* ← ∅
    δ* ← −1
    for all x ∈ E ∖ S do
        Δ(S,x) := J(S ∪ {x}) − J(S)
        if Δ(S,x) > δ* then
            x* ← {x}
            δ* ← Δ(S,x)
    S ← S ∪ x*
    B ← B − 1
Output(S)
```

The computational time is $O(B|N|^3|E|)$ for each evader, which is strongly polynomial since $|B| \leq |E|$. The linear growth in this bound as a function of the number of evaders could sometimes be significantly reduced. Suppose one is interested in interdicting flow $f(s,t)$ that has a small number of sources but a larger number of targets. In the current formulation the cost grows linearly in the number of targets (evaders) but is independent of the number of sources. Therefore for this $f(s,t)$ it is advantageous to reformulate UME by inverting the source-target relationship by deriving a Markov process which describes how an evader moves from a given source s to each of the targets. In this formulation the cost would be independent of the number of targets and grow linearly in the number of sources.

4.3 Solution Quality

The quality of the approximation can be bounded as a fraction of the optimal solution by exploiting the submodularity property [13]. In submodular set functions such as $J(S)$ there is an interference between the elements of S in the sense that sum of the individual contributions is greater than the contribution when part of S. Let S_B^* be the optimal interdiction set with a budget B and let S_B^g be the solution with a greedy algorithm. Consider just the first edge x_1 found by the greedy algorithm. By the design of the greedy algorithm the gain from x_1 is greater than the gain for all other edges y, including any of the edges in the optimal set S^*. It follows that

$$\Delta(\varnothing, x_1)B \geq \sum_{y \in S_B^*} \Delta(\varnothing, y) \geq J(S_B^*)\,. \tag{16}$$

Thus x_1 provides a gain greater than the average gain for all the edges in S_B^*,

$$\Delta(\varnothing, x_1) \geq \frac{J(S_B^*)}{B}\,. \tag{17}$$

A similar argument for the rest of the edges in S_B^g gives the bound,

$$J(S_B^g) \geq \left(1 - \frac{1}{e}\right) J(S_B^*)\,, \tag{18}$$

where e is Euler's constant [13, p.268]. Hence, the greedy algorithm achieves at least 63% of the optimal solution.

This performance bound depends on the assumption that the cost of an edge is a constant. Fortunately, good discrete optimization algorithms for submodular functions are known even for the case where the cost of an element (here, an edge) is variable. These algorithms are generalizations of the simple greedy algorithm and provide a constant-factor approximation to the optimum [14,15]. Moreover, for any particular instance of the problem one can bound the approximation ratio, and such an "online" bound is often better than the "offline" *a priori* bound [16].

4.4 Exploiting Submodularity with Priority Evaluation

In addition to its theoretical utility, submodularity can be exploited to compute the same solution much faster using a priority evaluation scheme. The basic greedy algorithm recomputes the objective function change $\Delta(S_l, x)$ for each edge $x \in E \setminus S_l$ at each step l. Submodularity, however, implies that the gain $\Delta(S_l, x)$ from adding any edge x would be less than or equal to the gain $\Delta(S_k, x)$ computed at any earlier step $k < l$. Therefore, if at step l for some edge x', we find that $\Delta(S_l, x') \geq \Delta(S_k, x)$ for all x and any past step $k \leq l$, then x' is the optimal edge at step l; there is no need for further computation (as was suggested in a different context [16].) In other words, one can use stale values of $\Delta(S_k, x)$ to prove that x' is optimal at step l.

As a result, it may not be necessary to compute $\Delta(S_l, x)$ for all edges $x \in E \setminus S$ at every iteration. Rather, the computation should prioritize the edges in descending order of $\Delta(S_l, x)$. This "lazy" evaluation algorithm is easily implemented with a priority queue

which stores the gain $\Delta(S_k,x)$ and k for each edge where k is the step at which it was last calculated. (The step information k determines whether the value is stale.)

The priority algorithm (Alg. 2) combines lazy evaluation with the following fast initialization step. Unlike in other submodular problems, in UME one can compute $\Delta(\varnothing,x)$ simultaneously for all edges $x \in E$ because in this initial step, $\Delta(\varnothing,x)$ is just the probability of transition through edge x multiplied by the interdiction efficiency d_x, and the former could be found for all edges in just one operation. For the "non-retreating" model of Ref. [10] the probability of transition through $x = (i, j)$ is just the expected number of transitions though x because in that model an evader moves through x at most once. This expectation is given by the i, j element in $\mathbf{a}(\mathbf{I}-\mathbf{M})^{-1} \odot \mathbf{M}$ (derived from Eq. (2)). The probability is multiplied by the weight of the evader and then by d_x: $\Delta(\varnothing,x) = \sum_k \left(\mathbf{a}^{(k)}(\mathbf{I}-\mathbf{M}^{(k)})^{-1}\right)_i M_{ij}^{(k)} w^{(k)} d_x$. In addition to these increments, for disconnected graphs the objective $J(S)$ also contains the constant term $\sum_k w^{(k)} \left(\sum_{i \in Z^{(k)}} a_i\right)$, where $Z^{(k)} \subset N$ are nodes from which evader k cannot reach his target $t^{(k)}$.

In subsequent steps this formula is no longer valid because interdiction of x may reduce the probability of motion through other interdicted edges. Fortunately, in many instances of the problem the initialization is the most expensive step since it involves computing the cost for all edges in the graph. As a result of the two speedups the number of cost evaluations could theoretically be linear in the budget and the number of evaders and independent of the size of the solution space (the number of edges).

The performance gain from priority evaluation can be very significant. In many computational experiments, the second best edge from the previous step was the best in the current step, and frequently only a small fraction of the edges had to be recomputed at each iteration. In order to systematically gauge the improvement in performance, the algorithm was tested on 50 synthetic interdiction problems. In each case, the

Algorithm 2. Priority greedy construction of the interdiction set S with budget B

```
S ← ∅
PQ ← ∅ {Priority Queue: (value, data, data)}
for all x = (i, j) ∈ E do
    Δ(x) ←{The cost found using fast initialization}
    PUSH (PQ, (Δ(x), x, 0))
s ← 0
while B > 0 do
    s ← s + 1
    loop
        (Δ(x), x, n) ← POP(PQ)
        if n = s then
            S ← S ∪ {x}
            break
        else
            Δ(x) ← J(S ∪ {x}) − J(S)
            PUSH (PQ, (Δ(x), x, s))
    B ← B − 1
Output(S)
```

underlying graph was a 100-node Geographical Threshold Graph (GTG), a possible model of sensor or transportation networks [17], with approximately 1600 directed edges (the threshold parameter was set at $\theta = 30$). Most of the networks were connected. We set the cost of traversing an edge to 1, the interdiction efficiency d_x to 0.5, $\forall x \in E$, and the budget to 10. We used two evaders with uniformly distributed source nodes based on the model of [10] with an equal mixture of $\lambda = 0.1$ and $\lambda = 1000$. For this instance of the problem the priority algorithm required an average of 29.9 evaluations of the objective as compared to 31885.2 in the basic greedy algorithm - a factor of 1067.1 speedup.

The two algorithms find the same solution, but the basic greedy algorithm needs to recompute the gain for all edges uninterdicted edges at every iteration, while the priority algorithm can exploit fast initialization and stale computational values. Consequently, the former algorithm uses approximately $B|E|$ cost computations, while the latter typically uses much fewer (Fig. 2a).

Simulations show that for the priority algorithm the number of edges did not seem to affect the number of cost computations (Fig. 2b), in agreement with the theoretical limit. Indeed, the only lower bound for the number of cost computations is B and this bound is tight (consider a graph with B evaders each of which has a distinct target separated from each evader's source by exactly one edge of sufficiently small cost). The priority algorithm performance gains were also observed in other example networks.[1]

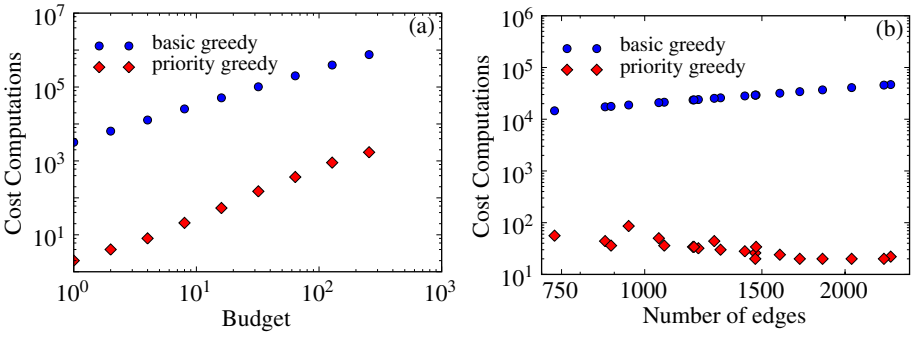


Fig. 2. Comparison between the basic greedy (blue circles) and the priority greedy algorithms (red diamonds) for the number of cost evaluations as a function of (a) budget, and (b) number of edges. In (a) each point is the average of 50 network interdiction problems. The average coefficient of variation (the ratio of the standard deviation to the mean) is 0.10 for basic greedy and 0.15 for the priority greedy. Notice the almost perfectly linear trends as a function of budget (shown here on a log-log scale, the power ≈ 1.0 in both.) In (b), the budget was fixed at 10 and the number of edges was increased by decreasing the connectivity threshold parameter from $\theta = 50$ to $\theta = 20$ to represent, e.g., increasingly dense transportation networks.

[1] Specifically, the simulations were a two evader problem on a grid-like networks consisting of a lattice (whose dimensions were grown from 8-by-8 to 16-by-16) with random edges added at every node. The number of edges in the networks grew from approximately 380 to 1530 but there was no increasing trend in the number of cost evaluations.

The priority algorithm surpasses a benchmark solution of the corresponding mixed integer program (See Appendix) using a MIP solver running CPLEX (version 10.1) in consistency, time, and space. For example, in runs on 100-node GTG networks with 4 evaders and a budget of 10, the priority algorithm terminates in 1 to 20 seconds, while CPLEX terminated in times ranging from under 1 second to 9.75 hours (the high variance in CPLEX run times, even on small problems, made systematic comparison difficult.) The difference in solution optimality was zero in the majority of runs. In the hardest problem we found (in terms of its CPLEX computational time - 9.75 hours), the priority algorithm found a solution at 75% of the optimum in less than 10 seconds.

For our implementation, memory usage in the priority algorithm never exceeded 300MiB. Further improvement could be made by re-implementing the priority algorithm so that it would require only order $O(|E|)$ to store both the priority queue and the vectors of Eq. (4). In contrast, the implementation in CPLEX repeatedly used over 1GiB for the search tree. As was suggested from the complexity proof, in runs where the number of evaders was increased from 2 to 4 the computational time for an exact solution grew rapidly.

5 Outlook

The submodularity property of the UME problem provides a rich source for algorithmic improvement. In particular, there is room for more efficient approximation schemes and practical value in their invention. Simultaneously, it would be interesting to classify the UME problem into a known approximability class. It would also be valuable to investigate various trade-offs in the interdiction problem, such as the trade-off between quality and quantity of interdiction devices.

As well, to our knowledge little is known about the accuracy of the assumptions of the unreactive Markovian model or of the standard max-min model in various applications. The detailed nature of any real instance of network interdiction would determine which of the two formulations is more appropriate.

Acknowledgments

AG would like to thank Jon Kleinberg for inspiring lectures, David Shmoys for a helpful discussion and assistance with software, and Vadas Gintautas for support. Part of this work was funded by the Department of Energy at Los Alamos National Laboratory under contract DE-AC52-06NA25396 through the Laboratory Directed Research and Development Program.

References

1. Corley, H.W., Sha, D.Y.: Most vital links and nodes in weighted networks. Oper. Res. Lett. 1(4), 157–160 (1982)
2. McMasters, A.W., Mustin, T.M.: Optimal interdiction of a supply network. Naval Research Logistics Quarterly 17(3), 261–268 (1970)

3. Ghare, P.M., Montgomery, D.C., Turner, W.C.: Optimal interdiction policy for a flow network. Naval Research Logistics Quarterly 18(1), 37 (1971)
4. Pourbohloul, B., Meyers, L., Skowronski, D., Krajden, M., Patrick, D., Brunham, R.: Modeling control strategies of respiratory pathogens. Emerg. Infect. Dis. 11(8), 1246–1256 (2005)
5. Farley, J.D.: Breaking Al Qaeda cells: A mathematical analysis of counterterrorism operations (a guide for risk assessment and decision making). Studies in Conflict and Terrorism 26, 399–411 (2003)
6. Pan, F., Charlton, W., Morton, D.P.: Interdicting smuggled nuclear material. In: Woodruff, D. (ed.) Network Interdiction and Stochastic Integer Programming, pp. 1–19. Kluwer Academic Publishers, Boston (2003)
7. Ball, M.O., Golden, B.L., Vohra, R.V.: Finding the most vital arcs in a network. Oper. Res. Lett. 8(2), 73–76 (1989)
8. Bar-Noy, A., Khuller, S., Schieber, B.: The complexity of finding most vital arcs and nodes. Technical report, University of Maryland, College Park, MD, USA (1995)
9. Boros, E., Borys, K., Gurevich, V.: Inapproximability bounds for shortest-path network interdiction problems. Technical report, Rutgers University, Piscataway, NJ, USA (2006)
10. Gutfraind, A., Hagberg, A., Izraelevitz, D., Pan, F.: Interdicting a Markovian evader (preprint) (2009)
11. Grinstead, C.M., Snell, J.L.: Introduction to Probability. Second revised edn. American Mathematical Society, USA (July 1997)
12. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations, pp. 85–103. Plenum, New York (1972)
13. Nemhauser, G., Wolsey, L., Fisher, M.: An analysis of the approximations for maximizing submodular set functions-I. Mathematical Programming 14, 265–294 (1978)
14. Khuller, S., Moss, A., Naor, J.S.: The budgeted maximum coverage problem. Information Processing Letters 70(1), 39–45 (1999)
15. Krause, A., Guestrin, C.: A note on the budgeted maximization on submodular functions. Technical report, Carnegie Mellon University, CMU-CALD-05-103 (2005)
16. Leskovec, J., Krause, A., Guestrin, C., Faloutsos, C., VanBriesen, J., Glance, N.: Cost-effective outbreak detection in networks. In: KDD 2007: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 420–429. ACM, New York (2007)
17. Bradonjić, M., Kong, J.S.: Wireless ad hoc networks with tunable topology. In: Forty-Fifth Annual Allerton Conference, UIUC, Illinois, USA, pp. 1170–1177 (2007)

Appendix: Mixed Integer Program for UME

In the unreactive Markovian evader interdiction (UME) problem an evader $k \in K$ is sampled from a source distribution $\mathbf{a}^{(k)}$, and moves to a sink $t^{(k)}$ with a path specified by the matrix $\mathbf{M}^{(k)}$. This matrix is the Markov transition matrix with zeros in the row of the absorbing state (sink). The probability that the evader arrives at $t^{(k)}$ is $(\mathbf{a}^{(k)}(\mathbf{I} - \mathbf{M}^{(k)})^{-1})_{t^{(k)}}$ and is 1 without any interdiction (removal of edges).

Notation summary

$G(N,E)$: simple graph with node and edge sets N and E, respectively.
K: the set of evaders.
$w^{(k)}$: probability that the evader k occurs.
$a_i^{(k)}$: probability that node i is the source node of evader k.

$t^{(k)}$: the sink of evader k.
$\mathbf{M}^{(k)}$: the modified transition matrix for the evader k.
d_{ij}: the conditional probability that interdiction of edge (i,j) would remove an evader who traverses it.
B: the interdiction budget.
$\pi_i^{(k)}$: decision variable on conditional probability of node evader k traversing node i.
r_{ij}: interdiction decision variable, 1 if edge (i,j) is interdicted and 0 otherwise.

Definition 4. Unreactive Markovian Evader *interdiction (UME) problem*

$$\begin{aligned} \min_{\mathbf{r}} \ & H(\mathbf{r}) = \sum_{k \in K} w^{(k)} h^{(k)}(\mathbf{r}), \\ \text{s.t.} \ & \sum_{(i,j) \in E} r_{ij} = B, \\ & r_{ij} \in \{0,1\}, \ \forall (i,j) \in E, \end{aligned}$$

where

$$\begin{aligned} h^{(k)}(\mathbf{r}) = \ & \min_{\pi} \pi_{t^{(k)}}, \\ \text{s.t.} \ & \pi_i^{(k)} - \sum_{(j,i) \in E} (M_{ji}^{(k)} - M_{ji}^{(k)} d_{ji} r_{ji}) \pi_j^{(k)} = a_i^{(k)}, \ \forall i \in N, \qquad (19) \\ & \pi_i^{(k)} \geq 0, \ \forall i \in N. \qquad (20) \end{aligned}$$

The constraint (19) is nonlinear. We can replace this with a set of linear constraints, and the evader problem becomes

$$\begin{aligned} h^{(k)}(\mathbf{r}) = \ & \min_{\pi, \theta} \pi_{t^{(k)}}, \\ \text{s.t.} \ & \pi_i^{(k)} - \sum_{(j,i) \in E} \theta_{ji}^{(k)} = a_i^{(k)}, \ \forall i \in N, \\ & \theta_{ji}^{(k)} \geq M_{ji}^{(k)} \pi_j^{(k)} - M_{ji}^{(k)} d_{ji} r_{ji}, \ \forall (j,i) \in E, \qquad (21a) \\ & \theta_{ji}^{(k)} \geq M_{ji}^{(k)} (1 - d_{ji}) \pi_j^{(k)}, \ \forall (j,i) \in E, \qquad (21b) \\ & \theta_{ij}^{(k)} \geq 0, \ \forall (i,j) \in E, \\ & \pi_i^{(k)} \geq 0, \ \forall i \in N. \end{aligned}$$

If we set $r_{ij} = 0$, the constraint (21a) is dominating (21b), and θ_{ij} will take value $M_{ij}^{(k)} \pi_i^{(k)}$ at optimal because of the minimization. If we set $r_{ij} = 1$, the constraint (21b) is dominating since $\pi_j^{(k)} \leq 1$. Although formulation (21) has an additional variable θ, at the optimum the two formulations are equivalent because π and $\mathbf{r}$ have the same values.

Using Model Counting to Find Optimal Distinguishing Tests

Stefan Heinz[1,⋆] and Martin Sachenbacher[2]

[1] Zuse Institute Berlin, Takustr. 7, 14195 Berlin, Germany
heinz@zib.de
[2] Technische Universität München, Institut für Informatik
Boltzmannstraße 3, 85748 Garching, Germany
sachenba@in.tum.de

Abstract. Testing is the process of stimulating a system with inputs in order to reveal hidden parts of the system state. In the case of non-deterministic systems, the difficulty arises that an input pattern can generate several possible outcomes. Some of these outcomes allow to distinguish between different hypotheses about the system state, while others do not.

In this paper, we present a novel approach to find, for non-deterministic systems, modeled as constraints over variables, tests that allow to distinguish among the hypotheses as good as possible. The idea is to assess the quality of a test by determining the ratio of distinguishing (good) and not distinguishing (bad) outcomes. This measure refines previous notions proposed in the literature on model-based testing and can be computed using model counting techniques. We propose and analyze a greedy-type algorithm to solve this test optimization problem, using existing model counters as a building block. We give preliminary experimental results of our method, and discuss possible improvements.

1 Introduction

In natural sciences, it often occurs that one has several different hypotheses (models) for a system or parts of its state. *Testing* asks whether one can reduce their number by stimulating the system with appropriate inputs, called test patterns, in order to validate or falsify hypotheses from observing the generated outputs. Applications include, for example, model-based fault analysis (checking technical systems for the absence or presence of faults [9,17]), autonomous systems (acquiring sensory inputs to discriminate among competing state estimates [4]), and bioinformatics (designing experiments that help to distinguish between different possible explanations of biological phenomena [18]).

For deterministic systems where each input generates a unique output, such as digital circuits, it has been shown how generating test inputs can be formulated and solved as a satisfiability problem [6,11]. The *non-deterministic* case, however,

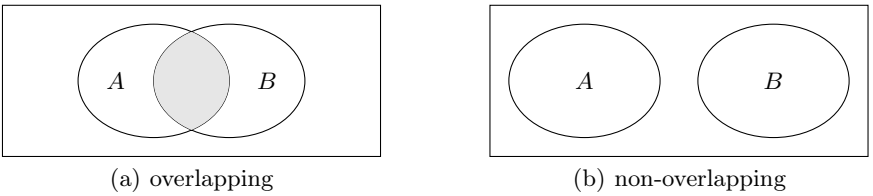
⋆ Supported by the DFG Research Center Matheon *Mathematics for key technologies* in Berlin.

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 117–131, 2009.

where the output is not uniquely determined by the inputs, is more frequent in practice. One reason is that in order to reduce the size of a model, for example, to fit it into an embedded controller [14,20], it is common to aggregate the domains of system variables into small sets of values such as 'low', 'med', and 'high'; a side-effect of this abstraction is that the resulting models can no longer be assumed to be deterministic functions, even if the underlying system behavior was deterministic [19]. Another reason is the test situation itself: even in a rigid environment such as an automotive test-bed, there are inevitably variables or parameters that cannot be completely controlled while testing the device.

The difficulty of test generation with non-deterministic models is that each input pattern can generate a set of possible outcomes instead of a single outcome. For two hypotheses and a fixed test input, let A and B be the sets of possible outputs. These sets can either overlap or be disjoint as illustrated in Figure 1. Assuming that at least one hypothesis captures the actual behavior of the system, there are two possible cases: (i) the actual observed output of the system could either fall into the intersection of A and B or (ii) outside the intersection. In the first case no information is gained, as none of the hypotheses can be refuted. In the latter case, however, one of the hypotheses can be refuted. Thus, if the sets overlap as depicted in Figure 1(a), the test input *might* distinguish between the two hypotheses, whereas if the sets are disjunct as shown in Figure 1(b), the test input *will certainly* distinguish among them. Note that, if the assumption that at least one hypothesis captures the actual behavior of the system fails, there is a third possible outcome, where the observed output lies outside of both sets. In this case, both hypotheses can be refuted since they do not describe the actual behavior of the system.

This *qualitative* distinction of tests for non-deterministic models was noted in several research areas. In the field of model-based diagnosis with first-order logical models, Struss [17] introduced so-called *possibly* and *definitely discriminating tests*, for short PDT and DDT, respectively. The first type of test (PDT) might distinguish between fault hypotheses and corresponds to Figure 1(a), whereas the second type (DDT) will necessarily do so, which corresponds to Figure 1(b). Struss [17] further provided a characterization of PDTs and DDTs in terms of relational (logical) models, together with an ad-hoc algorithm to compute them. In the field of automata theory, Alur et al. [3] have studied the analogous problem of generating so-called *weak* and *strong distinguishing sequences*. These are



(a) overlapping (b) non-overlapping

Fig. 1. Given two non-deterministic models, a test input can either lead to overlapping (a) or non-overlapping (b) output sets A and B

input sequences for a non-deterministic finite state machine, such that based on the generated outputs, one can determine the internal state either for some or all feasible runs of the machine. Finding weak and strong sequences with a length less than or equal to a bound $k \in \mathbb{N}$ can be reduced to the problem of finding PDTs and DDTs, by unrolling automata into a constraint network using k copies of the transition relation and the observation relation [9].

In previous work [13], we have shown how PDTs and DDTs can be formalized and computed using *quantified constraint satisfaction problems*, a game-theoretic extension of constraint satisfaction problems. In the next section, we summarize this constraint-based framework for testing. In section 3, we then propose a novel, *quantitative* distinction of tests that refines and generalizes the previous notions of weak versus strong and possibly versus definitely discriminating tests. The key idea is to measure the quality of a test by determining the actual ratio of distinguishing and not distinguishing outcomes, corresponding to the ratio of non-intersecting and intersecting areas in Figure 1. Because test inputs that maximize this measure distinguish among given hypotheses as good as possible, we call them *optimal distinguishing tests* (ODTs). We show how in a constraint-based framework, ODTs can be defined and computed using model counting techniques. In Section 4, we propose a greedy algorithm that can quickly find distinguishing tests, using existing model counters as a building block (in our experiments, we used a model counting extension of a constraint integer programming solver SCIP [1,2]). We give preliminary experimental results of our method using a small real-world problem from automotive industry. Finally, in the last section we discuss possible improvements and directions for future work.

2 Distinguishing Tests

We briefly introduce the theory of constraint-based testing similar to [13,17]. We first define the notion of a constraint satisfaction problem (CSP).

Definition 1 (Constraint Satisfaction Problem). *A* constraint satisfaction problem M *is a triple* $M = (\mathcal{V}, \mathcal{D}, \mathcal{C})$, *where* $\mathcal{D} = D(v_1) \times \ldots \times D(v_n)$ *are the finite domains of finitely many variables* $v_j \in \mathcal{V}$, $j = 1, \ldots, n$, *and* $\mathcal{C} = \{C_1, \ldots, C_m\}$ *is a finite set of constraints with* $C_i \subseteq \mathcal{D}$, $i = 1, \ldots, m$. *The task is to find an* assignment $\boldsymbol{x} \in \mathcal{D}$ *to the variables such that all constraints are satisfied, that is,* $\boldsymbol{x} \in C_i$ *for* $i = 1, \ldots, m$.

We denote by X the set of all solutions of a given constraint satisfaction problem. That is,

$$X = \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathcal{D},\ \mathcal{C}(\boldsymbol{x})\},\ \text{with } \mathcal{C}(\boldsymbol{x}) :\Leftrightarrow \boldsymbol{x} \in C_i\ \forall i = 1, \ldots, m.$$

Testing attempts to discriminate between hypotheses about a system – for example, about different kinds of faults – by stimulating it in such a way that the hypotheses become observationally distinguishable. Thereby, the system under investigation defines a set of *controllable* (input) variables $\mathcal{I}$ and a set of *observable* (output) variables $\mathcal{O}$. Formally, a hypothesis M for a system is a CSP where the variable set $\mathcal{V}$ contains the input and output variables of the system.

Definition 2 (Hypothesis). *A* hypothesis *for a system is a CSP whose variables are partitioned into* $\mathcal{V} = \mathcal{I} \cup \mathcal{O} \cup \mathcal{S}$*, such that* $\mathcal{I}$ *and* $\mathcal{O}$ *are the input and output variables of the system, and for all assignments to* $\mathcal{I}$*, the CSP is solvable. The remaining variables* $\mathcal{S} = \mathcal{V} \setminus (\mathcal{I} \cup \mathcal{O})$ *are called internal state variables.*

Note that the internal state variable sets $\mathcal{S}$ can differ for two different hypotheses. We denote by $\mathcal{D}(\mathcal{I})$ and $\mathcal{D}(\mathcal{O})$ the cross product of the domains of the input and output variables, respectively:

$$\mathcal{D}(\mathcal{I}) = \bigtimes_{v \in \mathcal{I}} D(v) \quad \text{and} \quad \mathcal{D}(\mathcal{O}) = \bigtimes_{v \in \mathcal{O}} D(v).$$

The goal of testing is then to find assignments of the input variables $\mathcal{I}$ that will cause different assignments of output variables $\mathcal{O}$ for different hypotheses. For a given hypothesis M we define the *output function* $\mathcal{X}$ as follows:

$$\mathcal{X} : \mathcal{D}(\mathcal{I}) \to 2^{\mathcal{D}(\mathcal{O})} \text{ with } \boldsymbol{t} \mapsto \{\boldsymbol{y} \mid \boldsymbol{y} \in \mathcal{D}(\mathcal{O}), \exists\, \boldsymbol{x} \in X : \boldsymbol{x}[\mathcal{I}] = \boldsymbol{t} \wedge \boldsymbol{x}[\mathcal{O}] = \boldsymbol{y}\},$$

where $2^{\mathcal{D}(\mathcal{O})}$ denotes the power set of $\mathcal{D}(\mathcal{O})$, and $\boldsymbol{x}[\mathcal{I}]$, $\boldsymbol{x}[\mathcal{O}]$ denote the restriction of the assignment vector $\boldsymbol{x}$ to the input variables $\mathcal{I}$ and the output variables $\mathcal{O}$, respectively. Note that since M will always yield an output, $\mathcal{X}(\boldsymbol{t})$ is non-empty for all $\boldsymbol{t} \in \mathcal{D}(\mathcal{I})$.

Definition 3 (Distinguishing Tests). *Consider* $k \in \mathbb{N}$ *hypotheses* $M_1, \ldots, M_k$ *with input variables* $\mathcal{I}$ *and output variables* $\mathcal{O}$*. Let* $\mathcal{X}_i$ *be the output function of hypothesis* M_i *with* $i \in \{1, \ldots, k\}$*. An assignment* $\boldsymbol{t} \in \mathcal{D}(\mathcal{I})$ *to the input variables* $\mathcal{I}$ *is a* possibly distinguishing test *(PDT), if there exists an* $i \in \{1, \ldots, k\}$ *such that*

$$\mathcal{X}_i(\boldsymbol{t}) \setminus \bigcup_{j \neq i} \mathcal{X}_j(\boldsymbol{t}) \neq \varnothing.$$

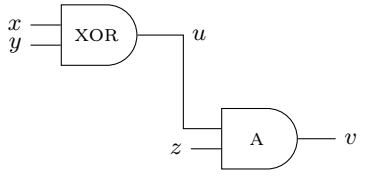
An assignment $\boldsymbol{t} \in \mathcal{D}(\mathcal{I})$ *is a* definitely distinguishing test *(DDT), if for all* $i \in \{1, \ldots, k\}$ *it holds that*

$$\mathcal{X}_i(\boldsymbol{t}) \setminus \bigcup_{j \neq i} \mathcal{X}_j(\boldsymbol{t}) = \mathcal{X}_i(\boldsymbol{t}).$$

Verbally, a test input is a PDT if there exists a hypothesis for which this test input can lead to an output which is not reachable for any other hypothesis. On the other hand, an assignment to the input variables is a DDT if for all hypotheses the possible outputs are pairwise disjoint. This means, there exists no overlapping of the possible outcomes at all.

In the following, we restrict ourselves to the case where there are only two possible hypotheses, for example corresponding to normal and faulty behavior of the system.

To illustrate the above definitions, consider the system in Figure 2. It consists of five variables x, y, z, u, and v, where x, y, and z are input variables and v is an output variable. Furthermore, the system has two components, one comparing



XOR:

x	y	u
L	L	L
L	H	H
H	L	H
H	H	L

A:

u	z	v
L	L	L
L	H	L
L	H	H
H	L	L
H	L	H
H	H	H

A′:

u	z	v
L	L	L
L	H	L
H	L	L
H	H	L

Fig. 2. Circuit with a possibly faulty adder

signals x and y with result u and the other adding signals u and z. The signals have been abstracted into qualitative values 'low' (L) and 'high' (H). This means, each variable of the system has the same domain set $\{L, H\}$; thus, for instance, values L and H can add up to the value L or H, and so on. Assume we have two hypotheses M_1 and M_2 about the system that we want to distinguish from each other: the first hypothesis is that the system is functioning normally, which is modeled by the constraint set $\{\text{XOR}, \text{A}\}$ (see Figure 2). The second hypothesis is that the adder is *stuck-at-L*, which is modeled by the constraints $\{\text{XOR}, \text{A}'\}$. Note that only the second constraint of both hypotheses contains a non-deterministic behavior. The assignment $(x, y, z) = (L, H, L)$, for example, is a PDT, since it leads to the observation $v = L$ or $v = H$ for M_1, and $v = L$ for M_2. One the other hand, the assignment $(x, y, z) = (L, H, H)$ is a DDT for the two hypotheses, since this assignment leads to the observation $v = H$ and $v = L$ for the hypotheses M_1 and M_2, respectively.

Testing can be extended from the above case of logical, state-less models to the more general case of *automata models* that have internal states. This means that we are no longer searching for a single assignment to input variables, but rather for a sequence of inputs over different time steps. The following definitions are adapted from [5] and [7]:

Definition 4 (Plant Hypothesis). *A (partially observable) plant is a tuple* $P = \langle x_0, S, I, \delta, O, \lambda \rangle$*, where* S, I, O *are finite sets, called the* state space*,* input space*, and* output space*, respectively,* $x_0 \in S$ *is the* start state*,* $\delta \subseteq S \times I \times S$ *is the* transition relation*, and* $\lambda \subseteq S \times O$ *is the* observation relation*.*

Such plant models are for instance used in NASA's Livingstone [21] or MIT's Titan model-based system [20]. Note that a plant need not be deterministic, that is, the state after a transition may not be uniquely determined by the state before the transition and the input. Likewise, a plant state may be associated with several possible observations.

For technical convenience, it is assumed that the relations δ and λ are *complete*, that is for every $x \in S$ and $i \in I$ there exists at least one $x' \in S$ such that $(x, i, x') \in \delta$ and at least one $o \in O$ such that $(x, o) \in \lambda$. We write $\delta(x, i, x')$ for $(x, i, x') \in \delta$, and $\lambda(s, o)$ for $(x, o) \in \lambda$. A *feasible trace* of a plant P is a pair (σ, ρ), where $\sigma = i_1, i_2, \ldots, i_k \in I^*$ is a sequence of k inputs and $\rho = o_0, o_1, \ldots, o_k \in O^*$

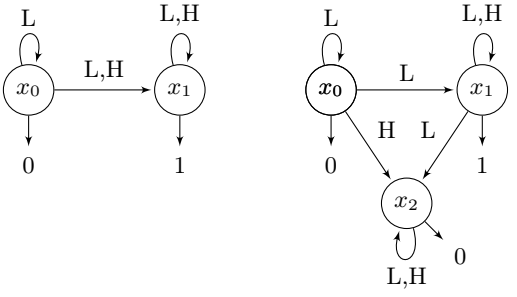


Fig. 3. Two plants P_1 (left) and P_2 (right)

is a sequence of $k+1$ outputs, such that there exists a sequence $x_0, x_1, \ldots, x_k$ of states with $\delta(x_{j-1}, i_j, x_j)$ for all $1 \leq j \leq k$ and $\lambda(x_j, o_j)$ for all $0 \leq j \leq k$.

Definition 5 (Distinguishing Test Sequences). *Given two plants* $P_1 = \langle x_0, S, I, \delta, O, \lambda \rangle$ *and* $P_2 = \langle y_0, Y, I, \eta, O, \mu \rangle$*, a sequence of inputs* $\sigma \in I^*$ *is a* weak test*, if there exists a sequence of outputs* $\rho \in O^*$ *such that* (σ, ρ) *is a feasible trace of* P_1 *but not of* P_2*. The sequence* σ *is a* strong test *for* P_1 *and* P_2*, if and only if for all sequences of outputs* ρ*, it holds that if* (σ, ρ) *is a feasible trace* P_1 *then it is not a feasible trace of* P_2*.*

Notice that due to the assumptions about completeness, for every input sequence $\sigma \in I^*$ there exist output sequences $\rho, \tau \in O^*$ such that (σ, ρ) is a feasible trace of P_1 and (σ, τ) is a feasible trace of P_2.

Analogous to PDTs and DDTs, a weak test is a sequence that *may* reveal a difference between two hypotheses, whereas a strong test is a sequence that *will necessarily* do so. For example, Figure 3 shows two plants P_1 and P_2 with $I = \{L, H\}$ and $O = \{0, 1\}$. The input sequence $\sigma = L, L$ is a weak test for the two plants, because, for example, $0, 1, 0$ is a possible output sequence of P_2 but not of P_1. The sequence $\sigma' = H, H$ is a strong test for P_2 and P_1, because the only possible output sequence $0, 0, 0$ of P_2 cannot be produced by P_1.

From a practical point of view, it is often sufficient to consider *bounded test sequences* that do not exceed a certain length $k \in \mathbb{N}$. In this case, the problem of finding weak and strong tests for automata models can be reduced to finding PDTs and DDTs:

Remark 1. Finding weak and strong tests with a length less than or equal to a bound $k \in \mathbb{N}$ can be reduced to the problem of finding PDTs and DDTs, by unrolling automata into a constraint network using k copies of the transition relation and the observation relation [9].

In the following, we consider only tests with such a bounded length. Therefore, we assume the hypotheses are given as CSPs over finite-domain variables (Definition 2). This covers both the case of logical models and (bounded) automata models.

3 Optimal Distinguishing Tests

In [13], we have shown how PDTs and DDTs can be formalized and computed using *quantified constraints satisfaction problems* (QCSP), a game-theoretic extension of CSPs. However, for larger hypotheses, the computational cost of solving such QCSPs can be prohibitive. Moreover, due to limited observability or a high degree of non-determinism in the system under investigation, it is not uncommon that a DDT for the hypotheses does not exist, and one can instead only find PDTs.

In the following, we therefore propose a novel, *quantitative measure* for tests that refines and generalizes the previous, qualitative notions of PDTs and DDTs. The key idea is to determine the ratio of distinguishing and not distinguishing outcomes of a test input, corresponding to the degree of overlap between the output sets shown in Figure 1. This measure provides a way to further distinguish between different PDTs. In addition, even if computing this measure is by itself not easier than finding PDTs and DDTs, approximations of it can be used as a *guiding heuristic* in the search for tests, providing a basis for greedy methods to quickly find good tests.

The main assumption underlying our approach is that for a test input and a non-deterministic hypothesis, the possible outcomes (feasible assignments to the output variables) are all (roughly) equally likely. Then, a PDT will be more likely to distinguish among two given hypotheses compared to another PDT, if the ratio of possible outcomes that are unique to a hypothesis versus the total number of possible outcomes is higher.

This intuition is captured in the following definitions.

Definition 6 (Distinguishing Ratio). *Given a test input $\boldsymbol{t} \in \mathcal{D}(\mathcal{I})$ for two hypotheses M_1, M_2 with input variables $\mathcal{I}$ and output variables $\mathcal{O}$, we define $\Gamma(\boldsymbol{t})$ to be the ratio of feasible outputs that distinguish among the hypotheses versus all feasible outputs:*

$$\Gamma(\boldsymbol{t}) := \frac{|\mathcal{X}_1(\boldsymbol{t}) \cup \mathcal{X}_2(\boldsymbol{t})| - |\mathcal{X}_1(\boldsymbol{t}) \cap \mathcal{X}_2(\boldsymbol{t})|}{|\mathcal{X}_1(\boldsymbol{t}) \cup \mathcal{X}_2(\boldsymbol{t})|} = 1 - \frac{|\mathcal{X}_1(\boldsymbol{t}) \cap \mathcal{X}_2(\boldsymbol{t})|}{|\mathcal{X}_1(\boldsymbol{t}) \cup \mathcal{X}_2(\boldsymbol{t})|}.$$

Γ is a measure for test quality that can take on values in the interval $[0, 1]$. It refines the notion of PDTs and DDTs in the following precise sense: if Γ is 0, then the test does not distinguish at all, as both hypotheses lead to the same observations (output patterns). If the value is 1, then the test is a DDT, since both hypotheses always lead to different observations. If the value is between 0 and 1, then the test is a PDT (there is some non-overlap in the possible observations). Note that Γ is well-defined since for any chosen $\boldsymbol{t} \in \mathcal{D}(\mathcal{I})$, the sets $\mathcal{X}_1(\boldsymbol{t})$ and $\mathcal{X}_2(\boldsymbol{t})$ are non-empty (see Definition 2).

Remark 2. For computing the distinguishing ratio for a fixed test input $\boldsymbol{t}$ it is only necessary to compute (model count) the value $|\mathcal{X}_1(\boldsymbol{t}) \cap \mathcal{X}_2(\boldsymbol{t})|$, $|\mathcal{X}_1(\boldsymbol{t})|$, and $|\mathcal{X}_2(\boldsymbol{t})|$, since

$$\Gamma(\boldsymbol{t}) = 1 - \frac{|\mathcal{X}_1(\boldsymbol{t}) \cap \mathcal{X}_2(\boldsymbol{t})|}{|\mathcal{X}_1(\boldsymbol{t}) \cup \mathcal{X}_2(\boldsymbol{t})|} = 1 - \frac{|\mathcal{X}_1(\boldsymbol{t}) \cap \mathcal{X}_2(\boldsymbol{t})|}{|\mathcal{X}_1(\boldsymbol{t})| + |\mathcal{X}_2(\boldsymbol{t})| - |\mathcal{X}_1(\boldsymbol{t}) \cap \mathcal{X}_2(\boldsymbol{t})|}.$$

Based on this measure, we can formalize our goal of finding tests that discriminate among two hypotheses as good as possible:

Definition 7 (Optimal Distinguishing Test). *An assignment* $\boldsymbol{t} \in \mathcal{D}(\mathcal{I})$ *is an* optimal distinguishing test (ODT) *for two hypotheses* M_1, M_2 *with input variables* $\mathcal{I}$ *and output variables* $\mathcal{O}$ *if its distinguishing ratio is maximal, that is,* $\Gamma(\boldsymbol{t}) = \max_{\boldsymbol{x} \in \mathcal{D}(\mathcal{I})} \Gamma(\boldsymbol{x})$.

Note that each DDT is also an ODT. To illustrate the previous definition, consider again the example in Figure 3. The input sequence $\boldsymbol{t} = (L, L)$ is a weak test or equivalently, a PDT if the automata are expanded into suitable constraint networks. The possible outcomes (output patterns) for P_1 and P_2 are

$$\mathcal{X}_1(\boldsymbol{t}) = \{(0,0,0),(0,0,1),(0,1,1)\} \quad \mathcal{X}_2(\boldsymbol{t}) = \{(0,0,0),(0,0,1),(0,1,1),(0,1,0)\}.$$

Thus, for this test there is only one possible outcome $(0,1,0)$ that is unique to a hypothesis, out of a total of four possible outcomes. Hence, $\Gamma(\boldsymbol{t}) = \frac{1}{4}$. There exists another weak test (PDT), namely the input sequence $\boldsymbol{t'} = (L, H)$, with possible outcomes

$$\mathcal{X}_1(\boldsymbol{t'}) = \{(0,0,1),(0,1,1)\} \quad \mathcal{X}_2(\boldsymbol{t'}) = \{(0,0,0),(0,1,1)\}.$$

This test has two possible outcomes $\{(0,0,0),(0,0,1)\}$ that are unique to a hypothesis, out of three possible outcomes $\{(0,0,0),(0,0,1),(0,1,1)\}$. This leads to $\Gamma(\boldsymbol{t'}) = \frac{2}{3}$. Note that for this example, there exists a test $\boldsymbol{t''} = (H, H)$ with $\Gamma(\boldsymbol{t''}) = 1$, which is a DDT and therefore an ODT.

Now we present a general lower bound on the optimal distinguishing ratio.

Theorem 1. *Consider a system with input variable set* $\mathcal{I}$ *and output variable set* $\mathcal{O}$. *Furthermore, let* M_1 *and* M_2 *be two hypotheses for this system. Let*

$$X_i[\mathcal{I},\mathcal{O}] = \{(\boldsymbol{x},\boldsymbol{y}) \mid \boldsymbol{x} \in \mathcal{D}(\mathcal{I}),\ \boldsymbol{y} \in \mathcal{D}(\mathcal{O}),\ \exists\, \boldsymbol{t} \in X_i : \boldsymbol{t}[\mathcal{I}] = \boldsymbol{x} \wedge \boldsymbol{t}[\mathcal{O}] = \boldsymbol{y}\},$$

where X_i *is the set of all feasible solutions of the hypothesis* M_i, $i \in \{1,2\}$. *Then,*

$$1 - \frac{|X_1[\mathcal{I},\mathcal{O}] \cap X_2[\mathcal{I},\mathcal{O}]|}{|X_1[\mathcal{I},\mathcal{O}] \cup X_2[\mathcal{I},\mathcal{O}]|}$$

is a lower bound on the optimal distinguishing ratio.

Proof. Let $\mathcal{I}$ and $\mathcal{O}$ be the input and output variable sets of an arbitrary system and M_1 and M_2 two hypotheses. Furthermore, let $X_1[\mathcal{I},\mathcal{O}]$ and $X_2[\mathcal{I},\mathcal{O}]$ be the sets to the hypotheses as defined in the theorem.

Given an input variable $v \in \mathcal{I}$, we denote by T_d the subset of $\mathcal{D}(\mathcal{I})$ which is restricted to the elements where the input variable v is fixed to $d \in D(v)$. That is, $T_d = \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathcal{D}(\mathcal{I}) \wedge \boldsymbol{x}[\{v\}] = d\}$. These subsets form a partition of $\mathcal{D}(\mathcal{I})$. This means, $\mathcal{D}(\mathcal{I}) = \bigcup_{d \in D(v)} T_d$ and $T_d \cap T_k = \varnothing$ for all $d, k \in D(v)$ with $d \neq k$. Hence, these subsets can be used to partition $X_i[\mathcal{I},\mathcal{O}]$ as follows:

$$X_i[\mathcal{I},\mathcal{O}] = \bigcup_{d \in D(v)} \{(\boldsymbol{x},\boldsymbol{y}) \mid \boldsymbol{x} \in T_d,\ \boldsymbol{y} \in \mathcal{D}(\mathcal{O}),\ \exists\, \boldsymbol{t} \in X_i : \boldsymbol{t}[\mathcal{I}] = \boldsymbol{x} \wedge \boldsymbol{t}[\mathcal{O}] = \boldsymbol{y}\}.$$

Therefore,

$$|X_i[\mathcal{I},\mathcal{O}]| = \sum_{d\in D(v)} |\{(\boldsymbol{x},\boldsymbol{y}) \mid \boldsymbol{x}\in T_d,\, \boldsymbol{y}\in\mathcal{D}(\mathcal{O}),\, \exists\boldsymbol{t}\in X_i : \boldsymbol{t}[\mathcal{I}]=\boldsymbol{x}\wedge\boldsymbol{t}[\mathcal{O}]=\boldsymbol{y}\}|.$$

We claim that

$$\frac{|X_1[\mathcal{I},\mathcal{O}]\cap X_2[\mathcal{I},\mathcal{O}]|}{|X_1[\mathcal{I},\mathcal{O}]\cup X_2[\mathcal{I},\mathcal{O}]|} \geq \min_{d\in D(v)} \frac{|X_1[\mathcal{I},\mathcal{O}]\cap X_2[\mathcal{I},\mathcal{O}]\cap(T_d\times\mathcal{D}(\mathcal{O}))|}{|(X_1[\mathcal{I},\mathcal{O}]\cup X_2[\mathcal{I},\mathcal{O}])\cap(T_d\times\mathcal{D}(\mathcal{O}))|}.$$

To this end, let $d^*\in D(v)$ be a domain value of v, which attains the minimum on the right hand side. In a first step, we decompose the left hand side using that the subsets T_d are a partition of $\mathcal{D}(\mathcal{I})$:

$$\frac{|X_1[\mathcal{I},\mathcal{O}]\cap X_2[\mathcal{I},\mathcal{O}]|}{|X_1[\mathcal{I},\mathcal{O}]\cup X_2[\mathcal{I},\mathcal{O}]|} = \frac{\sum_{d\in D(v)}|X_1[\mathcal{I},\mathcal{O}]\cap X_2[\mathcal{I},\mathcal{O}]\cap(T_d\times\mathcal{D}(\mathcal{O}))|}{\sum_{d\in D(v)}|(X_1[\mathcal{I},\mathcal{O}]\cup X_2[\mathcal{I},\mathcal{O}])\cap(T_d\times\mathcal{D}(\mathcal{O}))|}$$

Now we substitute

$$\begin{aligned} a_d &:= |X_1[\mathcal{I},\mathcal{O}]\cap X_2[\mathcal{I},\mathcal{O}]\cap(T_d\times\mathcal{D}(\mathcal{O}))| \\ \tilde{a}_d &:= |(X_1[\mathcal{I},\mathcal{O}]\cup X_2[\mathcal{I},\mathcal{O}])\cap(T_d\times\mathcal{D}(\mathcal{O}))|. \end{aligned}$$

To prove the claim, it is left to show that

$$\frac{\sum_{d\in D(v)} a_d}{\sum_{d\in D(v)}\tilde{a}_d} \geq \frac{a_{d^*}}{\tilde{a}_{d^*}} \quad\text{with}\quad \frac{a_d}{\tilde{a}_d}\geq\frac{a_{d^*}}{\tilde{a}_{d^*}} \quad \forall\, d\in D(v).$$

This follows since

$$\frac{\sum_{d\in D(v)} a_d}{\sum_{d\in D(v)}\tilde{a}_d} \geq \frac{a_{d^*}}{\tilde{a}_{d^*}} \Leftrightarrow \sum_{d\in D(v)} a_d \geq \sum_{d\in D(v)} \tilde{a}_d\cdot\frac{a_{d^*}}{\tilde{a}_{d^*}}$$

and

$$\sum_{d\in D(v)} a_d = \sum_{d\in D(v)} \frac{a_d\cdot\tilde{a}_d}{\tilde{a}_d} \geq \sum_{d\in D(v)} \frac{a_{d^*}\cdot\tilde{a}_d}{\tilde{a}_{d^*}} = \sum_{d\in D(v)} \tilde{a}_d\cdot\frac{a_{d^*}}{\tilde{a}_{d^*}}.$$

Therefore, we have proven, that it is possible to fix any input variable such that the claimed lower bound holds. Doing this sequentially for all input variables leads to an assignment which has as distinguishing ratio which is at least as good as the claimed lower bound. □

4 Greedy Algorithm for Distinguishing Test Generation

In the previous section we stated the optimization problem of computing an optimal distinguishing test (ODT). In this section, we propose and analyze a *greedy-type* algorithm to solve this problem, which can use existing model counting methods (exact or approximate) as a building block.

The idea of the greedy algorithm is to select at each step an input variable which is not fixed yet. For each possible value of this variable, the algorithm computes a local form of the distinguishing ratio (comparison of model counts, as defined in Section 3) for assigning this value. The variable is then fixed to a value that attains the maximal (local) distinguishing ratio.

To formalize this idea, we canonically extend the function Γ from single assignments $\boldsymbol{t} \in \mathcal{D}(\mathcal{I})$ to sets of assignments $T \subseteq \mathcal{D}(\mathcal{I})$ by defining

$$\mathcal{X}(T) = \bigcup_{\boldsymbol{t} \in T} \mathcal{X}(\boldsymbol{t}).$$

Algorithm 1 shows the algorithm GREEDY. It takes as input the controllable and observable variable sets $\mathcal{I}$ and $\mathcal{O}$ defined by the system under investigation and two hypotheses M_1 and M_2 to distinguish. As output it returns an assignment for the input variables.

For example, consider the system shown in Figure 4. It has two input variables $\mathcal{I} = \{v_1, v_2\}$ and one output variable $\mathcal{O} = \{v_3\}$. Let $D(v_1) = D(v_2) = \{0, 1\}$ and $D(v_3) = \{0, 1, 2\}$. Consider two hypotheses M_1 and M_2 for this system, where both hypotheses have no internal state variables. Each hypothesis has one constraint

$$C_1 = D(v_1) \times D(v_2) \times D(v_3)$$
$$C_2 = D(v_1) \times D(v_2) \times D(v_3) \setminus \{(0,0,1), (0,0,2), (0,1,2), (1,0,2)\},$$

where C_1 and C_2 belong to hypothesis M_1 and M_2, respectively. Assume the algorithm selects the variables in the order v_1, v_2. Then for the two values of

Input: Hypotheses M_1 and M_2 with set of input and output variables $\mathcal{I}$ and $\mathcal{O}$
Output: Test $\boldsymbol{t} \in \mathcal{D}(\mathcal{I})$

$T \leftarrow \mathcal{D}(\mathcal{I})$;
foreach $v \in \mathcal{I}$ **do**
  bestratio $\leftarrow -1$;
  bestfixing $\leftarrow \infty$;
  foreach $d \in D(v)$ **do**
    $T' \leftarrow \{\boldsymbol{x} \mid \boldsymbol{x} \in T \wedge \boldsymbol{x}[\{v\}] = d\}$;
    ratio $\leftarrow \Gamma(T')$;
    if ratio > bestratio **then**
      bestratio $\leftarrow$ ratio;
      bestfixing $\leftarrow d$;
    end
  end
  $T \leftarrow T \cap \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathcal{D}(\mathcal{I}) \wedge \boldsymbol{x}[\{v\}] = \text{bestfixing}\}$;
end
return $\boldsymbol{t} \in T$

Algorithm 1. GREEDY algorithm for distinguishing test input generation

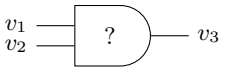


Fig. 4. Example system schema

v_1, it computes the two ratios $v_1 = 0 \rightarrow \Gamma(T') = \frac{1}{3}$ and $v_1 = 1 \rightarrow \Gamma(T') = 0$. It chooses value 0 for v_1, since it has the highest ratio. Continuing with v_2, its ratios are determined as $v_2 = 0 \rightarrow \Gamma(T') = \frac{2}{3}$ and $v_2 = 1 \rightarrow \Gamma(T') = \frac{1}{3}$. Thus value 0 for v_2 is chosen. The computed input $(0, 0)$ is an ODT for this example.

4.1 Properties of the Algorithm

Note that if the system consists only of one input variable, GREEDY computes an ODT, since the algorithm just enumerates all possible variable assignments for the input variable and selects the assignment that maximizes the distinguishing ratio. In general, however, the GREEDY algorithm has no constant approximation factor.

Theorem 2. *The* GREEDY *algorithm has no constant approximation factor. That is, there exists no constant c such that for all instances*

$$\max_{\boldsymbol{t} \in D(\mathcal{I})} \Gamma(\boldsymbol{t}) \leq c \cdot \Gamma(\boldsymbol{x}^*),$$

where $\boldsymbol{x}^*$ is the solution computed by GREEDY.

Proof. Consider again the system stated in Figure 4, and let the domain of the input variables be $D(v_1) = D(v_2) = \{0, 1\}$ and of the output variable v_3 be $D(v_3) = \{0, \ldots, n\}$, $n \in \mathbb{N}$, and $n > 2$. W.l.o.g. let the domains be ordered as:

$$\mathcal{D} = D(v_1) \times D(v_2) \times D(v_3).$$

Let M_1 and M_2 be defined by the following sets of feasible solutions:

$$X_1 = \{(0,0,0), (1,0,0)\} \cup \{(x, 1, z) \mid x \in D(v_1) \wedge z \in \{2, \ldots, n\}\} \subset \mathcal{D}$$
$$X_2 = \{(0,1,0), (1,0,0), (1,1,1)\} \cup \{(x, x, z) \mid x \in \{0,1\} \wedge z \in \{2, \ldots, n\}\} \subset \mathcal{D},$$

where X_1 and X_2 belong to hypothesis M_1 and M_2, respectively. Both hypotheses have no internal state variables. It is assumed that the GREEDY algorithm selects v_1 first. The best possible assignment for this variable is $v_1 = 1$, since $v_1 = 0 \rightarrow \Gamma(T') = 0$ and $v_1 = 1 \rightarrow \Gamma(T') > 0$. In the final step GREEDY has to fix variable v_2 with respect to the previous fixing of $v_1 = 1$. The best possible decision is, to fix v_2 also to 1, since fixing v_2 to 0 leads to an distinguishing ratio of zero and for $v_2 = 1$ we have $\Gamma((1,1)) = \frac{1}{n}$. Note that the computed test input $(v_1, v_2) = (1, 1)$ is independent of the chosen n.

An ODT for this problem, however, is $(v_1, v_2) = (0, 0)$, which is also a DDT. This test input has, therefore, a distinguishing ratio of 1. For n tending to infinity, the distinguishing ratio of the test input computed by GREEDY tends to zero. This proves that the GREEDY algorithm has in general no constant approximation factor. □

Note that if GREEDY would choose variable v_2 first, it would compute an ODT for this example. This rises the question, whether there exist always a permutation of the input variables such that the GREEDY algorithm computes an ODT. The following theorem answers this question.

Theorem 3. *In general, the* GREEDY *algorithm does not compute an ODT even if it is allowed to try all possible input variable permutations.*

Proof. Again, consider the abstract system depicted in Figure 4 with the input variable set $\mathcal{I} = \{v_1, v_2\}$, $D(v_1) = D(v_2) = \{0, 1\}$, the output variable set $\mathcal{O} = \{v_3\}$, and $D(v_3) = \{1, 2, 3, 4\}$. Let M_1 and M_2 be two hypotheses given through the constraints:

$$C_1 = \{(0,0,1), (1,0,2), (0,1,2), (1,1,2), (1,1,3), (1,1,4)\} \subset \mathcal{D}$$
$$C_2 = \{(x,y,2) \mid x,y \in \{0,1\}\} \subset \mathcal{D},$$

where C_1 belongs to hypothesis M_1, C_2 to hypothesis M_2, and $\mathcal{D} = D(v_1) \times D(v_2) \times D(v_3)$.

The test input $(v_1, v_2) = (0, 0)$ is the unique DDT and, therefore, the unique ODT. If we show that the GREEDY algorithm fixes in the first iteration, independently of the chosen input variable, this variable to 1, then we have proven the theorem.

Independently from the chosen input variable, GREEDY fixes this variable to 1 since for $i \in \{1, 2\}$ it follows $v_i = 0 \rightarrow \Gamma(T') = \frac{1}{2}$ and $v_i = 1 \rightarrow \Gamma(T') = \frac{2}{3}$. □

In the example at the beginning of Section 4, the sequence of distinguishing ratios $\Gamma(T')$ computed by GREEDY increases monotonically. This observation can also be made later in the computational results for the automotive example (see Table 2). However, this needs not be the case in general.

Theorem 4. *In general, the sequence of distinguishing ratios computed by* GREEDY *is not monotonically increasing.*

Proof. Consider the system stated in Figure 4, and let the domain of the input variables be $D(v_1) = \{0\}$, $D(v_2) = \{0, 1\}$, and of the output variable v_3 be $D(v_3) = \{0, 1, 2\}$. W.l.o.g. let the domains be ordered as:

$$\mathcal{D} = D(v_1) \times D(v_2) \times D(v_3).$$

Let M_1 and M_2 be defined by the following sets of feasible solutions:

$$X_1 = \{(0,0,0), (0,1,0), (0,1,1)\} \subset \mathcal{D}$$
$$X_2 = \{(0,0,0), (0,1,0), (0,0,2)\} \subset \mathcal{D}$$

where X_1 and X_2 belong to hypothesis M_1 and M_2, respectively. Both hypotheses have no internal state variables. It is assumed that the GREEDY algorithm selects v_1 first. Since v_1 has only one possible value in its domain, GREEDY fixes v_1 to this value 0. The (local) distinguishing ratio yields:

$$\Gamma(T) = 1 - \frac{|\{0,1\} \cap \{0,2\}|}{|\{0,1\} \cup \{0,2\}|} = \frac{2}{3} \quad \text{with} \quad T = \mathcal{D}$$

In the final step GREEDY has to fix variable v_2 with respect to the previous fixing of $v_1 = 0$:

$$\Gamma(T') = 1 - \frac{|\{0\} \cap \{0,2\}|}{|\{0\} \cup \{0,2\}|} = \tfrac{1}{2} \quad \text{with} \quad T' = \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathcal{D} \wedge \boldsymbol{x}[\{v_2\}] = 0\}$$
$$\Gamma(T'') = 1 - \frac{|\{0,1\} \cap \{0\}|}{|\{0,1\} \cup \{0\}|} = \tfrac{1}{2} \quad \text{with} \quad T'' = \{\boldsymbol{x} \mid \boldsymbol{x} \in \mathcal{D} \wedge \boldsymbol{x}[\{v_2\}] = 1\}.$$

Independently of the chosen fixing for the variable v_2, the (local) distinguishing ratio decreases. □

4.2 Computational Results

We have implemented Algorithm 1 using the constraint integer programming solver SCIP [1,2] as (exact) model counter.

We ran our prototype implementation on a small real-world automotive example. The example is based on a mixed discrete-continuous model of an engine air intake test-bed [12]. It has been turned into a coarse CSP model by abstracting continuous system variables into suitable finite domains with up to 12 values, corresponding to different operating regions. The system consists of the three major components ENGINE, PIPE, and THROTTLE; for each component, a fault model is defined that simply omits the respective constraint from the model. Thus, there are four diagnostic hypotheses (CORRECT, NO-ENGINE, NO-PIPE, and NO-THROTTLE), corresponding to all components functioning normally and one of them failing. The goal is to find an assignment to two controllable variables (throttle angle v_1, valve timing v_2), such that one can discriminate among hypotheses based on two observable variables (engine speed and air flow) in

Table 1. Model counts for the four hypotheses in the automotive example

	CORRECT	NO-ENGINE	NO-PIPE	NO-THROTTLE
$\lvert X \rvert$	329	6552	25356	8560
$\lvert X[\mathcal{I}, \mathcal{O}] \rvert$	43	552	168	127
$\lvert \mathcal{X}(\mathcal{D}(\mathcal{I})) \rvert$	13	72	41	22

Table 2. Distinguishing ratios computed by GREEDY for the automotive example

permutation	lower bound	sequence of distinguishing ratio $\Gamma(T')$ 0 iteration	1 iteration	2 iterations
CORRECT VS. NO-ENGINE				
(v_1, v_2)	$1 - \frac{43}{552} = 0.922$	$1 - \frac{13}{72} = 0.819$	$1 - \frac{3}{24} = 0.875$	$1 - \frac{1}{24} = 0.958$
(v_2, v_1)	$1 - \frac{43}{552} = 0.922$	$1 - \frac{13}{72} = 0.819$	$1 - \frac{4}{72} = 0.944$	$1 - \frac{1}{24} = 0.958$
CORRECT VS. NO-PIPE				
(v_1, v_2)	$1 - \frac{43}{168} = 0.744$	$1 - \frac{13}{41} = 0.683$	$1 - \frac{3}{27} = 0.889$	$1 - \frac{1}{15} = 0.933$
(v_2, v_1)	$1 - \frac{43}{168} = 0.744$	$1 - \frac{13}{41} = 0.683$	$1 - \frac{4}{23} = 0.826$	$1 - \frac{1}{15} = 0.933$
CORRECT VS. NO-THROTTLE				
(v_1, v_2)	$1 - \frac{43}{127} = 0.661$	$1 - \frac{13}{22} = 0.409$	$1 - \frac{3}{22} = 0.864$	$1 - \frac{1}{9} = 0.889$
(v_2, v_1)	$1 - \frac{43}{127} = 0.661$	$1 - \frac{13}{22} = 0.409$	$1 - \frac{5}{10} = 0.5$	$1 - \frac{5}{10} = 0.5$

the system. Table 1 shows the model counts (total number of solutions $|X|$ and the total number of projected solutions $|X[\mathcal{I}, \mathcal{O}]|$ and $|\mathcal{X}(\mathcal{D}(\mathcal{I}))|$) for the four hypotheses. Table 2 shows the computational results of the GREEDY algorithm for finding tests to distinguish the normal system behavior from the faults. The first column states the used permutation, the second column gives the general lower bound on the optimal distinguishing ratio, as stated in Theorem 1, and the last three columns the sequence of the distinguishing ratios as GREEDY iterates through the input variables. In all cases except the last (finding a test input to identify a NO-PIPE fault given the variable order v_2, v_1), the test input generated by the algorithm is an ODT. The last test shows that in general, the GREEDY algorithm does not compute a test input whose distinguishing ratio is at least as good as the general lower bound. The run-time of the algorithm on this example is in the order of a few seconds.

5 Conclusion and Future Work

We presented a method for generating tests to distinguish system hypotheses modeled as constraints over variables. It is based on maximizing the number of non-overlapping versus overlapping observable outcomes and extends previous notions of testing for non-deterministic systems. We showed how this proposed test quality measure can be computed as a ratio of model counts. Challenges arise from the computational cost of generating optimal distinguishing tests, since computing the optimal distinguishing ratios can be very expensive.

We proposed an algorithm that greedily assigns input variables and thus requires only a limited number of model counts, but sometimes misses the optimal solution. An alternative approach that we would like to investigate in the future is to use a complete (branch-and-bound like) algorithm, but to combine it with *approximate* counting methods that compute confidence intervals for solution counts [10]. Also, in practice, testing problems often have additional structure: for instance, in the automotive example in Section 4.2, pairs of hypotheses share significant identical portions. There exist decomposition techniques in test generation that can exploit this fact [5]; therefore, an interesting question is whether these can be adapted to model counting approaches. In [16], we have recently developed an approach that exploits model structure by pre-compiling the ODT problem into decomposable negation normal form (DNNF) [8].

Another extension concerns relaxing the simplifying assumption that the possible outcomes of a non-deterministic hypothesis all have similar likelihood. In this context, methods for *weighted* model counting [15] could be used to capture, for instance, probability distributions in the hypotheses.

References

1. Achterberg, T.: Constraint Integer Programming, PhD thesis, TU Berlin (2007)
2. Achterberg, T., Heinz, S., Koch, T.: Counting solutions of integer programs using unrestricted subtree detection. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 278–282. Springer, Heidelberg (2008)

3. Alur, R., Courcoubetis, C., Yannakakis, M.: Distinguishing tests for nondeterministic and probabilistic machines. In: Proc. of the twenty-seventh annual ACM symposium on Theory of computing, pp. 363–372 (1995)
4. Blackmore, L., Williams, B.C.: Finite horizon control design for optimal discrimination between several models. In: Proc. IEEE Conference on Decision and Control, pp. 1147–1152 (2006)
5. Boroday, S., Petrenko, A., Groz, R.: Can a model checker generate tests for nondeterministic systems? Electron. Notes Theor. Comput. Sci 190, 3–19 (2007)
6. Brand, S.: Sequential automatic test pattern generation by constraint programming. In: Proc. CP 2001 Workshop on Modelling and Problem Formulation (2001)
7. Cimatti, A., Pecheur, C., Cavada, R.: Formal verification of diagnosability via symbolic model checking. In: Proc. of IJCAI 2003, pp. 363–369 (2003)
8. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. 17, 229–264 (2002)
9. Esser, M., Struss, P.: Fault-model-based test generation for embedded software. In: Proc. of IJCAI 2007, pp. 342–347 (2007)
10. Gomes, C., Sabharwal, A., Selman, B.: Model counting: A new strategy for obtaining good bounds. In: Proc. of AAAI 2006, pp. 54–61 (2006)
11. Larrabee, T.: Test pattern generation using boolean satisfiability. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 11, 4–15 (1992)
12. Luo, J., Pattipati, K.R., Qiao, L., Chigusa, S.: An integrated diagnostic development process for automotive engine control systems. IEEE Transactions on Systems, Man, and Cybernetics 37, 1163–1173 (2007)
13. Sachenbacher, M., Schwoon, S.: Model-based testing using quantified CSPs: A map. In: ECAI 2008 Workshop on Model-based Systems, pp. 37–41 (2008)
14. Sachenbacher, M., Struss, P.: Task-dependent qualitative domain abstraction. Artif. Intell. 162, 121–143 (2005)
15. Sang, T., Beame, P., Kautz, H.: Solving bayesian networks by weighted model counting. In: Proc. of AAAI 2005 (2005)
16. Schumann, A., Sachenbacher, M., Huang, J.: Computing optimal tests for nondeterministic systems using DNNF graphs. In: Proc. Fifth Workshop on Model-Based Testing (MBT 2009) (2009)
17. Struss, P.: Testing physical systems. In: Proc. of AAAI 1994, pp. 251–256 (1994)
18. Vatcheva, I., de Jong, H., Bernard, O., Mars, N.J.: Experiment selection for the discrimination of semi-quantitative models of dynamical systems. Artif. Intell. 170, 472–506 (2006)
19. Weld, D.S., de Kleer, J. (eds.): Readings in qualitative reasoning about physical systems. Morgan Kaufmann, San Francisco (1990)
20. Williams, B.C., Ingham, M.D., Chung, S.H., Elliott, P.H.: Model-based programming of intelligent embedded systems and robotic space explorers. Proc. of the IEEE 91, 212–237 (2003)
21. Williams, B.C., Nayak, P.P.: A model-based approach to reactive self-configuring systems. In: Proc. of AAAI 1996, pp. 971–978 (1996)

Reformulating Global Grammar Constraints*

George Katsirelos[1], Nina Narodytska[2], and Toby Walsh[2]

[1] NICTA, Sydney, Australia
george.katsirelos@nicta.com.au
[2] NICTA and University of NSW, Sydney, Australia
ninan@cse.unsw.edu.au, toby.walsh@nicta.com.au

Abstract. An attractive mechanism to specify global constraints in rostering and other domains is via formal languages. For instance, the REGULAR and GRAMMAR constraints specify constraints in terms of the languages accepted by an automaton and a context-free grammar respectively. Taking advantage of the fixed length of the constraint, we give an algorithm to transform a context-free grammar into an automaton. We then study the use of minimization techniques to reduce the size of such automata and speed up propagation. We show that minimizing such automata after they have been unfolded and domains initially reduced can give automata that are more compact than minimizing before unfolding and reducing. Experimental results show that such transformations can improve the size of rostering problems that we can "model and run".

1 Introduction

Constraint programming provides a wide range of tools for modelling and efficiently solving real world problems. However, modelling remains a challenge even for experts. Some recent attempts to simplify the modelling process have focused on specifying constraints using formal language theory. For example the REGULAR [1] and GRAMMAR constraints [2,3] permit constraints to be expressed in terms of automata and grammars. In this paper, we make two contributions. First, we investigate the relationship between REGULAR and GRAMMAR. In particular, we show that it is often beneficial to reformulate a GRAMMAR constraint as a REGULAR constraint. Second, we explore the effect of minimizing the automaton specifying a REGULAR constraint. We prove that by minimizing this automaton *after* unfolding and initial constraint propagation, we can get an exponentially smaller and thus more efficient representation. We show that these transformations can improve runtimes by over an order of magnitude.

2 Background

A constraint satisfaction problem consists of a set of variables, each with a domain of values, and a set of constraints specifying allowed combinations of values for given subsets of variables. A solution is an assignment to the variables satisfying the constraints. A constraint is *domain consistent* iff for each variable, every value in its domain can be extended to an assignment that satisfies the constraint. We will consider

* NICTA is funded by the Australian Government's Department of Broadband, Communications, and the Digital Economy and the Australian Research Council.

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 132–147, 2009.

constraints specified by automata and grammars. An automaton $A = \langle \Sigma, Q, q_0, F, \delta \rangle$ consists of an alphabet Σ, a set of states Q, an initial state q_0, a set of accepting states F, and a transition relation δ defining the possible next states given a starting state and symbol. The automaton is deterministic (DFA) is there is only one possible next state, non-deterministic (NFA) otherwise. A string s is *recognized* by $\mathcal{A}$ iff starting from the state q_0 we can reach one of the accepting states using the transition relation δ. Both DFAs and NFAs recognize precisely regular languages. The constraint REGULAR$(\mathcal{A}, [X_1, \ldots, X_n])$ is satisfied iff X_1 to X_n is a string accepted by $\mathcal{A}$ [1]. Pesant has given a domain consistency propagator for REGULAR based on unfolding the DFA to give a n-layer automaton which only accepts strings of length n [1].

Given an automaton $\mathcal{A}$, we write $unfold_n(\mathcal{A})$ for the unfolded and layered form of $\mathcal{A}$ that just accepts words of length n which are in the regular language, $\min(\mathcal{A})$ for the canonical form of $\mathcal{A}$ with minimal number of states, $simplify(\mathcal{A})$ for the simplified form of $\mathcal{A}$ constructed by deleting transitions and states that are no longer reachable after domains have been reduced. We write $f_{\mathcal{A}}(n) \ll g_{\mathcal{A}}(n)$ iff $f_{\mathcal{A}}(n) \leq g_{\mathcal{A}}(n)$ for all n, and there exist $\mathcal{A}$ such that $\log \frac{g_{\mathcal{A}}(n)}{f_{\mathcal{A}}(n)} = \Omega(n)$. That is, $g_{\mathcal{A}}(n)$ is never smaller than $f_{\mathcal{A}}(n)$ and there are cases where it is exponentially larger.

A context-free grammar is a tuple $G = \langle T, H, P, S \rangle$, where T is a set of *terminal* symbols called the *alphabet* of G, H is a set of *non-terminal* symbols, P is a set of productions and S is a unique starting symbol. A production is a rule $A \rightarrow \alpha$ where A is a non-terminal and α is a sequence of terminals and non-terminals. A string in Σ^* is *generated* by G if we start with the sequence $\alpha = \langle S \rangle$ and non deterministically generate α' by replacing any non-terminal A in α by the right hand side of any production $A \rightarrow \alpha$ until α' contains only terminals. A context free language $\mathcal{L}(G)$ is the language of strings generated by the context free grammar G. A context free grammar is in Chomsky normal form if all productions are of the form $A \rightarrow BC$ where B and C are non terminals or $A \rightarrow a$ where a is a terminal. Any context free grammar can be converted to one that is in Chomsky normal form with at most a linear increase in its size. A grammar G_a is *acyclic* iff there exists a partial order $\prec$ of the non-terminals, such that for every production $A_1 \rightarrow A_2 A_3$, $A_1 \prec A_2$ and $A_1 \prec A_3$. The constraint GRAMMAR$([X_1, \ldots, X_n], G)$ is satisfied iff X_1 to X_n is a string accepted by G [2,3].

Example 1. As the running example we use the GRAMMAR$([X_1, X_2, X_3], G)$ constraint with domains $D(X_1) = \{a\}$, $D(X_2) = \{a, b\}$, $D(X_3) = \{b\}$ and the grammar G in Chomsky normal form [3] $\{S \rightarrow AB, A \rightarrow AA \mid a, B \rightarrow BB \mid b\}$.

Since we only accept strings of a fixed length, we can convert any context free grammar to a regular grammar. However, this may increase the size of the grammar exponentially. Similarly, any NFA can be converted to a DFA, but this may increase the size of the automaton exponentially.

3 GRAMMAR Constraint

We briefly describe the domain consistency propagator for the GRAMMAR constraint proposed in [2,3]. This propagator is based on the CYK parser for context-free grammars. It constructs a dynamic programing table V where an element A of $V[i, j]$ is a

non-terminal that generates a substring from the domains of variables $X_i, \ldots, X_{i+j-1}$ that can be extended to a solution of the constraint using the domains of the other variables. The table V produced by the propagator for Example 1 is given in Figure 1.

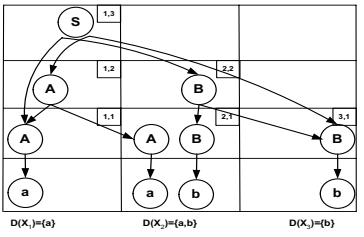


Fig. 1. Dynamic programming table produced by the propagator of the GRAMMAR constraint. Pointers correspond to possible derivations.

An alternative view of the dynamic programming table produced by this propagator is as an AND/OR graph [4]. This is a layered DAG, with layers alternating between AND-NODES or OR-NODES. Each OR-NODE in the AND/OR graph corresponds to an entry $A \in V[i,j]$. An OR-NODE has a child AND-NODE for each production $A \rightarrow BC$ so that $A \in V[i,j]$, $B \in V[i,k]$ and $C \in V[i+k, j-k]$. The children of this AND-NODE are the OR-NODES that correspond to the entries $B \in V[i,k]$ and $C \in V[i+k, j-k]$. Note that the AND/OR graph constructed in this manner is equivalent to the table V [4], so we use them interchangeably in this paper.

Every derivation of a string $s \in \mathcal{L}(G)$ can be represented as a tree that is a subgraph of the AND/OR graph and therefore can be represented as a trace in V. Since every possible derivation can be represented this way, both the table V and the corresponding AND/OR graph are a compilation of all solutions of the GRAMMAR constraint.

4 Reformulation into an Automaton

The time complexity of propagating a GRAMMAR constraint is $O(n^3|G|)$, as opposed to $O(n|\delta|)$ for a REGULAR constraint. Therefore, reformulating a GRAMMAR constraint as a REGULAR constraint may improve propagation speed if it does not require a

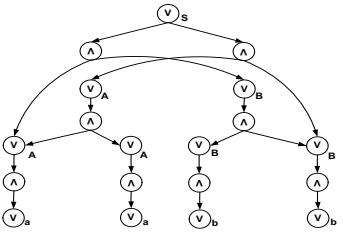


Fig. 2. AND/OR graph

large transition relation. In addition, we can perform optimizations such as minimizing the automaton. In this section, we argue that reformulation is practical in many cases (sections 4.1-4.3), and there is a polynomial test to determine the size of the resulting NFA (section 4.4). In the worst case, the resulting NFA is exponentially larger then the original GRAMMAR constraint as the following example shows. Therefore, performing the transformation itself is not a suitable test of the feasibility of the approach.

Example 2. Consider GRAMMAR$([X_1, \ldots, X_n], G)$ where G generates $L = \{ww^R | w \in \{0,1\}^{n/2}\}$. Solutions of GRAMMAR can be compiled into the dynamic programming table of size $O(n^3)$, while an equivalent NFA that accepts the same language has exponential size. Note that an exponential separation does not immediately follow from that between regular and context-free grammars, because solutions of the GRAMMAR constraint are the strict subset of $\mathcal{L}(G)$ which have length n.

In the rest of this section we describe the reformulation in three steps. First, we convert into an acyclic grammar (section 4.1), then into a pushdown automaton (section 4.2), and finally we encode this as a NFA (section 4.3). The first two steps are well known in formal language theory but we briefly describe them for clarity.

4.1 Transformation into an Acyclic Grammar

We first construct an acyclic grammar, G_a such that the language $\mathcal{L}(G_a)$ coincides with solutions of the GRAMMAR constraint. Given the table V produced by the GRAMMAR propagator (section 3), we construct an acyclic grammar in the following way. For each possible derivation of a nonterminal A, $A \rightarrow BC$, such that $A \in V[i,j]$, $B \in V[i,k]$ and $C \in V[i+k, j-k]$ we introduce a production $A_{i,j} \rightarrow B_{i,k}C_{i+k,j-k}$ in G_a (lines 11- 17 of algorithm 1). The start symbol of G_a is $S_{1,n}$. By construction, the obtained grammar G_a is acyclic. Every production in G_a is of the form $A_{i,j} \rightarrow B_{i,k}C_{i+k,j-k}$ and nonterminals $B_{i,k}$, $C_{i+k,j-k}$ occur in rows below jth row in V. Example 3 shows the grammar G_a obtained by Algorithm 1 on our running example.

Example 3. The acyclic grammar G_a constructed from our running example.

$$S_{1,3} \rightarrow A_{1,2}B_{3,1} \mid A_{1,1}B_{2,2} \qquad A_{1,2} \rightarrow A_{1,1}A_{2,1} \qquad B_{2,2} \rightarrow B_{2,1}B_{3,1}$$
$$A_{i,1} \rightarrow a_i \qquad B_{i,1} \rightarrow b_i \qquad \forall i \in \{1,2,3\}$$

To prove equivalence, we recall that traces of the table V represent all possible derivations of GRAMMAR solutions. Therefore, every derivation of a solution can be simulated by productions from G_A. For instance, consider the solution (a,a,b) of GRAMMAR from Example 1. A possible derivation of this string is $S|_{S \in V[1,3]} \rightarrow AB|_{A \in V[1,2], B \in V[3,1]} \rightarrow AAB|_{A \in V[1,1], A \in V[2,1], B \in V[3,1]} \rightarrow aAB|_{\ldots} \rightarrow aaB|_{\ldots} \rightarrow aab|_{\ldots}$. We can simulate this derivation using productions in G_a: $S_{1,3} \rightarrow A_{1,2}B_{3,1} \rightarrow A_{1,1}A_{2,1}B_{3,1} \rightarrow a_1A_{2,1}B_{3,1} \rightarrow a_1a_2B_{3,1} \rightarrow a_1a_2b_3$.

Observe that, the acyclic grammar G_a is essentially a labelling of the AND/OR graph, with non-terminals corresponding to OR-NODES and productions corresponding to AND-NODES. Thus, we use the notation G_a to refer to both the AND/OR graph and the corresponding acyclic grammar.

Algorithm 1. Transformation to an Acyclic Grammar

```
1: procedure ConstructAcyclicGrammar(in : X, G, V; out : G_a)
2:     T = ∅                                   ▷ T is the set of terminals in G_a
3:     H = ∅                                   ▷ H is the set of nonterminals in G_a
4:     P = ∅                                   ▷ P is the set of productions in G_a
5:     for i = 1 to n do
6:         V[i, 1] = {A | A → a ∈ G, a ∈ D(X_i)}
7:         for A ∈ V[i, 1] s.t A → a ∈ G, a ∈ D(X_i) do
8:             T = T ∪ {a_i}
9:             H = H ∪ {A_{i,1}}
10:            P = P ∪ {A_{i,1} → a_i}
11:    for j = 2 to n do
12:        for i = 1 to n − j + 1 do
13:            for each A ∈ V[i, j] do
14:                for k = 1 to j − 1 do
15:                    for each A → BC ∈ G s.t. B ∈ V[i, k], C ∈ V[i + k, j − k] do
16:                        H = H ∪ {A_{i,j}, B_{i,k}, C_{i+k,j−k}}
17:                        P = P ∪ {A_{i,j} → B_{i,k} C_{i+k,j−k}}
```

4.2 Transformation into a Pushdown Automaton

Given an acyclic grammar $G_a = (T, H, P, S_{1,n})$ from the previous section, we now construct a pushdown automaton $P_a(\langle S_{1,n}\rangle, T, T \cup H, \delta, Q_P, F_P)$, where $\langle S_{1,n}\rangle$ is the initial stack of P_a, T is the alphabet, $T \cup H$ is the set of stack symbols, δ is the transition function, $Q_P = F_P = \{q_P\}$ is the single initial and accepting state. We use an algorithm that encodes a context free grammar into a pushdown automaton (PDA) that computes the leftmost derivation of a string[5]. The stack maintains the sequence of symbols that are expanded in this derivation. At every step, the PDA non-deterministically uses a production to expand the top symbol of the stack if it is a non-terminal, or consumes a symbol of the input string if it matches the terminal at the top of the stack.

We now describe this reformulation in detail. There exists a single state q_P which is both the starting and an accepting state. For each non-terminal $A_{i,j}$ in G_a we introduce the set of transitions $\delta(q_P, \varepsilon, A_{i,j}) = \{(q_P, \beta) | \forall A_{i,j} \to \beta \in G_a\}$. For each terminal $a_i \in G_a$, we introduce a transition $\delta(q_P, a_i, a_i) = \{(q_P, \varepsilon)\}$. The automaton P_a accepts on the empty stack. This constructs a pushdown automaton accepting $\mathcal{L}(G_a)$.

Example 4. The pushdown automaton P_a constructed for the running example.

$$\delta(q_P, \varepsilon, S_{1,3}) = \delta(q_P, A_{1,2}B_{3,1}) \qquad \delta(q_P, \varepsilon, S_{1,3}) = \delta(q_P, A_{1,1}B_{2,2})$$
$$\delta(q_P, \varepsilon, A_{1,2}) = \delta(q_P, A_{1,1}A_{2,1}) \qquad \delta(q_P, \varepsilon, B_{2,2}) = \delta(q_P, B_{2,1}B_{3,1})$$
$$\delta(q_P, \varepsilon, A_{i,1}) = \delta(q_P, a_i) \qquad \delta(q_P, \varepsilon, B_{i,1}) = \delta(q_P, b_i) \forall i \in \{1, 2, 3\}$$
$$\delta(q_P, a_i, a_i) = \delta(q_P, \varepsilon) \qquad \delta(q_P, b_i, b_i) = \delta(q_P, \varepsilon) \forall i \in \{1, 2, 3\}$$

4.3 Transformation into a NFA

Finally, we construct an NFA$(\Sigma, Q, Q_0, F_0, \sigma)$, denoted N_a, using the PDA from the last section. States of this NFA encode all possible configurations of the stack of the PDA that can appear in parsing a string from G_a. To reflect that a state of the NFA represents a stack, we write states as sequences of symbols $\langle\alpha\rangle$, where α is a possibly empty sequence of symbols and $\alpha[0]$ is the top of the stack. For example, the initial

Algorithm 2. Transformation to NFA

```
1: procedure PDA TO NFA(in : P_a, out : N_a)
2:     Q_u = {⟨S_{1,n}⟩}                          ▷ Q_u is the set of unprocessed states
3:     Q = ∅                                      ▷ Q is the set of states in N_a
4:     σ = ∅                                      ▷ σ is the set of transitions in N_a
5:     Q_0 = {⟨S_{1,n}⟩}                          ▷ Q_0 is the initial state in N_a
6:     F_0 = {⟨⟩}                                 ▷ F_0 is the set of final states in N_a
7:     while Q_u is not empty do
8:         if q ≡ ⟨A_{i,j}, α⟩ then
9:             for each transition δ(q_P, ε, A_{i,j}) = (q_P, β) ∈ δ do
10:                σ = σ ∪ {σ(⟨A_{i,j}, α⟩, ε) = ⟨β, α⟩}
11:                if ⟨β, α⟩ ∉ Q then
12:                    Q_u = Q_u ∪ {⟨β, α⟩}
13:            Q = Q ∪ {⟨A_{i,j}, α⟩}
14:        else if q ≡ ⟨a_i, α⟩ then
15:            for each transition δ(q_P, a_i, a_i) = (q_P, ε) ∈ δ do
16:                σ = σ ∪ {σ(⟨a_i, α⟩, a_i) = ⟨α⟩}
17:                if ⟨α⟩ ∉ Q then
18:                    Q_u = Q_u ∪ {⟨α⟩}
19:            Q = Q ∪ {⟨a_i, α⟩}
20:        Q_u = Q_u \ {q}
21:    N_a(Σ, Q, Q_0, F_0, σ) = ε − Closure(N_a(Σ, Q, Q_0, F_0σ)).
```

state is $\langle S_{1,n} \rangle$ corresponding to the initial stack $\langle S_{1,n} \rangle$ of P_a. Algorithm 2 unfolds the PDA in a similar way to unfolding the DFA. Note that the NFA accepts only strings of length n and has the initial state $Q_0 = \langle S_{1,n} \rangle$ and the single final state $F_0 = \langle \rangle$.

We start from the initial stack $\langle S_{1,n} \rangle$ and find all distinct stack configurations that are reachable from this stack using transitions from P_a. For each reachable stack configuration we create a state in the NFA and add the corresponding transitions. If the new stack configurations are the result of expansion of a production in the original grammar, these transitions are $\varepsilon-$transitions, otherwise they consume a symbol from the input string. Note that if a non-terminal appears on top of the stack and gets replaced, then it cannot appear in any future stack configuration due to the acyclicity of G_a. Therefore $|\alpha|$ is bounded by $O(n)$ and Algorithm 2 terminates. The size of N_a is $O(|G_a|^n)$ in the worst case. The automaton N_a that we obtain before line 21 is an acyclic NFA with ε transitions. It accepts the same language as the PDA P_a since every path between the starting and the final state of N_A is a trace of the stack configurations of P_a. Figure 3(a) shows the automaton N_a with ε-transitions constructed from the running example. After applying the ε-closure operation, we obtain a layered NFA that does not have ε transitions (line 21) (Figure 3(b)).

4.4 Computing the Size of the NFA

As the NFA may be exponential in size, we provide a polynomial method of computing its size in advance. We can use this to decide if it is practical to transform it in this way. Observe first that the transformation of a PDA to an NFA maintains a queue of states that correspond to stack configurations. Each state corresponds to an OR-NODE in the AND/OR graph and each state of an OR-NODE v is generated from the states of the parent OR-NODES of v. This suggests a relationship between paths in the AND/OR graph of the CYK algorithm and states in N_a. We use this relationship to compute

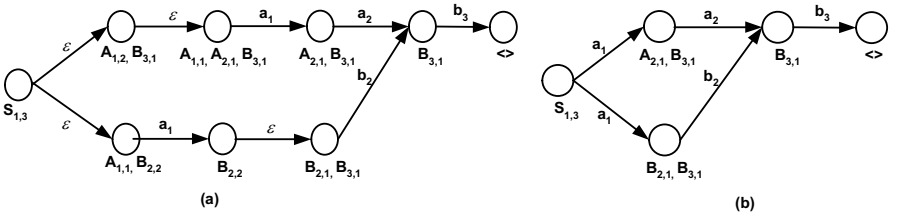


Fig. 3. N_a produced by Algorithm 2

a loose upper bound for the number of states in N_a in time linear in the size of the AND/OR graph by counting the number of paths in that graph. Alternatively, we compute the exact number of states in N_a in time quadratic in the size of the AND/OR graph.

Theorem 1. *There exists a surjection between paths in G_a from the root to* OR-NODES *and stack configurations in the PDA P_a.*

Proof. Consider a path p from the root of the AND/OR graph to an OR-NODE labelled with $A_{i,j}$. We construct a stack configuration $\Gamma(p)$ that corresponds to p. We start with the empty stack $\Gamma = \langle\rangle$. We traverse the path from the root to $A_{i,j}$. For every AND-NODE $v_1 \in p$, with left child v_l and right child v_r, if the successor of v_1 in p is v_l, then we push v_r on Γ, otherwise do nothing. When we reach $A_{i,j}$, we push it on Γ. The final configuration Γ is unique for p and corresponds to the stack of the PDA after having parsed the substring $1 \dots i-1$ and having non-deterministically chosen to parse the substring $i \dots i+j-1$ using a production with $A_{i,j}$ on the LHS.

We now show that all stack configurations can be generated by the procedure above. Every stack configuration corresponds to at least one partial left most derivation of a string. We say a stack configuration $\langle\alpha\rangle$ corresponds to a derivation $dv = \langle a_1, \dots, a_{k-1}, A_{k,j}, \alpha\rangle$ if α is the context of the stack after parsing the prefix of the string of length $k+j$. Therefore, it is enough to show that all partial left most derivation (we omit the prefix of terminals) can be generated by the procedure above. We prove by a contradiction. Suppose that $\langle a_1, \dots, a_{i-1}, B_{i,j}, \beta\rangle$ is the partial left most derivation such that $\Gamma(p(root, B_{i,j})) \neq \beta$, where $p(root, B_{i,j})$ is the path from the root to the OR-NODE $B_{i,j}$ and for any partial derivation $\langle a_1, \dots, a_{k-1}, A_{k,j}, \alpha\rangle$, such that $k < i$ $A_{k,j} \in G_a$ $\Gamma(p(root, A_{k,j})) = \alpha$. Consider the production rule that introduces the nonterminal $B_{i,j}$ to the partial derivation. If the production rule is $D \rightarrow C, B_{i,j}$, then the partial derivation is $\langle a_1, \dots, a_f, D, \beta\rangle \Rightarrow|_{D \rightarrow C, B_{i,j}} \langle a_1, \dots, a_f, C, B_{i,j}, \beta\rangle$. The path from the root to the node $B_{i,j}$ is a concatenation of the paths from D to $B_{i,j}$ and from the root to D. Therefore, $\Gamma(p(root, B_{i,j}))$ is constructed as a concatenation of $\Gamma(p(D, B_{i,j}))$ and $\Gamma(p(root, D))$. $\Gamma(p(D, B_{i,j}))$ is empty because the node $B_{i,j}$ is the right child of AND-NODE that corresponds to the production $D \rightarrow C, B_{i,j}$ and $\Gamma(p(root, D)) = \beta$ because $f < i$. Therefore, $\Gamma(p(root, B_{i,j})) = \beta$. If the production rule is $D \rightarrow B_{i,j}, C$, then the partial derivation is $\langle a_1, \dots, a_{i-1}, D, \gamma\rangle \Rightarrow|_{D \rightarrow B_{i,j}, C}$ $\langle a_1, \dots, a_{i-1}, B_{i,j}, C, \gamma\rangle = \langle a_1, \dots, a_{i-1}, B_{i,j}, \beta\rangle$. Then, $\Gamma(p(root, D)) = \gamma$, because $i - 1 < i$ and $\Gamma(p(D, B_{i,j})) = \langle C\rangle$, because the node $B_{i,j}$ is the left

child of AND-NODE that corresponds to the production $D \rightarrow C, B_{i,j}$. Therefore, $\Gamma(p(root, B_{i,j})) = \langle C, \gamma \rangle = \beta$. This leads to a contradiction. □

Example 5. An example of the mapping described in the last proof is in Figure 4(a) for the grammar of our running example. Consider the OR-NODE $A_{1,1}$. There are 2 paths from $S_{1,3}$ to $A_{1,1}$. One is direct and uses only OR-NODES $\langle S_{1,3}, A_{1,1} \rangle$ and the other uses OR-NODES $\langle S_{1,3}, A_{1,2}, A_{1,1} \rangle$. The 2 paths are mapped to 2 different stack configurations $\langle A_{1,1}, B_{2,2} \rangle$ and $\langle A_{1,1}, A_{2,1}, B_{3,1} \rangle$ respectively. We highlight edges that are incident to AND-NODES on each path and lead to the right children of these AND-NODES. There is exactly one such edge for each element of a stack configuration. □

Note that theorem 1 only specifies a surjection from paths to stack configurations, not a bijection. Indeed, different paths may produce the same configuration Γ.

Example 6. Consider the grammar $G = \{S \rightarrow AA, A \rightarrow a|AA|BC, B \rightarrow b|BB, C \rightarrow c|CC\}$ and the AND/OR graph of this grammar for a string of length 5. The path $\langle S_{1,5}, A_{2,4}, B_{2,2} \rangle$ uses the productions $S_{1,5} \rightarrow A_{1,1}A_{2,4}$ and $A_{2,4} \rightarrow B_{2,2}C_{4,2}$, while the path $\langle S_{1,5}, A_{3,3}, B_{3,1} \rangle$ uses the productions $S_{1,5} \rightarrow A_{1,2}A_{3,3}$ and $A_{3,3} \rightarrow B_{3,1}C_{4,2}$. Both paths map to the same stack configuration $\langle C_{4,2} \rangle$. □

By construction, the resulting NFA has one state for each stack configuration of the PDA in parsing a string. Since each path corresponds to a stack configuration, the number of states of the NFA before applying ε-closure is bounded by the number of paths from the root to any OR-NODE in the AND/OR graph. This is cheap to compute using the following recursive algorithm [6]:

$$PD(v) = \begin{cases} 1 & \text{If } v \text{ has no incoming edges} \\ \sum_p PD(p) & \text{where } p \text{ is a parent of } v \end{cases} \quad (1)$$

Therefore, the number of states of the NFA N_a is at most $\sum_v PD(v)$, where v is an OR-NODE of G_a (Figure 4).

We can compute the exact number of paths in N_a before ε-closure without constructing the NFA by counting paths in the *stack graph* G_v for each OR-NODE v. The stack graph captures the observation that each element of a stack configuration generated

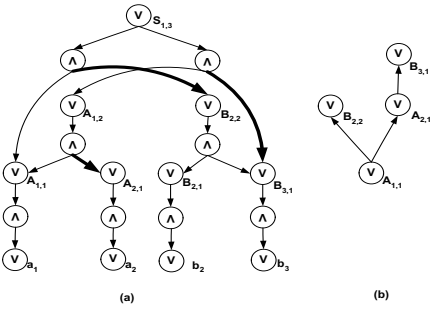


Fig. 4. Computing the size of N_a. (a) AND/OR graph G_a. (b) Stack graph $G_{A_{1,1}}$.

from a path p is associated with exactly one edge e that is incident on p and leads to the right child of an AND-NODE. G_v contains one path for each sequence of such edges, so that if two paths p and p' in G_a are mapped to the same stack configuration, they are also mapped to the same path in G_v. Formally, the stack graph of an OR-NODE $v \in V(G_a)$ is a DAG G_v, such that for every stack configuration Γ of P_a with k elements, there is exactly one path p in G_v of length k and v' is the i^{th} vertex of p if and only if v' is the i^{th} element from the top of Γ.

Example 7. Consider the grammar of the running example and the OR-NODE $A_{1,1}$ in the AND/OR graph. The stack graph $G_{A_{1,1}}$ for this OR-NODE is shown in figure 4(b). Along the path $\langle S_{1,3}A_{1,1}\rangle$, only the edge that leads to $B_{2,2}$ generates a stack element. This edge is mapped to the edge $(A_{1,1}, B_{2,2})$ in $G_{A_{1,1}}$. Similarly, the edges that lead to $A_{2,1}$ and $B_{3,1}$ are mapped to the edges $(A_{1,1}, A_{2,1})$ and $(A_{2,1}, B_{3,1})$ respectively. □

Since G_v is a DAG, we can efficiently count the number of paths in it. We construct G_v using algorithm 3. The graph G_v computed in algorithm 3 for an OR-NODE v has as many paths as there are unique stack configurations in P_a with v at the top.

Algorithm 3. Computing the stack DAG G_v of an OR-NODE v

```
1: procedure StackGraph((in : G_a, v, out : G_v))
2:     V(G_v) = {v}
3:     label(v) = {v}
4:     Q = {(v, v_p)|v_p ∈ parents(v)}                          ▷ queue of edges
5:     while Q not empty do
6:         (v_c, v_p) = pop(Q)
7:         if v_p is an AND-NODE v_c is left child of v_p then
8:             v_r = children_r(v_p)
9:             V(G_v) = V(G_v) ∪ {v_r}
10:            E(G_v) = E(G_v) ∪ {(v_l, v_r)|v_l ∈ label(v_c)}
11:            label(v_p) = label(v_p) ∪ {v_r}
12:        else
13:            label(v_p) = label(v_p) ∪ label(v_c)
14:        Q = Q ∪ {(v_p, v'_p)|v'_p ∈ parents(v_p)}             ▷
```

Theorem 2. *There exists a bijection between paths in G_v and states in the NFA N_a which correspond to stacks with v at the top.*

Proof. Let p be a path from the root to v in G_a. First, we show that every path p' in G_v corresponds to a stack configuration, by mapping p to p'. Therefore p' corresponds to $\Gamma(p)$. We then show that p' is unique for $\Gamma(p)$. This establishes a bijection between paths in G_v and stack configurations.

We traverse the inverse of p, denoted $inv(p)$ and construct p' incrementally. Note that every vertex in $inv(p)$ is examined by algorithm 3 in the construction of G_v. If $inv(p)$ visits the left child of an AND-NODE, we append the right child of that AND-NODE to p'. This vertex is in G_v by line 7. By the construction of $\Gamma(p)$ in the proof of theorem 1, a symbol is placed on the stack if and only if it is the right child of an AND-NODE, hence if and only if it appears in p'. Moreover, if a vertex is the i^{th} vertex in a path, it corresponds to the i^{th} element from the top of $\Gamma(p)$. We now see that p' is unique for $\Gamma(p)$. Two distinct paths of length k cannot map to the same stack configuration,

because they must differ in at least one position i, therefore they correspond to stacks with different symbols at position i. Therefore, there exists a bijection between paths in G_v and stack configurations with v at the top. □

Hence $|Q(Ng)| = \sum_v \#paths(G_v)$, where v is an OR-NODE of G_a. Computing the stack graph G_v of every OR-NODE v takes $O(|G_a|)$ time, as does counting paths in G_v. Therefore, computing the number of states in N_a takes $O(|G_a|^2)$ time. We can also compute the number of states in the ε-closure of N_a by observing that if none of the OR-NODES that are reachable by paths of length 2 from an OR-NODE v correspond to terminals, then any state that corresponds to a stack configuration with v at the top will only have outgoing $\varepsilon-$transitions and will be removed by the $\varepsilon-$closure. Thus, to compute the number of states in N_a after $\varepsilon-$closure, we sum the number of paths in G_v for all OR-NODES v such that a terminal OR-NODE can be reached from v by a path of length 2.

4.5 Transformation into a DFA

Finally, we convert the NFA into a DFA using the standard subset construction. This is optional as Pesant's propagator for the REGULAR constraints works just as well with NFAs as DFAs. Indeed, removing non-determinism may increase the size of the automaton and slow down propagation. However, converting into a DFA opens up the possibility of further optimizations. In particular, as we describe in the next section, there are efficient methods to minimize the size of a DFA. By comparison, minimization of a NFA is PSPACE-hard in general [7]. Even when we consider just the acyclic NFA constructed by unfolding a NFA, minimization remains NP-hard [8].

5 Automaton Minimization

The DFA constructed by this or other methods may contain redundant states and transitions. We can speed up propagation of the REGULAR constraint by minimizing the size of this automaton. Minimization can be either offline (i.e. before we have the problem data and have unfolded the automaton) or online (i.e. once we have the problem data and have unfolded the automaton). There are several reasons why we might prefer an online approach where we unfold before minimizing. First, although minimizing after unfolding may be more expensive than minimizing before unfolding, both are cheap to perform. Minimizing a DFA takes $O(Q \log Q)$ time using Hopcroft's algorithm and $O(nQ)$ time for the unfolded DFA where Q is the number of states [9]. Second, thanks to Myhill-Nerode's theorem, minimization does not change the layered nature of the unfolded DFA. Third, and perhaps most importantly, minimizing a DFA after unfolding can give an exponentially smaller automaton than minimizing the DFA and then unfolding. To put it another way, unfolding may destroy the minimality of the DFA.

Theorem 3. *Given any DFA $\mathcal{A}$,* $|\min(unfold_n(\mathcal{A}))| \ll |unfold_n(\min(\mathcal{A}))|$.

Proof: To show $|\min(unfold_n(\mathcal{A}))| \leq |unfold_n(\min(\mathcal{A}))|$, we observe that both $\min(unfold_n(\mathcal{A}))$ and $unfold_n(\min(\mathcal{A}))$ are automata that recognize the same language. By definition, minimization returns the smallest DFA accepting this language. Hence $\min(unfold_n(\mathcal{A}))$ cannot be larger than $unfold_n(\min(\mathcal{A}))$.

To show unfolding then minimizing can give an exponentially smaller sized DFA, consider the following language L. A string of length k belongs to L iff it contains the symbol j, $j = k \bmod n$, where n is a given constant. The alphabet of the language L is $\{0, \ldots, n-1\}$. The minimal DFA for this language has $\Omega(n2^n)$ states as each state needs to record which symbols from 0 to $n-1$ have been seen so far, as well as the current length of the string mod n. Unfolding this minimal DFA and restricting it to strings of length n gives an acyclic DFA with $\Omega(n2^n)$ states. Note that all strings are of length n and the equation $j = n \bmod n$ has the single solution $j = 0$. Therefore, the language L consists of the strings of length n that contain the symbol 0. On the other hand, if we unfold and then minimize, we get an acyclic DFA with just $2n$ states. Each layer of the DFA has two states which record whether 0 has been seen. □

Further, if we make our initial problem domain consistent, domains might be pruned which give rise to possible simplifications of the DFA. We show here that we should also perform such simplification before minimizing.

Theorem 4. *Given any DFA* $\mathcal{A}$, $|\min(simplify(unfold_n(\mathcal{A})))| \ll |simplify(\min(unfold_n(\mathcal{A})))|$.

Proof: Both $\min(simplify(unfold_n(\mathcal{A})))$ and $simplify(\min(unfold_n(\mathcal{A})))$ are DFAs that recognize the same language of strings of length n. By definition, minimization must return the smallest DFA accepting this language. Hence $\min(simplify(unfold_n(\mathcal{A})))$ is no larger than $simplify(\min(unfold_n(\mathcal{A})))$.

To show that minimization after simplification may give an exponentially smaller sized automaton, consider the language which contains sequences of integers from 1 to n in which at least one integer is repeated and in which the last two integers are different. The alphabet of the language L is $\{1, \ldots, n\}$. The minimal unfolded DFA for strings of length n from this language has $\Omega(2^n)$ states as each state needs to record which integers have been seen.Suppose the integer n is removed from the domain of each variable. The simplified DFA still has $\Omega(2^n)$ states to record which integers 1 to $n-1$ have been seen.On the other hand, suppose we simplify before we minimize. By a pigeonhole argument, we can ignore the constraint that an integer is repeated. Hence we just need to ensure that the string is of length n and that the last two integers are different. The minimal DFA accepting this language requires just $O(n)$ states. □

6 Empirical Results

We empirically evaluated the results of our method on a set of shift-scheduling benchmarks [11,14] [1]. Experiments were run with the Minisat+ solver for pseudo-Boolean instances and Gecode 2.2.0 for constraint problems, on an Intel Xeon 4 CPU, 2.0 Ghz, 4G RAM. We use a timeout of 3600 sec in all experiments. The problem is to schedule employees to activities subject to various rules, e.g. a full-time employee has one hour for lunch. This rules are specified by a context-free grammar augmented with restrictions on productions [4]. A schedule for an employee has $n = 96$ slots of 15 minutes

[1] We would like to thank Louis-Martin Rousseau and Claude-Guy Quimper for providing us with the benchmark data.

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 specified by the following grammar:

$$S \rightarrow RPR, f_P(i,j) \equiv 13 \leq j \leq 24, \quad P \rightarrow WbW, \; L \rightarrow lL|l, f_L(i,j) \equiv j = 4$$
$$S \rightarrow RFR, f_F(i,j) \equiv 30 \leq j \leq 38, \quad R \rightarrow rR|r, \; W \rightarrow A_i, f_W(i,j) \equiv j \geq 4$$
$$A_i \rightarrow a_i A_i|a_i, f_A(i,j) \equiv open(i), \quad F \rightarrow PLP$$

where functions $f(i,j)$ are predicates that restrict the start and length of any string matched by a specific production, and $open(i)$ is a function that returns 1 if the business is open at i^{th} slot and 0 otherwise. In addition, the business requires a certain number of employees working in each activity at given times during the day. We minimize the number of slots in which employees work such that the demand is satisfied.

As shown in [4], this problem can be converted into a pseudo-Boolean (PB) model. The GRAMMAR constraint is converted into a SAT formula in conjunctive normal form using the AND/OR graph. To model labour demand for a slot we introduce Boolean variables $b(i,j,a_k)$, equal to 1 if j^{th} employee performs activity a_k at i^{th} time slot. For each time slot i and activity a_k we post a pseudo-Boolean constraint $\sum_{j=1}^{m} b(i,j,a_k) > d(i,a_k)$, where m is the number of employees. The objective is modelled using the function $\sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{a} b_{i,j,a_k}$. Additionally, the problem can be formulated as an optimization problem in a constraint solver, using a matrix model with one row for each employee. We post a GRAMMAR constraint on each row, AMONG constraints on each column for labour demand and LEX constraints between adjacent rows to break symmetry. We use the static variable and value ordering used in [4].

We compare this with reformulating the GRAMMAR constraint as a REGULAR constraint. Using algorithm 3, we computed the size of an equivalent NFA. Surprisingly, this is not too big, so we converted the GRAMMAR constraint to a DFA then minimized. In order to reduce the blow-up that may occur converting a NFA to a DFA, we heuristically minimized the NFA using the following simple observation: two states are equivalent if they have identical outgoing transitions. We traverse the NFA from the last to the first layer and merge equivalent states and then apply the same procedure to the reversed NFA. We repeat until we cannot find a pair of equivalent states. We also simplified the original CYK table, taking into account whether the business is open or closed at each slot. Theorem 4 suggests such simplification can significantly reduce the size both of the CYK table and of the resulting automata. In practice we also observe a significant reduction in size. The resulting minimized automaton obtained before simplification is about ten times larger compared to the minimised DFA obtained after simplification. Table 1 gives the sizes of representations at each step. We see from this that the minimized DFA is always smaller than the original CYK table. Interestingly, the subset construction generates the minimum DFA from the NFA, even in the case of two activities, and heuristic minimization of the NFA achieves a notable reduction.

For each instance, we used the resulting DFA in place of the GRAMMAR constraint in both the CP model and the PB model using the encoding of the REGULAR constraint (DFA or NFA) into CNF [10]. We compare the model that uses the PB encoding of the GRAMMAR constraint (GR_1) with two models that use the PB encoding of the

Table 1. Shift Scheduling Problems. G_a is the acyclic grammar, N_a^ε is NFA with ε-transitions, N_a is NFA without ε-transitions, $\min(N_a)$ is minimized NFA, $\mathcal{A}$ is DFA obtained from $\min(N_a)$, $\min(\mathcal{A})$ is minimized $\mathcal{A}$, a is the number of activities, $\#$ is the benchmark number.

$\#act$	$\#$	G_a		NFA_a^ε		NFA_a		$\min(NFA_a)$		DFA		$\min(DFA)$	
		terms	prods	states	trans	states	trans	states	trans	states	trans	states	trans
1	2/3/8	4678	/ 9302	69050	/ 80975	29003	/ 42274	**3556**	**/ 4505**	3683	/ 4617	3681	/ 4615
1	4/7/10	3140	/ 5541	26737	/ 30855	11526	/ 16078	**1773**	**/ 2296**	1883	/ 2399	1881	/ 2397
1	5/6	2598	/ 4209	13742	/ 15753	5975	/ 8104	**1129**	**/ 1470**	1215	/ 1553	1213	/ 1551
2	1/2/4	3777	/ 6550	42993	/ 52137	19654	/ 29722	**3157**	**/ 4532**	3306	/ 4683	3303	/ 4679
2	3/5/6	**5407**	/ 10547	111302	/ 137441	50129	/ 79112	5975	**/ 8499**	6321	/ 8846	6318	/ 8842
2	8/10	**6087**	/ 12425	145698	/ 180513	65445	/ 104064	7659	**/ 10865**	8127	/ 11334	8124	/ 11330
2	9	4473	/ 8405	76234	/ 93697	34477	/ 53824	**4451**	**/ 6373**	4691	/ 6614	4688	/ 6610

REGULAR constraint (REGULAR$_1$, REGULAR$_2$), a CP model that uses the GRAMMAR constraint (GR$_1^{CP}$) and a CP model that uses a REGULAR constraint (REGULAR$_1^{CP}$). REGULAR$_1$ and REGULAR$_1^{CP}$ use the DFA, whilst REGULAR$_2$ uses the NFA constructed after simplification by when the business is closed.

The performance of a SAT solver can be sensitive to the ordering of the clauses in the formula. To test robustness of the models, we randomly shuffled each of PB instances to generate 10 equivalent problems and averaged the results over 11 instances. Also, the GRAMMAR and REGULAR constraints were encoded into a PB formula in two different ways. The first encoding ensures that unit propagation enforces domain consistency on the constraint. The second encoding ensures that UP detects disentailment of the constraint, but does not always enforce domain consistency. For the GRAMMAR constraint we omit the same set of clauses as in [4] to obtain the weaker PB encoding. For the REGULAR constraint we omit the set of clauses that performs the backward propagation of the REGULAR constraint. Note that Table 2 shows the median time and the number of backtracks to prove optimality over 11 instances. For each model we show the median time and the corresponding number of backtracks for the best PB encoding between the one that achieves domain consistency and the weaker one.

Table 2 shows the results of our experiments using these 5 models. The model REGULAR$_2$ outperforms GR$_1$ in all benchmarks, whilst model REGULAR$_1$ outperforms GR$_1$ in most of the benchmarks. The model REGULAR$_2$ also proves optimality in several instances of hard benchmarks. It should be noted that performing simplification before minimization is essential. It significantly reduces the size of the encoding and speeds up MiniSat+ by factor of 5[2]. Finally, we note that the PB models consistently outperformed the CP models, in agreement with the observations of [4]. Between the two CP models, REGULAR$_1^{CP}$ is significantly better than GR$_1^{CP}$, finding a better solution in many instances and proving optimality in two instances. In addition, although we do not show it in the table, Gecode is approximately three orders of magnitude faster per branch with the REGULAR$_1^{CP}$ model. For instance, in benchmark number 2 with 1 activity and 4 workers, it explores approximately 80 million branches with the REGULAR$_1^{CP}$ and 24000 branches with the GR$_1^{CP}$ model within the 1 hour timeout.

[2] Due to lack of space we do not show these results.

Table 2. Shift Scheduling Problems. GR_1 is the PB model with GRAMMAR, REGULAR_1 is the PB model with $\min(simplify(DFA))$, REGULAR_2 is the PB model with $\min(simplify(NFA))$, GR_1^{CP} is the CSP model with GRAMMAR, REGULAR_1^{CP} is the CSP model with $\min(simplify(DFA))$. We show **t**ime and number of **b**acktracks to prove optimality (the median time and the median number of backtracks for the PB encoding over **s**olved shuffled instances), number of **a**ctivities, the number of **w**orkers and the benchmark number #.

			PB/Minisat+									CSP/Gecode			
			GR_1			REGULAR_1			REGULAR_2			GR_1^{CP}		REGULAR_1^{CP}	
a	#	w	cost	s	t / b	cost	s	t / b	cost	s	t / b	cost	t / b	cost	t / b
1	2	4	**26.00**	11	27 / 8070	**26.00**	11	9 / 11053	**26.00**	11	**4 / 7433**	26.75	- / -	**26.00**	- / -
1	3	6	**36.75**	11	530 / 101560	**36.75**	11	94 / 71405	**36.75**	11	**39 / 58914**	37.00	- / -	37.00	- / -
1	4	6	**38.00**	11	31 / 16251	**38.00**	11	12 / 10265	**38.00**	11	**6 / 7842**	**38.00**	- / -	**38.00**	- / -
1	5	5	**24.00**	11	5 / 3871	**24.00**	11	2 / 4052	**24.00**	11	**2 / 2598**	**24.00**	- / -	**24.00**	- / -
1	6	6	**33.00**	11	9 / 5044	**33.00**	11	4 / 4817	**33.00**	11	**3 / 4045**	-	- / -	**33.00**	- / -
1	7	8	**49.00**	11	22 / 7536	**49.00**	11	9 / 7450	**49.00**	11	**7** / 8000	**49.00**	- / -	**49.00**	- / -
1	8	3	**20.50**	11	13 / 4075	**20.50**	11	4 / 5532	**20.50**	11	**2 / 1901**	21.00	- / -	**20.50**	92 / 2205751
1	10	9	**54.00**	11	242 / 106167	**54.00**	11	111 / **91804**	**54.00**	11	**110** / 109123	-	- / -	-	- / -
2	1	5	**25.00**	11	92 / 35120	**25.00**	11	96 / 55354	**25.00**	11	**32 / 28520**	**25.00**	- / -	**25.00**	90 / 1289554
2	2	10	**58.00**	1	3161 / **555249**	**58.00**	0	- / -	**58.00**	4	**2249** / 701490	-	- / -	**58.00**	- / -
2	3	6	**37.75**	0	- / -	**37.75**	1	3489 / 590649	**37.75**	9	**2342 / 570863**	42.00	- / -	40.00	- / -
2	4	11	**70.75**	0	- / -	71.25	0	- / -	71.25	0	- / -	-	- / -	-	- / -
2	5	4	**22.75**	11	739 / 113159	**22.75**	11	823 / 146068	**22.75**	11	**308 / 69168**	23.00	- / -	23.00	- / -
2	6	5	**26.75**	11	86 / 25249	**26.75**	11	153 / 52952	**26.75**	11	**28 / 21463**	**26.75**	- / -	**26.75**	- / -
2	8	5	**31.25**	11	1167 / 135983	**31.25**	11	383 / 123612	**31.25**	11	**74 / 47627**	32.00	- / -	31.50	- / -
2	9	3	**19.00**	11	1873 / 333299	**19.00**	11	629 / 166908	**19.00**	11	**160 / 131069**	19.25	- / -	**19.00**	- / -
2	10	8	**55.00**	0	- / -	**55.00**	0	- / -	**55.00**	0	- / -	-	- / -	-	- / -

7 Other Related Work

Beldiceanu *et al* [12] and Pesant [1] proposed specifying constraints using automata and provided filtering algorithms for such specifications. Quimper and Walsh [3] and Sellmann [2] then independently proposed the GRAMMAR constraint. Both gave a monolithic propagator based on the CYK parser. Quimper and Walsh [4] proposed a CNF decomposition of the GRAMMAR constraint, while Bacchus [10] proposed a CNF decomposition of the REGULAR constraint. Kadioglu and Sellmann [13] improved the space efficiency of the propagator for the GRAMMAR constraint by a factor of n. Their propagator was evaluated on the same shift scheduling benchmarks as here. However, as they only found feasible solutions and did not prove optimality, their results are not directly comparable. Côté, Gendron, Quimper and Rousseau proposed a mixed-integer programming (MIP) encoding of the GRAMMAR constraint [14], Experiments on the same shift scheduling problem used here show that such encodings are competitive.

There is a body of work on other methods to reduce the size of constraint representations. Closest to this work is Lagerkvist who observed that a REGULAR constraint represented as a multi-value decision diagram (MDD) is no larger than that represented by a DFA that is minimized and then unfolded [15]. A MDD is similar to an unfolded and then minimized DFA except a MDD can have long edges which skip over layers. We extend this observation by proving an exponential separation in size between such representations. As a second example, Katsirelos and Walsh compressed table constraints representing allowed or disallowed tuples using decision tree methods [16]. They also used a compressed representation for tuples that can provide exponentially savings in space. As a third example, Carlsson proposed the CASE constraint which can be

represented by a DAG where each node represents a range of values for a variable, and a path from the root to a leaf represents a set of satisfying assignments [17].

8 Conclusions

We have shown how to transform a GRAMMAR constraint into a REGULAR constraint specified. In the worst case, the transformation may increase the space required to represent the constraint. However, in practice, we observed that such transformation reduces the space required to represent the constraint and speeds up propagation. We argued that transformation also permits us to compress the representation using standard techniques for automaton minimization. We proved that minimizing such automata after they have been unfolded and domains initially reduced can give automata that are exponentially more compact than those obtained by minimizing before unfolding and reducing. Experimental results demonstrated that such transformations can improve the size of rostering problems that can be solved.

References

1. 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)
2. Sellmann, M.: The theory of grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 530–544. Springer, Heidelberg (2006)
3. Quimper, C.G., Walsh, T.: Global grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 751–755. Springer, Heidelberg (2006)
4. Quimper, C.G., Walsh, T.: Decomposing global grammar constraints. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 590–604. Springer, Heidelberg (2007)
5. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation. Addison Wesley Publishing Company, Reading (1979)
6. Darwiche, A.: On the tractable counting of theory models and its application to truth maintenance and belief revision. J. of Applied Non-Classical Logics 11, 11–34 (2001)
7. Meyer, A., Stockmeyer, L.: The equivalence problem for regular expressions with squaring requires exponential space. In: 13th Annual Symposium on Switching and Automata Theory, pp. 125–129. IEEE, Los Alamitos (1972)
8. Amilhastre, J., Janssen, P., Vilarem, M.C.: FA minimisation heuristics for a class of finite languages. In: Boldt, O., Jürgensen, H. (eds.) WIA 1999. LNCS, vol. 2214, pp. 1–12. Springer, Heidelberg (2001)
9. Revuz, D.: Minimization of ayclic deterministic automata in linear time. TCS 92, 181–189 (1992)
10. Bacchus, F.: GAC via unit propagation. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 133–147. Springer, Heidelberg (2007)
11. 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, pp. 140–154. Springer, Heidelberg (2005)
12. Beldiceanu, N., Carlsson, M., Petit, T.: Deriving filtering algorithms from constraint checkers. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 107–122. Springer, Heidelberg (2004)

13. Kadioglu, S., Sellmann, M.: Efficient context-free grammar constraints. In: AAAI 2008, pp. 310–316 (2008)
14. Cote, M.C., Bernard, G., Claude-Guy, Q., Louis-Martin, R.: Formal languages for integer programming modeling of shift scheduling problems. TR (2007)
15. Lagerkvist, M.: Techniques for Efficient Constraint Propagation. PhD thesis, KTH, Sweden, Licentiate thesis (2008)
16. Katsirelos, G., Walsh, T.: A compression algorithm for large arity extensional constraints. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 379–393. Springer, Heidelberg (2007)
17. Carlsson, M.: Filtering for the case constraint, Talk given at Advanced School on Global Constraints, Samos, Greece (2006)

IBM ILOG CP Optimizer for Detailed Scheduling Illustrated on Three Problems

Philippe Laborie

ILOG an IBM company,
9, rue de Verdun, 94253 Gentilly Cedex, France
laborie@fr.ibm.com

Abstract. Since version 2.0, IBM ILOG CP Optimizer provides a new scheduling language supported by a robust and efficient automatic search. This paper illustrates both the expressivity of the modelling language and the robustness of the automatic search on three problems recently studied in the scheduling literature. We show that all three problems can easily be modelled with CP Optimizer in only a few dozen lines (the complete models are provided) and that on average the automatic search outperforms existing problem specific approaches.

Keywords: Constraint Programming, Scheduling.

1 Introduction

Since version 2.0, IBM ILOG CP Optimizer provides a new scheduling language supported by a robust and efficient automatic search. This new-generation scheduling model is based on ILOG's experience in applying Constraint-Based Scheduling to industrial applications. It was designed with the following requirements in mind [1,2]:

- It should be accessible to software engineers and to people used to mathematical programming;
- It should be simple, non-redundant and use a minimal number of concepts so as to reduce the learning curve for new users;
- It should fit naturally into a CP paradigm with clearly identified variables, expressions and constraints;
- It should be expressive enough to handle complex industrial scheduling applications, which often are over-constrained, involve optional activities, alternative recipes, non-regular objective functions, *etc.*
- It should support a robust and efficient automatic search algorithm so that the user can focus on the declarative model without necessity to write any complex search algorithm (model-and-run development process).

The scheduling language is available in C++, Java, C# as well as in the OPL Optimization Programming Language[1]. The automatic search is based on a

[1] A trial version of OPL supporting this language can be downloaded on http://www.ilog.com/products/oplstudio/trial.cfm

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 148–162, 2009.

Self-Adapting Large Neighbourhood Search that iteratively *unfreezes* and *re-optimizes* a selected fragment of the current solution. The search algorithm is out of the scope of this paper, the principles of the approach have been described in [3] whereas more details about constraint propagation are available in [1].

The present paper illustrates the new modelling language and the efficiency and robustness of the automatic search on three problems recently studied in the scheduling literature. These problems were selected for several reasons:

- They are quite different in nature, covering cumulative and disjunctive scheduling, non-preemptive and preemptive scheduling, alternative modes, structured and unstructured temporal networks, *etc.*;
- All three problems are optimization problems with realistic non-regular objective functions (earliness/tardiness costs, number of executed tasks, complex temporal preference functions);
- They cover a range of different application domains (manufacturing, aerospace, project scheduling);
- Benchmarks and recent results are available to evaluate the efficiency of CP Optimizer's automatic search.

Section 2 recaps the modelling concepts of CP Optimizer used in the paper. Sections 3 to 5 are dedicated to the three scheduling problems: a flow-shop problem with earliness and tardiness costs [4], the oversubscribed scheduling problem studied in [5] and the personal task scheduling problem introduced in SelfPlanner [6]. Each of these sections starts with a description of the problem followed by a problem formulation in OPL. We show that all problems are easily modelled with CP Optimizer and that the resulting models are very concise (ranging from 15 to 42 lines of code). These models are then solved using the automatic search of CP Optimizer 2.1.1 with default parameter values on a 3GHz Linux desktop. We show that, in spite of its generality, the default search of CP Optimizer outperforms state-of-the-art approaches on all three problems.

2 CP Optimizer Model for Detailed Scheduling

This section recaps the conditional interval formalism introduced in [1,2]. It extends classical constraint programming by introducing with parsimony additional mathematical concepts (such as intervals, sequences or functions) as new variables or expressions to capture the temporal aspects of scheduling[2]. In this section we focus on the modelling concepts that are sufficient to understand the three models detailed in sections 3-5. A more formal and exhaustive description of the CP Optimizer concepts for detailed scheduling as well as several examples are provided in [7].

[2] In the present paper, a few concepts have been renamed so as to be consistent with their implementation in IBM ILOG CP Optimizer. In particular, we speak of *present*/*absent* rather than *executed*/*non-executed* interval variable and the notion of interval *duration* is replaced by the notion of interval *length*.

2.1 Interval Variables

An **interval variable** a is a decision variable whose domain $\text{dom}(a)$ is a subset of $\{\perp\} \cup \{[s,e) | s,e \in \mathbb{Z}, s \leq e\}$. An interval variable is said to be **fixed** if its domain is reduced to a singleton, i.e., if $\underline{a}$ denotes a fixed interval variable then:

- either interval is **absent**: $\underline{a} = \perp$;
- or interval is **present**: $\underline{a} = [s,e)$. In this case, s and e are respectively the **start** and **end** of the interval and $l = e - s$ its **length**.

Absent interval variables have special meaning. Informally speaking, an absent interval variable is not considered by any constraint or expression on interval variables it is involved in. For example, if an absent interval variable a is used in a precedence constraint between interval variables a and b then, this constraint does not influence interval variable b. Each constraint and expression specifies how it handles absent interval variables.

By default interval variables are supposed to be present but they can be specified as being **optional** meaning that $\perp$ is part of the domain of the variable and thus, it is a decision of the problem to have the interval present or absent in the solution. Optional interval variables provide a powerful concept for efficiently reasoning with optional or alternative activities. The following constraints on interval variables are introduced to model the basic structure of scheduling problems. Let a, a_i and b denote interval variables and z an integer variable:

- A **presence constraint** $\text{presenceOf}(a)$ states that interval a is present, that is $a \neq \perp$. This constraint can be composed, for instance $\text{presenceOf}(a) \Rightarrow \text{presenceOf}(b)$ means that the presence of a implies the presence of b.
- A **precedence constraint** (*e.g.* $\text{endBeforeStart}(a,b,z)$) specifies a precedence between interval end-points with an integer or variable minimal distance z provided both intervals a and b are present.
- A **span constraint** $\text{span}(a, \{a_1, ..., a_n\})$ states that if a is present, it starts together with the first present interval in $\{a_1, ..., a_n\}$ and ends together with the last one. Interval a is absent if and only if all the a_i are absent.
- An **alternative constraint** $\text{alternative}(a, \{a_1, ..., a_n\})$ models an exclusive alternative between $\{a_1, ..., a_n\}$: if interval a is present then exactly one of intervals $\{a_1, ..., a_n\}$ is present and a starts and ends together with this chosen one. Interval a is absent if and only if all the a_i are absent.

These constraints make it easy to capture the structure of complex scheduling problems (hierarchical description of the work-breakdown structure of a project, representation of optional activities, alternative modes/recipes/processes, etc.) in a well-defined CP paradigm.

Sometimes the intensity of "work" is not the same during the whole interval. For example let's consider a worker who does not work during weekends (his work intensity during weekends is 0%) and on Friday he works only for half a day (his intensity during Friday is 50%). For this worker, 7 man-days work will last for longer than just 7 days. In this example 7 man-days represent what we call the *size* of the interval: that is, the length of the interval would be if the intensity function was always at 100%. In CP Optimizer, this notion is captured

by an **integer step function** that describes the instantaneous *intensity* - expressed as a percentage - of a work over time. An interval variable is associated with an **intensity function** and a **size**. The intensity function F specifies the instantaneous ratio between size and length. If an interval variable a is present, the intensity function enforces the following relation:

$$100 \times size(a) \leq \int_{start(a)}^{end(a)} F(t).dt < 100 \times (size(a) + 1)$$

By default, the intensity function of an interval variable is a flat function equal to 100%. In this case, the concepts of *size* and *length* are identical.

It may also be necessary to state that an interval cannot start, cannot end at or cannot overlap a set of fixed dates. CP Optimizer provides the following constraints for modelling it. Let a denote an interval variable and F an integer stepwise function.

- **Forbidden start constraint.** Constraint forbidStart(a, F) states that whenever interval a is present, it cannot start at a value t where $F(t) = 0$.
- **Forbidden end constraint.** Constraint forbidEnd(a, F) states that whenever interval a is present, it cannot end at a value t where $F(t - 1) = 0$.
- **Forbidden extent constraint.** Constraint forbidExtent(a, F) states that whenever interval a is present, it cannot overlap a point t where $F(t) = 0$.

Integer expressions are provided to constrain the different components of an interval variable (start, end, length, size). For instance the expression startOf(a, dv) returns the start of interval variable a when a is present and returns integer value dv if a is absent (by default if argument dv is omitted it assumes $dv = 0$). Those expressions make it possible to mix interval variables with more traditional integer constraints and expressions.

2.2 Sequence Variables

Many problems involve scheduling a set of activities on a disjunctive resource that can only perform one activity at a time (typical examples are workers, machines or vehicles). From the point of view of the resource, a solution is a sequence of activities to be processed. Besides the fact that activities in the sequence do not overlap in time, additional constraints such as resource setup times or constraints on the relative position of activities in the sequence are common. To capture this idea we introduce the notion of *sequence variable*, a new type of decision variable whose value is a permutation of a set of interval variables. Constraints on sequence variables are provided for ruling out illegal permutations (sequencing constraints) or for stating a particular relation between the order of intervals in the permutation and the relative position of their start and end values (no-overlap constraint).

A **sequence variable** p is defined on a set of interval variables A. A value of p is a permutation of all present intervals of A. For instance, if $A = \{a, b\}$ is a set of two interval variables with a being necessarily present and b optional, the domain of the sequence p defined on A consists of 3 permutations: $\{(a), (a, b), (b, a)\}$.

If p denotes a sequence variable and a, b two interval variables in the sequence, the **sequencing constraints** first(p, a) and last(p, a) respectively mean that if interval a is present, it is the first or last in sequence p. Sequencing constraints before(p, a, b) and prev(p, a, b) respectively mean that if both intervals a and b are present, then a is before or immediately before b in sequence p.

It is to be noted that the sequencing constraints above do not have any impact on the start and end values of intervals, they only constrain the possible values (permutations) of the sequence variable. The **no-overlap constraint** noOverlap(p) on a sequence variable p states that permutation p defines a chain of non-overlapping intervals, any interval in the chain being constrained to end before the start of the next interval in the permutation.

For modelling sequence dependent setup times, each interval variable a in a sequence p can be associated with a non-negative integer **type** $T(p, a)$ and the no-overlap constraint can be associated with a transition distance. A **transition distance** M is a function $M : [0, n) \times [0, n) \rightarrow \mathbb{Z}^+$. If a and b are two successive non-overlapping present intervals, the no-overlap constraint noOverlap(p, M) will express a minimal distance $M(T(p, a), T(p, b))$ between the end of a and the start of b.

2.3 Cumul Function Expressions

For cumulative resources, the cumulated usage of the resource by the activities is a function of time. An activity usually increases the cumulated resource usage function at its start time and decreases it when it releases the resource at its end time. For resources that can be produced and consumed by activities (for instance the content of an inventory or a tank), the resource level can also be described as a function of time: production activities will increase the resource level whereas consuming activities will decrease it. In these problem classes, constraints are imposed on the evolution of these functions of time, for instance a maximal capacity or a minimum safety level.

CP Optimizer introduces the notion of a *cumul function expression* which is a constrained expression that represents the sum of individual contributions of intervals. A set of elementary cumul functions is available to describe the individual contribution of an interval variable (or a fixed interval of time or a fixed date). These elementary functions cover the use-cases mentioned above: *pulse* for usage of a cumulative resource, and *step* for resource production/consumption (see Figure 1). It is important to note that the elementary cumul functions

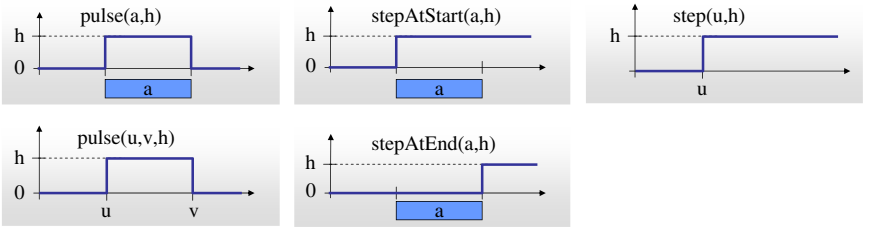


Fig. 1. Elementary cumul function expressions

defined on an interval variable are equal to the zero function when the interval variable is absent.

A **cumul function expression** f is defined as the sum of a set of elementary functions f_i or their negations: $f = \sum_i \epsilon_i \cdot f_i$ where $\epsilon_i \in \{-1, +1\}$. When the elementary cumul functions f_i that define a cumul function f are fixed (and thus, so are their related intervals), the cumul function itself is fixed and its value is a stepwise integer function. Several constraints are provided over cumul functions. These constraints allow restricting the possible values of the function over the complete horizon or over some fixed or variable interval. Let $u, v \in \mathbb{Z}$, $h, h_{min}, h_{max} \in \mathbb{Z}^+$ and a denote an interval variable. The following constraints are available on a cumul function f to restrict its possible values:

- alwaysIn$(f, u, v, h_{min}, h_{max})$ means that the values of function f must remain in the range $[h_{min}, h_{max}]$ everywhere on the fixed interval $[u, v)$.
- alwaysIn(f, a, h_{min}, h_{max}) means that if interval a is present, the values of function f must remain in the range $[h_{min}, h_{max}]$ between the start and the end of interval variable a.
- $f \leq h$: function f cannot take values greater than h.
- $f \geq h$: function f cannot take values lower than h.

3 Flow-Shop with Earliness and Tardiness Costs

3.1 Problem Description

The first problem studied in the paper is a flow-shop scheduling problem with earliness and tardiness costs on a set of instances provided by Morton and Pentico [8] that have been used in a number of studies including GAs [9] and Large Neighbourhood Search [4]. In this problem, a set of n jobs is to be executed on a set of m machines. Each job i is a chain of exactly m operations, one per machine. All jobs require the machines in the same order that is, the position of an operation in the job determines the machine it will be executed on. Each operation j of a job i is specified by an integer processing time $pt_{i,j}$. Operations cannot be interrupted and each machine can process only one operation at a time. The objective function is to minimize the total earliness/tardiness cost. Typically, this objective might arise in just-in-time inventory management: a late job has negative consequence on customer satisfaction and time to market, while an early job increases storage costs. Each job i is characterized by its release date rd_i, its due date dd_i and its weight w_i. The first operation of job i cannot start before the release date rd_i. Let C_i be the completion date of the last operation of job i. The earliness/tardiness cost incurred by job i is $w_i \cdot abs(C_i - dd_i)$. In the instances of Morton and Pentico, the total earliness/tardiness cost is normalized by the sum of operation processing times so the global cost to minimize is:

$$\frac{\sum_{i\in[1,n]} \left(w_i \cdot abs(C_i - dd_i)\right)}{W} \quad \text{where } W = \sum_{i\in[1,n]} \left(w_i \cdot \sum_{j\in[1,m]} pt_{i,j}\right)$$

Model 1 - OPL Model for Flow-shop with Earliness and Tardiness Costs

```
 1: using CP;
 2: int n = ...;
 3: int m = ...;
 4: int rd[1..n] = ...;
 5: int dd[1..n] = ...;
 6: float w[1..n] = ...;
 7: int pt[1..n][1..m] = ...;
 8: float W = sum(i in 1..n) (w[i] * sum(j in 1..m) pt[i][j]);
 9: dvar interval op[i in 1..n][j in 1..m] size pt[i][j];
10: dexpr int C[i in 1..n] = endOf(op[i][m]);
11: minimize sum(i in 1..n) w[i]*abs(C[i]-dd[i])/W;
12: subject to {
13:    forall(i in 1..n) {
14:       rd[i] <= startOf(op[i][1]);
15:       forall(j in 1..m-1)
16:          endBeforeStart(op[i][j],op[i][j+1]);
17:    }
18:    forall(j in 1..m)
19:       noOverlap(all(i in 1..n) op[i][j]);
20: }
```

3.2 Model

A complete OPL model for this problem is shown in Model 1. The instruction at *line 1* tells the model is a CP model to be solved by CP Optimizer. The section between *line 2* and *line 8* is data reading and data manipulation. The number of jobs n is read from the data file at *line 2* and the number of machines m at *line 3*. A number of arrays are defined to store, for each on the n jobs, the release date (*line 4*), due date (*line 5*), earliness/tardiness cost weight (*line 6*) and, for each machine, the processing time of each operation on the machine (*line 7*). The normalization factor W is computed at line *line 8*. The model itself is declared between *line 9* and *line 20*. *Line 9* creates a 2-dimensional array of interval variables indexed by the job i and the machine j. Each interval variable represents an operation and is specified with a size corresponding to the operation's processing time. *Line 10* creates one integer expression $C[i]$ for each job i equal to the end of the m^{th} (last) operation of the job. These expressions are used in *line 11* to state the objective function. The constraints are defined between *line 13* and *line 19*. For each job, *line 14* specifies that the first operation of job i cannot start before the job release date whereas precedence constraints between operations of job i are defined at *lines 15-16*. *Lines 18-19* state that for each machine j, the set of operations requiring machine j do not overlap.

3.3 Experimental Results

Table 1 compares the results obtained by the default automatic search of CP Optimizer using the above model (col. CPO) with the best results obtained by various genetic algorithms as reported in [9] (col. GA-best) and the results of the

Table 1. Results for Flow-shop Scheduling with Earliness and Tardiness Costs

Problem	GA-best	S-LNS-best	*CPO*	Problem	GA-best	S-LNS-best	*CPO*
jb1	0.474	0.191	*0.191*	ljb1	0.279	0.215	*0.215*
jb2	0.499	0.137	*0.137*	ljb2	0.598	0.508	*0.509*
jb4	0.619	0.568	*0.568*	ljb7	0.246	0.110	*0.137*
jb9	0.369	0.333	*0.334*	ljb9	0.739	1.015	*0.744*
jb11	0.262	0.213	*0.213*	ljb10	0.512	0.525	*0.549*
jb12	0.246	0.190	*0.190*	ljb12	0.399	0.605	*0.518*

best Large Neighbourhood Search (S-LNS) studied in [4] (col. S-LNS-best). A time limit of one hour was used on a 3GHz processor for CP Optimizer similar to the two hours limit used in [4] on a 1.5GHz processor. The average improvement (using the geometric mean over the ratio $value_{CPO}/value_{Other}$) over the best GA is about 25% whereas the average improvement over the best LNS is more modest (1.7%).

4 Satellite Scheduling

4.1 Problem Description

The second illustrative model is an oversubscribed scheduling problem described in [5]. This model is a generalization of two real-world oversubscribed scheduling domains, the USAF Satellite Control Network (AFSCN) scheduling problem and the USAF Air Mobility Command (AMC) airlift scheduling problem. These two domains share a common core problem structure:

- A problem instance consists of n tasks. In AFSCN, the tasks are communication requests; in AMC they are mission requests.
- A set Res of resources are available for assignment to tasks. Each resource $r \in Res$ has a finite capacity $cap_r \geq 1$. The resources are air wings for AMC and ground stations for AFSCN. The capacity in AMC corresponds to the number of aircraft for a wing; in AFSCN it represents the number of antennas available at the ground station.
- Each task T_i has an associated set Res_i of feasible resources, any of which can be assigned to carry out T_i. Any given task T_i requires 1 unit of capacity (i.e., one aircraft in AMC or one antenna in AFSCN) of the resource $r_j \in Res_i$ that is assigned to perform it. The duration $Dur_{i,j}$ of task T_i depends on the allocated resource r_j.
- Each of the feasible alternative resources $r_j \in Res_i$ specified for a task T_i defines a time window within which the duration of the task needs to be allocated. This time window corresponds to satellite visibility in AFSCN and mission requirements for AMC.
- All tasks are optional; the objective is to minimize the number of unassigned tasks[3].

[3] A second type of model with task priorities is also described in [5]. In the present paper, we focus on the version without task priorities.

4.2 Model

A complete OPL model for this problem is shown in Model 2 using the AFSCN semantics. The section between *line* 2 and *line* 6 is data reading and data manipulation. A tuple defining ground stations data (with a name, a unique integer identifier and a capacity) is defined at *line* 2 and read from the data file at *line* 4. A tuple defining a possible resource assignment for a task (specifying a task, a station, a task minimal start time, a task duration and a task maximal end time) is defined at *line* 3 and read from the data file at *line* 5. The set of all tasks $Tasks$ is computed at *line* 6 as the set of tasks used in at least one possible assignments.

Model 2 - OPL Model for Satellite Scheduling

```
 1: using CP;
 2: tuple Station { string name; key int id; int cap; }
 3: tuple Alternative { string task; int station; int smin; int dur; int emax; }
 4: {Station} Stations = ...;
 5: {Alternative} Alternatives = ...;
 6: {string} Tasks = { a.task | a in Alternatives };
 7: dvar interval task[t in Tasks] optional;
 8: dvar interval alt[a in Alternatives] optional in a.smin..a.emax size a.dur;
 9: maximize sum(t in Tasks) presenceOf(task[t]);
10: subject to {
11:    forall(t in Tasks)
12:       alternative(task[t], all(a in Alternatives: a.task==t) alt[a]);
13:    forall(s in Stations)
14:       sum(a in Alternatives: a.station==s.id) pulse(alt[a],1) <= s.cap;
15: }
```

Variables and constraints are defined between *line* 7 and *line* 15. *Line* 7 defines an array of interval variables indexed by the set of tasks $Tasks$. As tasks are optional and may be left unassigned, each of these interval variable is declared optional so that it can be ignored in the solution schedule. Each of the possible task assignments is defined as an optional interval variable in *line* 8. When present, these interval variables will be of size dur and belong to the time window $[smin, emax]$ of the assignment. This is expressed by the `size` and `in` OPL keywords in the interval variable declaration. The objective function is to maximize the number of assigned tasks, that is, the number of present tasks in the schedule; this is specified by a sum of presence constraints at *line* 9.

The constraints *lines* 11-12 state that each task, if present, is the alternative among the set of possible assignments for this task, this is modelled by an alternative constraint: if interval $task[t]$ is present, then one and only one of the intervals $alt[a]$ representing a ground station assignment for $task[t]$ will be present and $task[t]$ will start and end together with this selected interval. As specified by the semantics of the alternative constraint, if the task is absent, then all the possible assignments related with this task are absent too. The limited capacity (number of antennas) of ground stations is modelled by *lines* 13-14.

For each ground station s, a cumul function is created that represents the time evolution of the number of antennas used by the present assignments on this station s. This is a sum of unit pulse functions pulse($alt[a], 1$). Note that when the assignment $alt[a]$ is absent, the resulting pulse function is the zero function so it does not impact the sum. The resulting sum is constrained to be lower than the maximal capacity *cap* of the station. An interesting feature of the CP Optimizer model is that it handles optional tasks in a very transparent way: here, the fact that tasks are optional only impacts the declaration of task intervals at *line* 7. The notion of optional interval variable and the handling of absent intervals by the constraints and expressions of the model (here the alternative constraint and the cumul function expressions) allows an elegant modelling of scheduling problems involving optional activities and, more generally, optional and/or alternative tasks, recipes or modes.

4.3 Experimental Results

Table 2 compares the results obtained by the default automatic search of CP Optimizer using the above model (col. CPO) with the TaskSwap (TS) and Squeaky Wheel Optimization (SWO) approaches studied in [5] (col. TS and SWO). Figures represent the average number of unscheduled tasks for each problem set of the benchmark. The time limit for each instance was fixed to 120s for problem sets $x.1$, 180s for problem sets $x.2$ and 360s for problem sets $x.3$. In average, compared to the best approach described in [5] (SWO), the default automatic search of CP Optimizer assigns 5.3% more tasks.

Table 2. Results for Satellite Scheduling

Problem set	TS	SWO	*CPO*	Problem set	TS	SWO	*CPO*
1.1	30.44	26.60	*27.50*	4.1	3.20	2.00	*1.96*
1.2	114.02	104.72	*98.10*	4.2	13.34	7.90	*7.48*
1.3	87.92	84.52	*86.04*	4.3	16.60	12.46	*9.68*
2.1	11.46	7.80	*7.84*	5.1	3.90	3.80	*3.76*
2.2	45.54	34.26	*30.64*	5.2	32.98	31.98	*31.72*
2.3	33.96	31.18	*32.14*	5.3	46.18	45.22	*44.34*
3.1	2.64	2.32	*2.28*	6.1	1.56	1.28	*1.24*
3.2	15.50	12.82	*11.82*	6.2	11.62	9.56	*8.92*
3.3	32.10	28.58	*24.00*	6.3	25.28	22.60	*19.48*

5 Personal Task Scheduling

5.1 Problem Description

The third problem treated in this paper is the personal task scheduling problem introduced in [6] and available as an add-on to Google Calendar (`selfplanner.uom.gr/`). It consists of a set of n tasks $\{T_1, ..., T_n\}$. Each task T_i has a duration denoted dur_i. All tasks are considered preemptive, i.e. they

can be split into parts that can be scheduled separately. The decision variable p_i denotes the number of parts in which the i^{th} task has been split, where $p_i \geq 1$. T_{ij} denotes the j^{th} part of task T_i, $1 \leq j \leq p_i$. The sum of the durations of all parts of a task T_i must equal its total duration dur_i. For each task T_i, a minimum and maximum allowed duration for its parts, $smin_i$ and $smax_i$, as well as a minimum allowed temporal distance between every pair of its parts, $dmin_i$ are given. Depending on the values of $smax_i$ and $smin_i$ and the overall duration of the task dur_i, implicit constraints are imposed on p_i. For example, if $dur_i < 2 * smin_i$, then necessarily $p_i = 1$ and task T_i is non-preemptive. Each task T_i is associated with a domain $D_i = [s_{i1}, e_{i1}] \cup [s_{i2}, e_{i2}] \cup \cdots \cup [s_{iF_i}, e_{iF_i}]$, consisting of a set of F_i time windows within which all of its parts have to be scheduled. We denote respectively $L_i = s_{i1}$ and $R_i = e_{iF_i}$ the leftmost and rightmost values of domain D_i. A set of m locations is given, $Loc = \{l_1, l_2, \cdots, l_m\}$ as well as a 2-dimensional matrix $Dist$ with their temporal distances represented as non-negative integers. Each task T_i has its own spatial reference, $loc_i \in Loc$, denoting the location where the user should be in order to execute each part of the task. A set of ordering constraints, denoted $\prec (T_i, T_j)$ between some pairs of tasks is also defined, meaning that no part of task T_j can start its execution until all parts of task T_i have finished their execution. Time preferences are expressed for each task T_i. Five types of preference functions are available; they are depicted on Figure 2:

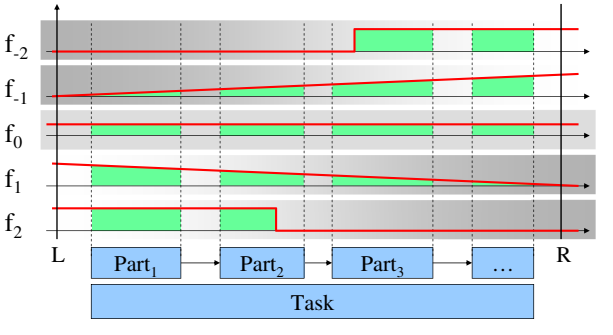


Fig. 2. Preference functions

f_{-2} Execute as much as possible of task T_i after a date d.
f_{-1} Execute as much as possible of task T_i as late as possible.
f_0 No preference.
f_1 Execute as much as possible of task T_i as early as possible.
f_2 Execute as much as possible of task T_i before a date d.

For a given preference function f_i associated with a task T_i that is split into p_i parts $P_{i,1}, ..., P_{i,p_i}$, the satisfaction related with the execution of task T_i is computed as:

$$\text{satisfaction}(T_i) = \sum_{j=1}^{p_i} \sum_{t \in P_{i,j}} f_i(t)$$

It is to be noted that functions f_i are normalized in the interval [0,1] in such a way that an upper bound for the satisfaction for a task T_i is 1.

5.2 Model

A complete OPL model for the personal task scheduling problem is shown in Model 3. The section between *line* 2 and *line* 16 is data reading and data manipulation. A tuple representing a task description is declared at *line* 2, it specifies a unique integer task identifier, the location of the task, the task duration, the minimal and maximal duration of task parts, the minimal delay between two consecutive task parts, an identifier of the type of preference function for the task in $\{-2, -1, 0, 1, 2\}$, the threshold date in case preference function is of type f_{-2} or f_2 and two sets of integers ds and de respectively representing the start and end dates of the intervals $[s_i, e_i]$ of the task domain. The set of tasks is read from the data file at *line* 3. A triplet representing the temporal distance between two locations is declared at *line* 4 and the transition distance matrix represented as a set of such triplets is read from the data file at *line* 5. A tuple storing an ordering constraint is defined on *line* 6 and a set of such tuples is read from the data at *line* 7. *Lines* 8-10 respectively compute, for each task t the leftmost value, rightmost value and diameter of the task domain. A tuple representing the i^{th} part of a task is defined at *line* 11 and the total set of possible parts is computed at *line* 12 considering that for each task of duration dur and minimal part duration $smin$, the maximal number of parts is $\lfloor dur/smin \rfloor$. *Lines* 13-16 define a step function $holes[t]$ for each task t that is equal to 1 in the domain of t and to 0 everywhere else.

Variables and constraints are defined between *lines* 17 *and* 42. An array of interval variables, one interval $task[t]$ for each task t, is declared at *line* 17; each task is constrained to end before the schedule horizon (500 in the benchmark). *Line* 18 defines an optional interval variable for each possible task part with a minimal and a maximal size given by $smin$ and $smax$. A sequence variable is created at *line* 19 on the set of all parts p, each part being associated with an integer type in the sequence corresponding to the location of the part. The satisfaction expression for each task t is modelled on *lines* 20-25 depending on the preference function type; it uses the OPL conditional expression `c?e1:e2` where c is a boolean condition and $e1$ is the returned expression if c is true and $e2$ the returned expression if c is false. The normalization factors are the ones used in [6][4]. The objective function, as defined on *line* 26 is to maximize the sum of all tasks satisfaction.

The constraints on *line* 29 forbid any part of a task t to overlap a point where the step function $holes[t]$ is zero; this will constrain each task part to be executed in its domain. Constraints on *lines* 31-32 state that the set of parts of a given task t forms a chain of optional intervals with minimum separation time $dmin$ among which only the first ones will be executed, that is, each part $a[p]$ if present is constrained to be executed before its successor part $a[s]$ and the presence of

[4] The objective expression being quite complex, we used the solution checker provided with the instances to check that the constraints and objective function of our model are equivalent to the ones used in [6].

part $a[s]$ implies the presence of part $a[p]$. Constraints on *line 36* state that the total duration of the part of a task must equal the specified task duration *dur*. Note that when part $a[p]$ is absent, by default the value of sizeOf($a[p]$) is 0. *Line 37* constrains each task t to span its parts, that is to start at the start of first

Model 3 - OPL Model for Personal Task Scheduling

```
 1: using CP;
 2: tuple Task { key int id; int loc; int dur; int smin; int smax; int dmin; int f; int
    date; {int} ds; {int} de; }
 3: {Task} Tasks = ...;
 4: tuple Distance { int loc1; int loc2; int dist; };
 5: {Distance} Dist = ...;
 6: tuple Ordering { int pred; int succ; };
 7: {Ordering} Orderings = ...;
 8: int L[t in Tasks] = min(x in t.ds) x;
 9: int R[t in Tasks] = max(x in t.de) x;
10: int S[t in Tasks] = R[t]-L[t];
11: tuple Part { Task task; int id; }
12: {Part} Parts = { <t,i> | t in Tasks, i in 1 .. t.dur div t.smin };
13: tuple Step { int x; int y; }
14: sorted  {Step} Steps[t in  Tasks] =
15:    {<x,0> | x in t.ds} union {<x,1> | x in t.de};
16: stepFunction holes[t in Tasks] = stepwise(s in Steps[t]) {s.y -> s.x; 0};
17: dvar interval tasks[t in Tasks] in 0..500;
18: dvar interval a[p in Parts] optional size p.task.smin..p.task.smax;
19: dvar sequence seq in all(p in Parts) a[p] types all(p in Parts) p.task.loc;
20: dexpr float satisfaction[t in Tasks] = (t.f==0)? 1 :
21:    (1/t.dur)* sum(p in Parts: p.task==t)
22:       (t.f==-2)? maxl(endOf(a[p]),t.date)-maxl(startOf(a[p]),t.date) :
23:       (t.f==-1)? lengthOf(a[p])*(R[t]-(startOf(a[p])+endOf(a[p])-1)/2)/S[t] :
24:       (t.f== 1)? lengthOf(a[p])*((startOf(a[p])+endOf(a[p])-1)/2-L[t])/S[t] :
25:       (t.f== 2)? minl(endOf(a[p]),t.date)-minl(startOf(a[p]),t.date) : 0;
26: maximize sum(t in Tasks) satisfaction[t];
27: subject to {
28:    forall(p in Parts) {
29:       forbidExtent(a[p], holes[p.task]);
30:       forall(s in Parts: s.task==p.task && s.id==p.id+1) {
31:          endBeforeStart(a[p], a[s], p.task.dmin);
32:          presenceOf(a[s]) => presenceOf(a[p]);
33:       }
34:    }
35:    forall(t in Tasks) {
36:       t.dur == sum(p in Parts: p.task==t) sizeOf(a[p]);
37:       span(tasks[t], all(p in Parts: p.task==t) a[p]);
38:    }
39:    forall(o in Orderings)
40:       endBeforeStart(tasks[<o.pred>], tasks[<o.succ>]);
41:    noOverlap(seq, Dist);
42: }
```

part and to end with the end of the last executed part. Ordering constraints are declared on *line 40* whereas *line 41* states that task parts cannot overlap and that they must satisfy the minimal transition distance between task locations defined by the set of triplets *Dist*.

5.3 Experimental Results

Table 3 compares the results obtained by the default automatic search of CP Optimizer using the above model (col. CPO) and a time limit of 60s for each problem with the Squeaky Wheel Optimization (SWO) approach implemented in SelfPlanner [6] (col. SWO). CP Optimizer finds a solution to more problems than the approach described in [6]: the SWO could not find any solution for the problems with 55 tasks whereas the automatic search of CP Optimizer solves 70% of them. Furthermore, SWO could not find any solution to 4 of the smaller problems with 50 tasks whereas CP Optimizer solves them all but for problem 50-2. On problems where SWO finds a solution, the average task satisfaction (average of the ratio between the total satisfaction and the number of tasks) is 78% whereas it is 87.8% with CP Optimizer. It represents an improvement of about 12.5% in solution quality.

Table 3. Results for Personal Task Scheduling

#	SWO	*CPO*	#	SWO	*CPO*	#	SWO	*CPO*	#	SWO	*CPO*
15-1	12.95	*14.66*	30-6	28.09	*29.28*	40-1	24.72	*28.95*	45-6	32.70	*37.35*
15-2	12.25	*13.16*	30-7	23.80	*24.20*	40-2	23.48	*32.07*	45-7	32.40	*35.77*
15-3	13.71	*13.90*	30-8	24.06	*26.89*	40-3	33.57	*37.74*	45-8	31.79	*35.23*
15-4	11.57	*12.55*	30-9	23.42	*24.86*	40-4	31.46	*35.45*	45-9	35.79	*38.86*
15-5	12.64	*14.67*	30-10	22.04	*27.18*	40-5	28.05	*34.21*	45-10	32.78	*40.68*
15-6	14.30	*14.63*	35-1	28.80	*31.56*	40-6	29.46	*34.01*	50-1	42.04	*43.53*
15-7	13.08	*14.46*	35-2	29.17	*32.33*	40-7	33.13	*37.51*	50-2	×	×
15-8	11.46	*12.37*	35-3	27.84	*28.58*	40-8	29.72	*34.90*	50-3	×	*37.17*
15-9	11.44	*11.61*	35-4	26.64	*29.67*	40-9	33.03	*36.89*	50-4	×	*36.52*
15-10	12.07	*13.51*	35-5	25.15	*32.13*	40-10	30.28	*34.19*	50-5	34.25	*43.55*
30-1	24.17	*29.13*	35-6	26.12	*29.49*	45-1	37.42	*42.90*	50-6	38.32	*41.87*
30-2	24.69	*27.55*	35-7	29.28	*31.69*	45-2	33.97	*39.71*	50-7	32.59	*42.48*
30-3	25.61	*26.53*	35-8	25.71	*30.07*	45-3	35.44	*39.40*	50-8	34.70	*43.67*
30-4	27.13	*28.49*	35-9	23.74	*29.60*	45-4	33.02	*37.41*	50-9	×	*42.75*
30-5	23.89	*26.46*	35-10	30.70	*33.41*	45-5	30.83	*36.65*	50-10	37.46	*41.84*
55-1	×	*36.84*	55-4	×	*40.36*	55-7	×	×	55-10	×	×
55-2	×	*38.56*	55-5	×	*42.70*	55-8	×	*45.27*			
55-3	×	×	55-6	×	*35.92*	55-9	×	*42.14*			

6 Conclusion

This paper illustrates the new scheduling support in IBM ILOG CP Optimizer. We selected three problems recently studied in the scheduling literature and provide a simple and concise CP Optimizer model for each of them. The size of the

OPL models range from 15 to 42 lines of code. These models are then solved using the automatic search of CP Optimizer with default parameter values. We show that on average, CP Optimizer outperforms state-of-the-art problem specific approaches on all the problems which is quite a remarkable result given the generality of the search and the large spectrum of problem characteristics. These results are consistent with our experience of using CP Optimizer on industrial detailed scheduling applications. In spite of the relative simplicity of the new scheduling language based on optional interval variables, it was shown to be expressive and versatile enough to model a large range of complex problems for which the automatic search proved to be efficient and robust. The major part of the future development of CP Optimizer will be the continued improvement of the automatic search process.

References

1. Laborie, P., Rogerie, J.: Reasoning with Conditional Time-intervals. In: Proc. 21th International FLAIRS Conference (FLAIRS 2008), pp. 555–560 (2008)
2. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: Reasoning with Conditional Time-intervals, Part II: an Algebraical Model for Resources. In: Proc. 22th International FLAIRS Conference (FLAIRS 2009) (2009)
3. Laborie, P., Godard, D.: Self-Adapting Large Neighborhood Search: Application to Single-mode Scheduling Problems. In: Proc. of the 3rd Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA), pp. 276–284 (2007)
4. Danna, E., Perron, L.: Structured vs. Unstructured Large Neighborhood Search: a Case Study on Job-shop Scheduling Problems with Earliness and Tardiness Costs. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 817–821. Springer, Heidelberg (2003)
5. Kramer, L.A., Barbulescu, L.V., Smith, S.F.: Understanding Performance Tradeoffs in Algorithms for Solving Oversubscribed Scheduling. In: Proc. 22nd AAAI Conference on Artificial Intelligence (AAAI 2007), pp. 1019–1024 (2007)
6. Refanidis, I.: Managing personal tasks with time constraints and preferences. In: Proc. 17th International Conference on Automated Planning and Scheduling Systems (ICAPS 2007), pp. 272–279 (2007)
7. Laborie, P., Rogerie, J., Shaw, P., Vilím, P., Wagner, F.: ILOG CP Optimizer: Detailed Scheduling Model and OPL Formulation. Technical Report 08-002, ILOG (2008), http://www2.ilog.com/techreports/
8. Morton, T., Pentico, D.: Heuristic Scheduling Systems. Wiley, Chichester (1993)
9. Vázquez, M., Whitley, L.D.: A Comparison of Genetic Algorithms for the Dynamic Job Shop Scheduling problem. In: Proc. GECCO 2000 (2000)

Open Constraints in a Boundable World

Michael J. Maher

NICTA* and University of NSW
Sydney, Australia
Michael.Maher@nicta.com.au

Abstract. Open forms of global constraints allow the addition of new variables to an argument during the execution of a constraint program. Such forms are needed for difficult constraint programming problems where problem construction and problem solving are interleaved. We introduce a new model of open global constraint where the length of the sequence of variables can be constrained but there is no a priori restriction on the variables that might be added. In general, propagation that is sound for a global constraint can be unsound when the constraint is open. We identify properties of constraints that simplify the design of algorithms for propagation by identifying when no propagation can be done, and use them to design propagation algorithms for several open global constraints.

1 Introduction

The classic CSP model of constraint satisfaction [5,14] has a fixed static collection of variables over which a solution must be found. However, in many problems it is natural for the presence of some variables to be contingent on the value of other variables. This is true of configuration problems and scheduling problems that involve process-dependent activities [1]. More generally, for difficult problems the intertwining of problem construction and problem solving provides a way to manage the complexity of a problem, and thus new variables and constraints may arise after solving has begun. Programming languages supporting constraint programming generally have the flexibility to add variables and constraints during the execution of a model.

However, global constraints do not have this flexibility: all variables must be available at the time the constraint is imposed, so variables cannot simply be added when they become available. The collection of variables they constrain is *closed*, rather than *open*. This can leave the filtering effect of the global constraint until too late in the execution, resulting in a large search space.

Recent work has focused on supporting open versions of global constraints. Barták [1] first formulated this issue and described a generic dynamisation technique to make open versions for the class of *monotonic* global constraints. But

* 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.

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 163–177, 2009.

this technique is inefficient, and he also provided a specific implementation of the open AllDifferent constraint by modifying Régin's algorithm [12] for the closed AllDifferent. He outlined a proposal for implementing open non-monotonic global constraints.

A notion of Open CSP was investigated in [6]. In that work the set of variables is closed but the domains are open, that is, extra values can be added to variable domains.

Later work [8] addressed the problem when the variables that might be added to a constraint are specified in advance and a set variable describes the set of variables that will participate in the constraint. In this formulation, it becomes possible to filter the set variable in addition to the individual variables. [8] also gives an implementation of the global cardinality constraint GCC and of multiple open GCC constraints over disjoint variables.

In this paper we propose a new model for open constraints, intermediate between the models of [1] and [8]. We define properties of constraints – called contractibility and extensibility – and show that constraints with these properties have simplified propagation. In particular, using contractibility we define, for three constraints, open domain consistent propagators under Barták's model of open constraints as simple adaptations of the corresponding closed propagators. However, these propagators are not sufficient to achieve open domain consistency under our model, and we describe stronger propagators to achieve this consistency. The ideas underlying these propagators also come from their closed counterparts, but the adaptation is more complex.

After some preliminaries in Section 2 we define and discuss the model of open constraints we propose in comparison to the models of [1,8]. We introduce the properties contractibility and extensibility in Section 4, and show how they affect propagation. We investigate, in particular, the GCC, Regular and CFG constraints. In Section 5 we establish tight approximations for these constraints, which can be the basis of propagators under Barták's model. Following sections describe open domain consistent propagators for these three constraints under our model.

2 Background

The reader is assumed to have a basic knowledge of constraint programming, CSPs, global constraints, and filtering, as might be found in [5,14,2]. In this section we discuss only conventional (closed) constraints.

To begin with, we view a global constraint as a relation over a sequence of variables. Other arguments of a constraint are considered parameters and are assumed to be fixed before execution. While some global constraints (such as AllDifferent and GCC) are more naturally represented as relations over an unordered collection of variables, we will find it convenient to ignore this abstraction: for some constraints (such as Sequence) a sequence of variables is necessary, and uniformity over all constraints will simplify our treatment of the issues. A sequence of variables will be denoted by $\boldsymbol{X}$ or $[X_1, \ldots, X_n]$. $|\boldsymbol{X}|$ denotes the length of $\boldsymbol{X}$.

We make no *a priori* restriction on the variables that may participate in the sequence except that, in common with most work on global constraints, we assume that no variable appears more than once in a single constraint.

There are some specific global constraints that we define for completeness. These and other global constraints are discussed more completely in [2] and the references therein. The constraint AllDifferent($[X_1, \ldots, X_n]$) [12] states that the variables $X_1, \ldots, X_n$ take distinct values. The global cardinality constraint GCC($[X_1, \ldots, X_n], \boldsymbol{v}, \boldsymbol{l}, \boldsymbol{u}$) [13] states that, for every i, the value v_i occurs between l_i and u_i times in the list of variables (u_i may be infinite). The constraint Regular($\mathcal{A}, [X_1, \ldots, X_n]$) [10] states that value of the list of variables, when considered as a word, is accepted by the automaton $\mathcal{A}$. Similarly, the constraint CFG($\mathcal{G}, [X_1, \ldots, X_n]$) [11,15] states that value of the list of variables, when considered as a word, is generated by the context-free grammar $\mathcal{G}$.

Each variable X has a static type which defines a (possibly infinite) set of values $T(X)$ which it may take. In addition, generally, each variable has an associated set $S \subseteq T(X)$ of values, called its *domain*. We will view this simultaneously as: a function $D : Vars \rightarrow 2^{Values}$ where $D(X) = S$, a unary relation $D(X)$ which is satisfied only when the value of X is some $s \in S$, and the pointwise extension of D to sequences of variables. With each use of a constraint $C(\boldsymbol{X})$ is an associated type T such that every variable X_i that appears in $\boldsymbol{X}$ satisfies $T(X_i) \subseteq T$. (In general, we will need a more sophisticated typing of $\boldsymbol{X}$, but this simple typing is sufficient for this paper.)

The reduction of a domain D to D' is *sound* wrt a constraint $C(\boldsymbol{X})$ iff $D(\boldsymbol{X}) \wedge C(\boldsymbol{X}) \leftrightarrow D'(\boldsymbol{X}) \wedge C(\boldsymbol{X})$. We define consistency with respect to a set of variables that may be different from $\boldsymbol{X}$. Given a set of variables $\boldsymbol{Y}$, a domain D and a constraint $C(\boldsymbol{X})$ we say that $D(\boldsymbol{Y})$ is *domain consistent with* C if for every $Y_i \in \boldsymbol{Y}$ and every $d \in D(Y_i)$ there is a solution of $C(\boldsymbol{X})$ in which $Y_i = d$.

3 A Model of Open Constraints

Open constraints pre-suppose the existence of a meta-program that can impose constraints, close an open constraint, add variables to a constraint, (possibly) create new variables, and interact with the execution of the constraint system, possibly controlling it. In this paper we will abstract away the details of the meta-program so that we can focus on the open constraints.

There are two models of open constraint that have been investigated. Barták's model of open global constraints [1] is straightforward: the constraint involves a sequence of variables to which variables may be added. Thus the arity and type of the constraint are unchanged, whether the constraint is open or closed. In this model there is no restriction on the variables to be added and there is no way within the constraint system to communicate information about the variables that have been added to other constraints or to constrain the possible additions to the constraint; that communication must be done by the meta-program.

The model of van Hoeve and Régin [8] uses a set variable S describing a set of object variables, rather than a sequence, to represent the collection of variables in

the constraint[1]. The lower bound of S is the set of variables that are committed to appear in the constraint; the upper bound is the set of variables that are permitted to appear in the constraint. Thus there is a finite set of variables that might appear in the constraint, and these are fixed in advance. The authors refer to the constraint as open "in a closed world" since the set of variables that might be added to the constraint is closed.

This model has the advantage that the effect of adding a variable to the constraint can be communicated to other global constraints via constraints on set variables. In this way, the constraint program can make explicit restrictions that would need to be embedded in the meta-program under Barták's model. For example, constraints that ensure that the collections of variables in different constraints are disjoint are discussed in [8]. The model makes elegant use of existing implementations of set variables and their associated bounds.

On the other hand, use of a constraint in this model requires knowing all the variables that might appear before imposing the constraint. As a result, it cannot deal well with contingent variables. They create a similar problem to the one faced by closed constraints: the constraint may be imposed late in the execution, creating a larger search space. An alternative might be to create all variables that might possibly be needed at the beginning of an execution, but that has several costs. It creates a large initialization cost, requires ensuring meaningless variable do not interfere with satisfiability or with search heuristics, and leaves set variables with very large upper bounds that might limit the opportunities for propagation both within a constraint and between constraints.

Furthermore, there is no easy and natural way to represent open constraints where the order of variables is significant. Among the constraints that are affected are Change, Contiguity, lexicographical ordering, Regular, Sequence, and Sort.

Finally, open constraint programming, by its nature, involves meta-programming explicitly instead of the declarative form of CSPs. However, introducing variables as objects to be reasoned on by constraints seems to be injecting too much of the meta-program into the constraints. It opens up many complications, including the question: Can the set variable S be an object variable in a global constraint and, in particular, can S be a member of itself? Nevertheless, this point is mostly a matter of taste.

The model of open constraints proposed here, is intermediate between that of Barták [1] and van Hoeve and Régin [8]. Under this model, a constraint $C(\boldsymbol{X}, N)$ acts on both a sequence of variables $\boldsymbol{X}$ and an integer variable N representing the length of the sequence once it is closed. Variables can only be added at one end of the sequence. This allows a natural representation of sequential constraints, such as Change, etc., and supports the use of constraints on the cardinality of the sequence of variables. In one sense, this model is an abstraction of the model of [8]: if N is subject only to lower and upper bounds, then the bounds on N correspond to the cardinalities of the bounds of S. However, it is able

[1] A set variable S ranges over sets and is constrained by two fixed finite sets L and U which are a lower and upper bound on the value of the variable: $L \subseteq S \subseteq U$. See [7].

to represent sequential constraints, and it is also able to treat N as a domain variable. The two models might be merged in a constraint $C(\boldsymbol{X}, S, N)$ but this possibility will not be addressed here.

In comparison to the model of [1], the variable N in this model gives more information to the propagator, which might be used to propagate further on $\boldsymbol{X}$, and can extract extra information from the propagator. For example, consider the constraint $\textsc{Regular}(a + a^2 + b^2b^*, \boldsymbol{X}, N)$. If we know $N \geq 3$ then we can infer $X_1 = b$. If we know $X_1 = a$ then we can infer $N \leq 2$.

In general, we will consider N to have a domain $D(N)$, but we will also want to refer to the bounds on N. We define l_N and u_N to be the minimum and maximum of $D(N)$. In consistent states we must have $|\boldsymbol{X}| \leq l_N$. On occasion, the argument N is irrelevant and we write $C(\boldsymbol{X})$ to denote $\exists N\ C(\boldsymbol{X}, N)$.

It is conventional to define the semantics of a constraint as a relation, that is, a set of tuples. However, the tuples defining the semantics of an open constraint are not of equal length, and hence do not form a relation. Instead, we propose to view the semantics of an open constraint C as a formal language.

Definition 1. *The* semantics *of an open constraint C with associated type T is a set of words $d_1 \dots d_n$ where $d_i \in T$ and n is not fixed. We denote this language by L_C.*

A solution *of an occurrence of an open constraint $C(\boldsymbol{X}, N)$ with $|\boldsymbol{X}| = k$ is a valuation mapping each X_i to some d_i and N to n such that $k \leq n$ and there is a word $d_1 \dots d_n$ in L_C.*

For example, the semantics of $\textsc{Regular}(a^* + bc, \boldsymbol{X}, N)$ is $\{\varepsilon, a, aa, bc, aaa, \dots, a^i, \dots\}$. The solutions of $\textsc{Regular}(a^* + bc, [X_1], N)$ are $X_1 = b, N = 2$ and $X_1 = a, N = n$ for each $n \geq 1$. The semantics of $\textsc{AllDifferent}(\boldsymbol{X}, N)$ over the type $T = \{a, b\}$ is $\{\varepsilon, a, b, ab, ba\}$.

We define some properties of formal languages, for later use. Let $P(L) = \{w \mid \exists u\ wu \in L\}$ denote the set of prefixes of a language L, called the prefix-closure of L. We say L is *prefix-closed* if $P(L) = L$. We say L is *continual* at threshold m for type T, $w \in L$ and $|w| > m \ \Rightarrow\ \exists a \in T$ *s.t.* $wa \in L$.

Because some of the variables in an open constraint will be unspecified during part of the execution, we need to adapt the definitions of consistency.

Definition 2. *Given a domain D, an occurrence of a constraint $C(\boldsymbol{X}, N)$ is* open domain consistent *if*

- *for every $X_i \in \boldsymbol{X}$ and every $d \in D(X_i)$ there is a word $d_1 \dots d_n$ in L_C such that $d_i = d$, $|\boldsymbol{X}| \leq n$, $n \in D(N)$, and $d_j \in D(X_j)$ for $j = 1, \dots, |\boldsymbol{X}|$; and*
- *for every $n \in D(N)$ there is a word $d_1 \dots d_n$ in L_C such that $d_j \in D(X_j)$ for $j = 1, \dots, |\boldsymbol{X}|$.*

In some cases we may wish to discuss weaker notions of consistency, in particular when $D(N)$ is approximated by a bound or bounds. We define open DL-consistency, DU-consistency, and DB-consistency as weakenings of open domain consistency where, respectively, the domain of N is abstracted to a lower bound, an upper bound, and a pair of bounds. If we abstract away from N altogether,

we obtain constraints $C(\boldsymbol{X})$, for which we need another notion of consistency. We use open D-consistency to refer to the first part of the above definition, ignoring the restriction imposed by $D(N)$. When C is closed, $D(N) = \{|\boldsymbol{X}|\}$ and open D-consistency is exactly domain consistency.

4 Contractibility and Extensibility

The properties of contractibility and extensibility are essentially converses of each other. Contractibility of a constraint C guarantees that any variable/value pair that is not part of any solution of C remains not part of any solution after the sequence of variables is extended. Extensibility of C guarantees that any variable/value pair that has support in C continues to have support after the sequence is extended.

The properties of contractibility and extensibility are independent of the argument N of the open constraints. For this section we will express a constraint C as $C(\boldsymbol{X})$.

We view the appending of a variable Y to a sequence of variables $\boldsymbol{X}$ as a two part process: the extension of the sequence, and then the communication of the domain of Y to the constraint. Some of the results of this section distinguish propagation induced by the extension of the sequence from propagation that is a consequence of the domain of Y, and focus only on the former.

Proposition 1. *The following three conditions are equivalent, where all variables range over T:*

1. *for all $n \geq 0$ we have*
$$C([X_1, \ldots, X_n, Y]) \rightarrow C([X_1, \ldots, X_n])$$
2. *$L_C \cap T^*$ is prefix-closed*
3. *for all $n \geq 0$, any domain D and domain reduction to D' that is sound wrt $C([X_1, \ldots, X_n])$*
$$D(\boldsymbol{X}) \wedge C([X_1, \ldots, X_n, Y]) \leftrightarrow D'(\boldsymbol{X}) \wedge C([X_1, \ldots, X_n, Y])$$

Definition 3. *We say a constraint $C([X_1, \ldots, X_n])$ is* contractible *if it satisfies any of the three conditions in Proposition 1.*

Contractibility is a variation of Barták's monotonicity [1] where we do not explicitly discuss variable domains. A constraint is contractible iff it is monotonic wrt every domain.

For a contractible constraint C, any sound form of filtering (such as domain consistency or bounds consistency) on $C([X_1, \ldots, X_n])$ is safe in the sense that any values deleted from domains in that process could also be deleted while filtering on $C([X_1, \ldots, X_n, Y])$. Consequently, for contractible constraints, filtering does not need to be undone if the sequence of variables is extended. That is, propagators for a closed contractible constraint are valid also for the corresponding open constraint.

Conversely, for any constraint that is not contractible, there is a domain and a domain reduction that would need to be undone if the sequence were extended. For example, a constraint $\sum_i X_i = 5$ would propagate $X_1 = 5$ if the sequence $\boldsymbol{X}$ contained just one variable, thus eliminating solutions such as $X_1 = 2, X_2 = 3$. When the second variable is added, all propagation that is a consequence of the inference $X_1 = 5$ must be undone.

It is exactly the contractible constraints for which we can interleave closed filtering and addition of new variables. There are many contractible global constraints, including AllDifferent, BinPacking, Contiguity, Cumulative, Diffn, Disjoint, InterDistance, Precedence, Sequence, SlidingSum and lexicographical ordering $\leq_{lex}$. Regular and CFG are contractible when the language of the automaton/grammar is prefix-closed. More details about contractibility and contractible constraints can be found in [9].

We might also consider a converse property to contractibility:

Proposition 2. *The following three conditions are equivalent, where all variables range over T:*

1. *for all $n > m$ we have*

$$C([X_1, \ldots, X_n]) \rightarrow \exists Y \; C([X_1, \ldots, X_n, Y])$$

2. *L_C is continual at threshold m for T*
3. *for all $n > m$ for any domain D and domain reduction to D' that is sound wrt $C([X_1, \ldots, X_n, Y])$*

$$D(\boldsymbol{X}) \wedge C([X_1, \ldots, X_n]) \leftrightarrow D'(\boldsymbol{X}) \wedge C([X_1, \ldots, X_n])$$

Definition 4. *We say a constraint $C([X_1, \ldots, X_n])$ is* extensible *if, there is a number m such that any of the three above conditions holds. The least such m is called the* extensibility threshold.

Extensibility guarantees that filtering would not delete more values from domains if the list were known to be longer. An extensible language must be infinite if it contains a word longer than its threshold. Sequence, SlidingSum, Precedence and Contiguity are extensible. GCC is extensible when there is a value in the associated type that does not have a finite upper bound. Regular and CFG are extensible when the language of the automaton/grammar is continual.

The first part of the following result implies that, for extensible constraints, the addition of a variable Y does not induce propagation, so that propagation can only come from the domain of Y.

Proposition 3. *Let C be a constraint and D a domain.*

If C is extensible with threshold m, $|\boldsymbol{X}| > m$ and $D(\boldsymbol{X})$ is domain consistent with $C(\boldsymbol{X})$ then $D(\boldsymbol{X})$ is domain consistent with $C(\boldsymbol{X}Y)$.

If C is contractible and $D(\boldsymbol{X}Y)$ is domain consistent with $C(\boldsymbol{X}Y)$ then $D(\boldsymbol{X})$ is domain consistent with $C(\boldsymbol{X})$.

In general, a change of bounds on N can induce filtering on $\boldsymbol{X}$ and domain reduction on $\boldsymbol{X}$ can induce a change of bounds on N. The following result shows that in some circumstances an update to the domain of $\boldsymbol{X}$ will not engender any further propagation on a bound of N and, similarly, an update to the bounds of N will not engender any further propagation on $\boldsymbol{X}$.

Proposition 4. *Consider a constraint $C(\boldsymbol{X}, N)$ and domain D, and let l_N and u_N be the lower and upper bounds of $D(N)$. Suppose D is DB-consistent with C and consider the problem of maintaining this consistency. If C is contractible then*

- *reduction in u_N does not constrain $\boldsymbol{X}$*
- *reduction in $D(\boldsymbol{X})$ does not increase l_N*

If C is extensible then

- *increase in l_N does not constrain $\boldsymbol{X}$*
- *reduction in $D(\boldsymbol{X})$ does not diminish u_N*

Hence, for constraints that are both contractible and extensible the variables $\boldsymbol{X}$ and N are essentially disconnected: no filtering of one can affect the other. Among constraints that have this pair of properties are Sequence, SlidingSum, Precedence and Contiguity. For these constraints an open version of the constraint can be implemented essentially by adding to the implementation of the closed version the ability to dynamically add variables. For these constraints there is no difference between the model of open constraints used here and Barták's model.

5 Approximating Constraints

Following a proposal of Barták [1], we can implement an uncontractible open constraint $C(\boldsymbol{X})$ by executing a safe contractible approximation C_{app} of C until $\boldsymbol{X}$ is closed, and then replacing C_{app} by C for the remainder of the execution. To employ this approach we need to identify a contractible language containing the language of C, and a propagator C_{app} that implements it. By Proposition 1, the tightest contractible approximation of a constraint is its prefix-closure.

The prefix-closure $P(L)$ of a language L often appears to be simpler than L. For example, if L_1 is $\{a^{n^2} \mid n \in \mathbb{N}\}$ then $P(L_1)$ is a^*. But in general the prefix-closure is no simpler than the original language. For example, if L_2 is $\{a^{n^2}b \mid n \in \mathbb{N}\}$ then $P(L_2)$ is $a^* \cup L_2$. In some cases it is easy to represent $P(L)$ when given a representation of L. In particular, when L is defined by a finite automaton the automaton accepting $P(L)$ is easily computed.

Proposition 5. *Let $\mathcal{A}$ be a (possibly nondeterministic) finite state automaton, and let $\mathcal{A}'$ be the finite state automaton obtained from $\mathcal{A}$ by making final all states on a path from the start state to a final state. Then $L(\mathcal{A}') = P(L(\mathcal{A}))$. $\mathcal{A}'$ can be computed in linear time.*

Similarly, we can use the structure of a context-free grammar to construct a grammar for its prefix-closure.

Proposition 6. *Given a context-free grammar $\mathcal{G}$ defining a language L, a context-free grammar $\mathcal{G}'$ for $P(L)$ can be generated in quadratic time, linear time if $\mathcal{G}$ is in Chomsky normal form.*

As a corollary to Proposition 5, we can check in linear time whether a language defined by a deterministic finite automaton is prefix-closed: we simply check whether the construction of $\mathcal{A}'$ in Proposition 5 made any new final states. This improves on a result of [3]. Unfortunately, recognising when a language defined by a nondeterministic finite automaton $\mathcal{A}$ is prefix-closed is not so simple; $\mathcal{A}$ need not have the property that all states on a path from start to final state are final. It is shown in [3] that this problem is PSPACE-complete. The problem is undecidable for languages defined by context-free grammars [3]. However, the decision problem is much less important than the ability to construct (the representation of) the prefix-closure, so these negative results are not significant.

Thus, the tightest contractible approximation of $\textsc{Regular}(\mathcal{A}, \boldsymbol{X}, N)$ is implemented by $\textsc{Regular}(\mathcal{A}', \boldsymbol{X}, N)$, and the tightest contractible approximation of $\mathrm{CFG}(\mathcal{G}, \boldsymbol{X}, N)$ is implemented by $\mathrm{CFG}(\mathcal{G}', \boldsymbol{X}, N)$. The constraint GCC is also easily approximated.

Proposition 7. *The tightest contractible approximation of the constraint* $\mathrm{GCC}(\boldsymbol{v}, \boldsymbol{l}, \boldsymbol{u}, \boldsymbol{X}, N)$ *is implemented by* $\mathrm{GCC}(\boldsymbol{v}, \mathbf{0}, \boldsymbol{u}, \boldsymbol{X}, N)$*, obtained by ignoring the lower bounds $\boldsymbol{l}$.*

With these approximations, we can provide propagators for these open constraints within Barták's model. Furthermore, the change from approximation to original propagator when the constraint is closed is particularly simple, with only small changes to the run-time structure.

These propagators all achieve open D-consistency. This is a consequence of Barták's proposal, the tight approximations, and the power of the closed propagators to maintain domain consistency. Under Barták's proposal, a closed propagator for C_{app} is dynamised to handle extensions of the sequence of variables (possibly through his generic dynamisation). This propagator is then executed until the sequence of variables is closed, at which point the propagator is replaced by a closed propagator for C.

Proposition 8. *Let C_{app} be the tightest contractible approximation to C, and suppose we have closed propagators for C_{app} and C that maintain domain consistency wrt $\boldsymbol{X}$. Then Barták's proposal maintains open D-consistency for C.*

6 Propagators for Open Constraints

While Barták's proposal successfully provides propagators under his model of open constraints, it is insufficient to achieve open domain consistency in the model discussed here. The prefix-closure approximation encodes the fact that,

under Barták's model, the ultimate length of the sequence is totally unknown, and remains so until the constraint is closed. But when we have dynamically changing knowledge about the length of the sequence, as in the model proposed here, we can obtain stronger propagation.

Example 1. Consider a constraint C with language L described by $abc + (cab)^*$. In the prefix-closure, C_{app}, $X_2 = a$ is always a possibility, independent of the length of the sequence. On the other hand, if we know $N < 5$ then we can infer $X_2 = b$.

Essentially, we can dynamically update the approximation as the length N is constrained. In the following sections we will describe adaptations of closed propagators that obtain open domain consistency.

When adapting a propagator for maintaining consistency for a closed constraint $C(\boldsymbol{X})$ to the corresponding open constraint $C(\boldsymbol{X}, N)$, we have the following extra events to consider

- a variable is appended to $\boldsymbol{X}$
- the domain (or the bounds) of N is reduced
- a reduction in $D(\boldsymbol{X})$ that necessitates a reduction of $D(N)$
- the constraint (or the sequence $\boldsymbol{X}$) is closed

7 The Global Cardinality Constraint

The GCC constraint is defined as follows

$$\mathrm{GCC}_T(\boldsymbol{X}, \boldsymbol{v}, \boldsymbol{l}, \boldsymbol{u}, N) = \{d_1 \dots d_n \mid d_i \in T, \text{ for } i = 1, \dots, n, \ n \geq 0 \\ \forall d \in Dom \ l_d \leq |\{i \mid d_i = d\}| \leq u_d, \}$$

where $\boldsymbol{v}$ is a list of values, $\boldsymbol{l}$ and $\boldsymbol{u}$ are corresponding lists of lower and upper bounds on the number of occurrences of each value, Dom is the set of values in $\boldsymbol{v}$, and T is the type of a use of the constraint. When the type is not significant we will leave it implicit.

As discussed in Section 2, we assume that there is a type T associated with each use of GCC and all variables appearing in this use are restricted to values of T. The importance of this assumption is that we know whether there may be values, other than those with bounded number of occurrences, that appear in the sequence (i.e. whether $Dom \subset T$). Usually [13,8] it is assumed that the domains of all variables that might appear in the sequence are known. Hence this information, which otherwise is not available in an open world, is already available in a closed world.

The GCC constraint will behave differently, depending on whether $Dom = T$ or not. If $Dom \subset T$ then variables may take values that are not in $\boldsymbol{v}$, that is, values whose number of occurrences is not directly constrained by the constraint. In this case, the constraint does not imply an upper bound on N and $\boldsymbol{X}$ may grow arbitrarily large. If $Dom = T$ then N is bounded above by $\sum_{d \in Dom} u_d$. In both cases, N is bounded below by $\sum_{d \in Dom} l_d$. In the former case GCC

is extensible, but in the latter case the constraint is effectively extensible until $N = \sum_{d \in Dom} u_d$.

The implementation of GCC is by network flow techniques, following [13,8]. Given a domain $D(X_i)$ for each variable X_i added to the sequence so far ($i = 1, \ldots, k$) and bounds L and U on the variable N, there is an associated network flow graph $\mathcal{G}$ defined as follows. There is a source node s, a sink node t, a node for each $d \in Dom$, a node for each variable X_i in $\boldsymbol{X}$, a node R_T representing the values of $T \backslash Dom$, and a node R_V representing all the variables that might be added to $\boldsymbol{X}$. It contains:

- arcs (s, X) with lower and upper bound 1, for each $X \in \boldsymbol{X}$
- arcs (X, d) with lower bound 0 and upper bound 1 for each $X \in \boldsymbol{X}$ and $d \in Dom \cap D(X)$
- arcs (d, t) with lower bound l_d and upper bound u_d for each $d \in Dom$
- an arc (s, R_V) with lower bound $\max\{0, L - k\}$ and upper bound $U - k$
- arcs (R_V, d) with lower bound 0 and no upper bound for each $d \in Dom$
- an arc (X, R_T) with lower bound 0 and upper bound 1 for each $X \in \boldsymbol{X}$ where $D(X) \backslash Dom \neq \emptyset$
- an arc (R_V, R_T) with no bounds
- an arc (R_T, t) with no bounds if $Dom \subset T$, and upper bound 0 if $Dom = T$

The first three kinds of arcs are essentially the same as the closed implementation of GCC [13]. The node R_V corresponds to the amalgamation of the variables X in [8] whose arcs (s, X) have lower bound 0.

If $Dom = T$ then the node R_T is superfluous and could be deleted, along with all related arcs. We choose instead to simply put an upper bound of 0 on the arc from R_T to t, which has the same effect since this is the only outbound arc from R_T.

Feasible integer flows in $\mathcal{G}$ and valuations of the GCC constraint *correspond* in the following sense. For each flow f in $\mathcal{G}$ there are corresponding valuations which map X_i to d iff $f(X_i, d) = 1$ and map N to $k + f(s, R_V)$. If $f(X_i, R_T) = 1$ then X_i is mapped to an element of $T \backslash Dom$. For each solution σ of $\mathrm{GCC}(\boldsymbol{XY}, \boldsymbol{v}, \boldsymbol{l}, \boldsymbol{u}, N) \wedge D(\boldsymbol{X}) \wedge L \leq N \leq U$, we define an integer flow f in $\mathcal{G}$ as follows: $f(s, X_i) = 1$ for each $X_i \in \boldsymbol{X}$; $f(X_i, d) = 1$ iff $\sigma(X_i) = d$; $f(X_i, R_T) = 1$ iff $\sigma(X_i) \in T \backslash Dom$; $f(s, R_V) = \sigma(N) - k$; $f(R_V, d) = c$ iff exactly c of the variables $\boldsymbol{Y}$ are mapped to d by σ, for each $d \in Dom$; and $f(d, t) = c$ iff exactly c of the variables $\boldsymbol{XY}$ are mapped to d by σ, for each $d \in Dom$.

By design there is the following close relationship between feasible integer flows of $\mathcal{G}$ and solutions of $\mathrm{GCC}(\boldsymbol{X}, \boldsymbol{v}, \boldsymbol{l}, \boldsymbol{u}, N) \wedge D(\boldsymbol{X}) \wedge L \leq N \leq U$.

Proposition 9. *Consider the constraint* $\mathrm{GCC}(\boldsymbol{X}, \boldsymbol{v}, \boldsymbol{l}, \boldsymbol{u}, N)$ *with domain* D *for* $\boldsymbol{X}$ *and bounds* L *and* U *on* N.

For each feasible integer flow in $\mathcal{G}$ *there is an extension* $\boldsymbol{Y}$ *to* $\boldsymbol{X}$ *such that the valuation corresponding to the flow can be extended to a solution of* $\mathrm{GCC}(\boldsymbol{XY}, \boldsymbol{v}, \boldsymbol{l}, \boldsymbol{u}, N) \wedge D(\boldsymbol{X}) \wedge L \leq N \leq U$.

For each extension $\boldsymbol{Y}$ *to* $\boldsymbol{X}$ *and solution to* $\mathrm{GCC}(\boldsymbol{XY}, \boldsymbol{v}, \boldsymbol{l}, \boldsymbol{u}, N) \wedge D(\boldsymbol{X}) \wedge L \leq N \leq U$ *there is a corresponding integer feasible flow in* $\mathcal{G}$.

Using this relationship we can characterize the open domain consistency of GCC, in preparation for describing a domain consistent propagator for GCC.

Proposition 10. *Consider the constraint* $\text{GCC}(\boldsymbol{X}, \boldsymbol{v}, \boldsymbol{l}, u, N)$ *with domain* D *for* $\boldsymbol{X}$ *and* N*. Let* $[l_N, u_N]$ *be the tightest bounding interval for* $D(N)$*. Consider the flow network* $\mathcal{G}$ *based on* D *and* $[l_N, u_N]$*.*

D is open domain consistent with the constraint if and only if

- *for every* $X \in \boldsymbol{X}$ *and* $d \in D(X)$ *some feasible integer flow in* $\mathcal{G}$ *has flow along the arc* (X, d)*,*
- *some feasible integer flow in* $\mathcal{G}$ *has a total flow of* l_N*, and*
- $u_N \leq u_T + \sum_{d \in Dom} u_d$*, where* u_T *is the upper bound on* (R_T, t)*, which is either 0 or* ∞

This result follows from Proposition 9 and a couple of extra points. By construction of $\mathcal{G}$, any feasible flow has total flow between l_N and u_N. Given a minimal feasible integer flow, any greater flow – up to $min(u_N, u_T + \sum_{d \in Dom} u_d)$ – can be obtained by adding flows along (s, R_V), (R_V, d) and (d, t) to the extent permitted by each u_d, and (R_V, R_T) and (R_T, t) if that is possible. Thus, if there is support for $N = l_N$ then there is support for all greater values up to u_N, assuming $u_N \leq u_T + \sum_{d \in Dom} u_d$.

Domains of variables are maintained by the standard techniques [13]. If a variable Y is added, we add a node for Y with arcs (s, Y) with bounds $[1, 1]$, (Y, d) with bounds $[0, 1]$ for each $d \in D(Y)$, and decrement both bounds on the arc (s, R_V). If the bounds on N are changed, we change the bounds on (s, R_V); deletions of values from within the interior of $D(N)$ require no adjustment. Since GCC is effectively extensible, propagation does not affect u_N. After propagation on $\boldsymbol{X}$ we must check the minimum feasible integer flow and if it has increased, increase l_N to the same value (and update the lower bound on (s, R_V)). Finally, when the constraint is closed we assign $l_N = u_N = |\boldsymbol{X}|$.

8 The Regular Constraint

The Regular constraint is defined as follows

$$\text{Regular}(\mathcal{A}, \boldsymbol{X}, N) = \{d_1 \dots d_n \mid d_1 \dots d_n \text{ is accepted by } \mathcal{A}, n \geq 0\}$$

For a language constraint C we can use the domain of N, which we assume to be fnite, to define the sublanguage that the constraint is restricted to by N. Let Len_N denote the set of words with length in $D(N)$. Then, given the domain of N, the language defined by C is $L_C \cap Len_N$.

For the Regular constraint, where the language is defined by an automaton $\mathcal{A}$, we can construct an automaton $\mathcal{A}_N$ accepting $L_C \cap Len_N$ by standard techniques. This would allow implementation techniques for closed Regular to be easily adapted for open Regular. In particular, the approach of [10] which essentially unfolds the automaton is easily adapted. That approach produces a

layered automaton where each layer is essentially a copy of the original automaton, except that transitions lead to the next layer. Thus, for each state q of $\mathcal{A}$ and layer i, there is a state q^i in $\mathcal{A}_N$, and for each transition $\delta(q, a) = p$ we have $\delta(q^i, a) = p^{i+1}$, for each i. q^i is a final state in $\mathcal{A}_N$ iff q is a final state in $\mathcal{A}$ and $i = |\boldsymbol{X}|$. The filtering algorithm of [10] deletes values from the domains of variables, deletes transitions and/or deletes states to ensure the following invariants: $d \in D(X_i)$ iff there is a transition to layer i by the value d, and every transition and state is on a path from the start state to a final state.

The implementation of open $\textsc{Regular}(\mathcal{A}, \boldsymbol{X}, N)$ is essentially the same as that of closed $\textsc{Regular}(\mathcal{A}_N, \boldsymbol{X})$, with some extra filtering steps. The unfolded automaton is a variation of the unfolding of $\mathcal{A}$. Let k be the length of $\boldsymbol{X}$. Rather than unfolding k times we must unfold u_N times. We must have $k \leq l_N$. The final states of the resulting automaton are the states in layers $i \in dom(N)$ that correspond to a final state of $\mathcal{A}$.

It remains to describe the extra filtering steps required. When there are no more final states in layer i, we eliminate i from the domain of N. If i is deleted from the domain of N by some other constraint then the states in layer i are updated to no longer be final. When a variable Y is added to $\boldsymbol{X}$, we eliminate k from $D(N)$ and delete transitions to layer $k + 1$ that are not compatible with $D(Y)$. The closing of $\boldsymbol{X}$ can be expressed by setting N to the current length of $\boldsymbol{X}$. In each of these cases, further filtering might be needed, to restore the invariant.

Proposition 11. *The propagator of* $\textsc{Regular}(\mathcal{A}, \boldsymbol{X}, N)$ *described above maintains open domain consistency.*

However, the size of the automaton used in the above implementation may be $O(|\mathcal{A}|u_N)$. In many cases the initial value of u_N may be extremely large, so this might present an unacceptable up-front cost. An alternative is to unfold only l_N times, and then to unfold further when l_N is increased. In this case we will not be able to achieve domain consistency for N, so we limit our attention to lower bound consistency.

Using a breadth-first search on $\mathcal{A}$ we find the shortest distance from each state q to a final state, denoted by $d(q, F)$. A state q^i in the unfolded automaton is final iff $i = l_N$ and $l_N + d(q, F) \leq u_N$. These are the states q^{l_N} that participate in an accepting run with a length between l_N and u_N. Now, when the lower bound on N is increased further unfolding must be performed, the old final states are made non-final and new final states are computed for the (new) layer l_N. When the upper bound on N is decreased, the final states must be updated. When a variable Y is appended to $\boldsymbol{X}$, if $k = l_N$ then l_N is incremented, the automaton unfolded, and new final states computed; in either case, the transitions to layer $k + 1$ incompatible with $D(Y)$ are deleted. When the constraint is closed, it is unsatisfiable if $k < l_N$; otherwise we assign $N = k$.

We also keep track of the set of states q^{l_N} such that q is final in $\mathcal{A}$. If this set becomes empty then no string of length l_N can be accepted. Let $m = \min\{d(q, F) \mid q^{l_N}$ exists and is a final state in the modified $\mathcal{A}_N\}$. Then l_N

is increased by m and m layers are added to the automaton. The old final states are made non-final and new final states are computed for the (new) layer l_N.

Proposition 12. *The propagator of* REGULAR$(\mathcal{A}, \boldsymbol{X}, N)$ *described above maintains open DL-consistency.*

When the REGULAR constraint is contractible (that is, $\mathcal{A}$ defines a prefix-closed language) then we can simplify the propagator above. Using the transformation described in Proposition 5, we can ensure that all states that lie on a path between start and final state are final. Other states can be ignored. Consequently, $d(q, F) = 0$ for every state q. If u_N is changed then no propagation is needed, by Proposition 4.

When the REGULAR constraint is also extensible (that is, $\mathcal{A}$ defines a continual language) then the propagator is as good as we can hope for. By Proposition 4, changes in l_N and $D(\boldsymbol{X})$ do not affect u_N. As remarked earlier, $\boldsymbol{X}$ and N are decoupled, and hence the above propagator maintains open domain consistency.

Proposition 13. *If the language of $\mathcal{A}$ is prefix-closed and continual at threshold m then the simplified propagator for* REGULAR$(\mathcal{A}, \boldsymbol{X}, N)$ *described above maintains open domain consistency when $|\boldsymbol{X}| > m$.*

9 The CFG Constraint

The CFG constraint is defined as follows

$$\text{CFG}(\mathcal{G}, \boldsymbol{X}, N) = \{d_1 \dots d_n \mid d_1 \dots d_n \text{ is generated by } \mathcal{G}, n \geq 0\}$$

A propagator for the closed form of this constraint, based on the CYK parser for grammars in Chomsky normal form, has been proposed [11,15][2]. That propagator creates a table that holds, for each substring of $\boldsymbol{X}$, the non-terminals that can generate the substring under the restrictions imposed by $D(\boldsymbol{X})$. Then, in another pass over the table, those non-terminals for substrings that are compatible with a parse of $\boldsymbol{X}$ are selected, starting with selecting S for the entire string. The set of terminal symbols a of productions $A \rightarrow a$ from selected non-terminals A for substrings of length 1 starting at i, form the new domain of X_i.

This propagator is easily adapted to an open version of the constraint. The table must consider strings of length up to u_N. If, after the table is created, a substring from position 1 to n where $n \in D(N)$, cannot be generated from the starting non-terminal S then n is deleted from $D(N)$. The second pass over the table is essentially the same as for the closed constraint: the only difference is that the starting point is the selection of S for every substring from 1 to n where $n \in D(N)$.

Proposition 14. *The propagator of* CFG$(\mathcal{G}, \boldsymbol{X}, N)$ *described above maintains open domain consistency.*

[2] [11] also proposes a propagator based on Earley's parser. We will not address that here.

10 Conclusions

We have introduced a model of open constraints that incorporates the length of the sequence of variables. With three case studies we have seen that propagators for closed constraints can be adapted to open constraints in this model. We have also explored the consequences of two properties of constraints that are likely to be important to open constraint programming, no matter which model is used. Along the way, we have obtained results on Barták's model of open constraints.

Acknowledgements. Thanks to the referees for their comments and to Andreas Bauer, Christian Dax, Boi Faltings, Rob van Glabeek, Sebastien Maneth and Toby Walsh for discussions related to this paper.

References

1. Barták, R.: Dynamic Global Constraints in Backtracking Based Environments. Annals of Operations Research 118, 101–119 (2003)
2. Beldiceanu, N., Carlsson, M., Rampon, J.-X.: Global Constraint Catalog, SICS Technical Report T2005:08, http://www.emn.fr/x-info/sdemasse/gccat/
3. Brzozowski, J., Shallit, J., Xu, Z.: Decision Problems for Convex Languages. In: Dediu, A.H., Ionescu, A.M., Martín-Vide, C. (eds.) LATA 2009. LNCS, vol. 5457, pp. 247–258. Springer, Heidelberg (2009)
4. Dechter, A., Dechter, R.: Belief Maintenance in dynamic constraint networks. In: AAAI 1988, pp. 37–42 (1988)
5. Dechter, R.: Constraint Processing. Morgan Kaufmann, San Francisco (2003)
6. Faltings, B., Macho-Gonzalez, S.: Open Constraint Programming. Artificial Intelligence 161(1-2), 181–208 (2005)
7. Gervet, C.: Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. Constraints 1(3), 191–244 (1997)
8. van Hoeve, W.-J., Régin, J.-C.: Open Constraints in a Closed World. In: Beck, J.C., Smith, B.M. (eds.) CPAIOR 2006. LNCS, vol. 3990, pp. 244–257. Springer, Heidelberg (2006)
9. Maher, M.J.: Open Contractible Global Constraints, IJCAI (2009)
10. 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)
11. Quimper, C.-G., Walsh, T.: Global Grammar Constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 751–755. Springer, Heidelberg (2006)
12. Régin, J.-C.: A filtering algorithm for constraints of difference in CSPs. In: AAAI 1994, pp. 362–367 (1994)
13. Régin, J.-C.: Generalized Arc Consistency for Global Cardinality Constraint. In: AAAI 1996, pp. 209–215 (1996)
14. Rossi, F., van Beek, P., Walsh, T. (eds.): Handbook of Constraint Programming. Elsevier, Amsterdam (2006)
15. Sellmann, M.: The Theory of Grammar Constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 530–544. Springer, Heidelberg (2006)

Sequencing and Counting with the `multicost-regular` Constraint

Julien Menana and Sophie Demassey

École des Mines de Nantes, LINA CNRS UMR 6241, F-44307 Nantes, France
{julien.menana,sophie.demassey}@emn.fr

Abstract. This paper introduces a global constraint encapsulating a `regular` constraint together with several cumulative costs. It is motivated in the context of personnel scheduling problems, where a schedule meets patterns and occurrence requirements which are intricately bound. The optimization problem underlying the `multicost-regular` constraint is NP-hard but it admits an efficient Lagrangian relaxation. Hence, we propose a filtering based on this relaxation. The expressiveness and the efficiency of this new constraint is experimented on personnel scheduling benchmark instances with standard work regulations. The comparative empirical results show how `multicost-regular` can significantly outperform a decomposed model with `regular` and `global-cardinality` constraints.

1 Introduction

Many combinatorial decision problems involve the simultaneous action of sequencing and counting objects, especially in the large class of routing and scheduling problems. In routing, a vehicle visits a sequence of locations following a path in the road network according to some numerical requirements on the whole travelling distance, the time spent, or the vehicle capacity. If only one numerical attribute is specified, finding a route is to solve a shortest/longest path problem. For several attributes, the problem – a Resource Constrained Shortest/Longest Path Problem (RCSPP) – becomes NP-hard. All these numerical requirements may drastically restrict the set of paths in the network which correspond to the actual valid routes. Hence, it is much more efficient to take these requirements into account throughout the search of a path, rather than each separately. Personnel scheduling problems can be treated analogously. Planning a worker schedule is to sequence activities (or shifts) over a time horizon according to many various work regulations, as for example: "a working night is followed by a free morning", "a night shift costs twice as musch as a day shift", "at least 10 days off a month", etc. Hence, a schedule meets both structural requirements – defined as allowed patterns of activities – and numerical requirements – defined as assignment costs or counters – which are intricately bounds. Modelling these requirements individually is itself a hard task, for which the expressiveness and the flexibility of Constraint Programming (CP) is recognized. Modelling these requirements efficiently is still a harder task as it means to aggregate all

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 178–192, 2009.

of them in order to process this set of tied requirements as a whole. By introducing the regular global constraint, Pesant [1] has proposed an elegant and efficient way to model and to enforce all the pattern requirements together. The allowed patterns are gathered in an acyclic digraph whose paths coincide with the valid sequences of activities. This approach was later extended to optimization constraints soft-regular [2] and cost-regular [3] for enforcing bounds on the global cost – a violation cost or any financial cost – of the sequence of assignments. The underlying problem is now to compute shortest and longest paths in the acyclic graph of patterns. The cost-regular constraint was successfully applied to solve real-world personnel scheduling problems under a CP-based column-generation approach [3]. Nevertheless, the authors complained about the weak interaction in their CP model between the cost-regular constraint and an external global-cardinality used for modelling occurrence requirements. Actually, with such a decomposition, the support graph of cost-regular maintains many paths which do not satisfy the cardinality constraints. In this paper, we still generalize this approach for handling several cost attributes within one global constraint multicost-regular. Such a constraint allows to reason simultaneously on the sequencing and counting requirements occurring in personnel scheduling problems. As mentioned above, the underlying optimization problem is a RCSPP and it remains NP-hard even when the graph is acyclic. Hence, the filtering algorithm we present achieves a relaxed level of consistency. It is based on the Lagrangian relaxation of the RCSPP following the principle by Sellmann [4] for Lagrangian relaxation-based filtering. Our implementation of multicost-regular is available in the distribution of the open-source CP solver *CHOCO* [1].

The paper is organized as follows. In Section 2, we present the class of regular constraints and provide a theoretical comparison between the path-finding approach of Pesant [1] and the decomposition-based approach of Beldiceanu et al. [5]. We introduce then the new constraint multicost-regular. In Section 3, we introduce the Lagrangian relaxation-based filtering algorithm. In Section 4, we describe a variety of standard work regulations and investigate a systematic way of building one instance of multicost-regular from a set of requirements. In Section 5, comparative empirical results on benchmark instances of personnel scheduling problems are given. They show how multicost-regular can significantly outperform a decomposed model with regular and globalcardinality constraints.

2 Regular Language Membership Constraints

In this section, we recall the definition of the regular constraint and report on related work, before introducing multicost-regular. First, we recall basic notions of automata theory and introduce notations used throughout this paper:

We consider a non empty set Σ called the *alphabet*. Elements of Σ are called *symbols*, sequences of symbols are called *words*, and sets of words are called

[1] http://choco.emn.fr/

languages over Σ. An *automaton* Π is a directed multigraph (Q, Δ) whose arcs are labelled by the symbols of an alphabet Σ, and where two non-empty subsets of vertices I and A are distinguished. The set Q of vertices is called the set of *states* of Π, I is the set of *initial states*, and A is the set of *accepting states*. The non-empty set $\Delta \subseteq Q \times \Sigma \times Q$ of arcs is called the set of *transitions* of Π. A word in Σ is said to be *accepted* by Π if it is the sequence of the arc labels of a path from an initial state to an accepting state in Π. Automaton Π is a *deterministic finite automaton (DFA)* if Δ is finite and if it has only one initial state ($I = \{s\}$) and no two transitions sharing the same initial extremity and the same label. The language accepted by a FA is a *regular language*.

2.1 Path-Finding and Decomposition: Two Approaches for `regular`

The `regular` language membership constraint was introduced by Pesant in [1]. Given a sequence $X = (x_1, x_2, ..., x_n)$ of finite domain variables and a deterministic finite automaton $\Pi = (Q, \Sigma, \Delta, \{s\}, A)$, the constraint `regular`$(X, \Pi)$ holds iff X is a word of length n over Σ accepted by DFA Π. By definition, the solutions of `regular`(X, Π) are in one-to-one correspondance with the paths of exactly n arcs connecting s to a vertex in A in the directed multigraph Π. Let $\delta_i \in \Delta$ denote the set of transitions that appears as the i-th arc of such a path, then a value for x_i is consistent iff δ_i contains a transition labelled by this value.

Coincidently, Pesant [1] and Beldiceanu et al [5] introduce two orthogonal approaches to achieve GAC on `regular` (see Figure 1). The approach proposed by Pesant [1] is to unfold Π as an acyclic DFA Π_n which accepts only the words of length n. By construction, Π_n is a layered multigraph with state s in layer 0 (the source), the accepting states A in layer n (the sinks), and where the set of arcs in any layer i coincides with δ_i. A breadth-first search allows to maintain the coherence between Π_n and the variable domains by pruning the arcs in δ_i whose labels are not in the domain of x_i, then by pruning the vertices and arcs which are not connected to a source and to a sink. In Beldiceanu et al [5], a `regular` is decomposed as n tuple constraints for modelling the sets $\delta_1, \delta_2, \ldots, \delta_n$. The decomposition introduces state variables $q_0 \in \{s\}$, $q_1, \ldots, q_{n-1} \in Q$, $q_n \in A$ and

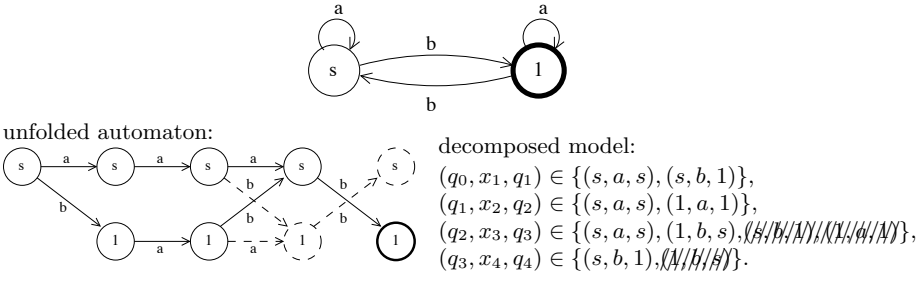


Fig. 1. Consider the DFA depicted above applied to $X \in \{a, b\} \times \{a\} \times \{a, b\} \times \{b\}$. The unfolded automaton of `regular` is depicted on the left and the decomposed model on the right. The dashed transitions are discarded in both models.

uses triplet relations defined in extension to enforce GAC on the transition constraints $(q_{i-1}, x_i, q_i) \in \delta_i$. Such a constraint network being Berge-acyclic, enforcing AC on the decomposition achieves GAC on regular.

In the first approach, a specialized algorithm is defined to maintain all the support paths, while in the second approach, the transitions are modeled with tuple constraints which are directly propagated by the CP solver. The two approaches are orthogonal. Actually, the second model may mimic the specialized algorithm depending on the chosen propagation.

If we assume w.l.o.g. that Σ is the union of the variable domains, then the initial run of Pesant's algorithm for the construction of Π_n is performed in $O(n|\Delta|)$ time and space (with $\Delta \leq |Q||\Sigma|$ if Π is a DFA). Incremental filtering is performed with the same worst-case complexity with a forward/backward traversal of Π_n. Actually, the complexity of the algorithm relies more on the size $|\Delta_n|$ of the unfolded automaton Π_n rather than on the size $|\Delta|$ of the specified automaton Π. Note for instance that when the specified automaton Π accepts only words of length n then it is already unfolded ($\Pi = \Pi_n$) and the first run of the algorithm is in $O(|\Delta|)$. In practice, as in our experiments (Section 5), Π_n can even be much smaller than Π, meaning that many accepting states in Π cannot be reached in exactly n transitions. The incremental filtering is performed in $O(|\Delta_n|)$ time with, in such a case, $|\Delta_n| \ll n|\Delta|$.

regular is a very expressive constraint. It is useful to model pattern constraints arising in many planning problems, but also to reformulate other global constraints [5] or to model tuples defined in extension. An other application of regular is to model a sliding constraint: recently, Bessière et al. [6] have introduced the slide meta constraint. In its more general form, slide takes as arguments a matrix of variables Y of size $n \times p$ and a constraint C of arity pk with $k \leq n$. $\texttt{slide}(Y, C)$ holds if and only of $C(y^1_{i+1}, \ldots, y^p_{i+1}, \ldots, y^1_{i+k}, \ldots, y^p_{i+k})$ holds for $0 \leq i \leq n-k$. Using the decomposition proposed in [5], $\texttt{regular}(X, \Pi)$ can be reformulated as $\texttt{slide}([Q, X], C_\Delta)$, where Q is the sequence of state variables and C_Δ is the transition constraint $C_\Delta(q, x, q', x') \equiv (q, x, q') \in \Delta$. Conversely [6], a slide constraint can be reformulated as a regular but it may require to enumerate all valid tuples for C. This reformulation can however be useful in the context of planning (especially for car sequencing) to model a sliding cardinality constraint also known as sequence. Even if powerful specialized algorithms exist for this constraint (see e.g. [7]), the automaton resulting from the reformulation can be integrated with other pattern requirements as we will show in Section 4. Finally, one should notice the work (see e.g. [8]) related to context-free grammar constraints. Though, most of the rules encountered in personnel scheduling can be described using regular languages.

2.2 Maintaining Patterns with Cumulative Costs and Cardinalities

Personnel scheduling problems are usually defined as optimization problems. Most often, the criterion to optimize is a *cumulative cost*, i.e. the sum of costs

associated to each assignment of a worker to a given activity at a given time. Such a cost has several meanings: it can model a financial cost, a preference, or a value occurrence. Now, designing a valid schedule for one worker is to enforce the sequence of assignments to comply with a given pattern while ensuring that the total cost of the assignments is bounded. This can be specified by means of a `cost-regular` constraint [3]. Given $c = (c_{ia})_{i\in[1..n]\times a\in\Sigma}$ a matrix of real assignment costs and $z \in [\underline{z}, \overline{z}]$ a bounded variable ($\underline{z}, \overline{z} \in \mathbb{R}$), `cost-regular`$(X, z, \Pi, c)$ holds iff `regular`(X, Π) holds and $\sum_{i=1}^{n} c_{ix_i} = z$. Note that it has the `knapsack` constraint [9] as a special case and that, unless $P = NP$, one can enforce GAC on a `knapsack` constraint at best in pseudo-polynomial time, i.e. the run time is polynomial in the values of the bounds of z. As a consequence, enforcing GAC on `cost-regular` is NP-hard.

The definition of `cost-regular` reveals a natural decomposition as a `regular` constraint channeled to a `knapsack` constraint. Actually, it is equivalent to the decomposition proposed by Beldiceanu et al. [5] when dealing with one cumulative[2] cost: cost variables k_i are now associated to the previous state variables q_i, with $k_0 = 0$ and $k_n = z$, and several arithmetic and `element` constraints model the `knapsack` and channeling constraints. In short, this formulation can be rewritten as `slide`$([Q, X, K], C^c_\Delta)$, with $C^c_\Delta(q_{i-1}, x_{i-1}, k_{i-1}, q_i, x_i, k_i) \equiv (q_{i-1}, x_{i-1}, q_i) \in \Delta \wedge k_i = k_{i-1} + c_{ix_i}$. Depending on the size of the domains of the cost variables, GAC can be enforced on `knapsack` in reasonable time. However, even in this case, since the constraint hypergraph of the decomposed model is no longer Berge-acyclic but α-acyclic, one has to enforce pairwise-consistency on the shared variables – a pair (q_i, k_i) of state and cost variables – of the transition constraints in order to achieve GAC. A similar option proposed for `slide` [6] is to enforce AC on the dual encoding of the hypergraph of the C^c_Δ constraints, but again it requires to explicit all the support tuples and then, it may be of no practical use.

The filtering algorithm presented in [3] for `cost-regular` is a slight adaptation[3] of Pesant's algorithm for `regular`. It is based on the computation of shortest and longest paths in the unfolded graph Π_n valued by the transition costs. To each vertex (i, q) in any layer i of Π_n are associated two bounded cost variables k^-_{iq} and k^+_{iq} modelling the lengths of the paths respectively from layer 0 to (i, q) and from (i, q) to layer n. The cost variables can trivially be initialized during the construction of Π_n: k^-_{iq} in the forward phase and k^+_{iq} in the backward phase. The bounds of variable z are then pruned according to the condition $z \subseteq k^+_{0s}$. Conversely, an arc $((i-1, q), a, (i, q')) \in \delta_i$ can be removed whenever:

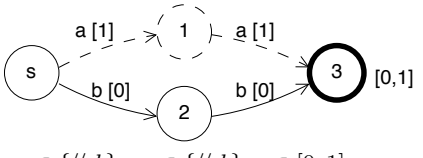
$$\underline{k^-_{(i-1)q}} + c_{ia} + \underline{k^+_{iq'}} > \overline{z} \quad \text{or} \quad \overline{k^-_{(i-1)q}} + c_{ia} + \overline{k^+_{iq'}} > \underline{z}.$$

[2] The model in [5] can deal not only with sum but also with various arithmetic functions on costs, but no example of use is provided.

[3] Previously, the algorithm was partially – for minimization only – applied to the special case `soft-regular[hamming]` in [5] and in [2].

As graph Π_n is acyclic, maintaining the cost variables, i.e. shortest and longest paths, can be performed by breadth-first traversal with the same time complexity $O(|\Delta_n|)$ than for maintaining the connexity of the graph in regular.

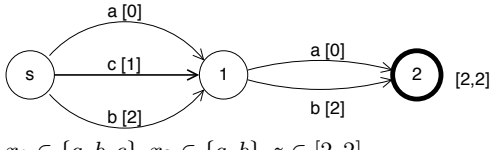
As said before, this algorithm achieves a hybrid level of consistency on cost-regular. As a matter of fact, it enforces a sort of pairwise-consistency on the decomposed model between each state variable and the bounds of the associated cost variable, according to the relation $q_i = (i, q) \iff k_i = k_{iq}^-$. Hence, it dominates the decomposed model knapsack∧regular when only Bound Consistency is enforced on the cost variables. Otherwise, if AC is enforced on knapsack then the two approaches are incomparable as show the two examples depicted in Figures 2 and 3.



$x_1 \in \{a, b\}, x_2 \in \{a, b\}, z \in [0, 1].$

$(q_0, x_1, q_1, j_1) \in \{(s, a, 1, 1), (s, b, 2, 2)\},$
$\texttt{element}(k_1, j_1, (k_0 + 1, k_0)),$
$(q_1, x_2, q_2, j_2) \in \{(1, a, 3, 1), (2, b, 3, 2)\},$
$\texttt{element}(z, j_2, (k_1 + 1, k_1)),$
$q_0 \in \{s\}, q_1 \in \{1, 2\}, q_2 \in \{3\},$
$k_0 \in \{0\}, k_1 \in \{0, 1\},$
$x_1 \in \{a, b\}, x_2 \in \{a, b\}, z \in \{0, 1\}.$

Fig. 2. Consider the depicted DFA with costs in brackets applied to $X = (x_1, x_2) \in \{a, b\} \times \{a, b\}$ and $z \in [0, 1]$. The cost-regular algorithm (on the left) discards the dashed transitions and hence achieves GAC. The decomposed model (on the right) is arc-consistent but not globally consistent.



$x_1 \in \{a, b, c\}, x_2 \in \{a, b\}, z \in [2, 2].$

$(x_1, j_1) \in \{(a, 1), (b, 3)\},$
$\texttt{element}(k_1, j_1, (k_0, k_0 + 2)),$
$(x_2, j_2) \in \{(a, 1), (b, 2)\},$
$\texttt{element}(z, j_2, (k_1, k_1 + 2)),$
$k_0 \in \{0\}, k_1 \in \{0, 2\},$
$x_1 \in \{a, b\}, x_2 \in \{a, b\}, z \in \{2\}.$

Fig. 3. Consider now the depicted DFA applied to $X = (x_1, x_2) \in \{a, b, c\} \times \{a, b\}$ and $z \in [2, 2]$. Enforcing AC on the decomposed model (on the right) achieves GAC. The cost-regular algorithm (on the left) does not achieve GAC since the minimum and maximum paths traversing arc $x_1 = c$ are consistent with the bounds on z.

2.3 The multicost-regular Constraint

A natural generalization of cost-regular is to handle several cumulative costs: given a vector $Z = (z^0, ..., z^R)$ of bounded variables and $c = (c_{ia}^r)_{i \in [1..n], a \in \Sigma}^{r \in [0..R]}$ a matrix of assignment costs, $\texttt{multicost-regular}(X, Z, \Pi, c)$ holds if and only if $\texttt{regular}(X, \Pi)$ holds and $\sum_{i=1}^{n} c_{ix_i}^r = z^r$ for all $0 \leq r \leq R$. Such a generalization has an important motivation in the context of personnel scheduling. Actually, apart a financial cost and pattern restrictions, an individual schedule is usually subject to a global-cardinality constraint bounding the number of

occurrences of each value in the sequence. These bounds can drastically restrict the language on which the schedule is defined. Hence, it could be convenient to tackle them within the `regular` constraint in order to reduce the support graph. As a generalization of `cost-regular` or of the `global-sequencing` constraint [10], we cannot hope to achieve GAC in polynomial time here. Note that the model by Beldiceanu et al [5] – and similarly the `slide` constraint – was also proposed for dealing with several costs but again, it amounts to decompose as a `regular` constraint channeled with one `knapsack` constraint for each cost.

Hence, we ought to exploit the structure of the support graph of Π_n to get a good relaxed propagation for `multicost-regular`. The optimization problems underlying `cost-regular` were shortest and longest path problems in Π_n. The optimization problems underlying `multicost-regular` are now the Resource Constrained Shortest and Longest Path Problems (RCSPP and RCLPP) in Π_n. The RCSPP (resp. RCLPP) is to find the shortest (resp. longest) path between a source and a sink in a valued directed graph, such that the quantities of resources accumulated on the arcs do not exceed some limits. Even with one resource on acyclic digraphs, this problem is known to be NP-hard[11]. Two approaches are most often used to solve RCSPP [11]: dynamic programming and Lagrangian relaxation. Dynamic programming-based methods extend the usual short path algorithms by recording the costs over every dimension at each node of the graph. As in `cost-regular`, this could easily be adapted for filtering by converting these cost labels as cost variables but it would make the algorithm memory expensive. Instead, we investigate a Lagrangian relaxation approach, which can also easily be adapted for filtering from the `cost-regular` algorithm without memory overhead.

3 A Lagrangian Relaxation-Based Filtering Algorithm

Sellmann [4] laid the foundation for using the Lagrangian relaxation of a linear program to provide a cost-based filtering for a minimization or maximization constraint. We apply this principle to the RCSPP/RCLPP for filtering `multicost-regular`. The resulting algorithm is a simple iterative scheme where filtering is performed by `cost-regular` on Π_n for different aggregated cost functions. In this section, we present the usual Lagrangian relaxation model for the RCSPP and explain how to solve it using a subgradient algorithm. Then, we show how to adapt it for filtering `multicost-regular`.

Lagrangian Relaxation for the RCSPP. Consider a directed graph $G = (V, E, c)$ with source s and sink t, and resources $(\mathcal{R}_1, ..., \mathcal{R}_R)$. For each resource $1 \leq r \leq R$, let $\overline{z_r}$ (resp. $\underline{z_r}$) denote the maximum (resp. minimum[4]) capacity of a path over the resource r, and c^r_{ij} denote the consumption of resource r on arc $(i, j) \in E$. A binary linear programming formulation for the RCSPP is as follows:

[4] In the original definition of RCSPP, there is no lower bound on the capacity: $\underline{z_r}$

$$\min \sum_{(i,j)\in E} c_{ij}x_{ij} \tag{1}$$

$$\text{s.t. } \underline{z^r} \leq \sum_{(i,j)\in E} c^r_{ij}x_{ij} \leq \overline{z^r} \qquad \forall r \in [1..R] \tag{2}$$

$$\sum_{j\in V} x_{ij} - \sum_{j\in V} x_{ji} = \begin{cases} 1 & \text{if } i = s, \\ -1 & \text{if } i = t, \\ 0 & \text{otherwise.} \end{cases} \qquad \forall i \in V \tag{3}$$

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

In this model, a binary decision variable x_{ij} defines whether arc (i,j) belongs to a solution path. Constraints (2) are the resource constraints and Constraints (3) are the usual path constraints.

Lagrangian relaxation consists in dropping "complicating constraints" and adding them to the objective function with a violation penalty cost $u \geq 0$, called the *Lagrangian multipliers*. The resulting program is called the Lagrangian subproblem with parameter u and it is a relaxation of the original problem. Solving the *Lagrangian dual* is to find the multipliers $u \geq 0$ which gives the best relaxation, i.e. the maximal lower bound.

The complicating constraints of the *RCSPP* are the $2R$ resource constraints (2). Indeed, relaxing these constraints leads to a shortest path problem, that can be solved in polynomial time. Let P denote the set of solutions $x \in \{0,1\}^E$ satisfying Constraints (3). P defines the set of paths from s to t in G. The Lagrangian subproblem with given multipliers $u = (u_-, u_+) \in \mathbb{R}^{2R}_+$ is:

$$SP(u): \quad f(u) = \min_{x\in P} cx + \sum_{r=1}^{R} u^r_+(c^r x - \overline{z^r}) - \sum_{r=1}^{R} u^r_-(c^r x - \underline{z^r}) \tag{5}$$

An optimal solution x^u for $SP(u)$ is then a shortest path in graph $G(u) = (V, E, c(u))$ where:

$$c(u) = c + \sum_{r=1}^{R}(u^r_+ - u^r_-)c^r, \kappa_u = \sum_{r=1}^{n}(u^r_-\underline{z^r} - u^r_+\overline{z^r}) \text{ and } f(u) = c(u)x^u + \kappa_u. \tag{6}$$

Solving the Lagrangian Dual. The Lagrangian dual problem is to find the best lower bound $f(u)$, i.e. to maximize the piecewise linear concave function f:

$$LD: \quad f_{LD} = \max_{u\in\mathbb{R}^{2R}_+} f(u) \tag{7}$$

Several algorithms exist to solve the Lagrangian dual. In our approach, we consider the subgradient algorithm [12] as it is rather easy to implement and it does not require the use of a linear solver. The subgradient algorithm iteratively solves one subproblem $SP(u)$ for different values of u. Starting from an arbitrary value, the position u is updated at each iteration by moving in the direction of a supergradient Γ of f with a given step length μ: $u^{p+1} = max\{u^p + \mu_p \Gamma(u^p), 0\}$. There

exist many ways to choose the step lengths for guaranteeing the convergence of the subgradient algorithm towards f_{LD} (see e.g. [13]). In our implementation, we use a standard step length $\mu_p = \mu_0 \epsilon^p$ with μ_0 and $\epsilon < 1$ "sufficiently" large (we have empirically fixed $\mu_0 = 10$ and $\epsilon = 0.8$). For the supergradient, solving $SP(u)$ returns an optimum solution $x^u \in P$ and $\Gamma(u)$ is computed as: $\Gamma(u) = ((c^r x^u - \overline{z^r})_{r\in[1..R]}, (\underline{z^r} - c^r x^u)_{r\in[1..R]})$.

From Lagrangian Relaxation to Filtering. The key idea of Lagrangian relaxation-based filtering, as stated in [4], is that if a value is proved to be inconsistent in at least one Lagrangian subproblem then it is inconsistent in the original problem:

Theorem 1. *(i) Let P be a minimization linear program with optimum value $f^* \leq +\infty$, $\overline{z} \leq +\infty$ be an upper bound for P, and $SP(u)$ be any Lagrangian subproblem of P, with optimum value $f(u)^* \leq +\infty$. If $f(u) > \overline{z}$ then $f^* > \overline{z}$.*

(ii) Let x be a variable of P and v a value in its domain. Consider $P_{x=v}$ (resp. $SP(u)_{x=v}$) the restriction of P (resp. $SP(u)$) to the set of solutions satisfying $x = v$ and let $f^*_{x=v} \leq +\infty$ (resp. $f(u)_{x=v} \leq +\infty$) its optimum value. If $f(u)_{x=v} > \overline{z}$ then $f^*_{x=v} > \overline{z}$.

Proof. Statement (i) of Theorem 1 is straightforward, since $SP(u)$ is a relaxation for P, then $f(u) \leq f^*$. Statement (ii) arises from (i) and from the fact that, adding a constraint $x = v$ within P and applying Lagrangian relaxation, or applying Lagrangian relaxation and then adding constraint $x = v$ to each subproblem, result in the same formulation.

The mapping between `multicost-regular`(X, Z, Π, c), with $|Z| = R + 1$ and an instance of the RCSPP (resp. RCLPP) is as follows: We single out one cost variable, for instance z^0, and create R resources, one for each other cost variable. The graph $G = (\Pi_n, c^0)$ is considered. A feasible solution of the RCSPP (resp. RCLPP) is a path in Π_n from the source (in layer 0) to a sink (in layer n) that consumes on each resource $1 \leq r \leq R$ is at least $\underline{z^r}$ and at most $\overline{z^r}$. Furthermore, we want to enforce an upper bound $\overline{z^0}$ on the minimal value for the RCSPP (resp. a lower bound $\underline{z^0}$ on the maximal value for the RCLPP). The arcs of Π_n are in one-to-one correspondance with the binary variables in the linear model of these two instances.

Consider a Lagrangian subproblem $SP(u)$ of the RCSPP instance (the approach is symmetric for the maximization instance of RCLPP) . We show that a slight modification of the `cost-regular` algorithm allows to solve $SP(u)$ but also to prune arcs of Π_n according to Theorem 1 and to shrink the lower bound $\underline{z^0}$. The algorithm starts by updating the costs on the graph Π_n with $c^0(u)$, as defined in (6) and then by computing, at each node (i, q), the shortest path $\underline{k^-_{iq}}$ from layer 0 and the shortest path $\underline{k^+_{iq}}$ to layer n. We get the optimum value $f(u) = \underline{k^+_{0s}} + \kappa_u$. As it is a lower bound for z^0, one can eventually update this lower bound as $\underline{z^0} = \max\{f(u), \underline{z^0}\}$. Then, by a traversal of Π_n, we remove each arc $((i-1, q), a, (i, q')) \in \delta_i$ such that $\underline{k^-_{(i-1)q}} + c^0(u)_{ia} + \underline{k^+_{iq'}} > \overline{z^0} - \kappa_u$.

The global filtering algorithm we developed for multicost-regular is as follows: starting from $u = 0$, a subgradient algorithm guides the choice of the Lagrangian subproblems to which the above cost-filtering algorithm is applied. The number of iterations for the subgradient algorithm is limited to 20 (it usually terminates far before). The subgradient algorithm is first applied to the minimization problem (RCSPP) then to the maximization problem (RCLPP). As a final step, we run the original cost-regular algorithm on each of the cost variables to shrink their bounds (by the way, it could deduce new arcs to filter, but it did not happen in our experiments). Note that due to the parameter dependancy of the subgradient algorithm, the propagation algorithm is not monotonic.

4 Modelling Personnel Scheduling Problems

In this section we show how to model standard work regulations arising in Personnel Scheduling Problems (PSP) as one instance of the multicost-regular constraint. The purpose is to emphasize the ease of modelling with such a constraint and also to derive a systematic way of modelling PSP.

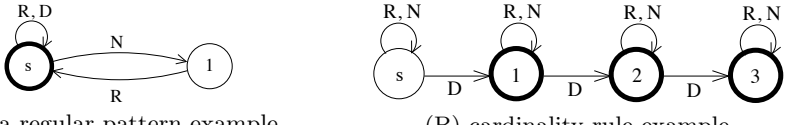
4.1 Standard Work Regulations

In PSP, many kinds of work regulations can be encountered, however, we can categorize most of them as rules enforcing either regular patterns, fixed cardinalities or sliding cardinalities.

To illustrate those categories, we consider a 7 days schedule and 3 activities: night shift (N), day shift (D) or rest shift (R). For example: R R D D N R D

Regular Patterns can be modelled directly as a DFA. For instance the rule "a night shift is followed by a rest" is depicted in Figure 4 (A). The rules can either be given as forbidden patterns or allowed patterns. In the first case, one just need to build the complement automaton.

Fixed Cardinality Rules bound the number of occurrences of an activity or a set of activities over a fixed subsequence of time slots. Such a rule can be modelled within an automaton or using counters. For example, the rule "at least 1 and at most 3 day shifts each week" can easily be modelled as the DFA depicted in Figure 4 (B). Taking a look at this automaton, we can see the initial state has been split into 3 different states that represent the maximum



(A) a regular pattern example. (B) cardinality rule example.

Fig. 4. Examples of automata representing work regulations

number of D transitions that can be taken. Such a formulation can be an issue when the maximum occurrence number increases. In this case, using a counter is more suitable, as we only need to create a new cost variable $z^r \in [1,3]$ with $c^r_{ij} = 1 \iff 1 \leq i \leq 7$ and $j = D$. More generally, one can also encounter cardinality rules over patterns. This also can be managed by means of a cost. One has to isolate the pattern within the automaton describing all the feasible schedules, then to price transitions entering it to 1.

Sliding Constraints can be modelled as a DFA using the reformulation stated in [6]. However, the width of the sliding sequence should not be too large as the reformulation requires to explicit all the feasible tuples of the constraint to slide. This is often the case in PSP or also in car sequencing problems.

4.2 Systematic `multicost-regular` Generation

A formalism to describe Personnel Scheduling Problems has been proposed in [14]. The set of predefined XML markups allows to specify a large scope of PSP. In order to automatically generate a CP model based on `multicost-regular` from such specifications, we developed a framework capable of interpreting those XML files. In a first step, we bounded each markup associated to a work regulation to one of the 3 categories described above. Hence, for each rule of a given PSP instance, we automatically generate either an automaton or a counter depending on the rule category. For instance, the forbidden pattern "no day shift just after a night shift" is defined in the xml file as

```
<Pattern weight="1350"><Shift>N</Shift><Shift>D</Shift></Pattern>
```

and is automatically turned into its equivalent regular expression $(D|N|R)*ND(D|N|R)*$. We use a java library for automata[5] in order to create a DFA from a regular expression and to operate on the set of generated DFA. We use the opposite, the intersection and the minimization operations to build an unique DFA. Once the DFA is built, we treat the rules that engender counters, and generate a `multicost-regular` instance for each employee. Last, we treat the transversal constraints and include them in the CP model. For example, cover requirements are turned into `global-cardinality` constraints. Note that we were not able to deal with two kinds of specifications: some rule violation penalties that the `multicost-regular` cannot model and the pattern cardinality rules that we do not yet know how to automatize the reformulation.

4.3 Two Personnel Scheduling Cases

We first tackled the *GPost* [14] problem. This PSP consists in building a valid schedule of 28 days for eight employees. Each day, an employee has to be assigned to a Day, Night, or Rest Shift. Each employee is bound to a (Fulltime or Part-time) contract defining regular pattern and cardinality rules. Regular pattern

[5] http://www.brics.dk/automaton/

rules are: "free days period should last at least two days", "consecutive working week-ends are limited" and "given shift sequences are not allowed". Using the automatic modelling method we presented earlier, we build a DFA for each kind of contract. Cardinality rules are: "a maximum number of worked days in the 28 days period is to be worked", "the amount of certain shifts in a schedule is limited" and "the number of working days per week is bounded". Cover requirements and employee availabilities are also modelled. The softness specification on rules has been ignored as well as the first pattern rule to avoid infeasibility.

The second case study is based on the generated benchmark set brought by Demassey et al. [3]. The work regulations arise from a real-world personnel scheduling problem. The goal is to build only one schedule for a day consisting of 96 fifteen minutes time slots. Each slot is assigned either a working activity, a break, a lunch or a rest. Each possible assignment carries a given cost. The purpose is to find a schedule of minimum cost meeting all the work regulations. As for the previous PSP, we can identify regular pattern work regulations: "A working activity lasts at least 1 hour", "Different work activities are separated by a break or a lunch", "Break, lunch and rest shifts cannot be consecutives", "Rest shifts are at the beginning or at the end of the day", and "A break lasts 15 minutes". And also fixed cardinality regulations with: "At least 1 and at most 2 breaks a day", "At most one lunch a day" and "Between 3 and 8 hours of work activities a day". In addition to those work regulations, some activities are not allowed to be performed during some period. These rules are trivial to model with unary constraints.

5 Experiments

Experimentations were run on an Intel Core 2 Duo 2Ghz processor with 2048MB of RAM running OS X. The two PSP problems were solved using the Java constraint library *CHOCO* with default value selection heuristics – min value.

5.1 On the Size of the Automaton

As explained in Section 2.1, the filtering algorithm complexity of the `regular` constraints depends on the size of the specified automaton. Thus it would seem natural that processing a big automaton is not a good idea. However, practical results points out two important facts. First of all, the operations we run for automatically building a DFA from several rules tends to generate partially unfolded DFA (by intersection) and to reduce the number of redundant states which lie in the same layers (by minimization). Hence, the unfolded automaton generated during the forward phase at the initialization of the constraint can even be smaller than the specified automaton Π. Secondly, pruning during the backward phase may produce an even smaller automaton Π_n as many accepting states cannot be reached in a given number n of transitions. Table 1 shows the number of nodes and arcs of the different automata during the construction of the `multicost-regular` constraints for the *GPost* problem: the sum of the

Table 1. Illustration of graph reduction during presolving

Contract	Count	sum of DFAs	Π	Forward	Backward (Π_n)
Fulltime	# Nodes	5782	682	411	230
	# Arcs	40402	4768	1191	400
Parttime	# Nodes	4401	385	791	421
	# Arcs	30729	2689	2280	681

DFAs generated for each rule, the DFA Π after intersection and minimization, the unfolded DFA after the forward and backward phases.

5.2 Comparative Experiments

The previous section showed the ease of modelling with `multicost-regular`. However, there would be no point in defining such a constraint if the solving was badly impacted. We then conduct experiments for comparing our algorithm with a decomposed model consisting of a `regular` (or `cost-regular` for optimization) channeled to a `global-cardinality` constraint (`gcc`).

Table 2. *GPost* problem results

	`multicost-regular`		`regular` ∧ `gcc`	
WE regulation	Time (s)	# Fails	Time (s)	# Fails
no	1.94	24	12.6	68035
yes	16.0	1576	449.2	2867381

Table 2 presents the computational results on the *GPost* instance. The models include 8 `multicost-regular` or 8 `regular` and `gcc` (for each employee) bound together by 28 transversal `gcc` (for each day). In the Table, the first row corresponds to the problem without the sliding rule over the maximum number of consecutive working week-ends. In the second row, this constraint was included. We tried various variable selection heuristics but found out assigning variables along the days gave the best results as it allows the constraint solver to deal with the transversal `gcc` more efficiently. Both models lead to the same solutions. Actually, the average time spent on each node is much bigger using `multicost-regular`. However, due to better filtering capabilities, the size of the search tree and the runtime to find a feasible solution are significantly decreased.

Our second experiment tested the scalability of `multicost-regular` (MCR) against `cost-regular`∧`gcc` (CR) on the optimization problem defined in Section 4.3. The models do not contain any other constraint. However the decomposed model CR requires additional channeling variables. Table 3 presents the results on a benchmark set made of 110 instances. The number n of working activities varies between 1 and 50. The assignment costs were randomly generated. We tested different variable selection heuristics and kept the best one for

Table 3. Shift generation results

	MCR model			CR model					
	Solved			Solved			Time out		
n	#	t	bt	#	t	bt	#	Δ	# opt
1	10	0.6	49	10	1.1	292	0	-	-
2	10	0.8	54	10	2.4	539	0	-	-
4	10	1.5	65	10	13.5	1638	0	-	-
6	10	1.6	44	10	53.6	4283	0	-	-
8	10	2.1	51	9	209.2	5132	1	3.5%	0
10	10	2.4	58	7	283.5	6965	3	4.6%	0
15	10	3.8	59	6	283.9	4026	4	4.7%	1
20	10	4.9	49	6	311.8	4135	4	4.2%	1
30	10	6.9	51	1	313.0	4303	9	3.1%	1
40	10	13.4	68	0	-	-	10	6.1%	0
50	10	14.4	51	1	486.0	1406	9	5.0%	1

each model. Note that the results of the CR model are more impacted by the heuristic.

The first columns in the table show that with the MCR model, we were able to solve all instances (Column #) in less than 15 seconds for the biggest ones (Column t). The average number of backtracks (Column bt) remains stable and low as n increases. On the contrary the CR model is impacted a lot as shown in the next columns. Indeed, as the initial underlying graph becomes bigger it contains more and more paths violating the cardinality constraints. Those paths are not discarded by `cost-regular`. Some instances with more than 8 activities could not be solved within the given 30 minutes (Column #). Considering only solved instances, the running time (t) and the number of backtracks (bt) are always much higher than the MCR model based results. Regarding unsolved instances, the best found solution within 30 minutes is rarely optimal (Column # opt), and the average gap (Column Δ) is up to 6% for 40 activities.

6 Conclusion

In this paper, we introduce the `multicost-regular` global constraint and provide a simple implementation of Lagrangian relaxation-based filtering for it. Experimentations on benchmark instances of personnel scheduling problems show the efficiency and the scalability of this constraint compared to a decomposed model dealing with pattern requirements and cardinality requirements separately. Furthermore, we investigate a systematic way to build an instance of `multicost-regular` from a given set of standard work regulations. In future works, we ought to get a fully systematic system linked to the *CHOCO* solver for modelling and solving a larger variety of personnel scheduling and rostering problems.

Acknowledgements

We thank Mats Carlsson for pointing out the example illustrating the propagation issue of the `cost-regular` algorithm. We also thank Christian Schulte for its insightful comments on the paper and its numerous ideas to improve the constraint.

References

1. 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)
2. van Hoeve, W.J., Pesant, G., Rousseau, L.M.: On global warming: Flow-based soft global constraints. J. Heuristics 12(4-5), 347–373 (2006)
3. Demassey, S., Pesant, G., Rousseau, L.M.: A cost-regular based hybrid column generation approach. Constraints 11(4), 315–333 (2006)
4. Sellmann, M.: Theoretical foundations of CP-based lagrangian relaxation. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 634–647. Springer, Heidelberg (2004)
5. Beldiceanu, N., Carlsson, M., Debruyne, R., Petit, T.: Reformulation of Global Constraints Based on Constraint Checkers. Constraints 10(3) (2005)
6. Bessière, C., Hebrard, E., Hnich, B., Kiziltan, Z., Quimper, C.G., Walsh, T.: Reformulating global constraints: The SLIDE and REGULAR constraints. In: Miguel, I., Ruml, W. (eds.) SARA 2007. LNCS, vol. 4612, pp. 80–92. Springer, Heidelberg (2007)
7. Maher, M., Narodytska, N., Quimper, C.G., Walsh, T.: Flow-Based Propagators for the sequence and Related Global Constraints. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 159–174. Springer, Heidelberg (2008)
8. Kadioglu, S., Sellmann, M.: Efficient context-free grammar constraints. In: AAAI, pp. 310–316 (2008)
9. Trick, M.: A dynamic programming approach for consistency and propagation for knapsack constraints (2001)
10. Régin, J.C., Puget, J.F.: A filtering algorithm for global sequencing constraints. In: Smolka, G. (ed.) CP 1997. LNCS, vol. 1330, pp. 32–46. Springer, Heidelberg (1997)
11. Handler, G., Zang, I.: A dual algorithm for the restricted shortest path problem. Networks 10, 293–310 (1980)
12. Shor, N., Kiwiel, K., Ruszcayǹski, A.: Minimization methods for non-differentiable functions. Springer, Heidelberg (1985)
13. Boyd, S., Xiao, L., Mutapcic, A.: Subgradient methods. Lecture notes of EE392o, Stanford University (Autumn Quarter 2004) (2003)
14. Personnel Scheduling Data Sets and Benchmarks, `http://www.cs.nott.ac.uk/~tec/NRP/`

Bandwidth-Limited Optimal Deployment of Eventually-Serializable Data Services

Laurent Michel[2], Pascal Van Hentenryck[1], Elaine Sonderegger[2], Alexander Shvartsman[2], and Martijn Moraal[2]

[1] Brown University, Box 1910, Providence, RI 02912
[2] University of Connecticut, Storrs, CT 06269-2155

Abstract. Providing consistent and fault-tolerant distributed object services is among the fundamental problems in distributed computing. To achieve fault-tolerance and to increase throughput, objects are replicated at different networked nodes. However, replication induces significant communication costs to maintain replica consistency. 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 revisits ESDS instances where bandwidth constraints are imposed on segments of the network interconnect. This class of problems was shown to be extremely challenging for both Mixed Integer Programming (MIP) and for Constraint Programming (CP), some instances requiring hours of computation time. The paper presents an improved constraint programming model, a constraint-based local search model that can obtain high-quality solutions quickly and a local search/constraint programming hybrid. The experimental results indicate that the resulting models significantly improve the state of the art.

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. Eventually-Serializable Data Services (ESDS) [5] is a system that was formulated to help reduce these costs. The algorithm implementing ESDS allows 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), and the users are able to require the results for certain operations to be consistent with the stable total order of all operations.

The design, analysis, and implementation of systems such as ESDS is not an easy task, and specification languages have been developed to express these algorithms formally. For instance, the framework of (timed) I/O automata [9,7] and their associated tools [10] allows theorem provers (e.g., PVS [13]) and model

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 193–207, 2009.

checkers (e.g., UPPAAL [8,3]) to reason about correctness. The ESDS algorithm is in fact formally specified with I/O automata and proved correct [5]. 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 [4] for a methodic implementation of the algorithm and a study of its performance. The deployment then focuses on mapping the software modules onto a distributed computing platform to maximize performance.

This research focuses on the last step: the deployment of the implementation on a specific architecture. The deployment 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 basic ESDS Deployment Problem (ESDSDP) was considered by [2] and was modeled as a MIP with disappointing results even on small instances. A highly competitive CP approach as well as a viable MIP model can be found in [11]. In [12], the basic ESDSDP model was extended to take into account bandwidth constraints on various segments of the network interconnect. This richer model proved harder for both MIP and CP solvers with running times in hours on some instances.

This paper focuses on the bandwidth-limited version of the ESDSDP and studies a constraint programming (CP), a Constraint-Based Local Search (CBLS), and an hybrid model. The empirical evaluation demonstrates that the CP model solves most instances in minutes (significantly outperforming the earlier CP model) and that the CBLS model can deliver high-quality solutions quickly. The CP model is a natural encoding of ESDSDP together with a simple search heuristic focusing on the objective. It improves earlier results [11] by exploiting a dominance property to rule out bandwidth-limited paths that are provably inferior to already considered paths. The CBLS model uses the same natural declarative model and its search procedure uses two neighborhoods that focus on the objective and the feasibility part of the model. The feasibility neighborhood is a simple constraint-directed search. The hybrid delivers optimality proofs for the biggest instances from 200 to 2800 times faster than the MIP and 30 to 90 times faster than the earlier CP model while improving the robustness.

The rest of this paper is organized as follows. Section 2 presents an overview of the bandwidth-limited ESDS and illustrates the deployment problem on a basic instance. Section 3 introduces the high-level deployment model and Section 4 presents the CP models. Section 5 presents the CBLS model, while Section 6 covers the hybridization. Section 7 reports the experimental results and analyzes the behavior of the models in detail. Section 8 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. The clients may request operations whose results are tentative, but can be quickly obtained, or they can request "strict" operations that are possibly slower, but whose results are guaranteed to be consistent with an eventual total order on the operations. Each replica maintains a set of the requested operations and a partial ordering on these operations that tends to the eventual total order on operations. 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. For any sequences of requests issued by the clients, the service guarantees eventual consistency of the replicated data [5].

ESDS is well-suited for implementing applications such as a distributed directory service, such as Internet's Domain Name System [6], 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 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. 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. A significant additional complication in deriving deployment mappings is due to bandwidth limitations placed on connections between the nodes in the target platform, e.g., a subnet based on a switched ethernet at 100Mbit/s.

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 abstract implementation of the ESDS. The ESDS software

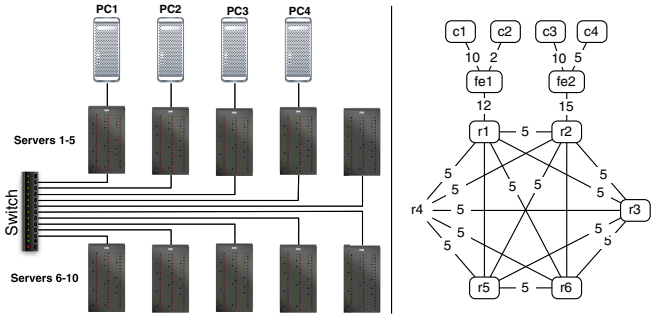


Fig. 1. A Simple ESDS Deployment Problem

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 $(r_1, \cdots, r_6)$. 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 $(c_4, fe_1, fe_2, r_1, \cdots, r_6)$ must run on the heavy-duty servers. Additionally, the replicas $r_1 \cdots r_6$ must execute on distinct servers to achieve fault tolerance. The deployment problem consists of finding an assignment of software components to servers that satisfies the constraints above and minimizes the overall network traffic expressed as the volume of messages sent given the host assignments.

3 Modeling Optimal ESDS Deployments

The deployment model for ESDS is based on [1,2]. 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 $\forall i, j \in S \ : \ x_i = x_j$.

Realistic target networks may impose bandwidth limitation on connections between hosts due to either physical channel limitations, input buffer limitations, or QoS guarantees. To reflect such constraints, the deployment model is extended to incorporate bandwidth limitations on some connections. Informally, a *connection* is a network inter-connect between a set of k machines. For instance, a connection can be a dedicated point-to-point link or a switched 802.11-wired Ethernet subnet. A deployment platform then reduces to a set of *connections* with some nodes (e.g., routers or machines with several network cards) appearing in several connections to establish bridges. More formally, a deployment platform is an hypergraph $H = (X, E)$ where X is the set of nodes and E is a set of hyperedges, i.e., $E \subseteq \mathcal{P}(X) \setminus \{\emptyset\}$ where $\mathcal{P}(X)$ is the power-set of X. Each hyperedge c carries a bandwidth capacity that is denoted by $c.bw$ and its vertices are denoted by $c.nSet$. If a hyperdge c (connection) has no bandwidth limitations, its bandwidth capacity is $c.bw = 0$.

Each pair of hosts is connected by one or more paths, where a path is an ordered collection of hyperedges (connections) from the source to the destination host. A path is bandwidth-limited if one or more of its hyperedges (connections) has limited (positive) bandwidth; otherwise a path is not bandwidth-limited. The network paths are represented by the following calculated parameters:

- $P_{i,j}$ denotes the set of paths from host i to host j for all $i, j \in N$. If $i = j$, $P_{i,j}$ contains a single path of zero length;
- A 3-D matrix h, where $h_{i,j,p}$ is the length of path p from host i to host j (replacing the 2-D hops matrix h);
- A boolean matrix $hasC$, where $hasC_{i,j,p,c}$ is *true* if path p from host i to host j uses the hyperedge (connection) c.

The model contains decision variables to choose the paths to be used by communicating software modules deployed on hosts. Specifically, a new decision variable $path_{a,b}$ is associated with each communicating pair of components a and b and represents the path component a uses to send messages to component b. This variable must satisfy the constraint $path_{a,b} \in P_{x_a, x_b}$. This reflects the current assumption that each pair of components uses a single directed path for data transmission. However, $path_{a,b}$ and $path_{b,a}$ may be different.

An optimal deployment minimizes

$$\sum_{a \in C} \sum_{b \in C} f_{a,b} \cdot h_{x_a, x_b, path_{a,b}}$$

subject to the supporting, separation, and co-location constraints presented earlier and the following bandwidth constraint for each $c \in E$ with $c.bw > 0$:

$$\sum_{a \in C} \sum_{b \in C} f_{a,b} \cdot hasC_{x_a, x_b, path_{a,b}, c} \leq c.bw$$

4 The CP Model

The COMET program for bandwidth-limited connections is shown in Figure 2. The data declarations are specified in lines 2–11, and the decision variables are declared in lines 12–13. Variable $x[c]$ specifies the host of component c, with its domain computed from the support matrix s. The variable $path[c1, c2]$ specifies the path used to send messages from component $c1$ to component $c2$, expressed as the rank of the selected path in the set $P[x[c1], x[c2]]$.

Lines 14–18 specify the objective function, which minimizes communication costs. The CP formulation uses a three-dimensional element constraint since the matrix h is indexed not only by variables for the two hosts but also by the variable for the particular communication path used between them.

Lines 19–24 contain the co-location and separation constraints. Lines 25–26 limit the ranges of the individual path variables to the number of paths between the hosts onto which the components are deployed. Lines 27–29 are the bandwidth constraints: for each hyperedge $c \in E$, the bandwidth $c.bw$ must be greater than or equal to the sum of the communication frequencies of all pairs of components with c in their chosen path. The `onDomains` annotations indicate that arc-consistency must be enforced for each constraint.

The search procedure, depicted in lines 31–45, operates in two phases. In the first phase (lines 31–40) all the components are assigned to hosts, beginning with the components that communicate most heavily. The search must *estimate* the communication cost between components a and b's potential deployment sites along *any* given path. Line 33 picks the first site k for component b, and the `tryall` instruction on line 34 considers the sites for component a in increasing order of path length based on an estimation equal to the shortest path one could take between the choice n and the selection k. (Symmetry breaking as in [11] optionally may be included.) The second phase (lines 41-45) labels the path variables, backtracking as needed over the initial component assignments.

Path Dominance. The initial CP bandwidth model [12] used a weak form of dominance. After finding a path that is not bandwidth-limited for a given pair of nodes, it discards all the other paths between those two nodes that are of the same length or longer. Longer paths need not be considered because the deployment model minimizes the number of "hops" for transmitted messages, and a shorter, non-bandwidth limited path always will be a better choice.

```
Solver<CP> cp();
range C = ...; // The components
range N = ...; // The host nodes
int[,] s = ...; // The supports matrix
int[,] f = ...; // The frequency matrix
int[,,] h = ...; // The hops matrix
set{set{int}} Sep = ...; // The separation sets
set{set{int}} Col = ...; // The co−location sets
set{connection} Conn = ...; // The connections
set{connection}[,] P = ...; // The paths matrix
int[,,,] hasC = ...; // The path/connection matrix
var<CP>{N} x[c in C](cp, setof(n in N) (s[c,n] == 1));
var<CP>{int} path [a in C, b in C](cp,0..max(i in N, j in N) P[i,j].getSize()−1);
var<CP>{int} obj(cp, 0..1000);
minimize<cp> obj
subject to {
   cp.post(obj == sum(a in C, b in C: f[a,b] != 0) f[a,b] ∗ h[x[a],x[b],path[a,b]],
      onDomains);
   forall(S in Col)
      select(c1 in S)
         forall (c2 in S: c1 != c2)
            cp.post(x[c1] == x[c2], onDomains);
   forall(S in Sep)
      cp.post(alldifferent(all(c in S) x[c]), onDomains);
   forall (a in C, b in C : f[a,b] != 0)
      cp.post (path[a,b] < P[x[a],x[b]].getSize(), onDomains);
   forall (c in Conn: c.bw > 0)
      cp.post (c.bw >= sum (a in C, b in C: f[a,b] != 0)
         hasC[x[a], x[b], path[a,b], c] ∗ f[a,b], onDomains);
} using {
   while (sum(k in C) x[k].bound() < C.getSize()) {
      selectMax(a in C: !x[a].bound(), b in C)(f[a,b]) {
         int k = min(k in N: x[b].memberOf(k)) k;
         tryall<cp>(n in N: x[a].memberOf(n))
               by (min (i in 0..P[n,k].getSize()−1) h[n,k,i])
            cp.post(x[a] == n);
         onFailure
            cp.post(x[a] != n);
      }
   }
   forall (a in C,b in C: f[a,b] != 0 && !path[a,b].bound())
      tryall<cp> (i in 0..P[x[a],x[b]].getSize()−1) by (h[x[a], x[b], i])
         cp.post(path[a,b] == i);
      onFailure
         cp.post(path[a,b] != i);
}
```

Fig. 2. The Bandwidth-Limited Model in Comet

Several other categories of paths need not be considered in determining an optimal deployment. In particular, let $p1$ and $p2$ be any two paths between nodes $n1$ and $n2$, and let $p1.bwSet$ and $p2.bwSet$ be the sets of bandwidth-limited connections of $p1$ and $p2$. Then $p1$ can be ignored if any of the following conditions hold:

- The path $p1$ is strictly longer than $p2$ and $p2.bwSet \subseteq p1.bwSet$.
- The path $p1$ is the same length as $p2$ and $p2.bwSet \subset p1.bwSet$.
- The path $p1$ is the same length as $p2$, $p2.bwSet = p1.bwSet$, and $p1$ is ordered after $p2$ in the set of paths between $n1$ and $n2$. (This is an arbitrary selection of one of two equivalent paths.)

The extended CP model uses these rules to eliminate all dominated paths during the initialization of the model. Alternatively, these path dominance properties can be expressed as constraints on the viable paths between components. However, this strategy is ineffective. Indeed, the set of paths P is indexed by its source and destination *hosts* whereas the `path` variable is indexed by its source and destination *components*. For any two components `c1` and `c2`, the constraint on the path variables would look like `validPath[x[c1],x[c2],path[c1,c2]]==1`. This constraint, though, is unable to fully reduce the domain of the path variable until the corresponding `x`'s are fixed. Avoiding the construction of dominated path altogether alleviates that difficulty as the constraint above can simplify to an upper-bound on the size of the domain of `path`.

5 The CBLS Model

The parameters of the CBLS model (components, nodes, frequency, hops, co-located, separated, and fixed) are identical to the CP model. Likewise, the CBLS model has two decision variables: an array $x[c]$ that specifies the node on which component c is deployed, and $path[c1, c2]$ that specifies the path used to connect components $c1$ and $c2$. The search procedure of the CBLS model uses a guided local search approach which increases the weights of the constraints that are hard to satisfy. The weight variables are created when the feasibility constraints are posted. A tabu list is represented by a simple dictionary that records which variables were changed recently (the initialization per se is not shown for brevity reasons).

The declarative part of the CBLS model is shown in Figure 3. It starts with the declaration of a weighted constraint system S for the guided local search. The feasibility constraints in lines 3–19 are essentially identical to those found in the CP model. Lines 3–6 declare the co-location constraints as mere equalities, while lines 7–8 specify the separation constraints with an alldifferent. Lines 10–13 require each path variable to range over the indices of available paths between the hosts onto which the components are deployed. Finally, lines 15–19 specify the bandwidth constraints.

The core objective function O is specified in lines 21–23 as the sum of the communication frequency between each pair of components multiplied by the

```
1 WeightedConstraintSystem<LS> S(m);
2 function var{int} mkWeight(Solver<LS> m) { var{int} x(m) := 1;return x;}
3 forall (t in Col)
4   selectMin(c1 in t.cSet)(c1)
5     forall (c2 in t.cSet : c1 != c2)
6       S.post(istrue(x[c1] == x[c2]), mkWeight(m));
7 forall (s in Sep)
8   S.post(alldifferent(all(c in s.cSet)x[c]), mkWeight(m));
9
10 ConstraintSystem<LS> S2(m);
11 forall (c1 in C, c2 in C : f[c1, c2] != 0)
12   S2.post (path[c1, c2] < numPaths[x[c1], x[c2]]);
13 S.post(S2, mkWeight(m));
14
15 ConstraintSystem<LS> S3(m);
16 forall (c in 0..Conn.getSize()−1 : Conn.atRank(c).bw > 0)
17   S3.post(Conn.atRank(c).bw >= sum(c1 in C, c2 in C : f[c1, c2] != 0)
18           (hasC[x[c1], x[c2], path[c1, c2], c] ∗ f[c1, c2]));
19 S.post(S3, mkWeight(m));
20
21 FunctionSum<LS> O(m);
22 forall (c1 in C, c2 in C : f[c1, c2] != 0)
23   O.post(f[c1, c2] ∗ h[x[c1], x[c2], path[c1, c2]]);
24
25 Function<LS> C = S + O;
26 m.close();
27 weights = all(k in S.getRange()) S.getWeight(k);
```

Fig. 3. The Constraint Systems for the CBLS Model in COMET

length of the chosen path between these components. Finally the objective function C declared in line 25 combines the feasibility constraint set S with the core objective O.

The search, illustrated in Figure 4, explores two different neighborhoods. During each iteration, one of the two neighborhoods is selected by line 2 with a fixed probability *objChance* (which defaults to 70%).

The first neighborhood (lines 3–9) focuses on the objective. Line 4 selects a non-tabu variable appearing in the objective and leading to the largest decrease to the overall objective function C. The selected variable is then assigned a value that delivers the largest decrease in the objective O, and tags the variable as tabu for the next *tLen* iterations. The second neighborhood (lines 11–17) is a standard constraint-directed search that attempts to reduce the number of violations on the constraints in S. It chooses a constraint k that causes the most violations and picks the worst variable from constraint k. Line 15 then selects the most promising value for the selected variable, and line 16 reassigns it. The best feasible solution is recorded in line 22, while line 23 updates the weight of the unsatisfied constraints in S when the algorithm hasn't progressed for

```
1  while (it < maxit) {
2    if (zo.get() <= objChance) {
3      var{int}[] ox = O.getVariables();
4      selectMax(i in ox.getRange() : tabu{ox[i].getId()} <= it)(C.decrease(ox[i])) {
5        selectMin(v in ox[i].getDomain())(O.getAssignDelta(ox[i], v)) {
6          ox[i] := v;
7          tabu{ox[i].getId()} = it + tLen;
8        }
9      }
10   } else {
11     selectMax(k in S.getRange())(S.getConstraint(k).violations()) {
12       Constraint<LS> cls = S.getConstraint(k);
13       var{int}[] cx = cls.getVariables();
14       selectMax(i in cx.getRange())(cls.violations(cx[i]))
15         selectMin(v in cx[i].getDomain())(C.getAssignDelta(cx[i],v))
16           cx[i] := v;
17     }
18   }
19   it++;
20   stableit++;
21   boolean feasible = S.violations()==0;
22   if (feasible && O.value() < bestValue) saveBest();
23   if (stableit >= 100) glsUpdate();
24   if (rounds >= 200) diversify(C);
25 }
```

Fig. 4. The Search Strategy for the CBLS Model in COMET

100 consecutive iterations. Finally, line 24 performs a diversification on all the variables appearing in the objective function C whenever 200 rounds of guided local search (weights updating) were unable to further improve the objective function. The diversification simply reassigns a fraction of the variables in C (chosen based on probability $diversifyChance$) uniformly at random, resets the weights of the guided local search, and updates the bounds on the length of the tabu list.

Co-Location Preprocessing. An alternative representation of the problem replaces each co-location constraint and its associated component variables with a single component variable representing the common location of the co-located components. This avoids a large collection of equality constraints. Simple preprocessing and postprocessing steps can then recast the solution in term of the initial formulation. This leaner formulation is beneficial for the CBLS model. In the original formulation, when a component is moved, all the co-location equality constraints are violated which induces a *bump* in the optimization value, which can make these moves less desirable. The preprocessed formulation does not suffer from this problem since all the co-located components are moved as one, allowing for aggregate moves. The experimental results confirm that this representation is beneficial.

Path Dominance. Unsurprisingly, the CBLS model also can make use of the path dominance rule during the initialization of the model to eliminate paths that are provably inferior. The experimental results consider local search models with path dominance included.

6 Hybrid CBLS-CP Model

The hybrid model is a sequential composition. The CBLS model runs for 10 seconds and then passes its best solution to the CP model to complete the optimization. A hybrid model using parallel composition also was considered, where the CBLS model runs in a separate thread and notifies the CP model each time it finds a better solution. The benefit of the parallel composition is only visible on *easy* instances where the CP model proves the optimality in less than 10 seconds (and therefore stops the search right away).

7 Experimental Results

The Benchmarks. The benchmarks fall into three categories: variants of the simple ESDS deployment problem depicted in Figure 1, variants of the `HYPER8` ESDS deployment problem shown in Figure 5, and variants of the `RING6` deployment problem shown in Figure 6. The `HYPER8` and `RING6` benchmarks are studied, not because they reflect actual network configurations, but because they are simple representations of networks with many equivalent alternative paths and networks with tightly coupled hosts. To model the capabilities of the communication infrastructure of a distributed system more realistically, all the benchmarks include, in addition to the components shown, one extra software module between each pair of replicas (components $r_1, \ldots, r_6$). These extra components are "drivers" that manage the communication channels and are required to be co-located with their sending replicas. The benchmarks are as follows:

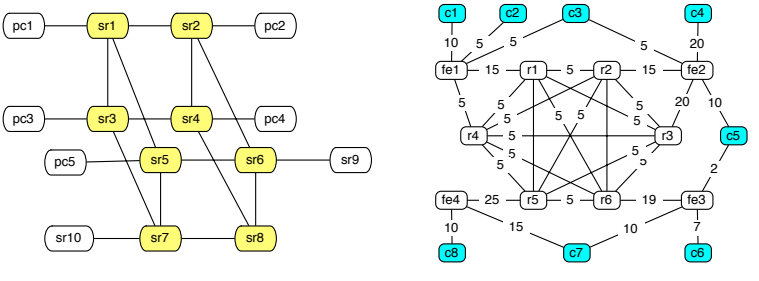


Fig. 5. Instance `HYPER8`: Deploying ESDS to a network with many equivalent paths

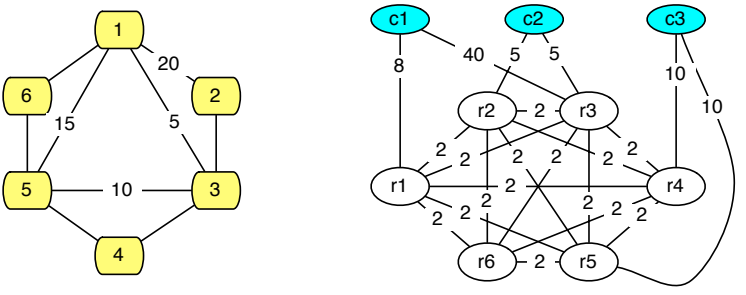


Fig. 6. Instance `RING6`: Deploying to a tightly coupled, bandwidth-limited network

- `SIM2BW1` is a variant of Figure 1 with a bandwidth limit of 5 on the connection between PC_2 and r_2.
- `SIM2BW2` is a variant of Figure 1 with a bandwidth limit of 5 on the connection between PC_2 and r_2 and a bandwidth limit of 10 on the connection between PC_3 and r_3.
- `RING4` is a variant of `RING6` with only four gossiping replicas (and no messages from $c3$ to $r5$).
- `RING5` is a variant of `RING6` with only five gossiping replicas.
- `RING6` is illustrated in Figure 6.
- `HYP8BW1` is a variant of `HYPER8` with a bandwidth limit of 10 on the connection between PC_1 and r_1.
- `HYP8BW4` is a variant of `HYPER8` with four bandwidth-limited connections.

Experimental Results for the CP Model. Table 1 reports the results for the CP model with Comet 1.1 (executing on an Intel Core 2 at 2.4Ghz with 2 gigabytes of RAM). The first three columns give the results for the initial CP bandwidth model ([12]) which only eliminates paths that are the same length or longer than the shortest non-bandwidth limited path. The next three columns give the results when the full path dominance described in Section 4 is exploited. The final three columns give the results when both path dominance and co-location

Table 1. Experimental Results for the CP Models

		Longest Path Dominance			Full Path Dominance			Path Dominance & Co-Location		
Benchmark	MIP	T_{end}	$\#Chpt$	T_{opt}	T_{end}	$\#Chpt$	T_{opt}	T_{end}	$\#Chpt$	T_{opt}
SIM2BW1	12.8	0.27	168	0.02	0.27	168	0.02	0.13	174	0.01
SIM2BW2	17.0	0.38	159	0.03	0.38	160	0.03	0.20	165	0.01
RING4	9.8	0.37	469	0.35	0.34	426	0.32	0.15	187	0.14
RING5	66.4	10.9	24422	10.4	9.6	20609	9.1	1.5	2101	1.3
RING6	327.9	132.8	300526	130.6	107.8	235161	105.6	14.4	27722	13.3
HYP8BW1	142388	4642.2	133047	893.3	123.0	43169	28.6	75.5	39137	11.3
HYP8BW4	71102	12572.9	108069	8346.5	1988.0	162227	1641.0	1175.7	103023	770.9

preprocessing are performed. Within each group, 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. The results are averages of 50 runs (except for the HYP8BW4 Longest Path results which are averages of 10 runs). The column MIP repeats the results (CPLEX version 11 running on an AMD Athlon at 2Ghz) for the MIP model described in [12]. It is useful to review these results in more detail.

1. Full path dominance preprocessing is faster for all benchmarks than longest path dominance preprocessing as expected. Adding co-location preprocessing also improves performance for all benchmarks.
2. For SIM2BW1 and SIM2BW2, the performance is almost identical with both path dominance techniques. There is only one path between each pair of hosts, so there are no paths to eliminate and thus no benefits.
3. RING4, RING5, and RING6 all have the same network and path characteristics. Although full path dominance only eliminates one more path than longest path dominance (20 vs. 19 eliminated paths out of 98 total paths), the impact on performance is non-negligible for all three benchmarks.
4. In HYP8BW1, full path dominance eliminates all but one path between each pair of hosts, resulting in a dramatic improvement (factor of 37) over longest path dominance which has up to 18 paths between pairs of nodes.
5. In HYP8BW4, full path dominance retains two paths between 16 pairs of hosts and one path between all other pairs. Even this relatively small number of path options make processing HYP8BW4 considerably more complex than HYP8BW1. The improvement with full path dominance is still substantial (over a factor of 6), but not as dramatic.
6. The improvement with co-location preprocessing appears to be related to the fraction of components that can be combined. The largest improvement is for RING6 where 45 components are reduced to 8, and the smallest improvement is for HYP8BW1 and HYP8BW4 where 54 components are reduced to 18.
7. With all three preprocessing techniques, the CP model finds the optimum relatively quickly for SIM2BW1, SIM2BW2, and HYP8BW1. Unfortunately, the optimum is not found until late in the search for the other benchmarks.
8. The MIP results are also likely improve with path and co-location preprocessing.

Experimental Results for the CBLS Model. Table 2 reports the results for the CBLS models. The *Opt* column gives the optimum. The $\mu(Path)$ and $\sigma(Path)$ columns report the averages and standard deviations for the best solution (Q), the time to the best solution in seconds (TB) and the total running time (TT) of the CBLS model with path dominance. The $\mu(Path\&Col)$ and $\sigma(Path\&Col)$ columns give the averages and standard deviations with both path dominance and co-location pre-processing. All the results are reported over 50 runs.

The experimental results indicate that CBLS delivers high-quality solutions in a few seconds. The elimination of the co-location constraints is beneficial in several respects. First, it reduces the running time significantly (both to termination and to the best solution), and it has a positive impact on the average

Table 2. Experimental Results for the Local Search Models

		$\mu(Path)$			$\mu(Path \,\&\, Col)$			$\sigma(Path)$			$\sigma(Path \,\&\, Col)$		
Benchmark	Opt	Q	TB	TT	Q	TB	TT	Q	TB	TT	Q	TB	TT
RING4	54	54.1	1.79	6.06	54.0	0.17	4.22	0.7	1.23	0.04	0.0	0.12	0.04
RING5	88	90.1	3.74	8.83	88.0	0.61	5.20	2.7	2.61	0.06	0.0	0.48	0.05
RING6	120	131.1	6.87	14.70	120.8	3.76	7.72	21.7	5.08	0.78	1.0	0.00	0.09
HYP8BW1	522	594.0	2.62	5.22	523.1	1.28	3.56	37.4	1.76	0.20	1.8	0.59	0.07
HYP8BW4	526	552.3	7.42	17.58	554.5	5.41	8.92	18.5	5.60	0.50	23.1	2.13	0.34

Table 3. Experimental Results for the Sequential Hybrid Models

	μ(Path)		μ(Path & Col)		σ(Path)		σ(Path & Col)	
Benchmark	T_{end}	$\#Chpt$	T_{end}	$\#Chpt$	T_{end}	$\#Chpt$	T_{end}	$\#Chpt$
RING4	10.09	11	10.03	6	0.00	1	0.01	3
RING5	11.02	88	10.14	26	0.08	15	0.01	2
RING6	23.64	17718	10.51	81	25.26	50147	0.04	7
HYP8BW1	140.70	40799	51.06	29602	9.35	4404	0.74	463
HYP8BW4	1325.02	63343	339.68	32045	294.22	28785	26.54	3675

best solution found. Indeed, as the standard deviation shows, the local search algorithm could deliver the best solution on all 50 runs on `RING4` and `RING5` and the average best solution across the board. Second, all the standard deviations improved significantly, indicating that the algorithm is more robust.

Experimental Results for the Hybrid Model. Table 3 reports the results for a sequential hybrid model that runs the best CBLS model for 10 seconds before initiating a CP search with an upper bound based on the best solution delivered in the first phase. All the results are averages based on 50 runs. The column groups are the same as for the CP model. Within each group, T_{end} denotes the time in seconds to find the optimum and prove optimality, and $\#Chpt$ denotes the number of choice points.

The first hybrid algorithm composes models relying only on the path dominance. Nonetheless, the benefit is already visible when the pure CP model is compared to the hybrid. The second hybrid composes models using both path dominance and co-location pre-processing and clearly dominates the earlier CP models. On the hardest instance (HYP8BW4) it proves optimality in 340s when the pure CP model needed 1176s. The good results can be attributed to an excellent phase 1 that delivers a high quality solution to bootstrap the second phase. On the ring instances the runtime is dominated by the fixed 10s of local search, while on the HYPER8 it spends most of the computation in the CP phase.

8 Conclusion

This paper revisited the Bandwidth-Limited ESDS Deployment Problem and considered an improved CP model that leverages dominance properties, a CBLS model featuring the same declarative model, as well as a CP/CBLS hybrid model. The CBLS model is particularly compelling given the similarity of its declarative

part and its ability to deliver high-quality solutions quickly. Its search procedure composes a standard constraint-directed neighborhood for the feasibility part of the model with a tabu-based greedy gradient descent for the objective function. The path dominance and the co-location preprocessing steps proved very effective for CP and CBLS, both in terms of the solution quality and the time to solve the model. Constraint Programming now appears to be the ideal methodology to solve this class of problem for which hard instances can be solved to optimality in 5 to 10 minutes.

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 and support from AFOSR under contract FA955007C0114.

References

1. Bastarrica, M., Demurjian, S., Shvartsman, A.: Software architectural specification for optimal object distribution. In: SCCC 1998: Proc. of the XVIII Int-l Conf. of the Chilean Computer Science Society, Washington, DC, USA (1998)
2. Bastarrica, M.C.: Architectural specification and optimal deployment of distributed systems. PhD thesis, University of Connecticut (2000)
3. 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)
4. Cheiner, O., Shvartsman, A.: Implementing an eventually-serializable data service as a distributed system building block. Networks in Distributed Computing 45, 43–71 (1999)
5. Fekete, A., Gupta, D., Luchangco, V., Lynch, N.A., Shvartsman, A.A.: Eventually-serializable data services. Theor. Comput. Sci. 220(1), 113–156 (1999)
6. IETF. Domain name system, rfc 1034 and rfc 1035 (1990)
7. Kaynar, D.K., Lynch, N., Segala, R., Vaandrager, F.: The Theory of Timed I/O Automata. Synthesis Lectures in Computer Science. Morgan & Claypool Publishers (2006)
8. 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)
9. Lynch, N., Tuttle, M.: An introduction to Input/Output Automata. CWI-Quarterly 2(3), 219–246 (1989)
10. 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
11. Michel, L., Shvartsman, A.A., Sonderegger, E.L., Hentenryck, P.V.: Optimal deployment of eventually-serializable data services. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 188–202. Springer, Heidelberg (2008)
12. Michel, L., Shvartsman, A.A., Sonderegger, E.L., Hentenryck, P.V.: Optimal deployment of eventually-serializable data services. CPAIOR (2008); extended version of the CPAIOR 2008 paper (submitted)
13. 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)

Tightening the Linear Relaxation of a Mixed Integer Nonlinear Program Using Constraint Programming

Sylvain Mouret[1], Ignacio E. Grossmann[1], and Pierre Pestiaux[2]

[1] Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh PA 15213, USA
{smouret,grossmann}@cmu.edu
[2] Total Refining & Marketing, Research Division, 76700 Harfleur, France
pierre.pestiaux@total.com

Abstract. This paper aims at solving a nonconvex mixed integer nonlinear programming (MINLP) model used to solve a refinery crude-oil operations scheduling problem. The model is mostly linear but contains bilinear products of continuous variables in the objective function. It is possible to define a linear relaxation of the model leading to a weak bound on the objective value of the optimal solution. A typical method to circumvent this issue is to discretize the continuous space and to use linear relaxation constraints based on variables lower and upper bounds (e.g. McCormick convex envelopes) on each subdivision of the continuous space. This work explores another approach involving constraint programming (CP). The idea is to use an additional CP model which is used to tighten the bounds of the continuous variables involved in bilinear terms and then generate cuts based on McCormick convex envelopes. These cuts are then added to the mixed integer linear program (MILP) during the search leading to a tighter linear relaxation of the MINLP. Results show large reductions of the optimality gap of a two step MILP-NLP solution method due to the tighter linear relaxation obtained.

1 Introduction

Many optimization problems arising in the chemical industry involve nonconvex nonlinear functions which makes them difficult to solve to global optimality. In this paper, the crude-oil scheduling problems introduced in [1] are solved with the original objective of minimizing the logistics cost. The problem is formulated as a nonconvex mixed integer nonlinear programming (MINLP) model developed in [2] which is based on a continuous-time scheduling representation. The solution procedure returns a solution with an estimate of the gap with the optimal solution. In earlier work, an operation specific event point continuous-time formulation has been developed (see [3]) and applied to the problems from [1] using a linear approximation of storage costs. However, no proof of optimality or estimate of optimality gap is given with this approximation. The global optimization of this model was later addressed by [4] using an outer-approximation algorithm.

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 208–222, 2009.

where the MILP master problem is solved by a Lagrangean decomposition. While rigorous, this method is still computationally expensive.

The main contribution of this work is to reduce the optimality gap by tightening the linear relaxation of the MINLP using McCormick convex envelopes for bilinear products of continuous variables (see [5]). This can be done by using discretization techniques and applying McCormick convex and concave estimators on each discrete element of the continuous space (see [6], [7], [8]). On the other hand, it has been shown how Constraint Programming (CP) techniques can be efficiently integrated with Branch & Bound procedures for the global optimization of MINLPs (see [9]). In this paper, CP is used during the search to tighten variable bounds and generate new McCormick cuts, thus efficiently tightening the mixed integer linear program (MILP) relaxation. When applied to the sequential MILP-NLP approach in [2], this extension of the Branch & Cut algorithm leads to reduced optimality gaps and allows finding good suboptimal solutions with lower logistics cost. The concepts developed in this work can be applied to a generic solver as in the global MINLP solver BARON or can be used to efficiently solve other optimization problems to global optimality by exploiting their specific structure.

This paper is organized as follows. Section 2 gives some details about the crude-oil scheduling problems to be solved. Section 3 presents the MINLP mathematical formulation with a nonlinear objective function. Section 4 explains how this model is intended to be solved. Section 5 shows how the model can be reformulated in order to get a linear objective function and gives a simple MILP relaxation using McCormick convex envelopes. Section 6 presents how to improve this MILP relaxation by adding tighter McCormick cuts for some subproblems explored during the Branch & Cut procedure. Finally, Section 7 gives computational results showing the impact of this approach in terms of relaxation and CPU time.

2 Problem Statement

A crude-oil operations scheduling problem has as an objective to prepare various types of crude-oil blends throughout the horizon in order to continuously feed each crude distillation unit (CDU), while satisfying the demand for each crude blend. The problem is composed of four types of resources: crude marine vessels, storage tanks, charging tanks and CDUs. Three types of operations, all transfers between resources, can be executed: unloading from crude-oil marine vessels to storage tanks, transfer between tanks, and transfers from charging tanks to CDUs. The following parameters are given: (a) a time horizon, (b) arrival time of marine vessels, (c) capacity limits of tanks, (d) transfer flowrate limitations, (e) initial composition of vessels and tanks, (f) crude property specifications for distillations, (g) and demands for each crude blend. The logistics constraints involved in the problem are defined as follows.

(i) Only one berth is available at the docking station for vessel unloadings,
(ii) simultaneous inlet and outlet transfers on tanks are forbidden,
(iii) a tank may charge only one CDU at a time,
(iv) a CDU can be charged by only one tank at a time,
(v) and CDUs must be operated without interruption.

The goal is to determine which operation will be executed, how many times and when it will be performed, and the amount of crude to be transferred. In [2], the authors considers a linear objective function based on the maximization of production gross margins. In this work, the objective is to minimize the logistics costs which include sea waiting costs and unloading costs for marine vessels, storage costs in tanks, and CDU switching costs. A CDU switch is a transition from a crude blend to another. Figure 1 depicts the refinery configuration for problem 1 given in [1].

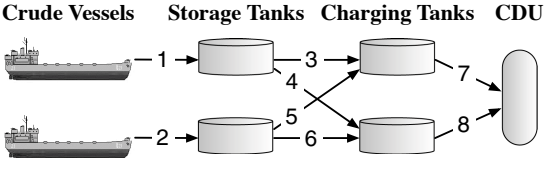


Fig. 1. Crude-oil operations system for problem 1

3 MINLP Model

This section presents the Single-Operation Sequencing (SOS) model introduced in [2]. It is based on the representation of a schedule as a sequence of operations. The optimal schedule is obtained by postulating a sequence of priority-slots and assigning exactly one operation to each priority-slot, thus forming a sequence of operations. The number of priority-slots, postulated a priori by the user, corresponds to the length of the sequence of operations, which is the total number of operations that will be executed during the scheduling horizon.

A common logistics constraint appearing in the chemical industry is the non-overlapping constraint between two operations v and w, noted $v][w$. The logistics constraints (i), (ii), (iii), and (iv) from the previous section can all be expressed as non-overlapping constraints. Indeed, for the problem 1 (see Fig. 1), (i) is equivalent to 1][2, (ii) to 1][3, 1][4, 2][5, 2][6, 3][7, 4][8, 5][7, 6][8, and (iv) to 7][8.

Assuming that two non-overlapping operations v and w are assigned to priority-slots i and j such that $i < j$ (i has a higher scheduling priority than j), we define S_{iv} and S_{jw} as their respective start times, and D_{iv} and D_{jw} as their respective durations. As the operation v has the highest priority, operation w must start after the end of operation v:

$$S_{iv} + D_{iv} \leq S_{jw}$$

The model involves the following set of constraints (see [2] for details):

(a) Assignment constraints for each priority-slot,
(b) variable constraints given from the variable definitions,
(c) sequencing constraints such as cardinality constraints,
(d) scheduling constraints such as non-overlapping constraints,
(e) operation constraints such as flowrate limitations,
(f) and resource constraints such as reservoir capacity limitations.

The objective function Z can be expressed as follows:

$$\begin{aligned} Z = {} & SWITCHINGCOST \sum_{r \in CDU} \left(\sum_{i,v \in I_r} Z_{iv} - 1 \right) \\ & + UNLOADINGCOST \sum_{i,v \in UNLOAD} D_{iv} \\ & + SEAWAITINGCOST \sum_{i,v \in UNLOAD} W_{iv} \\ & + FIXEDSTORAGECOST \\ & + \sum_{i,r_1,r_2,v \in O_{r_1} \cap I_{r_2}} \left(\dot{C}_{r_2} - \dot{C}_{r_1} \right) \left(H - S_{iv} - \frac{D_{iv}}{2} \right) V_{iv} \end{aligned}$$

where:

- $Z_{iv} = 1$ if operation v is assigned to slot i, $Z_{iv} = 0$ otherwise
- S_{iv} is the start time of operation v if it is assigned to slot i, $S_{iv} = 0$ otherwise
- D_{iv} is the duration of operation v if it is assigned to slot i, $D_{iv} = 0$ otherwise
- V_{iv} is the volume of crude transferred during operation v if it is assigned to slot i, $V_{iv} = 0$ otherwise
- W_{iv} is the waiting time before unloading v if it is assigned to slot i, $W_{iv} = 0$ otherwise
- CDU is the set of CDUs
- I_r is the set of inlet operations for resource r
- O_r is the set of outlet operations for resource r
- $UNLOAD$ is the set of unloading operations
- $SWITCHINGCOST$ is the cost associated with each CDU switch
- $UNLOADINGCOST$ is the unloading cost rate
- $SEAWAITINGCOST$ is the sea waiting cost rate
- $FIXEDSTORAGECOST$ is the total cost for storing the initial refinery inventory during the horizon H if no crude transfer is performed

The last term, which involves bilinearities making the model nonconvex, evaluates the variation of the total storage cost. For each transfer operation v between resources r_1 (storage cost rate $\dot{C}_1$) and r_2 (storage cost rate $\dot{C}_2$) assigned to priority-slot i, the corresponding variation of storage cost is calculated as follows:

$$\begin{aligned}\Delta COST_{iv} &= \left(\dot{C}_{r_2} - \dot{C}_{r_1}\right)\left(\int_{t=S_{iv}}^{t=S_{iv}+D_{iv}} \frac{V_{iv}}{D_{iv}}(t - S_{iv})dt + \int_{t=S_{iv}+D_{iv}}^{t=H} V_{iv}dt\right) \\ &= \left(\dot{C}_{r_2} - \dot{C}_{r_1}\right)\left(\frac{1}{2}D_{iv}V_{iv} + (H - S_{iv} - D_{iv})\,V_{iv}\right) \\ &= \left(\dot{C}_{r_2} - \dot{C}_{r_1}\right)\left(H - S_{iv} - \frac{D_{iv}}{2}\right)V_{iv}\end{aligned}$$

Appendix A contains the complete mathematical formulation for the MINLP.

4 Solution Method

The non-convex MINLP model given in the previous section can be solved using a generic MINLP solver such as DICOPT (outer-approximation method) or BARON (global solver using a branch-and-reduce procedure). The former code,which is only rigorous for convex MINLP problems, may converge to a poor suboptimal solution. The latter can be prohibitively expensive to use because many nodes in the Branch & Bound tree are required to close the gap within a specified tolerance.

Therefore, a simple two-step procedure consisting of one MILP and one NLP subproblem has been implemented. It leads to good suboptimal solutions with an estimate of the optimality gap. In the first step, an MILP relaxation, described in the following section, is solved. The solution returned during this step may not satisfy all nonlinear constraints. In this case, the binary variables Z_{iv} are fixed, which means that the sequence of operations is fixed, and the resulting nonlinear programming (NLP) model is solved. This NLP model contains all constraints from the MINLP, including nonlinear constraints, with all discrete variables fixed. The solution obtained during this step might not be the optimum of the full model, but the optimality gap can be estimated from the lower bound given by the MILP solution and the upper bound given by the NLP solution.

The quality of the final solution obtained with this procedure, estimated by the optimality gap, strongly depends on the tightness of the MILP relaxation. It should also be noted that the NLP subproblem is nonconvex and thus may lead to different local optimal solutions depending on the starting point. It could also be locally infeasible although this did not occur in our experiments. For such a case, one could add integer cuts (see [10]) to the MILP model and restart the procedure until a solution is found. However, there is no proof that the solution obtained with this algorithm is globally optimal. Also, it cannot be ensured that a solution will be found, even if the MINLP has feasible solutions.

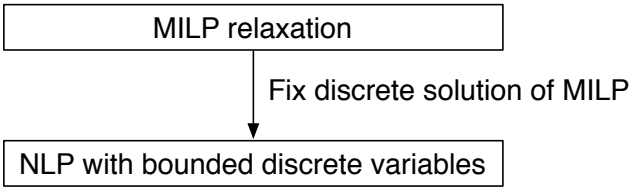


Fig. 2. Two-stage MILP-NLP solution method

5 Reformulation and Linear Relaxation

It is common to reformulate MINLP problems by introducing new variables to represent nonlinear terms. The MINLP model can then be rewritten as follows:

$$\begin{aligned}
\min Z = {} & SWITCHINGCOST \sum_{r \in CDU} \left(\sum_{i,v \in I_r} Z_{iv} - 1 \right) \\
& + UNLOADINGCOST \sum_{i,v \in UNLOAD} D_{iv} \\
& + SEAWAITINGCOST \sum_{i,v \in UNLOAD} W_{iv} \\
& + FIXEDSTORAGECOST \\
& + \sum_{i,r_1,r_2,v \in O_{r_1} \cap I_{r_2}} \left(\dot{C}_{r_2} - \dot{C}_{r_1} \right) X_{iv} \\
\text{s.t.} \quad & \begin{cases} X_{iv} = A_{iv} V_{iv} \\ A_{iv} = H - S_{iv} - \frac{D_{iv}}{2} \\ \text{constraints (1)-(26)} \end{cases}
\end{aligned}$$

where the variables X_{iv} represent the bilinear terms $A_{iv}V_{iv}$. Using McCormick convex envelopes introduced in [5], this MINLP can be linearly relaxed by replacing the constraint $X_{iv} = A_{iv}V_{iv}$ by:

$$\begin{aligned}
X_{iv} &\geq A_{iv}^L V_{iv} + A_{iv} V_{iv}^L - A_{iv}^L V_{iv}^L \\
X_{iv} &\geq A_{iv}^U V_{iv} + A_{iv} V_{iv}^U - A_{iv}^U V_{iv}^U \\
X_{iv} &\leq A_{iv}^L V_{iv} + A_{iv} V_{iv}^U - A_{iv}^L V_{iv}^U \\
X_{iv} &\leq A_{iv}^U V_{iv} + A_{iv} V_{iv}^L - A_{iv}^U V_{iv}^L
\end{aligned}$$

The terms A_{iv}^L, A_{iv}^U, V_{iv}^L, and V_{iv}^U represent the lower and upper bounds of variables A_{iv} and V_{iv}. It is important to note that the tightness of this linear relaxation strongly depends on the bounds of the variables A_{iv} and V_{iv}.

6 McCormick Cuts

Once the linear relaxation of the MINLP has been defined, the corresponding MILP can be solved by a Branch & Cut algorithm as implemented in the commercial tool Ilog Cplex. This approach consists in successively solving many subproblems (nodes) of the original MILP by solving its continuous relaxation and generating new subproblems when needed. User-defined constraints can also be added to each subproblem during the search using callbacks (see [11]).

A subproblem of the MILP is obtained by fixing some of the discrete variables Z_{iv} to either 0 or 1. Each of these subproblems correspond to a linear relaxation of the MINLP subproblem obtained by fixing the same discrete variables to the same values. If the LP relaxation of an MILP subproblem is integer feasible (i.e. all discrete variables take an integer value, whether it is fixed or not), it might still not satisfy all MINLP constraints. In such cases, it is possible to infer stronger McCormick constraints by contracting variables bounds for the current discrete solution.

We denote p a discrete solution defined by the discrete values taken by the variables Z_{iv}.The discrete solution p corresponds to a unique sequence of operations $(v_1^p, \ldots, v_n^p)$. We denote $(CP)^p$ the following CP model:

$$(CP)^p \begin{cases} \text{constraints (1)-(26)} & \\ Z_{iv} = 1 & \forall (i,v), v = v_i^p \\ Z_{iv} = 0 & \forall (i,v), v \neq v_i^p \end{cases}$$

When a discrete solution p is obtained at a given node of the search tree, Ilog Solver is used to perform constraint propagation on the model $(CP)^p$ leading to variable bounds $\left(A_{iv}^L\right)^p$, $\left(A_{iv}^U\right)^p$, $\left(V_{iv}^L\right)^p$, and $\left(V_{iv}^U\right)^p$ that are possibly tighter than the bounds defined at the root node (modeling stage). The McCormick constraints derived from these bounds are valid for the discrete solution p but are not valid for the current node as some variables Z_{iv} might not have been fixed yet. The MINLP subproblem corresponding to the discrete solution p is in fact a *restriction* of the MINLP subproblem corresponding to the current node. Therefore, the following modified big-M McCormick constraints are defined to be added to the current node subproblem:

$$X_{iv} \geq \left(A_{iv}^L\right)^p V_{iv} + A_{iv}\left(V_{iv}^L\right)^p - \left(A_{iv}^L\right)^p \left(V_{iv}^L\right)^p - M_1 \cdot (n - \sum_i Z_{iv_i^p})$$

$$X_{iv} \geq \left(A_{iv}^U\right)^p V_{iv} + A_{iv}\left(V_{iv}^U\right)^p - \left(A_{iv}^U\right)^p \left(V_{iv}^U\right)^p - M_2 \cdot (n - \sum_i Z_{iv_i^p})$$

$$X_{iv} \leq \left(A_{iv}^L\right)^p V_{iv} + A_{iv}\left(V_{iv}^U\right)^p - \left(A_{iv}^L\right)^p \left(V_{iv}^U\right)^p + M_3 \cdot (n - \sum_i Z_{iv_i^p})$$

$$X_{iv} \leq \left(A_{iv}^U\right)^p V_{iv} + A_{iv}\left(V_{iv}^L\right)^p - \left(A_{iv}^U\right)^p \left(V_{iv}^L\right)^p + M_4 \cdot (n - \sum_i Z_{iv_i^p})$$

where:

$$M_1 = \left(A_{iv}^L\right)^p V_{iv}^U + A_{iv}^U \left(V_{iv}^L\right)^p - \left(A_{iv}^L\right)^p \left(V_{iv}^L\right)^p$$
$$M_2 = \left(A_{iv}^U\right)^p V_{iv}^U + A_{iv}^U \left(V_{iv}^U\right)^p - \left(A_{iv}^U\right)^p \left(V_{iv}^U\right)^p$$
$$M_3 = A_{iv}^U V_{iv}^U - \left\{\left(A_{iv}^L\right)^p V_{iv}^L + A_{iv}^L \left(V_{iv}^U\right)^p - \left(A_{iv}^L\right)^p \left(V_{iv}^U\right)^p\right\}$$
$$M_4 = A_{iv}^U V_{iv}^U - \left\{\left(A_{iv}^U\right)^p V_{iv}^L + A_{iv}^L \left(V_{iv}^L\right)^p - \left(A_{iv}^U\right)^p \left(V_{iv}^L\right)^p\right\}$$

A lazy constraint callback (see [11]) has been implemented in order to generate these local McCormick cuts at each node where an integer feasible solution is found. If at least one McCormick cut is violated by the current discrete solution, it is added to the node subproblem and the LP relaxation is resolved as depicted in Fig. 3. This cut generation procedure can also be executed at nodes where no integer feasible solution is found. However, in order to reduce the computational expense corresponding to the generation of variable bounds and cuts, only integer feasible nodes are processed.

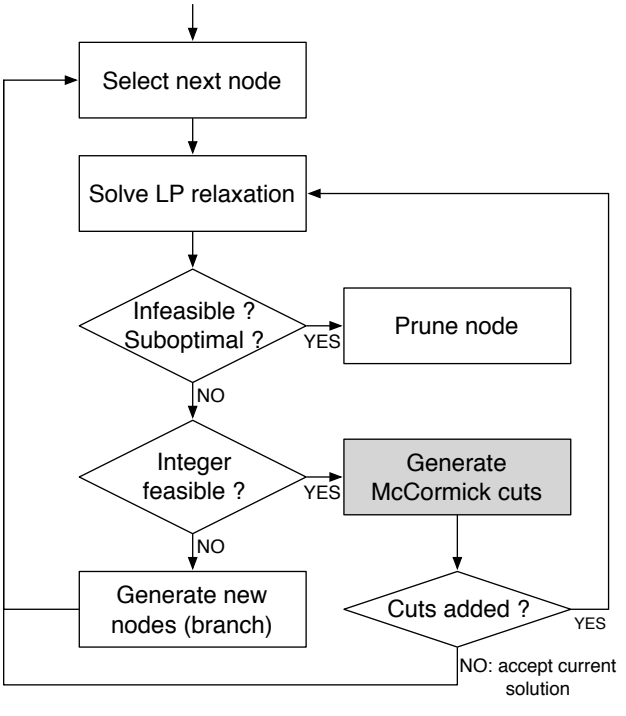


Fig. 3. Branch & Cut algorithm with McCormick cuts

7 Computational Results

The four problems introduced in [1] have been solved using two algorithms. The *BasicRelaxation* algorithm consists of initially adding McCormick constraints to the MILP model without generating new cuts during the search. The *ExtendedRelaxation* algorithm consists of adding McCormick constraints to the MILP and new cuts during the search, as explained in the previous section. Both approaches have been developed in C++ using Ilog Cplex 11.0 (MILP) and Ilog Solver 6.5 (CP). The NLPs have been solved using CONOPT 3. Experiments have been run on an Intel Core 2 Duo 2.16GHz processor.

In both basic and extended cases, Table 1 gives the solution of the MILP and the NLP, the corresponding optimality gap, and the total CPU time and the number of nodes explored during the search. In the ExtendedRelaxation case, the computational time for generating McCormick cuts using CP is also displayed in column "CP", it is already included in the total CPU time. The computational time for solving NLPs is not reported as it is always lower than 5s.

The results show that using the approach developed in this paper leads to important reductions of the optimality gap compared to the *BasicRelaxation* algorithm (3.48% vs 14.83% average gap). Besides, a better feasible solution (3.3% cost reduction) has been found for problem 2 using the two-step MILP-NLP procedure. This demonstrates how the linear relaxation of the MINLP can been tightened by adding McCormick constraints to the subproblems of the MILP leading to integer feasible solutions. More precisely, at the node where the optimal MILP solution of problem 1 is found, two rounds of cuts are added. The first round of cuts leads to an increase of the objective value from 199.1 to 212.4 (6.7% increase), while the second round increases the objective value to 213.7 (0.6% increase). This indicates that few rounds of cuts are necessary in this case and that the first round of cuts is the most important in order to tighten the MILP relaxation.

The optimality gaps obtained in the *BasicRelaxation* case are much larger than the optimality gaps obtained in [2]. This is due to the fact that the objective function considered in this previous work is linear, thus leading to a tighter MILP relaxation.

In terms of computational expense, the required CPU time increases by 9.5% for the *ExtendedRelaxation* procedure. This is due to the increase of the number of nodes explored in some cases (problems 2 and 3), the increase of the model

Table 1. Results obtained with BasicRelaxation and ExtendedRelaxation algorithms

	BasicRelaxation					ExtendedRelaxation					
Pb	MILP			NLP	Gap	MILP				NLP	Gap
	Solution	CPU	Nodes	Solution		Solution	CPU	CP	Nodes	Solution	
1	199.1	9s	22	222.3	11.7%	213.7	11s	1s	20	222.3	4.0%
2	297.8	215s	55	362.9	21.9%	343.1	246s	19s	57	351.2	2.4%
3	254.6	224s	73	287.6	13.0%	269.2	337s	16s	95	287.6	6.8%
4	331.8	600s	20	374.0	12.7%	371.3	554s	18s	19	374.0	0.7%

Table 2. Results obtained with several algorithms on problems 1 and 2

	Problem 1			Problem 2		
Algorithm	Solution	CPU time	Gap	Solution	CPU time	Gap
BasicRelaxation	222.3	9s	11.7%	362.9	215s	21.9%
ExtendedRelaxation	222.3	11s	4.0%	351.2	246s	2.4%
DICOPT	233.5	14s	-	351.2	1235s	-
sBB	Local Infeas.	14s	-	Local Infeas.	697s	-
Bonmin-OA	222.3	27s	-	No solution	+3600s	-
AlphaECP	222.3	260s	-	358.0	+3600s	-
BARON	222.3	+3600s	4.1%	No solution	+3600s	-

size for subproblems for which McCormick constraints have been generated, and the time used for performing constraint propagation on the CP model (4.7% of total CPU time). This last point can be improved with a better CP model and a more efficient implementation.

Table 2 shows computational results obtained with several algorithm on problem 1. Local optimizers DICOPT, sBB, Bonmin and AlphaECP have been tested as well as the global solver BARON. DICOPT and sBB did not return the best known solution, sBB failed to find any feasible solution. However, the corresponding CPU times are similar to the BasicRelaxation and ExtendedRelaxation procedures. Bonmin-OA found the best-known solution in reasonable time. AlphaECP and BARON also found the best known solution, but the former requires one order of magnitude increase in CPU time and does not give any optimality gap estimate, while the latter requires more than two orders of magnitude increase in CPU time (optimization is stopped after 1 hour) but has the smallest optimality gap.

8 Conclusion

We have presented a new approach for handling bilinear terms in MINLPs. It involves using CP bound contraction techniques in order to generate cuts based on McCormick convex envelopes for products of continuous variables. The procedure has the advantage of being implemented with the commercial solver (Ilog Cplex and Solver). This allows using their complementary strengths to obtain better solutions, reduce the optimality gap, with a reasonable increase of computational expense.

This approach may be improved by extending the interaction between MILP and CP as in the programming system SCIP (see [12]). Also, MINLP cuts such as McCormick cuts can be generated not only for integer feasible subproblems, but for other subproblems, thus pruning additional nodes during the search. Finally, optimality-based reduction techniques (see [9]) can be used to add new constraints to the CP model to remove feasible solutions that are not optimal. As a consequence variable bounds may be further contracted leading to a tighter MILP relaxation.

References

1. Lee, H., Pinto, J.M., Grossmann, I.E., Park, S.: Mixed-integer linear programming model for refinery short-term scheduling of crude oil unloading with inventory management. Industrial and Engineering Chemistry Research 35(5), 1630–1641 (1996)
2. Mouret, S., Grossmann, I.E., Pestiaux, P.: A novel priority-slot based continuous-time formulation for crude-oil scheduling problems. Industrial and Engineering Chemistry Research (to appear)
3. Jia, Z., Ierapetritou, M.G., Kelly, J.D.: Refinery short-term scheduling using continuous time formulation: Crude-oil operations. Industrial and Engineering Chemistry Research 42(13), 3085–3097 (2003)
4. Karuppiah, R., Furman, K.C., Grossmann, I.E.: Global optimization for scheduling refinery crude oil operations. Computers and Chemical Engineering 32(11), 2745–2766 (2008)
5. McCormick, G.P.: Computability of global solutions to factorable nonconvex programs: Part 1 - convex underestimating problems. Mathematical Programming 10, 147–175 (1976)
6. Bergamini, M.L., Grossmann, I.E., Scenna, N., Aguire, P.: An improved piecewise outer-approximation algorithm for the global optimization of minlp models involving concave and bilinear terms. Computers and Chemical Engineering 32, 477–493 (2008)
7. Wicaksono, D.S., Karimi, I.A.: Piecewise milp under- and overestimators for global optimization of bilinear programs. AIChE Journal 54(4), 991–1008 (2008)
8. Pham, V., Laird, C.D., El-Halwagi, M.: Convex hull discretization approach to the global optimization of pooling problems. Industrial and Engineering Chemistry Research (2009) (to be published)
9. Sahinidis, N.V.: Global Optimization and Constraint Satisfaction: The Branch-and-Reduce Approach. In: Bliek, C., Jermann, C., Neumaier, A. (eds.) COCOS 2002. LNCS, vol. 2861, pp. 1–16. Springer, Heidelberg (2003)
10. Balas, E., Jeroslow, R.G.: Canonical cuts on the unit hypercube. SIAM Journal on Applied Mathematics 23(1), 61–69 (1972)
11. ILOG Inc.: ILOG CPLEX 11.0 User's Manual (September 2007)
12. Achterberg, T.: Scip - a framework to integrate constraint and mixed integer programming. Technical report, Zuse Institute Berlin (2004)

Appendix A: MINLP Model [2]

Sets

- $T = \{1, \ldots, n\}$ is the set of priority-slots
- W is the set of all operations
- $W_U \subset W$ is the set of unloading operations
- $W_T \subset W$ is the set of tank-to-tank transfer operations
- $W_D \subset W$ is the set of distillation operations
- R is the set of resources (i.e. reservoirs)
- $R_V \subset R$ is the set of vessels
- $R_S \subset R$ is the set of storage tanks

- $R_C \subset R$ is the set of charging tanks
- $R_D \subset R$ is the set of distillation units
- $I_r \subset W$ is the set of inlet transfer operations on resource r
- $O_r \subset W$ is the set of outlet transfer operations on resource r
- C is the set of products (i.e. crudes)
- K is the set of product properties (e.g. crude sulfur concentration)

Parameters

- H is the scheduling horizon
- $[\underline{N_D}, \overline{N_D}]$ are the bounds on the number of distillation
- $[\underline{FR_v}, \overline{FR_v}]$ are the bounds on the flowrate of transfer operation v
- S_r is the arrival time of vessel r
- V_r is the minimum volume transferred during unloading of vessel r
- $[\underline{x_{vk}}, \overline{x_{vk}}]$ are the limits of property k of the blended products transferred during operation v
- x_{ck} is the value of the property k of crude c
- $[\underline{L_r}, \overline{L_r}]$ are the capacity limits of reservoir r
- $[\underline{D_r}, \overline{D_r}]$ are the bounds of the demand on products to be transferred out of the charging tank r during the scheduling horizon

Minimize

$$\begin{aligned} Z = {} & SWITCHINGCOST \sum_{r \in CDU} \left(\sum_{i,v \in I_r} Z_{iv} - 1 \right) \\ & + UNLOADINGCOST \sum_{i,v \in UNLOAD} D_{iv} \\ & + SEAWAITINGCOST \sum_{i,v \in UNLOAD} WAIT_{iv} \\ & + FIXEDSTORAGECOST \\ & + \sum_{i,r_1,r_2,v \in O_{r_1} \cap I_{r_2}} \left(\dot{C}_{r_2} - \dot{C}_{r_1} \right) \left(H - S_{iv} - \frac{D_{iv}}{2} \right) V_{iv} \end{aligned}$$

s.t.

- Assignment constraints:

$$\sum_{v \in W} Z_{iv} = 1 \qquad i \in T \tag{1}$$

– Variable constraints

$$S_{iv} + D_{iv} \leq H \cdot Z_{iv} \qquad i \in T, v \in W \tag{2}$$

$$V_{iv} \leq \overline{V_v} \cdot Z_{iv} \qquad i \in T, v \in W \tag{3}$$

$$\sum_{c \in C} V_{ivc} = V_{iv} \qquad i \in T, v \in W \tag{4}$$

$$L_{ir} = L_{0r} + \sum_{j \in T, j<i} \sum_{v \in I_r} V_{iv} - \sum_{j \in T, j<i} \sum_{v \in O_r} V_{iv} \qquad i \in T, r \in R \tag{5}$$

$$L_{irc} = L_{0rc} + \sum_{j \in T, j<i} \sum_{v \in I_r} V_{ivc} - \sum_{j \in T, j<i} \sum_{v \in O_r} V_{ivc} \quad i \in T, r \in R, c \in C \tag{6}$$

– Sequencing constraints:

$$\sum_{i \in T} \sum_{v \in O_r} Z_{iv} = 1 \qquad r \in R_V \tag{7}$$

$$\underline{N_D} \leq \sum_{i \in T} \sum_{v \in W_D} Z_{iv} \leq \overline{N_D} \tag{8}$$

$$\sum_{j \in T, j<i} \sum_{v \in O_{r_2}} Z_{jv} + \sum_{j \in T, j \geq i} \sum_{v \in O_{r_1}} Z_{jv} \leq 1 \quad i \in T, i \neq 1, r_1, r_2 \in R_V, r_1 < r_2 \tag{9}$$

– Scheduling constraints:

$$\sum_{v \in W_U} (S_{iv} + D_{iv}) \leq \sum_{v \in W_U} S_{jv} + H \cdot (1 - \sum_{v \in W_U} Z_{jv}) \qquad i, j \in T, i < j \tag{10}$$

$$\sum_{v \in I_r} (S_{iv} + D_{iv}) \leq \sum_{v \in O_r} S_{jv} + H \cdot (1 - \sum_{v \in O_r} Z_{jv}) \qquad i, j \in T, i < j, r \in R_S \cup R_C \tag{11}$$

$$\sum_{v \in O_r} (S_{iv} + D_{iv}) \leq \sum_{v \in I_r} S_{jv} + H \cdot (1 - \sum_{v \in I_r} Z_{jv}) \qquad i, j \in T, i < j, r \in R_S \cup R_C \tag{12}$$

$$\sum_{v \in O_r} (S_{iv} + D_{iv}) \leq \sum_{v \in O_r} S_{jv} + H \cdot (1 - \sum_{v \in O_r} Z_{jv}) \qquad i, j \in T, i < j, r \in R_C \tag{13}$$

$$\sum_{v \in I_r} (S_{iv} + D_{iv}) \leq \sum_{v \in I_r} S_{jv} + H \cdot (1 - \sum_{v \in I_r} Z_{jv}) \quad i, j \in T, i < j, r \in R_D \tag{14}$$

$$S_{iv} + D_{iv} \leq S_{jv} + H \cdot (1 - Z_{jv}) \quad i, j \in T, i < j, v \in W \tag{15}$$

$$\sum_{i \in T} \sum_{v \in I_r} D_{iv} = H \quad r \in R_D \tag{16}$$

$$S_{iv} = S_r \cdot Z_{iv} + W_{iv} \quad i \in T, r \in R_V, v \in O_r \tag{17}$$

– Operation constraints:

$$\underline{FR_v} \cdot D_{iv} \leq V_{iv} \leq \overline{FR_v} \cdot D_{iv} \quad i \in T, v \in W \tag{18}$$

$$V_{iv} \geq V_r \cdot Z_{iv} \quad i \in T, v \in W \tag{19}$$

$$\underline{x_{vk}} \cdot V_{iv} \leq \sum_{c \in C} x_{ck} V_{ivc} \leq \overline{x_{vk}} \cdot V_{iv} \quad i \in T, v \in W, k \in K \tag{20}$$

$$\frac{L_{irc}}{L_{ir}} = \frac{V_{ivc}}{V_{iv}} \text{ (ignored in MILP relaxation)} \quad i \in T, r \in R, v \in O_r, c \in C \tag{21}$$

– Resource constraints:

$$\underline{L_r} \leq L_{ir} \leq \overline{L_r} \quad i \in T, r \in R_S \cup R_C \tag{22}$$

$$0 \leq L_{irc} \leq \overline{L_r} \quad i \in T, r \in R_S \cup R_C, c \in C \tag{23}$$

$$\underline{L_r} \leq L_{0r} + \sum_{i \in T} \sum_{v \in I_r} V_{iv} - \sum_{i \in T} \sum_{v \in O_r} V_{iv} \leq \overline{L_r} \quad r \in R_S \cup R_C \tag{24}$$

$$0 \leq L_{0rc} + \sum_{i \in T} \sum_{v \in I_r} V_{ivc} - \sum_{i \in T} \sum_{v \in O_r} V_{ivc} \leq \overline{L_r} \quad r \in R_S \cup R_C, c \in C \tag{25}$$

$$\underline{D_r} \leq \sum_{i \in T} \sum_{v \in O_r} V_{iv} \leq \overline{D_r} \quad r \in R_C \tag{26}$$

– Symmetry-breaking constraints with deterministic finite automaton $M = (Q, \Sigma, \delta, q_1, F)$ (see [2]):

$$\sum_{q} S_{ivq} = Z_{iv} \qquad i \in T, v \in W \tag{27}$$

$$\sum_{v} S_{1vq_1} = 1 \tag{28}$$

$$\sum_{v,q',q=\delta(q',v)} S_{(i-1)vq'} - \sum_{v} S_{ivq} = 0 \qquad i \in T, i \neq 1, q \in Q \tag{29}$$

$$\sum_{v,q,\delta(q,v)\in F} S_{nvq} = 1 \tag{30}$$

The Polytope of Context-Free Grammar Constraints

Gilles Pesant[1], Claude-Guy Quimper[2], Louis-Martin Rousseau[1], and Meinolf Sellmann[3]

[1] Ecole Polytechnique de Montreal
Montreal, Canada
{Gilles.Pesant,Louis-Martin.Rousseau}@polymtl.ca
[2] Google Inc.
Waterloo, Canada
cquimper@gmail.com
[3] Brown University
Department of Computer Science
115 Waterman Street, P.O. Box 1910
Providence, RI 02912
sello@cs.brown.edu

Abstract. Context-free grammar constraints enforce that a sequence of variables forms a word in a language defined by a context-free grammar. The constraint has received a lot of attention in the last few years as it represents an effective and highly expressive modeling entity. Its application has been studied in the field of Constraint Programming, Mixed Integer Programming, and SAT to solve complex decision problems such as shift scheduling. In this theoretical study we demonstrate how the constraint can be linearized efficiently. In particular, we propose a lifted polytope which has only integer extreme points. Based on this result, for shift scheduling problems we prove the equivalence of Dantzig's original set covering model and a lately introduced grammar-based model.

Keywords: grammar constraints, polytope.

1 Introduction

Global constraints capture natural substructures of common combinatorial optimization or satisfaction problems. They facilitate the modeling process and, at the same time, offer the solver awareness of structures that it can exploit to boost performance. In constraint programming, global constraints allow greater filtering effectiveness. In integer programming, they offer the possibility to linearize specific structures more effectively than a non-expert is able to achieve. Systems like SCIP [1] and SIMPL [12] linearize entire substructures automatically and effectively.

In this short theoretical study, we show how context-free grammar constraints can be linearized effectively. We prove that the linearization proposed in [3] results in a polytope which has integer feasible corners only, thus giving us the convex hull of all integer feasible points and allowing us to obtain tight linear relaxation models when grammar constraints are used in combination with other constraints.

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 223–232, 2009.

2 Basic Concepts

We start by reviewing some well-known definitions from the theory of formal languages and the existing algorithm for filtering context-free grammar constraints. For a full introduction, we refer the interested reader to [5] and [11] where all the proofs that are omitted in this paper can be found.

Definition 1 (Alphabet and Words). *Given sets Z, Z_1, and Z_2, with Z_1Z_2 we denote the set of all* sequences *or* strings $z = z_1z_2$ *with $z_1 \in Z_1$ and $z_2 \in Z_2$, and we call Z_1Z_2 the* concatenation *of Z_1 and Z_2. Then, for all $n \in \mathbb{N}$ we denote by Z^n the set of all sequences $z = z_1z_2 \dots z_n$ with $z_i \in Z$ for all $1 \leq i \leq n$. We call z a* word *of length n, and Z is called an* alphabet *or set of* letters*. The empty word has length 0 and is denoted by ε. It is the only member of Z^0. We denote the set of all words over the alphabet Z by $Z^* := \bigcup_{n\in\mathbb{N}} Z^n$. In case that we wish to exclude the empty word, we write $Z^+ := \bigcup_{n\geq 1} Z^n$.*

Definition 2 (Context-Free Grammars). *A* grammar *is a tuple $G = (\Sigma, N, P, S_0)$ where Σ is the alphabet, N is a finite set of* non-terminals*, $P \subseteq (N \cup \Sigma)^*N(N \cup \Sigma)^* \times (N \cup \Sigma)^*$ is the set of* productions*, and $S_0 \in N$ is the start non-terminal. We will always assume that $N \cap \Sigma = \emptyset$. Given a grammar $G = (\Sigma, N, P, S_0)$ such that $P \subseteq N \times (N \cup \Sigma)^*$, we say that the grammar G and the language L_G are* context-free*. A context-free grammar $G = (\Sigma, N, P, S_0)$ is said to be in* Chomsky Normal Form *if and only if for all productions $(A \to \alpha) \in P$ we have that $\alpha \in \Sigma^1 \cup N^2$. Without loss of generality, we will then assume that each literal $a \in \Sigma$ is associated with exactly one unique non-literal $A_a \in N$ such that $(B \to a) \in P$ implies that $B = A_a$ and $(A_a \to b) \in P$ implies that $a = b$.*

Remark 1. Throughout the paper, we will use the following convention: Capital letters A, B, C, D, and E denote non-terminals, lower case letters a, b, c, d, and e denote letters in Σ, Y and Z denote symbols that can either be letters or non-terminals, u, v, w, x, y, and z denote strings of letters, and α, β, and γ denote strings of letters and non-terminals. Moreover, productions (α, β) in P can also be written as $\alpha \to \beta$.

Definition 3 (Derivation and Language).

- *Given a grammar $G = (\Sigma, N, P, S_0)$, we write $\alpha\beta_1\gamma \underset{G}{\Rightarrow} \alpha\beta_2\gamma$ if and only if there exists a production $(\beta_1 \to \beta_2) \in P$. We write $\alpha_1 \overset{*}{\underset{G}{\Rightarrow}} \alpha_m$ if and only if there exists a sequence of strings $\alpha_2, \dots, \alpha_{m-1}$ such that $\alpha_i \underset{G}{\Rightarrow} \alpha_{i+1}$ for all $1 \leq i < m$. Then, we say that α_m can be* derived *from α_1.*
- *The language given by G is $L_G := \{w \in \Sigma^* \mid S_0 \overset{*}{\underset{G}{\Rightarrow}} w\}$.*

2.1 Context-Free Grammar Constraints

Based on the concepts above, we review the definition of context-free grammar constraints introduced in [11]:

Definition 4 (Grammar Constraint). *For a given grammar $G = (\Sigma, N, P, S_0)$ and variables $X_1, \ldots, X_n$ with domains $D_1 := D(X_1), \ldots, D_n := D(X_n) \subseteq \Sigma$, we say that $Grammar_G(X_1, \ldots, X_n)$ is true for an instantiation $X_1 \leftarrow w_1, \ldots, X_n \leftarrow w_n$ if and only if it holds that $w = w_1 \ldots w_n \in L_G \cap D_1 \times \cdots \times D_n$. We denote a given grammar constraint $Grammar_G(X_1, \ldots, X_n)$ over a context-free grammar G in Chomsky Normal Form by $CFGC_G(X_1, \ldots, X_n)$.*

The filtering algorithm for $CFGC_G$ presented in [11] is based on the parsing algorithm from Cooke, Younger, and Kasami (CYK). CYK works as follows: Given a word $w \in \Sigma^n$, let us denote the subsequence of letters starting at position i with length j (that is, $w_i w_{i+1} \ldots w_{i+j-1}$) by w_{ij}. Based on a grammar $G = (\Sigma, N, P, S_0)$ in Chomsky Normal Form, CYK determines iteratively the set of all non-terminals from which we can derive w_{ij}, i.e. $S_{ij} := \{A \in N \mid A \overset{*}{\underset{G}{\Rightarrow}} w_{ij}\}$ for all $1 \leq i \leq n$ and $1 \leq j \leq n - i$. It is easy to initialize the sets S_{i1} just based on w_i and all productions $(A \to w_i) \in P$. Then, for j from 2 to n and i from 1 to $n - j + 1$, we have that

$$S_{ij} = \bigcup_{k=1}^{j-1} \{A \mid (A \to BC) \in P \text{ with } B \in S_{ik} \text{ and } C \in S_{i+k,j-k}\}. \tag{1}$$

Then, $w \in L_G$ if and only if $S_0 \in S_{1n}$. From the recursion equation it is simple to derive that CYK can be implemented to run in time $O(n^3|G|) = O(n^3)$ when we treat the size of the grammar as a constant.

The filtering algorithm for $CFGC_G$ that we sketch in Algorithm 1 works bottom-up by computing the sets S_{ij} for increasing j after initializing S_{i1} with all non-terminals that can produce in one step a terminal in the domains of X_i. Then, the algorithm works top-down by removing all non-terminals from each set S_{ij} which cannot be reached from $S_0 \in S_{1n}$. In [11], we showed:

Theorem 1. *Algorithm 1 achieves generalized arc-consistency for the $CFGC$ and requires time and space cubic in the number of variables.*

Example 1. Assume we are given the following context-free, normal-form grammar $G = (\{], [\}, \{A, B, C, S_0\}, \{S_0 \to AC, S_0 \to S_0 S_0, S_0 \to BC, B \to AS_0, A \to [\ , C \to\ ]\ \}, S_0)$that gives the language L_G of all correctly bracketed expressions (like,

Algorithm 1. CFGC Filtering Algorithm

1. We run the dynamic program based on recursion equation (1) with initial sets $S_{i1} := \{A \mid (A \to v) \in P, v \in D_i\}$.
2. We define the directed graph $Q = (V, E)$ with node set $V := \{v_{ijA} \mid A \in S_{ij}\}$ and arc set $E := E_1 \cup E_2$ with $E_1 := \{(v_{ijA}, v_{ikB}) \mid \exists\ C \in S_{i+k,j-k} :\ (A \to BC) \in P\}$ and $E_2 := \{(v_{ijA}, v_{i+k,j-k,C}) \mid \exists\ B \in S_{ik} :\ (A \to BC) \in P\}$ (see Figure 1).
3. Now, we remove all nodes and arcs from Q that cannot be reached from v_{1nS_0} and denote the resulting graph by $Q' := (V', E')$.
4. We define $S'_{ij} := \{A \mid v_{ijA} \in V'\} \subseteq S_{ij}$, and set $D'_i := \{v \mid \exists\ A \in S'_{i1} :\ (A \to v) \in P\}$.

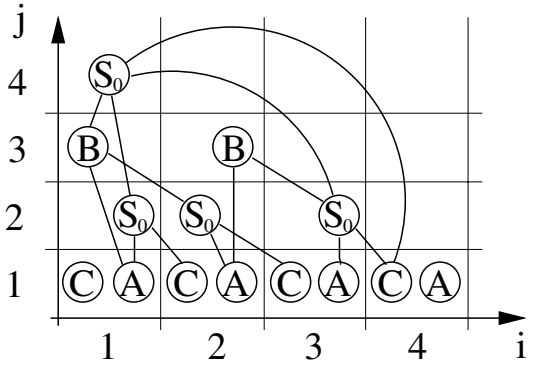


Fig. 1. The picture shows sets S_{ij} in Algorithm 1

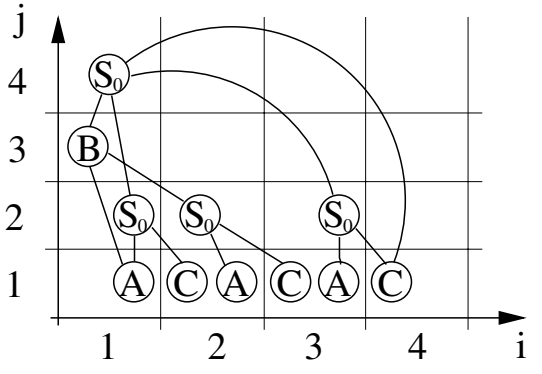


Fig. 2. The left picture shows the sets S'_{ij} in Algorithm 1. We see that the constraint filtering algorithm determines that the word may not start with a closing, nor end with an opening bracket.

for example, "[[][]]" or "[][[]]"). In Figures 1 and 2, we illustrate how Algorithm 1 works when the initial domain of all domains are $D_1 = \cdots = D_4 = \{[,]\}$: First, we work bottom-up, adding non-terminals to the sets S_{ij} if they allow to generate a word in $D_i \times \cdots \times D_{i+j-1}$. Then, in the second step, we work top-down and remove all non-terminals that cannot be reached from $S_0 \in S_{1n}$.

3 Linearization of Context-Free Grammar Constraints

In [9] an And/Or representation of the graph constructed in Algorithm 1 was given. The idea is to split each node v_{ijA^1} in the original graph into one *Or-node* $N^{or}_{ijA^1}$ which receives all incoming arcs of the original node. This Or-node then connects to *And-nodes* $N^{and}_{ijA^1A^2A^3k}$ for all productions $(A^1 \rightarrow A^2A^3) \in P$ and $k < j$. Each $N^{and}_{ijA^1A^2A^3k}$ then connects to $N^{or}_{ikA^2}$ and $N^{or}_{i+k,j-k,A^3}$. For Or-nodes $N^{or}_{i1A_a}$ on the lowest level, we add one And-node N^{and}_{i1a}, whereby we assume that the only production in P with non-terminal A_a on the left-hand side and a on the right is $(A_a \rightarrow a)$. In Figure 3 we show how the graph in Figure 2 is transformed in this way.

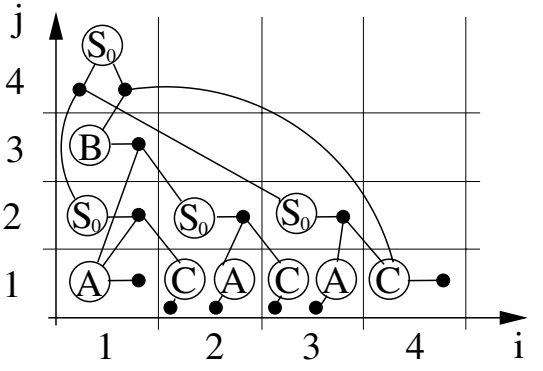


Fig. 3. The picture shows the modified And/Or-graph. The solid nodes denote And-nodes.

Based on this And/Or representation, in [3] a linearization of context-free grammar constraints was introduced which works as follows. A boolean variable is introduced for each node (AND/OR) of the graph, taking a value of 1 when the associated node is *true*. In the following, N^{or} and N^{and} respectively refer to the sets of all Or-nodes and And-nodes. Letting $u \in N^{or}$ (resp. $v \in N^{and}$) be an Or-node (resp. And-node) then o_u (resp. a_v) refers to its associated 0-1 variable. We also define $p(u)$ the set of u's parent nodes and, when not considering a leaf node, $c(u)$ the set of u's children nodes. The constraint in the associated MIP model was formulated as:

$$\sum_{v \in c(u)} a_v = o_u \quad \forall u \in N^{or} \tag{2}$$

$$\sum_{v \in p(u)} a_v = o_u \quad \forall u \in N^{or} \tag{3}$$

$$a_{1,n,S_0} = 1 \tag{4}$$

$$o_u, a_v \in \{0,1\} \qquad \forall u \in N^{or}, v \in N^{and} \tag{5}$$

The main difference between the MIP model and the AND/OR graph is that if there exist more than one parsing tree for a sequence, all nodes in the AND/OR graph that belong to at least one parsing tree are set to *true*, while for the MIP, one parsing tree is arbitrarily selected and only its variables are set to one. All other variables, including those that belong to other parsing trees, are set to zero. Choosing an arbitrary parsing tree simplifies the MIP without changing the solution space. Indeed, only one parsing tree is necessary to prove that a sequence belongs to a context-free language.

This model can however be simplified: we can remove the variable associated with Or-nodes (since (2) and (3) can be combined) and use non-negative variables instead of binary ones (as (4) imposes an upper bound on all variables). These changes will also significantly simplify the proof that all extreme points of the defined polytope are integer. The new model becomes:

$$\sum_{v \in c(u)} a_v = \sum_{v \in p(u)} a_v \quad \forall u \in N^{or} \tag{6}$$

$$a_{1,n,S_0} = 1 \tag{7}$$

$$a_v \geq 0, \qquad \forall v \in N^{and} \tag{8}$$

It is also possible to define the grammar polytope directly from the signature of the constraint, that is without referring to the AND/OR graph. The polytope is then defined as follows:

$$\text{IP} - \text{CF} = \min \sum c_{ijABCk} a_{ijABCk} + \sum c_{i1a} a_{i1a} \tag{9}$$

s.t.

$$\sum_{\substack{(A \to BC) \in P \\ 1 \leq k < j}} a_{ijABCk} = \sum_{\substack{(B \to AC) \in P \\ j < k < n}} a_{ikBACk} + \sum_{\substack{(B \to CA) \in P \\ j < k < n}} a_{i+j-k,k,BCAk} \qquad \begin{array}{l} \forall\, i, j \geq 1 \\ \forall\, A \in S_{ij} \\ \quad i+j \leq n+1 \end{array} \tag{10}$$

$$\sum_{(A_a \to a) \in P} a_{i1a} = \sum_{\substack{(B \to A_a C) \in P \\ 1 < k < n}} a_{ikBACk} + \sum_{\substack{(B \to C A_a) \in P \\ 1 < k < n}} a_{i+1-k,k,BCAk} \qquad \begin{array}{l} \forall\, i \in 1..n \\ \forall\, A_a \in S_{i1} \end{array} \tag{11}$$

$$1 = \sum_{\substack{(S_0 \to BC) \in P \\ 1 \leq k < n}} a_{1nS_0BCk} \tag{12}$$

$$0 \leq a_{ijABCk}, a_{i1a} \qquad \begin{array}{l} \forall\, A_a \in S_{i1} \\ \forall\, i, j \geq 1 \\ \quad i + j \leq n \end{array} \tag{13}$$

Example 2. Continuing with the bracketing example (Example 1), we illustrate the linearization of the corresponding `grammar` constraint. For the graph depicted in Figure 3, among other constraints, according to Equation 6, we enforce

$$a_{14S_0BC3} + a_{32S_0AC1} = a_{41]},$$

and according to Equation 4, we enforce

$$a_{14S_0BC3} + a_{14S_0S_0S_02} = 1.$$

4 The Context-Free Grammar Polytope

We now state and prove the main result of this theoretical study. Given a grammar constraint, the polytope defined by (6)-(8) has the following property:

Theorem 2. *The linearization IP-CF of a given context-free grammar constraint has integer-feasible corners only.*

Proof. We can write IP-CF in the form minimize $c^T a$ such that $Aa = b$, $a \geq 0$. Let a^1 denote an optimal solution to IP-CF, and u^1 the corresponding optimal dual solution. According to the well-known complementary slackness conditions, we know that all a with $Aa = b$ and $a^T(A^T u^1 - c) = 0$ are optimal. The complementary slackness conditions ensure that at optimality, we have $a_i^T = 0$ or $A_i^T u^1 - c_i = 0$ for every i. It is therefore sufficient to construct an integer-feasible solution a^0 to $Aa = b$ for which $A_i^T u^1 < c_i$ implies $a_i^0 = 0$. In particular, we want to construct an integer-feasible solution a^0 for which a_i^0 is greater than zero only if the non-integer solution satisfies $a_i^1 > 0$ or equivalently $A_i^T u^1 = c_i$. That is, it is sufficient to construct an integer-feasible solution a^0 to $Aa = b$ whose support (the set of variables that take non-zero values) is a subset of the support of a^1.

Let us consider any node in cell $S_1 n$ for which $a^1_{1nS_0BCk} > 0$ (at least one such node must exist according to Equation 12), and set $a^0_{1nS_0BCk} = 1$. Now, since the Or-nodes o_{1kB} and $o_{k,n-k,C}$ received a positive "flow" according to a^1, according to Equations 10 there must exist some And-nodes for which $a^1_{1kBDEh}, a^1_{k,n-k,CFGl} > 0$. Again, we set $a^1_{1kBDEh} = 1$ and $a^1_{k,n-k,CFGl} = 1$ and continue until we fade out at the bottom level. All other variables are set to zero. The tree of And-nodes which we constructed obeys all constraints in IP-CF. Moreover, we only utilized And-nodes that were already used in the fractional solution a^1. Consequently, a^0 obeys the complementary slackness conditions with respect to u^1 and is thus optimal. □

The result applies to models where costs are associated with variables taking specific values as studied in [6], but also models where using certain productions incurs specific costs as studied in [7].

5 Implication for Shift Scheduling Problems

We illustrate the use of grammar constraints in the domain of shift scheduling. Given a planning horizon divided into periods of equal length, a set of employees and a demand for different activities (work activities, lunch, break, rest) at each period, the shift scheduling problem consists in assigning one activity to each employee in each period in such a way that the demands are met. In this context, a *shift* is a sequence of activities corresponding to a continuous presence at work (that may include lunch and break, but not rest periods). The original objective (introduced by Dantzig in [2]) is to build a set of shifts that minimize labor costs while meeting labor regulations.

Given that Ω is the set of legal shifts, T the set of all time periods, d_i the required number of employees at time i, c_s the cost of shift s (equal to a weighted sum of the number of periods it covers), a_{is} an indicator specifying whether s covers time period i, and x_s is an integer variable that represents the number of employees assigned to s, we can state the original model of [2] as follows:

$$SC = \min \sum_{s \in \Omega} c_s x_s \tag{14}$$

$$s.t. \sum_{s \in \Omega} a_{is} x_s \geq d_i \quad \forall i \in T \tag{15}$$

$$x_s \geq 0, \text{integer} \quad \forall s \in \Omega \tag{16}$$

The most common methods used to solve this problem ([4]) are various set covering heuristics, which select from a set of *potentially* good shifts, the ones that together generate the best schedule. In order to generate an optimal solution the set covering model has to be defined over the *entire* set of possible shifts and solved through branch and price when Ω is too large.

For the purpose of comparison (and also many other practical issues that involve the need to track individual employees) this model can be disaggregated to identify the shift performed by each employee in set E. The boolean decision variable x_{se} indicates whether employee e performs shift s.

$$SCe = \min \sum_{s \in \Omega, e \in E} c_s x_{se} \tag{17}$$

$$s.t. \sum_{s \in \Omega, e \in E} a_{is} x_{se} \geq d_i \quad \forall i \in T \tag{18}$$

$$\sum_{s \in \Omega} x_{se} = 1 \quad \forall e \in E \tag{19}$$

$$x_{se} \in \{0, 1\} \quad \forall s \in \Omega, e \in E \tag{20}$$

In [3] Coté et al. showed that one could express implicitly the set of all shifts using a simple context-free grammar imposed on the sequence of fine grained decisions y_{ie} (employee e works at time i). At a high level, the model was:

$$SCg = \min \sum_{i \in T, e \in E} c_i y_{ie} \tag{21}$$

$$s.t. \sum_{e \in E} y_{ie} \geq d_i \ \ \forall i \in T \tag{22}$$

$$\texttt{grammar}(G, y_{i \in T, e}) \quad \forall e \in E \tag{23}$$

$$y_{ie} \in \{0, 1\} \ \forall i \in T, e \in E \tag{24}$$

where c_i is the cost of having an employee working at time i and G is the grammar defining how the shifts in Ω can be assembled. Note that even though Models SC and SCe could encapsulate more sophisticated costs than the weighted sum of the worked periods, it would also be possible to impose costs on the variables associated to the And-nodes of the grammar constraint formulations allowing for substantial flexibility in modeling.

Corollary 1. *Given that the* $c_s = \sum_{i \in T} a_{is} c_i \ \forall s \in \Omega$ *then models SCe and SCg are equivalent.*

Proof. From Theorem 2, the linearization of (23) yields a polytope that admits only integer extreme points which are, by definition of G, all the pairs (valid shifts $s \in \Omega$, employee $e \in E$). Thus any feasible solution $y_{.e}$ associated to one employee can be written as a convex combination of the extreme points (s,e). If we associate a Boolean variable to each of these extreme points, say x_{se}, we can convert any solution vector given in terms of the extreme points x back to the original variable y by applying

$$y_{ie} = \sum_{s \in \Omega} a_{is} x_{se} \quad \forall i \in T, \forall e \in E \tag{25}$$

Using (25), we can rewrite the y_{ie} variables in SCg (where (23) is no longer necessary) as a convex combination of the extreme point variables x_{se}:

$$SCg = \min \sum_{i \in T, e \in E} c_i \sum_{s \in \Omega} a_{is} x_{se} \tag{26}$$

$$s.t. \sum_{e \in E} \sum_{s \in \Omega} a_{is} x_{se} \geq d_i \quad \forall i \in T \tag{27}$$

$$\sum_{s \in \Omega} a_{is} x_{se} = 1 \; \forall i \in T, e \in E \tag{28}$$

$$x_{se} \in \{0,1\} \quad \forall s \in \Omega, e \in E \tag{29}$$

Given that $c_s = \sum_{i \in T} a_{is} c_i \; \forall s \in \Omega$ and that (28) is derived from (12), this is exactly SCe. □

6 Conclusion

Grammar constraints have received much attention in last few years, as they have been studied in the context of Constraint Programming [6,9,11], SAT [10], Mixed Integer Programming [3], and Large Neighborhood Search [8]. In this paper we studied the linearization of this global constraint and showed that it is possible to generate a polytope that possesses only integer feasible extreme points. We believe this result is fundamental as it means that the use of grammar constraints in the context of Mixed Integer Programming does not introduce any integrality gap. We know that the grammar constraint can be very useful to solve shift scheduling problems [8], and we plan to investigate other areas where these structures can be useful.

Acknowledgments. This work was supported by the National Science Foundation through the Career: Cornflower Project (award number 0644113).

References

1. Achterberg, T., Berthold, T., Koch, T., Wolter, K.: Constraint Integer Programming: a New Approach to Integrate CP and MIP. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 6–20. Springer, Heidelberg (2008)

2. Dantzig, G.: A comment on edieis traffic delay at toll booths. Operations Research 2, 339–341 (1954)
3. Coté, M.C., Gendron, B., Quimper, C.-G., Rousseau, L.-M.: Formal Languages for Integer Programming Modeling of Shift Scheduling Problems. Cirrelt Technical Report #CIRRELT-2007-64 (2007)
4. Ernst, A.T., Jiang, H., Krishnamoorthy, M., Owens, B., Sier, D.: An annotated bibliography of personnel scheduling and rostering. Annals of Operations Research 127, 21–144 (2004)
5. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (1979)
6. Kadioglu, S., Sellmann, M.: Efficient Context-Free Grammar Constraints. In: 23rd National Conference on Artificial Intelligence (AAAI), pp. 310–316 (2008)
7. Katsirelos, G., Narodytska, N., Walsh, T.: The Weighted CfgConstraint. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 323–327. Springer, Heidelberg (2008)
8. Quimper, C.-G., Rousseau, L.-M.: Language Based Operators for Solving Shift Scheduling Problems. In: Seventh Metaheuristics International Conference (2007)
9. Quimper, C.-G., Walsh, T.: Global grammar constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 751–755. Springer, Heidelberg (2006)
10. Quimper, C.-G., Walsh, T.: Decomposing global grammar constraints. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 590–604. Springer, Heidelberg (2007)
11. Sellmann, M.: The Theory of Grammar Constraints. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 530–544. Springer, Heidelberg (2006)
12. Yunes, T., Aron, I., Hooker, J.: An Integrated Solver for Optimization Problems Working Paper WPS-MAS-08-01, School of Business Administration, University of Miami (2008)

Determining the Number of Games Needed to Guarantee an NHL Playoff Spot

Tyrel Russell and Peter van Beek

Cheriton School of Computer Science
University of Waterloo
{tcrussel,vanbeek}@uwaterloo.ca

Abstract. Many sports fans invest a great deal of time into watching and analyzing the performance of their favourite team. However, the tools at their disposal are primarily heuristic or based on folk wisdom. This paper provides a concrete mechanism for calculating the minimum number of points needed to guarantee a playoff spot in the National Hockey League (NHL). Along with determining how many games need to be won to guarantee a playoff spot comes the notion of "must win" games. Our method can identify those games where, if a team loses, they no longer control their own playoff destiny. As a side effect of this, we can also identify when teams get lucky and still make the playoffs even though another team could have eliminated them.

1 Introduction

Hockey fans are interested in knowing when their team clinches a playoff spot. This problem can be modelled as a satisfaction problem and solved using constraint programming [1]. However, if the team has not clinched a playoff spot, this method provides no information about how close a team is to earning a playoff position. The problem of determining how close a team is to clinching a playoff spot can be modelled as an optimization problem that determines the minimum number of points that is necessary to guarantee a spot. This bound on the number of points can also be used to determine when a team has no guarantee of making the playoffs and when a team has lost a crucial game and left destiny in the hands of another team. These factors are interesting to hockey fans and can be generalized to other sports with playoff structures, such as baseball and basketball.

We solve the problem using a constraint model and a mixture of techniques from constraint programming and operations research including network flows, constraint propagation and dominance constraints. We decompose the problem so that we can use a multi-stage approach that adds constraints at each stage if no feasible solution is found. The NHL determines positioning by points and, if tied by points, by several tie breaking conditions. The stages of the solver correspond to the extra constraints needed to determine the extra tie breaking conditions. The first stage uses a combination of enumeration and network flows to determine a tight lower bound on the points needed and whether there exists

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 233–247, 2009.

a feasible solution using only points as a criterion. The second stage also uses network flows to check the first tie breaking condition. The third stage uses a backtracking constraint solver to determine if there are any solutions using the second tie breaking condition.

Our solver can determine the minimum number of points for a given team, at any point in the season, within ten minutes and, for dates near the end of the season, in seconds. In sports, analysts, reporters and coaches often refer to "must win" games. The method used in this paper can identify games where losing that game, the team puts its playoff aspirations into the hands of its opponents. While this does not mean the team will not qualify for the playoffs, it does mean that nothing the team does guarantees a playoff spot. We identify nine teams in the 2006-07 season that lost one of these "must win" games and found themselves in a position to earn a playoff spot again through the actions of their opponents. Several teams experienced this phenomenon four times during the season.

In the remainder of the paper, we give a brief description of the mechanics of playoff qualification in the NHL. Some related work is presented in Section 3. A formal definition of the optimization problem for determining the minimal number of points needed to guarantee a team makes the playoffs is presented. Afterwards, we introduce the concept of an elimination set and explain how this is used to determine tight lower bound values on the optimal value. Since the bound requires the relaxation of tie breaking conditions, we discuss how we can reincorporate those conditions using the same sets. Finally, the specific constraint model is introduced along with some optimizations in the form of exploited dominance and combined consistency checking.

2 The NHL Playoff System

Since the last expansion of the league in 2000, the NHL has consisted of thirty teams arranged into two conferences of fifteen teams. Each conference is broken into three divisions with five teams each. Each team in the NHL plays eighty two games with 41 home games and 41 away games. Eight teams make the playoffs from each conference and to earn a playoff position a team must either be a division leader or one of the five best teams in their conference not including division leaders, called wild card teams.

Positioning in the NHL is determined by one primary criterion and several secondary tie breaking criteria. The primary criterion is the number of points earned by a team. The two common secondary criteria are total wins and points earned against teams with the same number of points and wins. A third tie breaking criteria, which is the number of goals scored on opponents, is sometimes used to break ties between teams tied applying the first three criteria.

Like many North American sports, an NHL game must end in a win or loss. However, the NHL has a unique scoring system as there are points awarded for reaching overtime even if a team does not win the game. The game is separated into three twenty minute periods and, if teams are tied after sixty minutes, a five minute overtime period. If teams are tied after overtime, there is a shootout

to decide the winner. If the game ends during regulation time— the first sixty minutes— then the winner of the game is awarded two points and the loser earns no points. If, however, the game ends either during the overtime period, which is sudden death, i.e. ends when a goal is scored, or the shootout, which must conclude with a winner, then the winner still earns two points but the loser earns a single point in consolation.

3 Related Work

Russell and van Beek [1] created a constraint programming model for calculating whether a given team had clinched a playoff spot. However, the optimization of the decision model requires a relaxation of the dominance constraints. This relaxation of the constraints along with the increased search space leads to a significant increase in the execution time of the solver such that relatively late season instances could not be solved within a day.

Approaches for determining the minimum number of games needed to guarantee a playoff spot have been proposed for the Brazilian football championship [2] and in Major League Baseball [3]. These sports either have a simpler scoring model or a simpler playoff qualification method. Robinson [4] suggested a model for the NHL but his model did not allow for wild card teams or tie breaking conditions. As well, only a theoretical model was presented without experimental results. Gusfield and Martel [5] show that this problem is NP-Hard. They put forth a method for calculating bounds on when a team has been eliminated from the playoffs but their method only works for a single wild card team and a simple win-loss scoring model. Our technique uses a similar method but differs in approach as we must account for multiple wild card teams.

Wayne [6] introduced the concept of a constant that could be used to determine whether or not a team was eliminated from the playoffs. Specifically, he introduced the concept of lower bound constant W^* which denoted the minimum number of points needed to earn a playoff spot. Gusfield and Martel [5] show how this idea can be extended to include a single wild card team and multiple division leaders. In this paper, we will also discuss the existence of an upper bound constant which represents the minimum number of points needed to guarantee a playoff spot.

4 A Formal Problem Definition

To define the problem formally, certain concepts and notations must first be introduced. We denote the set of teams in the NHL as T. We denote the conference that team i belongs to as C_i and the division that team i belongs to as D_i. Table 1a shows the different time variables that we use to superscript the other feature-based variables. Table 1b shows the different features in the NHL and the notation for each feature based on the number and type of opponent. For each instance, there is a schedule and a date d_0 along with the results of games prior to d_0. We define a scenario S to be a completion of the schedule from d_0

Table 1. Variable Notation **(a)** The variables representing the different dates under consideration. **(b)** The variables representing the current state of the results at a given time d_t.

(a)

Date	Notation
Current	d_0
End	d_e
Generic	d_t

(b)

Feature	vs. j	vs. Opposite Conference	Total
Points	$p_{i,j}^{d_t}$	$ocp_i^{d_t}$	$p_i^{d_t}$
Wins	$w_{ij}^{d_t}$	$ocw_i^{d_t}$	$w_i^{d_t}$
Overtime Losses	$ol_{ij}^{d_t}$	$ocol_i^{d_t}$	$ol_i^{d_t}$
Games Remaining	$g_{ij}^{d_0}$	$ocg_i^{d_t}$	$g_i^{d_t}$

by assigning wins, losses, and overtime losses to the games scheduled after d_0. We refer to the maximum possible points that could be earned by a team i if they won all of their remaining games from a given date d_t as $mpp_i^{d_t}$. We refer to the maximum points over all teams $T' \subseteq T$ at a given time d_t as $\max^{d_t}(T')$ and the minimum points over all teams $T' \subseteq T$ at a given time d_t as $\min^{d_t}(T')$.

A team only qualifies for a playoff spot if they are a division leader or a wild card team. We define a division leader to be the team i that has the maximal points at the end of the season within their own division (i.e. $p_i^{d_e} = \max^{d_e}(D_i)$) and has better tie breakers than any team with equivalent points in their division at time d_e. We define a wild card team i to be any team that is not a division leader but has a $p_i^{d_e}$ greater than at least seven other teams in i's conference that are also not division leaders.

Given team k, a given date of the season, d_0, a given schedule of remaining games and given results up to d_0 in the season, a *Playoff Optimization Problem* is to determine the minimal number of points at the end of the season, $p_k^{d_e}$, such that there exists no scenario where k does not qualify for the playoffs as either the leader of the division or one of the five wild card teams. Note that we refer to the given team as either the elimination team or simply k for the remainder of the document.

5 Solution Overview

In this section, we provide an overview to the solver that we use to solve the optimization problem. In order to solve all instances, we use a multi-stage solver that applies different techniques at each stage. In the first stage, we enumerate all of the feasible elimination sets of teams (see Sec. 6) and derive a tight lower bound for the number of points needed. If $p_k^{d_0}$ is greater than the bound then they have already qualified and if $mpp_k^{d_0}$ is less than the bound then they can no longer guarantee. If the bound falls in between those values then with each set that obtains our lower bound, we check each tie break condition to determine if this lower bound is a feasible number of points to guarantee a playoff spot. The second stage checks to see if the first tie break condition, wins, is enough to eliminate k. We do this by enumerating the possible win values and checking

them with a feasible flow algorithm (see Sec. 7). If there is no feasible solution using only points and wins as criteria, the third stage again uses a feasible flow algorithm to check if there are any sets where teams are tied in both points and wins (see Sec. 7). If there exists feasible tie breaking sets, we use a backtracking constraint solver to determine if one of the sets can eliminate k (see Sec. 8). If there are no solutions at this point then k can guarantee a playoff spot if they earn enough points to reach the bound. Otherwise, the solution to the problem is one greater than our lower bound.

6 Generating and Bounding the Elimination Sets

In this section, we define elimination sets and present a method for determining a bound on the points achievable by that set. To calculate the lower bound, we generate all sets of eight teams that could compose the three division leaders and five wild card teams. Each of these sets has the potential to eliminate k at some point bound. The largest bound over all of these sets forms a tight lower bound on the solution to our problem, differing by at most one point.

We define an *Elimination Set*, E, as a set of eight teams from the same conference with at least one team from each division and does not include k. For each team i, they must either have $mpp_i^{d_0} > p_k^{d_0}$ or be the only team in the set from a division D_i such that $D_i \neq D_k$.

We define the bound of an elimination set, E, as the $\max\left(\min^{d_e}(E)\right)$ under all scenarios S where either $p_k^{d_e} = \min^{d_e}(E)$ or $p_k^{d_e} = mpp_k^{d_0}$. The maximum bound over all elimination sets is a tight lower bound on the solution to the complete problem differing by at most one point.

6.1 Calculating the Bound

To calculate the bound for a given elimination set, we adapt an idea by Brown [7] using iterative max flows to solve a sharing problem. We implemented a similar algorithm that shares the games between the teams so that the worst team in the set has the most points possible. By constructing a flow graph that allows us to determine a feasible share, we iterate until a valid distribution of games is found. We start out with a possible bound and determine its feasibility. If the bound is not feasible, we update the bound and check the feasibility of the new bound.

In order to find the best bound, teams win as many points as possible. This means that every loss by a team in the elimination set is an overtime loss and teams in the elimination set win all of their games against teams that are not except k. We formalize the points earned by a team i under this situation as,

$$p'_i = p_i^{d_0} + 2ocg_i^{d_0} + 2\sum_{j \notin E \cup \{k\}} g_{ij}^{d_0} + \sum_{j \in E \cup \{k\}} g_{ij}^{d_0} \quad . \tag{1}$$

Equation (1) represents the sum of the points already earned ($p_i^{d_0}$), the wins against teams not in the set $E \cup \{k\}$ ($2ocg_i^{d_0} + 2\sum_{j \notin E \cup \{k\}} g_{ij}^{d_0}$) and one point

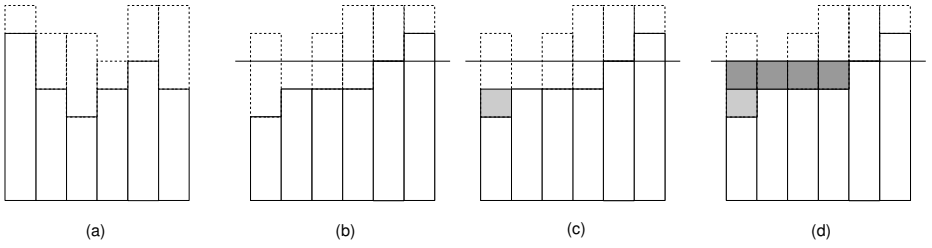


Fig. 1. The relaxed bound algorithm. **(a)** The original problem with 6 games remaining. The solid rectangles represents $p_i^{d_0}$ and the dashed rectangles represents $mpp_i^{d_0}$. The first step is to sort the teams by $p_i^{d_0}$. **(b)** shows the sorted teams with the $\min(mpp_i^{d_0})$ shown as the solid horizontal line. **(c)** We assign games to the teams with the least points. In this case, one game to the first team. **(d)** In the next iteration, four teams need games and we allocate four of the remaining five games. The bound is reached and the final solution has one game remaining.

each from games against teams in $E \cup \{k\}$ ($\sum_{j \in E \cup \{k\}} g_{ij}^{d_0}$). These preprocessing steps are valid dominance relations as we are looking for the scenario where we get the maximum $\min^{d_e}(E)$ and these steps either increase the points of a team in E or leave them the same while not affecting the maximum possible points of the teams in E.

6.2 The Relaxed Bound

To determine the starting point for the lower bound, we solve a relaxation of the bound calculation where we relax the constraint that a specific number of games must be played between two teams. Instead, we consider all games as a pool of unplayed games with no assigned opponents and assign them to the worst team until the games are used or the $min_{i \in E}(mpp_i^{d_0})$ is reached. Figure 1 shows an example bound calculation.

6.3 The Flow Network and the Bound

Once we have a starting point calculated by the relaxed bound algorithm, we are looking to find the first feasible bound when we include the constraints removed during the infeasible calculations. We formulate this as a feasible flow problem [8] with an artificial sink and source. These graphs look similar to the graphs constructed by Schwartz [9] in his paper on baseball elimination.

Every team in the elimination set and the team k needs to win a certain number of games to reach the bound and this must be incorporated into the graph. We define the *need* of a team i, n_i, to be $bound - p'_i$ (where *bound* is the current lower bound on points and p'_i is defined in (1)). The exception to this rule is k where the bound may be greater than mpp_k. In that case, we calculate n_k as $mpp_k - p'_k$. We use the p' values since we are still looking for best case results for the set $E \cup \{k\}$. A bound is feasible if the maximum flow in the graph

```
bound ← InfeasibleBound(E,k);
repeat
    Needs ← CalculateNeeds(E,k,bound);
    need ← Σ_{i∈E} n_i;
    G ← ConstructGraph(Needs);
    flow ← CalculateFlow(G);
    if flow < need then
        bound ← bound − 1;
until flow ≥ need ;
return bound
```

Algorithm 1. This algorithm shows the steps for calculating the bound for a given elimination set, E. First, we generate an infeasible bound as a starting point. From that starting point, we generate, for each team, the number of points needed to reach the bound, denoting the set of *needs* as $Needs$. Then we check feasibility using a flow algorithm. If the flow meets the *needs*, we return the bound. Otherwise, we reduce the bound and iterate.

is equal to the sum of the *needs* of the teams in the elimination set. If not, a new bound must be tried. Algorithm 1 describes the process by which the bound is calculated. We denote the maximum bound calculated by the algorithm over all elimination sets as **p** and prune any set that does not reach that bound.

Once we know the *needs* for a given elimination set, it is relatively simple to construct the graph. An example graph is in Fig. 2 showing the variables and the associated capacities in the graph. We create two nodes s and t to be the

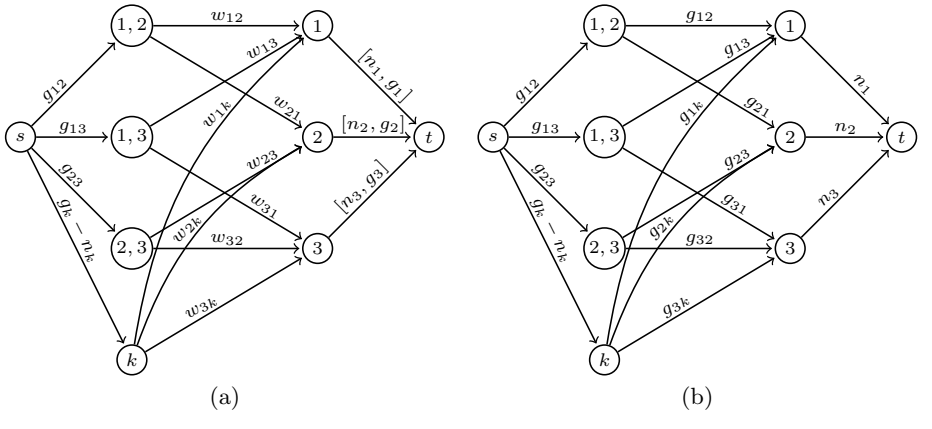


Fig. 2. (a) A variable representation of the values in the flow graph for a three team elimination set ($\{1, 2, 3\}$). The implementation of the graph uses the domain of the variables as the capacity bounds. **(b)** The capacities to determine feasibility for the flow graph. All variables are given their domain maximum. Since the flow is split in nodes $(1, 2)$, $(1, 3)$ and $(2, 3)$, a feasible flow is a valid assignment. If the max flow can saturate the *needs* of the teams then there is a feasible solution for this elimination set.

source and sink, respectively. We add one node for each pair of teams in the set and one node for each team in the set. On top of this, we add an extra node that represents games played against k by any team in E. Each node representing a pair of teams has three edges where one is an incoming edge from s with a capacity equal to the number of games between those two teams $g_{ij}^{d_0}$ and two are outgoing edges to the nodes for the teams with the same capacity as the incoming edge. There is also an edge from each node representing a team in the set to the sink node t with a capacity equal to the *need* of the node. Last, the node representing the games against k has an edge from the source with a capacity of $g_k^{d_0} - n_k$ and one link each to every team node with a capacity equal to the number of games played between them.

7 Win Values and Tie Breaking Sets

In this section, we describe how win values are used to determine if an instance has a solution and how to generate feasible tie break sets. Once we have a point bound and set of teams that could potentially reach that bound, we determine the possible values for the secondary criteria and only solve feasible instances. This means that we determine if the elimination set can eliminate k with only wins or whether some teams must be tied. We use a modification of the original flow problem to determine both of these quantities. First, observe that if a team earns $\mathbf{p}$ points then we have the following constraint,

$$(p_i^{d_e} = \mathbf{p}) \Rightarrow \quad (\mathbf{p} - p_i^{d_0}) - g_i^{d_0} \leq w_i^{d_e} - w_i^{d_0} \leq \left\lfloor \frac{(\mathbf{p} - p_i^{d_0})}{2} \right\rfloor \quad . \tag{2}$$

This constraint represents that any team i with $\mathbf{p}$ points at the end of the season would have earned the fewest extra wins if every loss was an overtime loss thus earning at least one point per game and the most extra wins when they win as many games as possible while still only earning $\mathbf{p}$ points. This constraint also holds true for k so we can determine a feasible range of wins for the elimination team given the elimination set, E, and the point bound $\mathbf{p}$. For each possible number of wins $\mathbf{w}$ for k, we determine if the set can eliminate with that number of wins and, if not, which sets of teams can be tied.

Both of these tasks can be solved by checking for feasible flows on the same graph using slightly different *needs* in each case. We modify the graph for calculating point bounds by allowing k as a proper team on the right hand side and adding links directly to the nodes for games outside the set (see Fig. 3a). Games against opponents outside $E \cup \{k\}$ do not have to be won by any team in the set so they have no lower bound but those in the set must be won by one of the teams; therefore, they have a lower bound. We construct the graph so that each team that must be tied with k in wins has a lower and upper bound equal to their *need*. The *need* calculation for this graph is different than the point graph. Since both points and wins are both fixed, we get the following equation for calculating *need*,

$$n_i = \begin{cases} \mathbf{w} - w_i^{d_0} & \text{if } (p_i^{d_e} = \mathbf{p}) \wedge (w_i^{d_e} = \mathbf{w}) \\ 0 & \text{if } (\mathbf{p} - p_i^{d_0}) - g_i^{d_0} < 0 \\ (\mathbf{p} - p_i^{d_0}) - g_i^{d_0} & \text{if } (\mathbf{p} - p_i^{d_0}) - g_i^{d_0} + w_i^{d_0} > \mathbf{w} \\ (\mathbf{p} - p_i^{d_0}) - g_i^{d_0} + 1 & \text{otherwise} \end{cases} . \quad (3)$$

Each condition of Equation (3) represents the number of wins needed to eliminate k in the best case scenario using as few wins as possible. The first condition denotes that the elimination team must have exactly $\mathbf{w}$ wins. The second condition denotes that teams that would have equal or more points using only extra points from overtime losses do not have to win any more games. The third condition ensures that teams that win the minimal number of games to reach $\mathbf{p}$ have more wins than k and eliminate k. The fourth condition corrects the number of wins needed when the second and third condition do not hold by adding an additional win to the minimal number of wins. For teams tied with k in wins, we introduce a tie break set. We define a *Tie Break Set* as any subset of the teams in C_k where every team can reach both the point bound $\mathbf{p}$ and the win bound $\mathbf{w}$ exactly. We test all subsets by setting the *need* of teams in the tie break equal to $\mathbf{w} - w_i^{d_0}$.

Since our graph has minimum and maximum capacities on the edges, we transform the graph into a different max flow problem as described by Ahuja et al. [8]. The transformation can be seen in Fig. 3. Once we have checked the wins tie breaker with the flow graph and determined which sets of nodes can be tied in both points and wins, we determine if any of those tie break sets can eliminate k with points against teams that are tied. We model this problem as a satisfaction problem and solve it using backtracking search as described next.

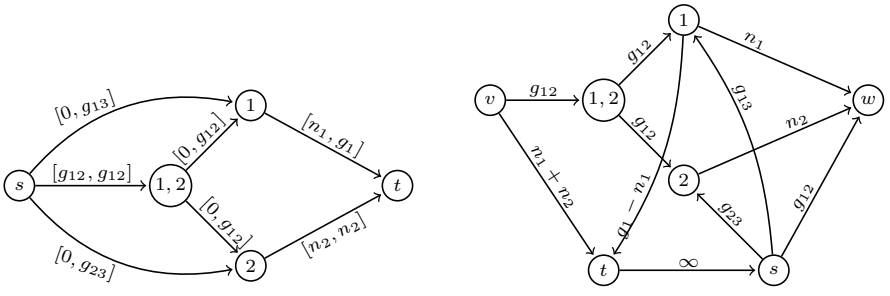


Fig. 3. (a) A network flow graph with three teams. Team 1 has a lower bound constraint on the number of wins and is in the elimination set and not in the tie break set, team 2 is in the tie break set and has a fixed number of possible wins, and team 3 is in neither the elimination set or the tie break set and has no bounds on either points or wins. **(b)** A flow graph transformed to remove the lower bound capacities. Two additional nodes are added v and w. A feasible flow exists in the original graph if the maximum flow is equal to the sum of the lower bounds on the original graph $(n_1 + n_2 + g_{12})$.

8 The Decision Problem

In this section, we describe the constraint model used to determine if the final tie breaks eliminate k. Once we have fixed the elimination set (E), point bound ($\mathbf{p}$), win bound ($\mathbf{w}$) and tie break set (TB), we verify this combination eliminates the team k. We examine possible scenarios of wins ($w_{i,j}$) and overtime losses ($ol_{i,j}$) as these are the two factors that affect the points and hence the outcome of a given scenario. We break the teams into four mutually exclusive classes to help describe our model.

$$A = \{i \mid i \in E \wedge i \notin TB\} \quad C = \{i \mid i \notin E \wedge i \in TB\}$$
$$B = \{i \mid i \in E \wedge i \in TB\} \quad D = \{i \mid i \notin E \wedge i \notin TB\}$$

8.1 The Model

There are four major constraints to this model. Each of which is modified slightly depending on which class a given team belongs. Constraint (4) represents the constraint that each of the teams must either meet or exceed the bounds depending on their class. These rules are derived from the NHL tie breaking rules. Constraint (5) represents the constraint that each game must have a winner. The exception to both of these constraints are those teams in D. These teams are not restricted by the bounds and thus we can ignore any game where they are playing other teams in D. We also must constrain the number of overtime losses so that the team does not earn more overtime losses than losses. This constraint is reflected in (6). Lastly, we must deal with constraints on games played against teams in the opposite conference. Teams in A can win these games freely, teams in D can lose them freely and teams in B and C can win them depending on the constraints applied in (4). We define these constraints explicitly in (7).

$$\begin{aligned}
p_i^{d_e} > \mathbf{p} \vee (p_i^{d_e} = \mathbf{p} \wedge w_i^{d_e} > \mathbf{w}) \qquad & \text{if } i \in A \ . \\
p_i^{d_e} = \mathbf{p} \wedge w_i^{d_e} = \mathbf{w} \ \wedge & \\
2 \sum_{j \in TB} w_{ij}^{d_e} + \sum_{j \in TB} ol_{ij}^{d_e} > 2 \sum_{j \in TB} w_{kj}^{d_e} + \sum_{j \in TB} ol_{kj}^{d_e} \qquad & \text{if } i \in B \ . \\
p_i^{d_e} = \mathbf{p} \wedge w_i^{d_e} = \mathbf{w} \qquad & \text{if } i \in C \ . \qquad (4)
\end{aligned}$$

$$\begin{aligned}
\forall_j \quad w_{ij}^{d_e} + w_{ji}^{d_e} = g_{ij} \qquad & \text{if } i \notin D \ . \\
(\forall_{j \in A} \quad w_{ij}^{d_e} = w_{ij}^{d_0} \wedge w_{ji}^{d_e} = w_{ji}^{d_0} + g_{ij}^{d_0} \ \wedge & \\
(\forall_{j \in B \cup C} \quad w_{ij}^{d_e} + w_{ji}^{d_e} = w_{ji}^{d_0} + g_{ij}^{d_0} \ \wedge & \\
(\forall_{j \in D} \quad w_{ij}^{d_e} = w_{ij}^{d_0} \wedge w_{ji}^{d_e} = w_{ij}^{d_0}) \qquad & \text{if } i \in D \ . \qquad (5) \\
\forall_j \quad w_{ij}^{d_e} + ol_{ij}^{d_e} = w_{ij}^{d_0} + ol_{ij}^{d_0} + g_{ij}^{d_0} \qquad & \text{if } i \in A \ . \\
\forall_j \quad w_{ij}^{d_e} + ol_{ij}^{d_e} \leq w_{ij}^{d_0} + ol_{ij}^{d_0} + g_{ij}^{d_0} \qquad & \text{if } i \in B \cup C \ . \\
\forall_j \quad ol_{ij}^{d_e} = ol_{ij}^{d_0} \qquad & \text{if } i \in D \ . \qquad (6)
\end{aligned}$$

$$\begin{aligned} ocw_i^{d_e} &= ocw_i^{d_0} + ocg_i^{d_0} \wedge ocol_i^{d_e} = ocol_i^{d_0} && \text{if } i \in A \ . \\ ocw_i^{d_e} + ocol_i^{d_e} &\leq ocw_i^{d_0} + ocol_i^{d_0} + ocg_i^{d_0} && \text{if } i \in B \cup C \ . \\ ocw_i^{d_e} &= ocw_i^{d_0} \wedge ocol_i^{d_e} = ocol_i^{d_0} && \text{if } i \in A \ . \end{aligned} \tag{7}$$

8.2 Updating Dominance during Search

As the search progresses, it is often possible to force the assignment of certain games. The most important dominance is to notice that only win variables within the tie break and elimination set must be set via search. Once those variables have been set, all that remains is to ensure teams meet or exceed **p** and **w** and to make sure teams trying to beat k earn as many of their necessary overtime losses within the tie break set and k wins as many of them as possible out of the tie break set. These dominances lead to a correct solution and makes sure teams in B earn as many points within the tie break as possible.

Another opportunity is when teams have satisfied (4). Specifically, once a team in A has met the conditions of (4), they may give points to other teams in A without any consequences. Another dominance is that once a team in the tie break set has achieved both **p** and **w** they must lose any remaining games in regulation time.

8.3 Pruning Values from Constrained Teams via Flow Manipulation

As mentioned in Sec. 7, the feasibility of the tie break set depends on whether there exists a max flow equal to the needs of the teams in the flow graph represented by Fig. 3. An important observation that can be made is that any feasible flow is a valid assignment of the win variables of the teams in the elimination and tie break sets. We can prune the variables within the solver by attempting to update an already existing flow to contain a specific test value using a method adapted from Maher et al.[10]. If there is a flow that contains the value then there is a support for that value and that value is kept. If not, then we prune the value from the domain of the variable. The idea is similar to the idea used in the Ford-Fulkerson algorithm. However, in our case, we are trying to find a path not from s to t but from j to i. We must repeat the update at most d times where d is the size of the domain of w_{ij} and w_{ji}.

To reduce the practical complexity of the algorithm, we reduce the residual graph to only those components that will be updated. In a graph containing a feasible flow, the edges out of v and into w are completely saturated. Since any modification must also be a feasible flow, these edges must remain saturated and any modification should not alter these edges. The other reduction that we can make to the graph depends on the symmetry between nodes representing teams and the links to their matched games. This allows us to remove the nodes representing the matched games and link the nodes directly together.

Example 1. Examine Fig. 4b and note that in the residual graph links into v and out of w are saturated and can be removed. Also observe that the edge

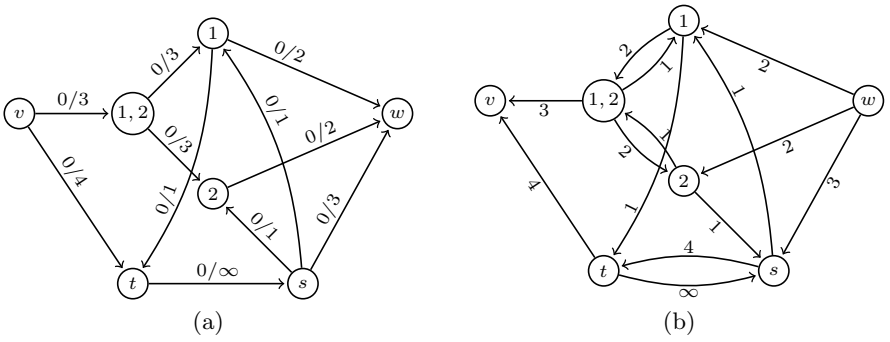


Fig. 4. (a) An example flow graph for three teams where Team 1 must earn between 2 and 3 games, Team 2 must earn exactly 2 games and Team 3 is unbounded. **(b)** The residual graph containing a maximum flow.

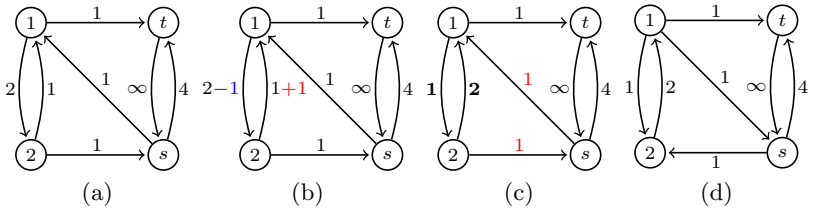


Fig. 5. Reduced Pruning Graph. **(a)** shows the reduced residual graph of Figure 4. In **(b)**, we reduce the link between nodes 1 and 2 and increase the link between nodes 2 and 1, which ensures the constraint that the flow between them equals some mutual capacity. **(c)** shows the path that is found from node 2 to node 1 correcting the imbalance. Once a path is found, the flow is redirected and the opposite edges are updated by the change. **(d)** shows the new stable solution showing support for the assignments of $w_{12} = 1$ and $w_{21} = 2$.

from node 1 to node $(1, 2)$ is the same as the edge from node $(1, 2)$ to node 2. Therefore, we can remove node $(1, 2)$ and directly link $(2, 1)$. Figure 5 shows the reduced pruning graph along with a single variable update.

9 Results

We implemented the solver in C++ using the Boost Graph Library [11] for the feasible flow calculations and ILOG Solver [12] to solve the final constraint model. In order to test our solver, we used the 2006-07 season results to calculate the minimum points needed to clinch a playoff spot. Table 2 shows the results of those calculations. In total, determining the bound for all 30 teams on all 181 game days of the 2006-07 NHL season (5430 problems) took a little over 46 hours. Each individual instance, representing a team at a given date, took less than ten minutes to calculate the bound and those problems near the end of the season,

Table 2. The counts of problems solved via the various stages of the solver. Positively solved instances means a solution was found and the bound must be increased. Negatively solved instances means that bound was valid for that instance. Any problem without a definitive solution was passed to the next phase of the solver.

Solver Stage & Result	Number of Instances (/5430)
Solved via Enumeration	1212
Solved via Win Checks (Positively)	2249
Solved via Win Checks (Negatively)	1524
Solved via Backtracking Search (Positively)	338
Solved via Backtracking Search (Negatively)	107

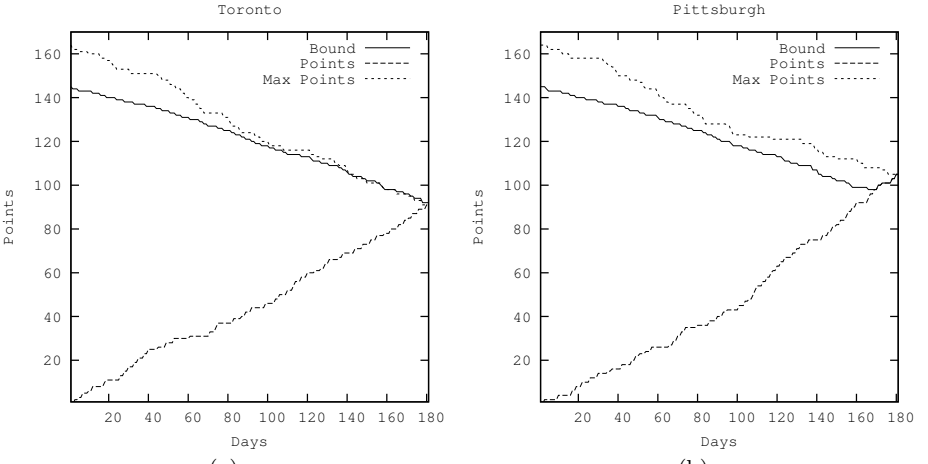


Fig. 6. The minimum number of points needed by Toronto and Pittsburgh to guarantee a playoff spot in the 2006-07 NHL season

where the results matter the most, were calculated in seconds. We note that our enumeration techniques solves 1212 of the 5430 of the problems and when we add first level tie breaking with wins we solve a further 3773 problems, which makes up about 92% of the problems. However, the remaining 8% problems require a backtracking constraint solver to calculate the final number. Also, note that in 47% of the total instances, which amounts to 61% of the instances not solved directly by enumeration, the answer differs from the initial lower bound given by enumeration.

We plot the result against both the current points of the team and maximum possible points of the team. If the result is greater than the maximum possible points, then the team is no longer able to guarantee a playoff spot. If the result is equal to the number of points needed by the team then that team has clinched a playoff spot. Figure 6 shows the result calculated for Toronto and Pittsburgh. Note that Toronto did not make the playoffs because they never reached the bound value. Also note that Toronto placed themselves in a position where they

Table 3. Shows some of the features that can be highlighted by calculating the minimum number of points needed to guarantee a playoff spot

Feature	Value	Team(s)
Earliest Day where a Team could not Guarantee	64 days	St. Louis
Most Days where a Team could not Guarantee	118 days	St. Louis
Most Times a Team got Lucky	4	Toronto, Boston and NY Rangers
Number of Teams that got Lucky and Earned a Spot	2	NY Islanders and NY Rangers
Number of Teams that got Lucky but Failed to Earn a Spot	7	Toronto, Boston, Washington, Carolina, Edmonton, Phoenix and Columbus

could not guarantee a playoff spot and got lucky four times. In other words, they lost a "must win" game five times during the 2006-07 season while Pittsburgh was never in that situation. Another interesting feature is that we can see, in both graphs, the bound on points, 145, needed at the start of the season to guarantee a playoff spot.

Table 3 shows an overview of the results of the 2006-07 NHL season in terms of the minimum points needed to guarantee a playoff spot. One interesting observation that can be made from this table is that of the nine teams that got a second chance only two of those teams ended up earning a playoff spot. As well, of those seven teams, two of them had four chances to make the playoffs after losing a must win game. Another interesting note is that St. Louis could not guarantee a playoff spot after only the sixty-fourth game day and never recovered during the final one hundred and eighteen game days.

10 Conclusion

As the season winds down, the fans of the NHL are interested in knowing how far their team is from clinching a playoff spot. We present a method for calculating the minimum number of points that must be earned in order to ensure that the team reaches a playoff spot. We preform this calculation efficiently by using a multi-stage solver that combines enumeration, flow network calculations and backtracking search.

A side effect of this calculation is the ability to determine when the team is in danger of losing control of their destiny. These games, often described by coaches as "must win" games, can be identified by their loss reducing the maximum possible points to below the bound of the team. We identified nine different teams in the 2006-07 NHL season that lost control of their fate and then gained that control back through mistakes by their opponents. We also noted that only two of these teams took full advantage of this situation and clinched a playoff spot.

Our solver used a decomposition of the problem to allow us to effectively apply several different strategies in several stages to ensure a quick solution to the problem. The results of this work could be applied to other sports. One area that seems to be missed entirely is basketball, especially NBA basketball, where that league shares many similarities with the NHL. The tie breaking conditions vary slightly and the NBA uses a simpler scoring model with only wins and losses.

References

1. Russell, T., van Beek, P.: Mathematically clinching a playoff spot in the NHL and the effect of scoring systems. In: Bergler, S. (ed.) Canadian AI. LNCS, vol. 5032, pp. 234–245. Springer, Heidelberg (2008)
2. Ribeiro, C.C., Urrutia, S.: An application of integer programming to playoff elimination in football championships. International Transactions in Operational Research 12, 375–386 (2005)
3. Adler, I., Erera, A.L., Hochbaum, D.S., Olinick, E.V.: Baseball, optimization and the world wide web. Interfaces 32, 12–22 (2002)
4. Robinson, L.W.: Baseball playoff eliminations: an application of linear programming. Operations Research Letters 10, 67–74 (1991)
5. Gusfield, D., Martel, C.E.: The structure and complexity of sports elimination numbers. Algorithmica 32, 73–86 (2002)
6. Wayne, K.D.: A new property and a faster algorithm for baseball elimination. SIAM Journal on Discrete Mathematics 14, 223–229 (2001)
7. Brown, J.R.: The sharing problem. Operations Research 27, 324–340 (1979)
8. Ahuja, R.K., Magnanti, T.L., Orlin, J.B.: Network Flows: Theory, Algorithms and Applications. Prentice-Hall, Englewood Cliffs (1993)
9. Schwartz, B.: Possible winners in partially completed tournaments. SIAM Review 8, 302–308 (1966)
10. Maher, M., Narodytska, N., Quimper, C.G., Walsh, T.: Flow-based propagators for the sequence and related global constraints. In: Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (2008)
11. Siek, J., Lee, L.Q., Lumsdaine, A.: Boost Graph Library: User Guide and Reference Manual. Addison-Wesley, Reading (2001)
12. ILOG S.A.: ILOG Solver 4.2 user's manual (1998)

Scalable Load Balancing in Nurse to Patient Assignment Problems

Pierre Schaus[1], Pascal Van Hentenryck[2], and Jean-Charles Régin[3]

[1] Dynadec, One Richmond Square, Providence, RI 02906, USA
pschaus@dynadec.com
[2] Brown University, Box 1910, Providence, RI 02912, USA
pvh@cs.brown.edu
[3] Université de Nice-Sophia Antipolis
930 Route des Colles - BP 145, 06903 Sophia Antipolis Cedex, France
regin@polytech.unice.fr

Abstract. This paper considers the daily assignment of newborn infant patients to nurses in a hospital. The objective is to balance the workload of the nurses, while satisfying a variety of side constraints. Prior work proposed a MIP model for this problem, which unfortunately did not scale to large instances and only approximated the objective function, since minimizing the variance cannot be expressed in a linear model. This paper presents constraint programming (CP) models of increasing complexity to solve large instances with hundreds of patients and nurses in a few seconds using the Comet optimization system. The CP models use the recent spread global constraint to minimize the variance, as well as an exact decomposition technique.

1 Introduction

This paper considers the daily assignment of newborn infant patients to nurses in a hospital described in [5]. In this problem, some infants require little attention, while others need significant care. The amount of work required by the infant during one shift is called the *acuity*. A nurse is in charge of a group of infants and the total amount of acuity is the workload of the nurse during that shift. For ensuring an optimal care quality and perceived fairness for the nurses, it is essential to balance the workload. In addition, the problem features various side constraints:

- A nurse can work in only one zone, but the patients are located in p different zones.
- A nurse cannot be responsible of more than $children^{\max}$ infants.
- The total amount of acuity of a nurse cannot exceed $acuity^{\max}$.

The balance objective and the various constraints make it very difficult to find a good solution in a reasonable time. Since nurses only work in one zone, the number of nurses assigned to each zone has already a huge impact on the quality of the balancing. In [5], the problem was tackled using a MIP model, but the

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 248–262, 2009.

results were not satisfactory. In this paper, we present a series of increasingly sophisticated constraint programming models in order to reach the required solution quality and scalability.

The rest of the paper is organized as follows. Section 2 presents the instances proposed in [5] and Section 3 describes the MIP model and its limitations. Section 4 reviews the *Spread* constraint for load balancing and characterizes its pruning (as implemented in COMET). Section 5 presents a first constraint programming (CP) model that can solve two-zones instances. Section 6 presents a two-step approach that first assigns the nurses in each zone and then assigns the infants to nurses to balance the load optimally. Finally, Section 7 shows that the second step can be decomposed by zones without losing the optimality guarantees. This final model is instrumental in solving large instances with dozens of zones and hundreds of patients.

2 Problem Instances

Reference [5] specifies a statistical model to generate instances very similar to their real instances. This statistical model was also used to measure the robustness of their solution technique with respect to the number of nurses, the number of infants, and the number of zones. The model contains a single parameter: the number of zones. The maximum acuity per nurse is fixed to $acuity^{\max} = 105$ and the maximum number of infants per nurse is fixed to $children^{\max} = 3$. The instance generator fixes the number of nurses, the number of infants, the acuity, and the zone of each infant. The different steps to generate an instance are as follows:

- The number of patients in a zone is specified by a Poisson random variable with mean 3.8 and offset by 10.
- The acuity Y of a patient is obtained by first generating a number $X \sim Binomial(n = 8, p = 0.23)$ and then choosing the number $Y \sim Unif(10 \cdot (X + 1), 10 \cdot (X + 1) + 9)$.
- The total number of nurses is obtained by solving a First Fit Decreasing (FFD) procedure in each zone. More precisely, the total number is the number of nurses found in each zone by the FFD procedure. The FFD procedure starts by ranking the patients in decreasing acuity. Then, the patient with the highest acuity is assigned to the first nurse. The next patients are assigned successively to the first nurse that can accommodate them without violating the maximum acuity and the number of patient constraints.

3 The MIP Model

We now review the main variables of the MIP model from [5]. We also describe the limitations of the MIP model and suggest why a CP approach may address them. Due to space reasons, we do not reproduce the entire MIP model but readers can consult [5] for more details. The technical details presented here are sufficient for our purposes. The MIP model contains four families of variables:

1. $X_{ij} = 1$ if infant i is assigned to nurse j and 0 otherwise;
2. $Z_{jk} = 1$ if nurse j is assigned zone k and 0 otherwise;
3. $Y_{k,\max}$ is the maximum acuity of a nurse in zone k;
4. $Y_{k,\min}$ is the minimum acuity of a nurse in zone k.

All these variables are linked with linear constraints to enforce the constraints of the problem. The objective function implements what we call the *range-sum* criterion and consists of minimizing the sum of the acuity ranges of the p zones, i.e.,

$$\sum_{k=1}^{p}(Y_{k,\max} - Y_{k,\min}).$$

The MIP model has a fundamental limitation: The objective function may produce poorly balanced workloads. It tends to equalize the workload inside the zones but may produce huge differences among the workload of different zones. This is illustrated in Figure 1. The workloads are depicted in the top-right corner of each COMET visualization. The left solution is obtained by minimizing the range-sum criterion and the right solution by minimizing the variance (L_2 norm in the next section). The range-sum objective is minimal on the left because the workloads inside each of the two zones are identical. Unfortunately, nurses in the first zone work twice as much as those in the second zone. The right solution is obtained by minimizing the variance and is significantly more appealing.

Fig. 1. Comparison of Two Solutions on a 6 Nurses, 14 Infants, and 2 zones Problem. Solution on the left is obtained by minimizing the range-sum criterion. Solution on the right is obtained by minimizing the variance.

This illustrates clearly that "the high level objective that all nurses should be assigned an equal amount of patient acuity" [5] is not properly captured with the range-sum criterion.

It is not immediately obvious how to remedy these problems. The variance is non-linear and is not easily modelled in a MIP approach. In addition, a CP approach may exploit the combinatorial structure in the bin-packing and the side-constraints, while the MIP relaxation is generally bad for bin-packing like problems. Finally, there are important symmetries that are not removed in their model: For a given solution, the nurses are completely interchangeable. We now review load balancing constraints before turning to the CP models.

4 Load Balancing Constraints

Balancing constraints arise in many real-world applications, most often to express the need of a fair distribution of items or work. For instance, Simonis [15] suggested a global constraint to balance the shift distribution among nurses and Pesant [7] proposed the use of balancing constraints for a fair allocation of individual schedules.

Two global constraints and their propagators are available in constraint programming for optimizing load balancing: `spread` [6,11], which constrains the variance and the mean of a set of variables, and `deviation` [12,13], which constrains the mean absolute deviation and the mean of a set of variables. We also say that `spread` and `deviation` respectively constrain the L_2 and L_1 norms of a set of variables $X_1..X_n$ with respect to their mean ($s = \sum_{i\in[1..n]} X_i$), i.e.,

- L_1: $\sum_{i\in[1..n]} |X_i - s/n|$;
- L_2: $\sum_{i\in[1..n]} (X_i - s/n)^2$.

These criteria are not equivalent: Minimizing L_1 or L_2 does not lead to the same solutions and it is not always obvious which one to choose. In fact, this is an old and recurrent debate (see for instance [3]). For this application, we use `spread` because the L_2 criteria is more sensitive to outliers, which we consider significant in this application.

We use the following definitions and notations to describe the semantics of the `spread` constraints and propagators.

Definition 1. *Let X be a finite-domain (discrete) variable. The domain of X is a set of ordered values that can be assigned to X and is denoted by $Dom(X)$. The minimum (resp. maximum) value of the domain is denoted by $X^{\min} = \min(Dom(X))$ (resp. $X^{\max} = \max(Dom(X))$. An integer interval with integer bounds a and b is denoted $[a..b] \subseteq \mathbb{Z}$, while a rational interval is denoted $[a, b] \subseteq \mathbb{Q}$. An assignment on the variables $\boldsymbol{X} = [X_1, X_2, ..., X_n]$ is denoted by the tuple $\mathbf{x}$ and the i-th entry of this tuple by $\mathbf{x}[i]$. The extended rational interval domain of X_i is $I_D^{\mathbb{Q}}(X_i) = [X_i^{\min}, X_i^{\max}]$ and its integer interval domain is $I_D^{\mathbb{Z}}(X_i) = [X_i^{\min} \,..\, X_i^{\max}]$.*

We now define the `spread` constraint with a fixed mean.

Definition 2. *Given finite domain variables* $\boldsymbol{X} = (X_1, X_2, ..., X_n)$*, an integer value* s *and a finite domain variable* Δ*,* $\texttt{spread}(\boldsymbol{X}, s, \Delta)$ *holds if and only if*

$$\sum_{i \in [1..n]} X_i = s \quad \text{and} \quad \Delta \geq n \times \sum_{i \in [1..n]} |X_i - s/n|^2.$$

Observe also

$$n \cdot \sum_{i \in [1..n]} |X_i - s/n|^2 = n \times \sum_{i \in [1..n]} X_i^2 - s^2. \tag{1}$$

Since s is an integer, this quantity is integer, which is why it is more convenient to work with $n \times \sum_{i \in [1..n]} X_i^2 - s^2$ than with $\sum_{i \in [1..n]} |X_i - s/n|^2$.

Example 1. Tuple $\mathbf{x} = (4, 6, 2, 5) \in \texttt{spread}([X_1, X_2, X_3, X_4], s = 17, \Delta = 40)$ but $\mathbf{x} = (3, 6, 2, 6) \notin \texttt{spread}([X_1, X_2, X_3, X_4], s = 17, \Delta = 40)$ because $4 \cdot (3^2 + 6^2 + 2^2 + 6^2) - 17^2 = 51 > 50$.

The filtering algorithm for `spread` achieves $\mathbb{Z}$-bound-consistency.

Definition 3 ($\mathbb{Q}$-bound-consistency and $\mathbb{Z}$-bound-consistency). *A constraint* $C(X_1, \ldots, X_n)$ *(*$n > 1$*) is* $\mathbb{Q}$*-bound-consistent (resp.* $\mathbb{Z}$*-bound-consistent) with respect to domains* $Dom(X_i)$ *if for all* $i \in \{1, \ldots, n\}$ *and each value* $v_i \in \{X_i^{\min}, X_i^{\max}\}$*, there exist values* $v_j \in I_D^{\mathbb{Q}}(X_j)$ *(resp.* $v_j \in I_D^{\mathbb{Z}}(X_j)$*) for all* $j \in \{1, \ldots, n\} - \{i\}$ *such that* $(v_1, \ldots, v_n) \in C$*.*

The propagators described in [6,11] achieve $\mathbb{Q}$-bound-consistency, which means that they assume that the variables can be assigned rational numbers. The propagators implemented in Comet implement the stronger $\mathbb{Z}$-bound-consistency by adapting the algorithms from [6,11]. In particular, to achieve $\mathbb{Z}$-bound-consistency, the propagators for `spread` compute $\underline{\Delta}^{\mathbb{Z}}$ to filter $\Delta^{\min}$, and $\overline{X}_i^{\mathbb{Z}}$ and $\underline{X}_i^{\mathbb{Z}}$ to filter $X_i^{\max}$ and $X_i^{\min}$:

$$\underline{\Delta}^{\mathbb{Z}} = \min_{\mathbf{x}} \{ n \cdot \sum_{i \in [1..n]} (\mathbf{x}[i] - s/n)^2 \text{ s.t. } \sum_{i \in [1..n]} \mathbf{x}[i] = s \text{ and } \forall i \in [1..n] : \mathbf{x}[i] \in I_D^{\mathbb{Z}}(X_i) \} \tag{2}$$

$$\overline{X}_i^{\mathbb{Z}} = \max_{\mathbf{x}} \{ \mathbf{x}[i] \text{ s.t. } n \cdot \sum_{j \in [1..n]} (x[j] - s/n)^2 \leq \Delta^{\max} \text{ and } \sum_{j \in [1..n]} \mathbf{x}[j] = s \text{ and } \forall j : x[j] \in I_D^{\mathbb{Z}}(X_j) \}. \tag{3}$$

The filtering of Δ is implemented in $O(n \cdot log(n))$ requiring to sort the bounds of the domains, and that of $\mathbf{X}$ in $O(n^2)$ in the Comet System [2,10].

5 A Basic CP Model

We now present a CP based resolution which addresses the issues raised for the MIP model.

The CP Model. Let m be the number of nurses, n the number of patients, and a_i be the acuity of patient i. The set of patients in zone k is denoted by $\mathcal{P}_k$ and $[\mathcal{P}_1, ..., \mathcal{P}_p]$ forms a partition of $\{1, ..., n\}$. For each patient i, we use a decision variable $N_i \in [1..n]$ representing her/his nurse. The workload of nurse j is represented by variable $W_j \in [0..acuity^{\max}]$. The objective and constraints are modelled as follows.

- The objective, i.e., minimizing the L_2 norm, is expressed by a `spread` constraint over the workload variables $[W_1, ..., W_m]$, the total acuity, and the acuity spread: `spread(`$[W_1, ..., W_m]$`,totalAcuity,spreadAcuity)`. Note that `spreadAcuity` is the variable to minimize.
- To express that nurses have a total acuity of atmost $acuity^{\max}$, we link the variables N_i, W_j, and the acuities with a global packing/multiknapsack constraint [14]: `multiknapsack`$([N_1, ..., N_n], [a_1, ..., a_n], [W_1, ..., W_m])$.
- To model that a nurse takes care of at most $children^{\max}$ infants and at least one, we use a global cardinality constraint [8]: `cardinality`$(1, [N_1, ..., N_n], children^{\max})$.
- The constraint that a nurse can work in at most one zone is modelled by a pairwise-disjoint constraint `pairwiseDisjoint(`$[Z_1, ..., Z_p]$`)`, where Z_k is an array of variables containing the variables N_i associated with zone k.

The Comet *Program.* The model in Comet is shown in Listing 1.1. Lines 1–3 declare the decision variables. Line 4 declares the arrays for the zones, which are filled in lines 5–7. The objective function is in lines 8–9 and 11. Lines 12–14 depict the constraints. The `pairwiseDisjoint` constraint introduces set-variables representing the set of nurses working in each zone $NS_k = \bigcup_{i \in \mathcal{P}_k} N_i$. The set NS_k is maintained with a global constraint `unionOf`. Then, the pairwise empty intersections between the set variables are enforced with a global disjoint constraint. Comet uses a reformulation with channeling constraints and a global cardinality constraint as explained in [9,1].

The search is implemented in the `using` block in lines 16–24. The search dynamically breaks the value symmetries originating from the nurse interchangeability. The patient having the largest acuity is selected first in line 17. Then the search tries to assign a nurse to this patient, starting first with those with the smaller load (lines 19–22). The symmetry breaking is implemented by considering the already assigned nurses and at most one additional nurse without any assigned patient (a similar technique was used for the steel mill slab problem in [4]). Value `mn` is the maximal index of a nurse already assigned to a patient. The `tryall` statement considers all the nurse indexes until `mn+1` (nurse `mn+1` having currently no patient).

Experimental Results. As a first experiment, we generated 10 instances with 2 zones, as was the case for the real instances studied in [5]. These instances have about 10–15 nurses, 20–30 infants, and cannot be solved by the MIP model. All the instances were solved optimally with our Comet model in less than 30

Listing 1.1. Patient-Nurse Assignment Model

```
1  var<CP>{int} N[patients](cp,nurses);
2  var<CP>{int} W[nurses](cp,1..MaxAcuity);
3  var<CP>{int} spreadAcuity(cp,0..System.getMAXINT());
4  var<CP>{int}[] Z[zones];
5  int k = 1;
6  forall(i in zones,j in 1..nbPatientsInZone[i])
7      Z[i][j] = N[k++];
8  minimize<cp>
9     spreadAcuity
10 subject to {
11    cp.post(spread(W,sum(p in patients) acuity[p],spreadAcuity));
12    cp.post(multiknapsack(N,acuity,W));
13    cp.post(cardinality(minNbPatients,N,maxNbPatients));
14    cp.post(pairwiseDisjoint(Z));
15 }
16 using {
17   forall(p in patients: !N[p].bound()) by (−acuity[p],N[p].getSize()) {
18     int mn = max(0,maxBound(N));
19     tryall<cp>(n in nurses: n <= mn + 1) by (W[n].getMin())
20         cp.label(N[p],n);
21     onFailure
22         cp.diff(N[p],n);
23   }
24 }
```

Table 1. Patients to Nurses Assignment Problem with 2 zones and minimization of L_2 with `spread`

m	n	#fails	time(s)	avg workload	sd. workload
11	28	511095	170.2	86.09	2.64
11	29	1126480	302.0	80.27	1.76
10	26	104931	24.7	76.50	2.29
12	30	259147	136.5	83.42	1.93
10	28	2990450	1138.5	91.80	6.84
10	26	779969	206.9	88.40	2.29
12	29	555243	198.2	80.08	2.72
10	27	931858	343.9	90.60	5.33
10	25	1616689	434.5	82.70	7.32
8	22	4160	1.2	87.50	3.12

minutes (the time constraint specified in [5] by the hospital to find the assignment). Table 1 depicts the experimental results. All results are using COMET 1.1 [2] on 2.4 GHz Intel Core Duo with 4GB running MacOS 10.5.6.

6 A Two-Step CP Model

The basic CP model can solve 2-zone instances but has great difficulty for 3 zones or more. We now show how to simplify the resolution by a two-step approach which first pre-computes the number of nurses assigned to each zone and then assigns the patients to nurses. This simplifies the resolution by

1. removing one degree of flexibility which is the number of nurses in each zone.
2. removing the disjointness constraint since the set of nurses that can be assigned to each patient can be pre-computed.

A Relaxation. This first step is important because the decomposition may be significantly sub-optimal if these numbers are not properly chosen. Indeed, the number of nurses assigned to each zone has a crucial impact on the quality of the balancing. However, after visualizing some optimal solutions, we observed that the workloads of the nurses are extremely well balanced (almost the same) inside the zones. This suggested solving a relaxation of the problem to discover a good distribution of the nurses to the zones. The relaxation allows the acuity of a child in a zone to be distributed among the nurses of that zone (continuous relaxation of the acuity). Since the acuity of a child can be split, the relaxed problem will have an optimal solution where the nurses of a zone have exactly the same workload $\frac{A_k}{x_k}$, i.e., the total acuity $A_k = \sum_{i \in \mathcal{P}_k} a_i$ of zone k divided by the number of nurses x_k in zone k. This is schematically illustrated on Figure 2 for a two-zone relaxation problem and stated in Theorem 1.

Theorem 1. *An optimal solution of the relaxed problem must have the same workload for all the nurses in a given zone.*

Proof. Otherwise, given m variables $[W_1, \ldots, W_m]$ with sum $s = \sum_{i=1}^{m} W_i$, the L_2 criterion can be improved on these variables if two of them can be made

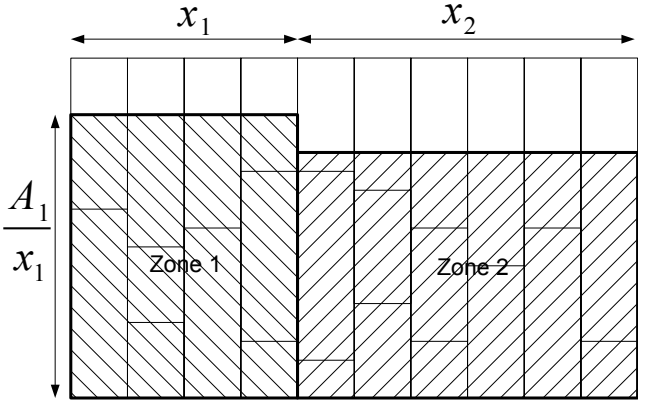


Fig. 2. Illustration of a solution of the relaxation solved to find the number of nurses in each zone

closer (2 nurses of the same zone with a different workload). Let W_i and W_j be the variables that can be made closer and assume without loss of generality that $W_i > W_j$. The variables after modification are respectively W'_i and W'_j. If W_i and W_j are made closer this means that $W'_i - W'_j < W_i - W_j$. Since the sum is fixed then $W'_i + W'_j = W_i + W_j$. Thus $W_i - W'_i = W'_j - W_j$ and so there exists δ with $\frac{(W_i - W_j)}{2} \geq \delta > 0$ such that $W_i - W'_i = \delta = W'_j - W_j$. That is $W'_i = W_i - \delta$ and $W'_j = W_j + \delta$. The starting sum of square deviations with formula (11) is $\Delta = m \cdot \sum_{i=1}^{m}(W_i)^2 - s^2$. With W'_i and W'_j it becomes $\Delta' = m \cdot (\sum_{k \neq i,j}(W_k)^2 + (W_i - \delta)^2 + (W_j + \delta)^2) - s^2 = \Delta - 2m\delta \cdot (W_i - W_j - \delta)$. Since $(W_i - W_j - \delta > 0)$, we have $\Delta' < \Delta$. □

Given Theorem 1, the mathematical formulation of the relaxed problem is

$$\min \sum_{k=1}^{p} x_k \cdot \left(\frac{A_k}{x_k} - \sum_{j=1}^{p} \frac{A_j}{m} \right)^2 \tag{4}$$

$$s.t. \sum_{k=1}^{p} x_k = m \tag{5}$$

$$x_k \in \mathbb{Z}_0^+ \tag{6}$$

The workload of all the nurses of zone k is $\frac{A_k}{x_k}$ and the average workload is $\sum_{j=1}^{p} \frac{A_j}{m}$. Hence the contribution to the L_2 criterion for the x_k nurses of zone k is $x_k \cdot \left(\frac{A_k}{x_k} - \sum_{j=1}^{p} \frac{A_j}{m} \right)^2$.

Solving the Relaxation. In our CP model, we approximate this relaxation in $O(p \cdot log(p))$ time. First, we solve the continuous relaxation of the problem, i.e., we drop the integrality constraint (6). The solution to this continuous optimization problem is $x_k = m \cdot \frac{A_k}{\sum_{j=1}^{p} A_j}$, which corresponds to assigning the average workload $\sum_{j=1}^{p} \frac{A_j}{m}$ to every nurse. The continuous solution $x_k = m \cdot \frac{A_k}{\sum_{j=1}^{p} A_j}$ can be transformed greedily into an integer solution using the following steps:

- By developing the objective (4), it appears that it is equivalent to minimize $\sum_{k=1}^{p} \frac{(A_k)^2}{x_k}$.
- The transformation into an integer solution starts by first rounding up the number of nurses in every zone $x_k = \lceil m \cdot \frac{A_k}{\sum_{j=1}^{p} A_j} \rceil$. The effect is that the constraint (5) may be violated and the objective might decrease.
- Then, the $x_k > 1$ are considered to be decreased by one unit until the constraint (5) is satisfied again. The index k of the next x_k to be decreased is $\text{argmin}_k \{ \frac{A_k^2}{x_k - 1} - \frac{A_k^2}{x_k} \}$, i.e., the variable that will increase the least its corresponding term in the equivalent objective $\sum_{k=1}^{p} \frac{A_k^2}{x_k}$.

Our experimental results show that this approximation is optimal on all the instances the first CP model solved.

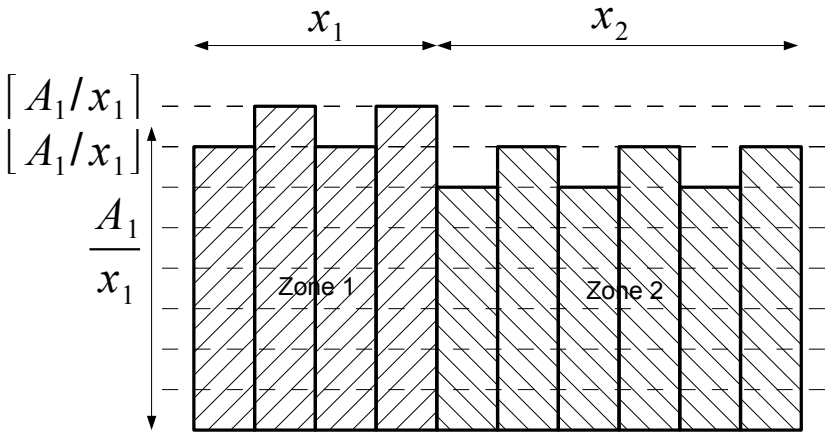


Fig. 3. Illustration of the Lower Bound on L_2 using the Pre-Computation of the Number of Nurses in Each Xone

Lower Bound on the Spread. The pre-computation of the number of nurses assigned to each zone is also instrumental in computing a lower bound on the L_2 criterion. Inside a zone, the average load is $\mu_k = A_k/x_k$. Since the acuity of patients are integers, we can strengthen the lower bound of the objective (4) by enforcing the workloads of nurses of zone k to be either $\lfloor \mu_k \rfloor$ or $\lceil \mu_k \rceil$. This is illustrated on Figure 3. Since the total workload of zone k must remain A_k, the distribution of the workload among $\lfloor \mu_k \rfloor$ and $\lceil \mu_k \rceil$ are given respectively by $\alpha_k = A_k + x_k \cdot (1 - \lceil \mu_k \rceil)$ and $\beta_k = x_k - \alpha_k$. The lower bound on the spread variable $\underline{\Delta}^{\mathbb{Z}}$ computed with formula (1) is thus

$$m \cdot \sum_{k=1}^{p} (\alpha_k \cdot \lceil \mu_k \rceil^2 + \beta_k \cdot \lfloor \mu_k \rfloor^2) - (\sum_{k=1}^{p} A_k)^2. \tag{7}$$

The Comet *Model.* The two-step CP model in Comet is given in Listing 1.2 and assumes that the x_k are already computed. The model does not create the N variables in line 2: These will be created at the same time as the zone arrays, since their domains are now restricted to a subset of the nurses. Lines 6–12 create the zone arrays, line 10 constructing the array for zone i. Note that the domains of these variables are defined in lines 9 and 11, using the number of nurses assigned in the zones. Lines 13–15 assign the zone variables to the nurse variables (the opposite of the first model, since the zone variables now have restricted domains). The constraints are similar but there is no longer a need for the `pairwiseDisjoint` constraint. The search in lines 23–34 is a little bit more complicated as the patients are assigned one zone at a time. The dynamic symmetry breaking scheme is the same but adapted to this by zone assignment.

Table 2 reports the results obtained on the same 2-zones instances as for Table 1 using the pre-computation of the number of nurses assigned to each

Listing 1.2. Two steps Patient-Nurse Assignment Model

```
1  Solver<CP> cp();
2  var<CP>{int} N[patients];
3  var<CP>{int} W[nurses](cp,1..MaxAcuity);
4  var<CP>{int} spreadAcuity(cp,0..System.getMAXINT());
5  var<CP>{int}[] Z[zones];
6  range nursesOfZone[zones];
7  int j=1;
8  forall(i in zones) {
9      nursesOfZone[i] = j..j+x[i]−1;
10     Z[i] = new var<CP>{int}[1..nbPatientsInZone[i]](cp,nursesOfZone[i]);
11     j += x[i];
12 }
13 int k = 1;
14 forall(i in zones,j in 1..x[i])
15      N[k++] = Z[i][j];
16 minimize<cp>
17     spreadAcuity
18 subject to {
19     cp.post(spread(W,sum(p in patients) acuity[p],spreadAcuity));
20     cp.post(multiknapsack(N,acuity,W));
21     cp.post(cardinality(minNbPatients,N,maxNbPatients));
22 }
23 using {
24   forall(i in zones){
25     forall(p in Z[i].rng(): !Z[i][p].bound()) by(−acuityByZone[i][p],Z[i][p].getSize()){
26         int shift = i==1? 0 : nursesOfZone[i−1].getUp();
27         int mn = max(0,maxBound(Z[i]))+shift;
28         tryall<cp>(n in nursesOfZone[i]: n <= mn + 1) by (W[n].getMin())
29             cp.label(Z[i][p],n);
30         onFailure
31             cp.diff(Z[i][p],n);
32     }
33   }
34 }
```

zone. The last column is the lower bound obtained with equation (7). A first observation is that the computation times are greatly reduced. They do not exceed 10 seconds with the new model, while they were over 1000 seconds for the most difficult instances with the old one. The CP model finds the correct number of nurses in the first step, since the standard deviation with previous model are exactly the same (hence optimum) as the optimal values in Table 1. It is also interesting to see that the lower bound is reasonably close to the optimum values which also validates the approach.

Since the instances with 2 zones can now be solved easily, we tried to solve instances with 3 zones. The results are presented on Table 3. Only 6 instances (out of 10) could be solved optimally within 30 minutes with this two-step approach.

Table 2. Patients to Nurses Assignment Problem with 2 zones with precomputation of the number of nurses in each zone

m	n	#fails	time(s)	avg workload	sd. workload	lb. sd.
11	28	25385	4.5	86.09	2.64	2.23
11	29	4916	1.4	80.27	1.76	0.62
10	26	458	0.1	76.50	2.29	2.29
12	30	17558	6.7	83.42	1.93	1.19
10	28	29865	4.8	91.80	6.84	6.81
10	26	3705	1.0	88.40	2.29	1.43
12	29	6115	1.2	80.08	2.72	0.64
10	27	1109	0.4	90.60	5.33	5.22
10	25	3299	0.6	82.70	7.32	6.71
8	22	127	0.0	87.50	3.12	3.04

Table 3. Patients to Nurses Assignment Problem with 3 zones with precomputation of the number of nurses in each zone

sol	m	n	#fails	time(s)	avg workload	sd. workload	lb. sd.
1	15	42	19488	5.3	84.20	3.04	2.93
1	18	43	3619310	919.2	79.78	5.84	5.49
0	17	43	9023072	1800.0	81.41	4.75	3.45
1	17	42	483032	106.9	83.82	5.65	5.59
0	18	43	7124370	1800.0	81.00	7.11	4.94
1	14	38	590971	145.2	85.36	3.08	2.16
0	19	48	3786580	1800.0	87.42	3.18	2.30
1	16	44	3888210	839.8	84.88	6.70	6.39
0	19	49	5697272	1800.0	86.00	2.70	1.95
1	17	41	61250	17.3	82.18	3.40	3.07

7 A Two-Step CP Model with Decomposition

The previous approach can solve easily two-zone problems but has difficulties to solve 3 zones problems and instances with more that 3 zones are intractable. It thus seems natural to decompose the problem by zone and to balance the workload of nurses inside each zone independently rather than balancing the workload of all the nurses globally. Interestingly, this decomposition preserves optimality, i.e., it reaches the same solution for the L_2 criterion as the two-step approach of Section 6 for a given pre-computation of the number of nurses assigned in each zone. In other words, given the pre-computed number of nurses in each zone, it is equivalent to minimize L_2 among all the nurses at once or to minimize L_2 separately inside each zone. We now prove this result formally.

Lemma 1. *Minimizing* $n \cdot \sum_{i=1}^{x_k}(y_i - A_k/x_k)^2$ *such that* $\sum_{i=1}^{x_k} y_i = A_k$ *is equivalent to minimizing* $n \cdot \sum_{i=1}^{x_k}(y_i - (A_k/x_k + c))^2$ *such that* $\sum_{i=1}^{x_k} y_i = A_k$.

Table 4. Patients to Nurses Assignment Problem with 3 zones with precomputation of the number of nurses in each zone and decomposition by zone

m	n	#fails	time(s)	avg workload	sd. workload	lb. sd.
15	42	203	0.1	84.20	3.04	2.93
18	43	608	0.1	79.78	5.84	5.49
17	43	8134	1.1	81.41	4.46	3.45
17	42	345	0.1	83.82	5.65	5.59
18	43	24994	3.2	81.00	5.77	4.94
14	38	151	0.0	85.36	3.08	2.16
19	48	3695	0.8	87.42	3.07	2.30
16	44	384	0.1	84.88	6.70	6.39
19	49	2056	0.4	86.00	2.49	1.95
17	41	776	0.2	82.18	3.40	3.07

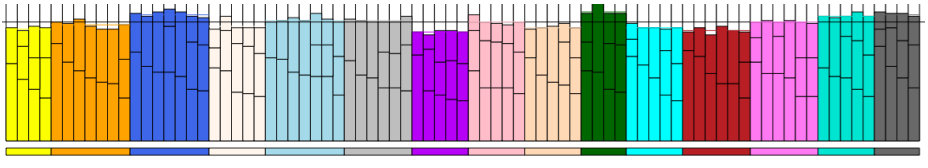


Fig. 4. Solution of a 15-Zone Instance

Proof. The first objective can be reformulated from formula 1 as $x_k \cdot \sum_{i=1}^{x_k} y_i^2 - A_k^2$. The second one can be reformulated after some algebraic manipulations as $c^2 \cdot x_k^2 + x_k \cdot \sum_{i=1}^{x_k} y_i^2 - A_k^2$. Since they differ only by a constant term, they produce the same set of optimal solutions. □

Theorem 2. *It is equivalent to minimize L_2 among all the nurses at once or to minimize L_2 separately inside each zone.*

Proof. This follows directly from Lemma 1. If the minimization of L_2 is performed globally for all the nurses, the least square L_2 criterion is computed with respect to the global average load of all the nurses that is wrt $\sum_{k=1}^{p} A_k/m$. This corresponds to choosing c in Lemma 1 equal to the difference between the average load in zone k and the global average load: $c = \sum_{k=1}^{p} A_k/m - A_k/x_k$. □

We solved again the 3-zone instances with the decomposition method. The results are given on Table 4. One can observe that, as expected, the objectives are the same for the instances that could be solved optimally in Table 3. For the remaining ones, the algorithm produces strictly better solutions. The time is also significantly smaller. Figure 4 shows a COMET visualization of a solution for a 15-zones instance with 81 nurses and 209 patients. This instance could be solved in only 7 seconds and 10.938 fails.

8 Conclusion

This paper considered the daily assignment of newborn infant patients to nurses in a hospital. The objective is to balance the workload of the nurses, while satisfying a variety of side constraints. Prior work proposed a MIP model for this problem which exhibits two limitations. It did not scale to large instances and its objective function did not balance the workload properly. The paper presented a direct CP model which balances the load appropriately and easily solve 2-zone instances. To scale the CP approach, the paper showed how to decompose the problem in two steps: an assignment of nurses to zones followed by the assignment of nurses to patients. The first step is obtained from a relaxation of the problem which could be solved quickly. The second step is solved by a simplification of the direct model. This 2-step approach dramatically improved the results on the 2-zone instances and could solve some 3-zone instances. The paper then showed that the zone problems can be solved independently without quality loss. This resulting CP model solves 3-zone problems almost instantly and is highly scalable. For instance, a 15-zone problem with 81 nurses and 209 patients was solved in 7 seconds.

There are a number of interesting issues left to investigate. It would be interesting to study the quality of the approximation performed in the first step. Our experimental results indicate that it is optimal on all our tested intances but a performance guarantee would be desirable. Alternatively, we could consider solving this first step exactly, an algorithmic issue we need to investigate. In addition, it would be interesting to study problems in which nurses have qualifications which restrict their possible zone assignments.

References

1. Bessière, 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. DYNADEC. Comet 1.1 release (2009), www.dynadec.com
3. Gorard, S.: Revisiting a 90-year-old debate: The advantages of the mean deviation. British Journal of Educational Studies, 417–439 (2005)
4. Van Hentenryck, P., Michel, L.: The steel mill slab design problem revisited. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 377–381. Springer, Heidelberg (2008)
5. Mullinax, C., Lawley, M.: Assigning patients to nurses in neonatal intensive care. Journal of the Operational Research Society 53, 25–35 (2002)
6. Pesant, G., Régin, J.C.: Spread: A balancing constraint based on statistics. In: van Beek, P. (ed.) CP 2005. LNCS, vol. 3709, pp. 460–474. Springer, Heidelberg (2005)
7. Pesant, G.: Constraint-based rostering. In: The 7th International Conference on the Practice and Theory of Automated Timetabling, PATAT 2008 (2008)
8. Régin, J.-C.: Generalized arc consistency for global cardinality constraint. In: AAAI 1996, pp. 209–215 (1996)
9. Régin, J.C.: Habilitation à diriger des recherches (hdr): modelization and global constraints in constraint programming. Université Nice (2004)

10. Schaus, P.: Balancing and bin-packing constraints in constraint programming. PhD thesis, Université catholique de Louvain, INGI (2009)
11. Schaus, P., Deville, Y., Dupont, P., Régin, J.C.: Simplification and extension of spread. In: 3rd Workshop on Constraint Propagation And Implementation (2006)
12. Schaus, P., Deville, Y., Dupont, P., Régin, J.C.: The deviation constraint. In: Van Hentenryck, P., Wolsey, L.A. (eds.) CPAIOR 2007. LNCS, vol. 4510, pp. 260–274. Springer, Heidelberg (2007)
13. Schaus, P., Deville, Y., Dupont, P.: Bound-consistent deviation constraint. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 620–634. Springer, Heidelberg (2007)
14. Shaw, P.: A constraint for bin packing. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 648–662. Springer, Heidelberg (2004)
15. Simonis, H.: Models for global constraint applications. Constraints 12, 63–92 (2007)

Learning How to Propagate Using Random Probing

Efstathios Stamatatos and Kostas Stergiou

Department of Information and Communication Systems Engineering
University of the Aegean, Samos, Greece
{stamatatos,konsterg}@aegean.gr

Abstract. In constraint programming there are often many choices regarding the propagation method to be used on the constraints of a problem. However, simple constraint solvers usually only apply a standard method, typically (generalized) arc consistency, on all constraints throughout search. Advanced solvers additionally allow for the modeler to choose among an array of propagators for certain (global) constraints. Since complex interactions exist among constraints, deciding in the modelling phase which propagation method to use on given constraints can be a hard task that ideally we would like to free the user from. In this paper we propose a simple technique towards the automation of this task. Our approach exploits information gathered from a random probing preprocessing phase to automatically decide on the propagation method to be used on each constraint. As we demonstrate, data gathered though probing allows for the solver to accurately differentiate between constraints that offer little pruning as opposed to ones that achieve many domain reductions, and also to detect constraints and variables that are amenable to certain propagation methods. Experimental results from an initial evaluation of the proposed method on binary CSPs demonstrate the benefits of our approach.

1 Introduction

Constraint propagation is a crucial reason for the success of constraint programming in solving hard combinatorial problems. Hence, this topic has attracted considerable interest and numerous generic and specialized constraint propagation techniques have been developed. As a result, when modelling a CSP there are, quite often, many choices regarding the propagation method to be used on the constraints of the problem. For example, advanced constraint solvers offer efficient filtering algorithms for both bounds consistency and generalized arc consistency (GAC), also known as domain consistency, for certain global constraints (e.g. alldifferent). The former are typically faster but the latter are stronger. As another example, there are numerous choices for local consistencies that can be applied on binary constraints. Despite the wealth of choices for constraint propagation, simple constraint solvers usually only apply a standard method, typically (G)AC, on all constraints throughout search. For instance, arc consistency is almost exclusively used on binary constraints. Advanced solvers can also apply a

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 263–278, 2009.

predetermined propagation method but in addition they allow for the modeler to choose among an array of propagators for certain (global) constraints. Finally, some solvers employ mechanisms for dynamically determining the propagation method during search based on the event that triggered propagation. Typically this is done on particular types of constraints such as arithmetic constraints.

Since complex interactions exist among constraints, which may only be revealed during search, deciding in the modelling phase which propagation method to use on given constraints can be a hard task that we would like to free the user from. For the case of a binary constraint, for example, it is very difficult to know a priori if choosing to propagate it using a strong local consistency such as singleton arc consistency or path consistency will pay off. Ideally, we would like to avoid using a strong propagation method on a constraint that will never, or rarely, cause domain reductions during search as this would result in needless cpu effort. Also, it would be preferable to choose say a cheap bounds consistency propagator for a constraint if we knew that stronger propagators achieve little extra pruning. But again this is very difficult to predict prior to search.

Deciding on which propagator to use for certain constraints based on static features of the problem is part of the modelling process and has attracted considerable interest. However, most of these works are problem-specific and require specialized modelling skills. The dynamic selection of propagators during search has also been investigated before, but to a far lesser extent (for example [10,17,15,19,20]). In this paper we propose a simple novel technique towards automating the task of choosing the right propagation method for individual constraints prior to search. Our approach differs from previous works as it does not the require the modeler's involvement in the process. Furthermore, it can be easily combined with dynamic methods or in itself extended to operate dynamically during search.

The proposed approach, which we call LPP (**L**earning **P**ropagators through **P**robing) uses information gathered from a random probing preprocessing phase to automatically decide on the propagation method to be used on each constraint. A *random probe* is a single run of a search algorithm with random variable ordering, a fixed cut-off, and propagation turned on. Random probes provide a sample of diverse areas in the search space and in our case can provide useful information regarding the percentage of fruitful revisions for each constraint, the number of value deletions caused by certain propagation methods, etc. We show that by exploiting such data the solver is able to accurately differentiate between constraints that offer little pruning as opposed to ones that achieve many domain reductions. As a result, the solver may automatically choose to propagate the former constraints using a low-cost propagation method and the latter using a stronger, and more expensive, propagator. Further to this, LPP can detect constraints and variables that are amenable to certain propagation methods. As we explain, these are accomplished through the use of a clustering algorithm that partitions the constraints into clusters having different features.

Although the method proposed is generic, we only present an initial evaluation on binary CSPs. To obtain the required data from random probing, we built

a *staged propagator* [18] for binary problems, i.e. a set of multiple propagators having varying cost and pruning power. This propagator progressively applies various local consistencies starting with bounds consistency and culminating in bounds singleton arc consistency. In a series of random probes where the propagator is applied after each variable assignment, we recorded the number of times each constraint was fruitfully revised, the local consistency that was responsible for each such revision, and the number of value deletions caused by each consistency. A comparison of these results to similar results obtained by running heuristically guided search to termination (using the same propagator) revealed interesting patterns. For instance, constraints that display a very low percentage of fruitful revisions can be accurately discovered through random probing.

Our methodology exploits the results of random probing to decide how to propagate each constraint during search using simple heuristic rules. Experimental results from various benchmarks demonstrate that LPP outperforms MAC, i.e. the standard search algorithm for binary problems, on hard instances, sometimes by a very large margin. Also, LPP is quite competitive with heuristics from [19] which dynamically switch between two local consistencies throughout search.

The rest of the paper is structured as follows. Section 2 gives some necessary background and introduces notation. In Section 3 we describe the staged propagator for binary constraints that we used in our experiments. Section 4 presents the LPP framework and gives experimental results demonstrating the accuracy of its predictions. In Section 5 we make an experimental evaluation of LPP on various binary problems. In Section 6 we discuss related work, and finally in Section 7 we conclude.

2 Background

A *Constraint Satisfaction Problem* (CSP) is a tuple (X, D, C) where: $X = \{x_1, \ldots, x_n\}$ is a set of n variables, $D = \{D(x_1), \ldots, D(x_n)\}$ is a set of domains, one for each variable, and $C = \{c_1, \ldots, c_e\}$ is a set of e constraints. Each constraint c is a pair $(var(c), rel(c))$, where $var(c) = \{x_1, \ldots, x_k\}$ is an ordered subset of X, and $rel(c)$ is a subset of the *Cartesian* product $D(x_1)\mathrm{x}\ldots\mathrm{x}D(x_k)$. In a binary CSP, a directed constraint c, with $var(c) = \{x_i, x_j\}$, is *arc consistent* (AC) iff for every value $a_i \in D(x_i)$ there exists a value $a_j \in D(x_j)$ s.t. the 2-tuple $<(x_i, a_i), (x_j, a_j)>$ satisfies c. In this case (x_j, a_j) is called an AC-support of (x_i, a_i) on c. A problem is AC iff there is no empty domain in D and all the constraints in C are AC. Maintaining arc consistency (MAC), which the most commonly used search algorithm for binary CSPs, applies AC to the problem after each variable assignment. A variable x_i is *singleton arc consistent* (SAC) iff for each value $a_i \in D(x_i)$ after assigning a_i to x_i and applying AC there is no empty domain [9]. A problem is SAC iff all variables are SAC.

Assuming finite integer domains for the variables, each domain $D(x_i)$ has a *minimum* and a *maximum* value, called the *bounds* of $D(x_i)$ and denoted by $min_{D(x_i)}$ and $max_{D(x_i)}$ respectively. A directed constraint c is *bounds consistent* (BC) iff both $min_{D(x_i)}$ and $max_{D(x_i)}$ have AC-supports on c. This definition of BC corresponds to BC(D) as defined in [4]. *Bounds SAC* (BSAC) is a

restricted version of SAC that only applies SAC on the bounds of the variables' domains [14].

A directed constraint c, with $var(c) = \{x_i, x_j\}$, is *max restricted path consistent* (maxRPC) iff it is AC and for each value (x_i, a_i) there exists a value $a_j \in D(x_j)$ that is an AC-support of (x_i, a_i) s.t. the 2-tuple $<(x_i, a_i), (x_j, a_j)>$ is *path consistent* (PC) [9]. A tuple $<(x_i, a_i), (x_j, a_j)>$ is PC iff for any third variable x_m there exists a value $a_m \in D(x_m)$ s.t. (x_m, a_m) is an AC-support of both (x_i, a_i) and (x_j, a_j).

The *revision* of a binary constraint c, with $var(c) = \{x_i, x_j\}$, using a local consistency A is the process of checking whether the values of x_i verify the property of A. For example, the revision of c using AC verifies if all values in $D(x_i)$ have AC-supports on c. We say that a revision is *fruitful* if it deletes at least one value, while it is *redundant* if it achieves no pruning.

3 A Staged Propagator for Binary Constraints

Staged propagators were introduced by Schulte and Stuckey as a way to efficiently apply the different propagators that may be available for certain types of constraints [18]. A staged propagator for a constraint c is a set of propagators for c, having varying pruning power and cost, that are combined together. Each staged propagator has an internal state variable, called the state of the propagator, which determines the individual propagation method to be used once an event that triggers propagation for c occurs. For example, assuming that variable x_i appears in c, the removal of $min_{D(x_i)}$ may force the staged propagator to enter a state where a bounds consistency algorithm will be applied.

Here we describe a simple staged propagator for binary constraints that combines together four local consistencies: BC, AC, maxRPC, and BSAC. For simplicity, we will use the term *stage* to refer to one of the local consistencies that are combined together. For example, value deletions caused by the AC stage will refer to value deletion caused by the application of AC. We slightly abuse the definition of a staged propagator as we have implemented a variable-oriented propagation scheme where variables are the entities added to and removed from the propagation queue. Although in constraint solvers like Ilog Solver and Gecode the entities handled by the propagation queue are propagators, in the case of binary constraints variable-oriented propagation is more efficient. This has been previously demonstrated for arc consistency algorithms (e.g. [6,1]), but it is also true for higher level consistencies. To be precise, our experimental results showed a speed-up of up to three times in favor of variable-oriented propagation compared to its constraint-oriented counterpart[1].

Figure 1 gives an abstract high-level description of the staged propagator used. During preprocessing with random probing this propagator is applied as shown in Figure 1 after each variable assignment (*current_variable* denotes the currently assigned variable). The propagator removes a variable x_i from the queue and revises all constraints involving x_i. That is, it applies all four stages successively,

[1] These experimental results are omitted because of space restrictions.

function `Binary_Staged_Propagation`(X,D,C,*current_variable*)
1: add *current_variable* to Q
2: **while** $Q \neq \emptyset$
3: remove variable x_i from Q;
4: **for** any constraint c, with $var(c) = \{x_j, x_i\}$
5: successively apply BC,AC,maxRPC to c;
6: apply BSAC to x_j;
7: **if** $D(x_j) = \emptyset$ **then return** FAILURE;
8: **else if** $D(x_j)$ has been reduced **then** add x_j to Q;
9: **return** TRUE;

Fig. 1. A staged propagator for binary CSPs

as long as no domain wipeout (DWO) occurs. After the application of each stage the propagator records information concerning the pruning effects of the relevant constraint and the currently applied stage, as detailed in the next section (this is not shown in Figure 1 for simplicity). Once the process terminates, the data gathered is processed as will be explained below to select the propagation method to be applied on each constraint during search. Note that using the staged propagator in its full power throughout search is prohibitively expensive as it incurs many redundant revisions resulting in cpu times that can be orders of magnitude slower than MAC. Also, using SAC instead of BSAC results in more domain reductions albeit with a much higher cost.

4 Learning through Random Probing

In this section we first show that results gathered through random probing, concerning the pruning effects of the constraints, often reflect similar results gathered by running search to completion. Then we explain how LPP exploits this to decide on the propagator for individual constraints prior to search.

4.1 Accuracy of Learning

The LPP methodology utilizes the staged propagator described previously to gather data concerning the filtering power of the various propagation stages on individual constraints. For each constraint c we record the following information:

1. the number of times c was revised,
2. the ratio of fruitful revisions over the total number of revisions,
3. the ratio of fruitful revisions over the total number of revisions for each of the propagator's stages,
4. the total number of value deletions caused by c,
5. the ratio of value deletions over the total number of deletions caused by each stage separately.

The third item above is computed by simply recording the stage that is responsible for each value deletion during a fruitful revision of a constraint. Table 1

Table 1. Sample data gathered by random probing in a frequency assignment problem

recorded data	c_1	c_2	c_3	...
#revisions	60	63	69	
fr-ratio	0.28	0.47	0.05	
fr_{BC}-ratio	0.08	0.27	0.01	
fr_{AC}-ratio	0.03	0.35	0.01	
fr_{maxRPC}-ratio	0.28	0.00	0.04	
fr_{BSAC}-ratio	0.00	0.00	0.00	
#deletions	51	136	8	
del_{BC}-ratio	0.06	0.60	0.25	
del_{AC}-ratio	0.03	0.40	0.25	
del_{maxRPC}-ratio	0.91	0.00	0.50	
del_{BSAC}-ratio	0.00	0.00	0.00	

depicts part of the data gathered by random probing in tabular form. There is one column for each constraint, and each row corresponds to a piece of information concerning the pruning achieved by the constraints. The sample data shown is taken from a frequency assignment problem where we run 20 random probes each being cut off once 100 nodes (i.e. variable assignments) have been counted.

As one can see, constraint c_1 displayed a relatively high ratio of fruitful to total revisions (28% in row 2), the maxRPC stage contributed at least one value deletion in each of the constraint's fruitful revisions (the number in row 5 is the same as in row 2), and most of the value deletions it caused were due to the maxRPC stage (91% in row 10). Constraint c_2 displayed an even higher ratio of fruitful revisions but this time all value deletions were contributed by the BC and AC stages. Finally, constraint c_3 had a low ratio of fruitful revisions (only 5%) and the 8 value deletions it caused were due to either BC, AC, or maxRPC. BSAC did not contribute any value deletions for these three constraints.

As the data in Table 1 demonstrates, the various constraints can display different behavior with respect to their revisions and the pruning they cause. The interesting question is whether this behavior observed during random probing is relevant to the corresponding behavior of the constraints during heuristically guided search. Or in other words, whether we can "predict" how each constraint will behave based on the random probing results. First of all, to get a better picture of the distribution of the constraints into different patterns of behavior, we run the clustering algorithm fuzzy c-means on the data gathered by random probing. The following paragraph briefly discusses fuzzy c-means and then we present some clustering results.

Fuzzy c-means (FCM) is one of the most frequently used clustering algorithms. FCM allows one piece of data to belong to more than one clusters [5]. To this end, data are bound to each cluster by means of a membership function. Given a predefined number of clusters, FCM iteratively optimizes an objective function that is based on the distance of each data point from the cluster centers and the degree of membership in each cluster. In comparison to k-means, another well-known clustering algorithm, the FCM objective function differs in taking into account the degrees of membership in each cluster as well as an additional parameter (the fuzzifier) that determines the level of cluster fuzziness. A large fuzzifier results in fuzzier clusters whereas a fuzzifier equal to 1 results in crisp

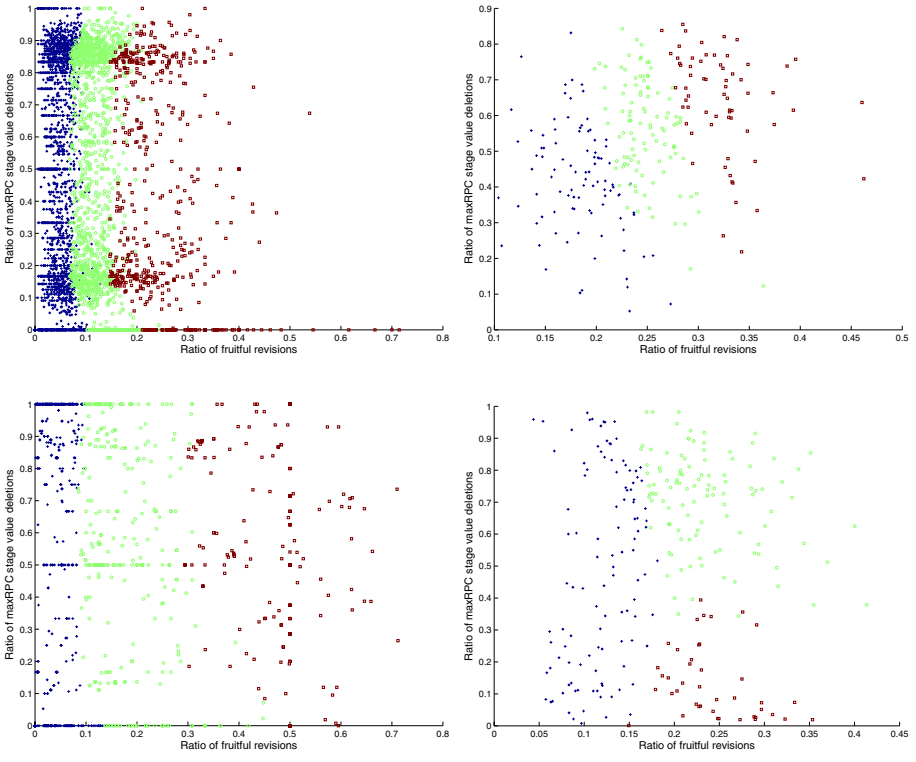


Fig. 2. Clustering results from a frequency assignment (left plots) and a random problem (right plots). The x-axis gives the ratio of fruitful revisions while the y-axis gives the ratio of value deletions due to the maxRPC stage. The top plots show clusters formed from the random probing results while the bottom ones show clusters formed from search results.

partitioning. The iteration stops when the degrees of membership of data in each cluster are not significantly modified in successive iteration steps. Similarly to k-means, FCM tends to group data spatially according to their distance from the cluster centers. However, this spatial partitioning is more flexible due to the fuzzifier parameter. In this paper, we used 3 clusters and set the fuzzifier to 2 based on preliminary experiments.

The top plots in Figures 2 and 3 show how constraints are clustered after running FCM on the data gathered by random probing for four different problems. The input parameters for FCM were the ratio of fruitful revisions and the corresponding ratios for the propagation stages. The horizontal axis in the figures gives the ratio of fruitful revisions while the vertical axis gives the ratio of value deletions caused by the maxRPC stage. As is evident, in all four problems the three clusters created partition the constraints mainly according to the ratio of fruitful revisions. Going from left to right the three clusters include constraints with increasing ratio. Apart from this differentiation additional useful

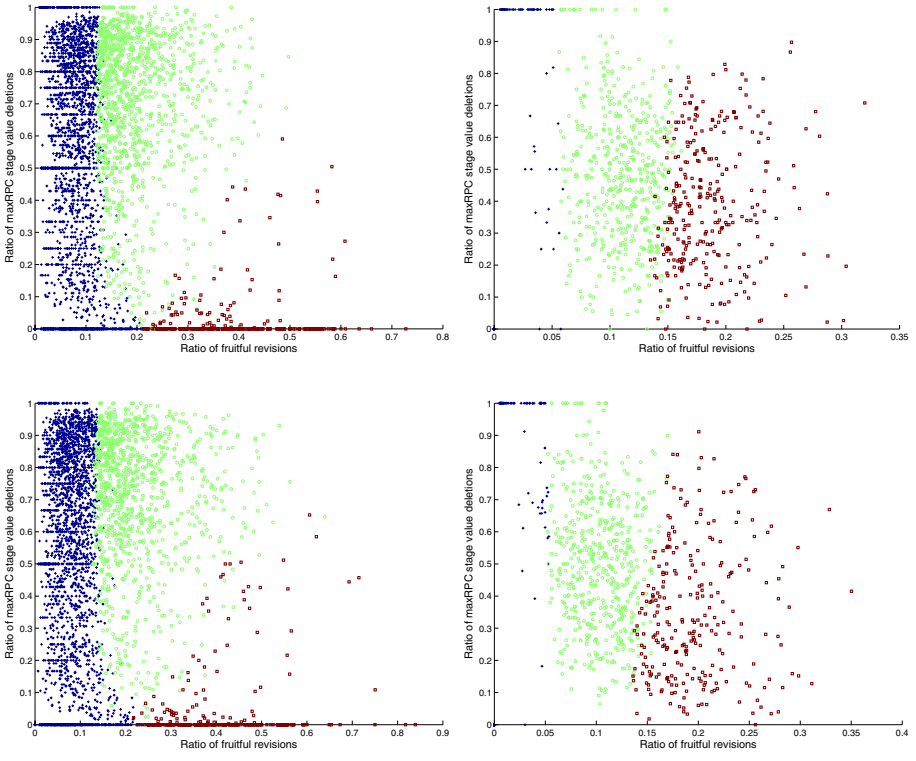


Fig. 3. Clustering results from a frequency assignment (left plots) and a quasigroup completion problem (right plots). The top plots show clusters formed from the random probing results while the bottom ones show clusters formed from accumulated probing and search results.

information can also be extracted. For example, the rightmost cluster in the top left plot of Figure 3 mostly includes constraints with low ratio of value deletions by maxRPC. Hence, it seems that for these constraints the maxRPC stage has little effect.

To answer the question posted above on whether the pruning behavior of the constraints during random probing is relevant to their behavior during search, we run a search algorithm that applied the staged propagator after each variable assignment. We also recorded the same information regarding revisions and value deletions as during random probing. The bottom plots in Figure 2 show how constraints are clustered after running FCM on this data for the same two problems as in the top plots. Figure 3 displays similar results but in these two cases random probing was applied prior to search. That is, the data is accumulated from both preprocessing and search. In the three structured problems (the left problem in Figure 2 and both problems in Figure 3) the distribution of the constraints in the three clusters resembles the corresponding distribution from the random probing results, especially in Figure 3. In contrast, the clusters in

the random problem (right plots in Figure 2) are quite different compared to the corresponding clusters from random probing. This indicates that in the absence of structure it is difficult to predict the behavior of the constraints using random probing.

Note that the bottom left plot in Figure 2 includes fewer data points (i.e. constraints) than the top left one. This is because heuristically guided search focuses on certain parts of the search space and as a result many constraints are not revised at all, which means that the corresponding data points have (0,0) coordinates on the plot. Also, in the bottom plot the three clusters are shifted to the right compared to the top one. However, the membership of constraints to clusters remains similar. That is, most constraints that belong to a particular cluster in the top plot, say the middle one, belong to the corresponding cluster in the bottom plot as well.

Table 2. Accuracy of clusters for various problems. The second column gives the number of constraints in each problem.

instance	e	**% accuracy**	% left cluster accuracy
scen11-f6	4102	70	96
scen11-f7	4102	79	97
driver9	17446	83	89
qcp15-120-9	3149	92	99
qwh20-166-1	7599	95	99
qwh20-166-8	7599	95	99
3-fullins-5-5	33750	58	94
myciel7-4	2359	65	72
frb40-19-0	320	59	73

Table 2 gives further evidence concerning the similarity of the clusters created using the random probing results compared to the clusters created using results from search. Each row in the table gives results from a benchmark problem. These problems are all structured (either real or academic) apart from the last one which was randomly generated (see Section 5 for more details). The third column gives the percentage of constraints that were assigned to corresponding clusters in both the random probing and the final clusterings. The fourth column gives the percentage of constraints that were assigned to the leftmost cluster in the preprocessing clustering and remained assigned to the leftmost cluster in the final clustering. This is particularly useful as it demonstrates the accuracy in identifying constraints that have a low ratio of fruitful revisions. For most problems the percentage is very high, getting close to 100%. As expected, the similarity between the clusterings is lower in the case of the random problem.

4.2 Exploiting Learning to Determine Propagators

Having shown that some important aspects of the pruning behavior that the constraints display can be predicted through random probing, the question that

naturally arises is how to exploit this in order to make informed automatic decisions about the propagators to use on individual constraints during search. A naive answer would be to simply look at the results gathered (e.g. Table 1) and select the propagation stage that caused the highest number of deletions for a constraint c as the propagator to be used on c. Although this is not entirely useless (experiments showed it outperforms MAC!), it suffers from certain drawbacks. Most notably it ignores the ratio of fruitful revisions which is a crucial piece of information. Choosing a strong propagator for constraints that have a low ratio is not cost-effective. For example, following this naive approach the solver would select to propagate constraint c_3 of Table 1 using maxRPC. This can result in many redundant revisions of high cost.

LPP answers the above question by exploiting the results provided by the FCM clustering algorithm and making the decisions using simple (heuristic) rules which basically constitute a decision tree. Note that it is easy to identify the three clusters by looking at the clusters' centers. The cluster whose center has the lowest value of fruitful revisions ratio is the one which includes constraints with low ratio of fruitful revisions. Accordingly, we can differentiate the other two clusters through their centers. The rules we have used are as follows.

- Any constraint belonging to the cluster whose center has the lowest ratio of fruitful revisions (the leftmost cluster in the plots) is propagated with AC or BC, depending on which one has the highest ratio of deletions.
- Any constraint belonging to the cluster whose center has the highest ratio of fruitful revisions (the rightmost cluster in the plots) is propagated with maxRPC if 1) the cluster center's ratio of fruitful revisions by maxRPC (fr_{maxRPC}-ratio in Table 1) is the highest among the three clusters and 2) maxRPC has the highest ratio of deletions (del_{maxRPC}-ratio in Table 1) compared to the other stages for this constraint. Otherwise, it is propagated using heuristic $H^{\vee}_{12}$ from [19]. This heuristic switches between AC and maxRPC during search according to certain conditions explained below.
- Any constraint belonging to the remaining cluster (the middle cluster) is propagated using heuristic $H^{\vee}_{12}$ except if: 1) the cluster center's ratio of fruitful revisions by maxRPC is the highest among the three clusters in which case it is propagated with maxRPC, or 2) the maxRPC stage does not cause any deletions at all, in which case it is propagated with AC.
- BSAC is applied on any variable whose ratio of fruitful calls to BSAC over the total number of calls is more that 0.5. That is, line 6 in Figure 1 is only executed for these variables.

Heuristic $H^{\vee}_{12}$ monitors and counts revisions, DWOs and value deletions for the constraints in the problem. It uses two (user defined) thresholds l_1 and l_2, set to 100 and 10 in this paper, to switch between a weak but cheap local consistency W and a stronger but more expensive one S. A constraint c is made S if the number of times it was revised since the last time it caused a DWO is less or equal to l_1, or if the number of times it was revised since the last time it caused a

value deletion is less or equal to l_2. If none of these conditions holds, c it is made W. In this paper W and S were set to AC and maxRPC respectively. Setting S to BSAC or SAC resulted in a very cost-inefficient method.

A drawback of our method is that the heuristic rules described above were predetermined based on intuition and preliminary experiments, and hence required expertise. The intuition is simple: we select a low-cost propagator for constraints that displayed many redundant revisions during preprocessing, and a high-cost but more efficient one for constraints that displayed many fruitful revisions most of which were due to the high-cost propagator. Automatic generation of heuristic rules is an important topic that requires further research.

5 Experimental Results

In this section we present an initial evaluation of LPP on binary CSPs. We compare the method to the widely used MAC algorithm and also to heuristic $H_{12}^{\vee}$ applied to all constraints of the problem as proposed in [19]. The solver we used applies d-way branching, lexicographic value ordering, the dom/wdeg variable ordering heuristic [7], and restarts. Concerning the restart policy, the initial number of allowed backtracks for the first run has been set to 10 and at each new run the number of allowed backtracks increases by a factor of 1.5. We set the number of random probes to 20 and the cut-off limit for each probe to 100 nodes. We noticed little variance in the results when these settings changed, but finding the "optimal" settings for each problem is an issue that requires further research. Another topic for future work is the use of random probes with random value ordering which may result in even more diverse sampling of the search space. To keep preprocessing times manageable BSAC, which can be quite time consuming, was only applied in 1/5th of the nodes (randomly selected).

We experimented with the following classes of problems: radio links frequency assignment (RLFAPs), graph coloring (GC), haystacks (H), quasigroup completion (QCP), quasigroups with holes (QWH), forced random problems (R). All apart from the last class are structured binary CSPs. Tables 3 and 4 give indicative experimental results. The specific benchmark instances taken from C. Lecoutre's web page. The first table gives results from insoluble problems while the second from soluble ones. For LPP we give both the total cpu time and the time required for random probing and clustering. Note that the time required for clustering is negligible compared to that for random probing.

As results demonstrate, LPP can be considerably more efficient than MAC on the majority of the problems, and especially on the hard insoluble ones. The random probing preprocessing phase consumes a significant portion of the execution time for easier instances, but in most cases this becomes negligible as the problems become harder. Comparing heuristic $H_{12}^{\vee}$ to LPP we can see that the methods are competitive with LPP often being faster despite the time spent on preprocessing. LPP, as well as $H_{12}^{\vee}$, is not competitive with MAC on random

Table 3. Nodes (n) and cpu times (t) in seconds from insoluble problems. The LPP column gives the total cpu time of preprocessing + search and in brackets the time required for preprocessing (i.e. random probing and clustering). A time out limit of 2 hours was set.

type	**instance**		MAC	$H_{12}^{\vee}$	LPP
RLFAP	scen11-f6	n	74,879	7,895	4,871
		t	58	14	66 (50)
RLFAP	scen11-f5	n	321,435	52,750	12,501
		t	254	94	99 (53)
RLFAP	scen11-f4	n	1,110,401	167,786	22,112
		t	856	266	110 (61)
RLFAP	scen11-f3	n	4,995,046	167,596	23,334
		t	3917	274	111 (62)
GC	homer-8	n	228,495	11,770	201,023
		t	102	7	118 (3)
GC	myciel5-5	n	22,640,358	22,640,358	22,640,358
		t	638	2021	842 (1)
GC	myciel6-5	n	6,915,618	6,915,618	6,915,618
		t	654	2815	896 (5)
GC	miles-500-10	n	19,996,866	9,693	18,048
		t	3596	3	15 (9)
H	haystacks-5	n	1,203,768	3,256	942
		t	13	0.5	0.2 (0.1)
H	haystacks-6	n	-	23,328	25,732
		t	>2h.	2	2 (0.2)
QCP	qcp15-120-10	n	8,580,800	2,747,682	113,487
		t	1860	1340	50 (5)
QCP	qcp15-120-13	n	1,007,089	155,971	230,591
		t	235	71	108 (4)

problems, which gives further evidence that the absence of structure hinders the accuracy of the learning process.

Interestingly, on the myciel graph coloring problems maxRPC and BSAC do not offer any more pruning than AC. LPP discovers this during preprocessing and does not select these two consistencies for any constraint. Hence the same node visits but reduced cpu times compared to $H_{12}^{\vee}$ which "blindly" switches between AC and maxRPC during search. However, the second rule of Section 4.2 selects to propagate some constraints using $H_{12}^{\vee}$ which accounts for the increased times compared to MAC. On a negative note, problem homer-8 is an example where LPP fails to interpret the random probing results in an efficient way. Although the leftmost cluster created includes constraints with low ratio of fruitful revisions, this is the only cluster that includes constraints where the maxRPC stage makes value deletions. Despite this, all constraints in this cluster are selected to be propagated with AC or BC which accounts for the significant difference in node visits and cpu time compared to $H_{12}^{\vee}$.

Table 4. Nodes (n) and cpu times (t) in seconds from various soluble problems

type	instance		MAC	$H_{12}^{\vee}$	LPP
QCP	qcp15-120-9	n	135,267	29,812	29,383
		t	30	12	31 (4)
QCP	qcp20-187-1	n	189,942	344,418	172,574
		t	102	262	149 (10)
QWH	qwh20-166-7	n	88,429	10,945	22,023
		t	206	19	49 (9)
QWH	qwh20-166-8	n	70,945	12,565	29,199
		t	160	23	72 (10)
GC	homer-10	n	-	3,505	2,994
		t	>2h.	3	6 (4)
R	frb35-17-0	n	59,910	10,155	4,320
		t	14	13	20 (10)
R	frb40-19-0	n	170,345	46,596	94,722
		t	45	80	238 (10)
R	frb45-21-0	n	1,028,028	767,550	1,205,280
		t	320	1862	1844 (10)

6 Related Work

Random probing has been used in constraint programming before, albeit in different contexts. Grimes and Wallace have used probing to initialize the scores of the dom/wdeg heuristic and in this way make it more informed at the initial stages of search [12][2]. Ruml proposed an adaptive probing scheme that iteratively adapts the search guiding heuristic in subsequent searches [16]. Beck used probing in the context of multi-point constructive search [2]. Probes are used to initialize a set of "elite" partial solutions some of which are thereafter used as starting points for subsequent searches. Finally, probing has been to measure the *promise* of variable ordering heuristics [3].

There have been several efforts, which are mainly related to the modelling of specific CSPs, on deciding which propagator to apply on certain constraints based on static features of the problem. As most of these works are not general but rather problem-specific, we will not review them in detail. Instead, we will focus on approaches that try to select the propagation method using dynamic features of the problem.

Adaptive constraint propagation has attracted interest in the past. The most common manifestation of adaptive propagation is the use of different propagators for different types of domain reductions in arithmetic constraints. When handling arithmetic constraints most solvers differentiate between events such as removing a value from the middle of a domain, or from a bound of a domain, or reducing a domain to a singleton, and apply suitable propagators accordingly.

[2] Note that we did not do this in our experiments to avoid adding bias to the results.

Works on adaptive propagation for general constraints include the following. El Sakkout et al. proposed a scheme called *adaptive arc propagation* for dynamically deciding whether to process individual constraints using AC or forward checking [10]. Freuder and Wallace proposed a technique, called *selective relaxation* which can be used to restrict AC propagation based on two criteria; the distance in the constraint graph of any variable from the currently instantiated one, and the proportion of values deleted [11]. Chmeiss and Sais presented a backtrack search algorithm, MAC (dist k), that also uses a distance parameter k as a bound to maintain a partial form of AC [8].

Schulte and Stuckey proposed techniques for dynamically selecting which propagator to apply to a given constraint using priorities and staged propagators [17]. Their proposed methods either select a single propagator from a given set or propagators or choose the order in which the propagator stages will be applied [17]. These methods are based on interpreting the event that triggers propagation for a constraint at any point in time, such as the reduction of a domain to a singleton or the removal of a value from a bound of a domain. Similar ideas are also implemented in constraint solvers such as Choco [13].

Probabilistic arc consistency is a scheme that can help avoid some consistency checks and constraint revisions that are unlikely to cause any domain pruning [15]. As in [10], the scheme is based on information gathered by examining the supports of values in constraints which can be very expensive for non-binary constraints. Szymanek and Lecoutre studied ways to select values on which to apply "shaving" (i.e. make the values SAC) using the semantics of global constraints (e.g. alldifferent) to suggest values that are most likely to be removed by shaving [20]. Finally, Stergiou proposed heuristics for dynamically switching between two propagators on individual constraints during search [19]. These heuristics take advantage of the fact that in structured problems propagation events usually occur in clusters, but it is difficult to see how they can be generalized to work with more than two propagators.

As discussed, our work makes a static selection of propagator for individual constraints, but it can be combined with most dynamic approaches as we demonstrated for [19]. Combining with such approaches is an interesting direction for future work. Also, we can extend LPP to a dynamic version where constraint propagation data acquired during search is taken into account to perhaps readjust the initial static propagator choices if necessary.

7 Conclusions

Choosing the right propagator for specific constraints prior to search is a difficult task for CP modelers. We have presented LPP, a simple approach toward automating this task. Our approach is based on gathering data concerning the pruning behavior of the constraints in a random probing preprocessing phase. A case study on binary constraints was presented, and as experimental results demonstrated, decisions taken using the random probing results can be quite accurate in many cases, resulting in improved cpu times during search. In addition, we believe that our work emphasizes the largely untapped potential of

using machine learning techniques, such as clustering, to boost the performance of CP systems.

A drawback of LPP is that the preprocessing phase can be too expensive on very large problems with many variables and constraints. To overcome this we may lift the requirement that all stages of the propagator used are applied at each node and for each constraint. In the future we plan to extend the work presented here to include a wider range of local consistencies for binary as well as non-binary constraints. Also, we would like to investigate the use of machine learning techniques to automatically build the decision tree which exploiting random probing results will be able to propose propagators for the constraints.

References

1. Balafoutis, T., Stergiou, K.: Exploiting constraint weights for revision ordering in Arc Consistency Algorithms. In: ECAI 2008 Workshop on Modeling and Solving Problems with Constraints (2008)
2. Beck, C.: Solution-Guided Multi-Point Constructive Search for Job Shop Scheduling. JAIR 29, 49–77 (2007)
3. Beck, C., Prosser, P., Wallace, R.: Variable Ordering Heuristics Show Promise. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 711–715. Springer, Heidelberg (2004)
4. Bessiere, C.: Constraint propagation. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, ch. 3. Elsevier, Amsterdam (2006)
5. Bezdek, J.C. (ed.): Pattern Recognition with Fuzzy Objective Function Algorithms. Kluwer Academic Publishers, Norwell (1981)
6. Boussemart, F., Hemery, F., Lecoutre, C.: Revision ordering heuristics for the Constraint Satisfaction Problem. In: CP 2004 Workshop on Constraint Propagation and Implementation (2004)
7. Boussemart, F., Heremy, F., Lecoutre, C., Sais, L.: Boosting systematic search by weighting constraints. In: ECAI 2004, pp. 482–486 (2004)
8. Chmeiss, A., Sais, L.: Constraint Satisfaction Problems: Backtrack Search Revisited. In: ICTAI 2004, pp. 252–257 (2004)
9. Debruyne, R., Bessière, C.: From restricted path consistency to max-restricted path consistency. In: Smolka, G. (ed.) CP 1997. LNCS, vol. 1330, pp. 312–326. Springer, Heidelberg (1997)
10. El Sakkout, H., Wallace, M., Richards, B.: An Instance of Adaptive Constraint Propagation. In: Freuder, E.C. (ed.) CP 1996. LNCS, vol. 1118, pp. 164–178. Springer, Heidelberg (1996)
11. Freuder, E., Wallace, R.J.: Selective relaxation for constraint satisfaction problems. In: ICTAI 1996 (1996)
12. Grimes, D., Wallace, R.J.: Sampling Strategies and Variable Selection in Weighted Degree Heuristics. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 831–838. Springer, Heidelberg (2007)
13. Laburthe, F., Ocre: Choco: implementation du noyau d'un systeme de contraintes. In: JNPC 2000, pp. 151–165 (2000)
14. Lecoutre, C., Prosser, P.: Maintaining Singleton Arc Consistency. In: 3rd International Workshop on Constraint Propagation And Implementation (CPAI 2006), pp. 47–61 (2006)

15. Mehta, D., van Dongen, M.R.C.: Probabilistic Consistency Boosts MAC and SAC. In: IJCAI 2007, pp. 143–148 (2007)
16. Ruml, W.: Incomplete Tree Search using Adaptive Probing. In: IJCAI 2001, pp. 235–241 (2001)
17. Schulte, C., Stuckey, P.J.: Speeding Up Constraint Propagation. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 619–633. Springer, Heidelberg (2004)
18. Schulte, C., Stuckey, P.J.: Efficient Constraint Propagation Engines. ACM Trans. Program. Lang. Syst. 31(1), 1–43 (2008)
19. Stergiou, K.: Heuristics for Dynamically Adapting Propagation. In: ECAI 2008, pp. 485–489 (2008)
20. Szymanek, R., Lecoutre, C.: Constraint-Level Advice for Shaving. In: ICLP 2008, pp. 636–650 (2008)

DFS* and the Traveling Tournament Problem

David C. Uthus[1], Patricia J. Riddle[1], and Hans W. Guesgen[2]

[1] University of Auckland, Auckland, New Zealand
[2] Massey University, Palmerston North, New Zealand
{dave,pat}@cs.auckland.ac.nz, h.w.guesgen@massey.ac.nz

Abstract. Our paper presents a new exact method to solve the traveling tournament problem. More precisely, we apply DFS* to this problem and improve its performance by keeping the expensive heuristic estimates in memory to help greatly cut down the computational time needed. We further improve the performance by exploiting a symmetry property found in the traveling tournament problem. Our results show that our approach is one of the top performing approaches for this problem. It is able to find known optimal solutions in a much smaller amount of computational time than past approaches, to find a new optimal solution, and to improve the lower bounds of larger problem instances which do not have known optimal solutions. As a final contribution, we also introduce a new set of problem instances to diversify the available instance sets for the traveling tournament problem.

1 Introduction

The traveling tournament problem (TTP)[3] emerged from the difficulties of scheduling real-world sports leagues. It simplified many of the constraints and requirements found in sports leagues to a problem manageable for theoretical research. Despite this simplification, it has been found to be a difficult problem: solutions have been proven optimal for only the smallest few problem instances. The difficulty lies in its unusual structure (round robin tournaments) and its feasibility constraints.

We present a new approach to the TTP by applying the search algorithm DFS*[9]. We are able to improve its performance for this application by using a new idea of storing expensive heuristic estimates in memory to greatly reduce the running time. In addition, we exploit a symmetry property found in the TTP, in which all schedules are symmetrically equivalent to one other schedule[5]. Our results show that this approach is one of the best performing approaches for the TTP, finding previously known optimal solutions in a fraction of the time of other approaches while being able to find both a new optimal solution and new lower bounds for unsolved problem instances. We also introduce a new set of problem instances for the TTP. This new set is derived from the rugby union league Super 14, which is composed of teams from Australia, New Zealand, and South Africa.

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 279–293, 2009.

2 Traveling Tournament Problem

The TTP is a sports scheduling combinatorial optimization problem. It involves n teams, n being even, and takes in an $n \times n$ matrix of distances between the teams. The problem consists of a double round robin tournament with $2 \cdot (n-1)$ rounds. The teams must play once during each round and are required to play every other team twice, once at home and once away. The objective is to create a tournament such that the total summed travel distance amongst all of the teams is minimized. The travel distance, which is calculated individually for each team, is the total distance a team must travel between locations during the tournament. All teams start at home prior to the first round and end at home after the final round. There is no travel cost for playing consecutive games at home.

The TTP has two feasibility constraints. The first is the *no_repeat* constraint. This forbids a team from playing the same team in consecutive rounds. The second constraint is the *at_most* constraint, which restricts a team to being able to play at most three games consecutively at home or away.

There are four problem sets for the TTP[7]. The NL set and the larger NFL set are derived from Major League Baseball and the National Football League respectively. Both of these sets use real distances between the cities where the teams are located. The other two problem sets use artificial distances. The CIRC set has all the teams placed on a circle, and the distance between two teams is the minimal number of teams a team has to go through to get to the other team. The CON set has all distances set to 1, which changes the problem to minimizing the number of trips[7].

In this paper, we focus our work on the NL and CIRC sets. We do not work with the NFL set since the size of the instances are too large for us to work with while the CON set has been solved for all instances except for 20 teams. So far, only the smallest few problem instances for the NL and CIRC sets have been solved to optimality. These are NL4, NL6, and NL8 for the NL set and CIRC4 and CIRC6 for the CIRC set[7]. For larger instances, only lower and upper bounds of the optimal solutions have been found. Some of the various techniques that have been used to find optimal solutions are integer and constraint programming[4], Lagrangian relaxation and constraint programming[1], independent lower bound estimations[8], and branch-and-price with column generation[5].

3 DFS*

We are taking a new approach to the TTP by applying DFS*. DFS* is an algorithm which combines IDA* with depth-first branch-and-bound (DFB&B) search. It goes through an iterative process like IDA*, starting with a small upper bound and increasing it after every iteration. However, it differs in that it uses DFB&B for the final iteration. This is due to the difference in how it creates the new upper bound thresholds. In IDA*, the new upper bound after every iteration is the minimal lower bound that exceeded the upper bound of the previous iteration. When the first solution is found, it knows that this solution is

optimal and the algorithm is finished. In DFS*, after every iteration, it increases the upper bound by a greater amount than IDA*. This will eventually cause DFS* to create an upper bound that is greater than or equal to the optimal solution. Because of this, when the first solution is found, it is unknown whether it is optimal or not. To prove optimality, the algorithm then works like a standard DFB&B approach, searching the rest of the solution space to prove either the solution is optimal or that there is another solution which is optimal.

We do take note that there are two similar algorithms to DFS* under different names called IDA*_CR[6] and MIDA*[10]. The original papers describing these three approaches were published at relatively the same time. Any of these works would have sufficed, as they shared the same underlying idea of using IDA* with DFB&B and only slightly differed on how to increase the upper bound thresholds.

One of the advantages of using DFS* over other depth-first search approaches like DFB&B is that it can find lower bounds of an unknown optimal solution for problems where it would take too long to find the optimal solution. For every iteration of DFS* in which it increases the upper bound, if it cannot find any solutions for that upper bound, then the upper bound is the lower bound of the optimal solution for these problem instances. This is important for the TTP since very few optimal solutions have been found so far. Much work has focused on decreasing the difference between the best known upper and lower bounds of the optimal solutions for the problem instances.

A disadvantage of using DFS* is that it may not return a feasible solution within a reasonable amount of time if it spends too much time on the iterations prior to the final iteration. If a feasible solution is needed, then it is better to use DFB&B by itself. Many of the improvements described in this paper can easily be applied to DFB&B to improve its performance on the TTP.

The following sections describe our application of DFS* to the TTP. We begin with how we perform the depth-first search, then continue on with heuristics and lower bounds, memory, symmetrical schedules, subtrees, upper bound thresholds, and conclude with the parallelization of DFS* for the TTP.

3.1 Depth-First Search

DFS* uses backtracking search with constraint propagation[2] to perform the depth-first search when applied to the TTP. It constructs the solutions from round 1 to $2 \cdot (n-1)$. Within each round, it pairs up all the teams prior to scheduling teams for the subsequent round. As the solution is being built, it propagates constraints after each pairing using forward checking. This involves making sure that teams play only once per round; that teams only play once at home and once away against every other team; and that teams do not violate the *no_repeat* and *at_most* constraints during the following round.

Algorithm 1 shows the specifics of the depth-first process. It first sets the index, i, to 1 and then begins to search through the solution space. At each index, it calculates the current round, $r \leftarrow \lceil \frac{2 \cdot i}{n} \rceil$, and chooses $team1$ and $team2$. For $team1$, it picks the first, unpaired team in numerical order for round r. It

then goes through $team1$'s domain that is associated with round r, $D_{(team1_i,r)}$, taking the first, untried team and designating it as $team2$. We note that with the domains described here, they list all of the feasible teams a team can play at home and then all of the feasible teams it can play away. Thus DFS* will always try pairing $team1$ with teams it can play at home before teams it can play away.

Algorithm 1. Constructing solutions

```
1: procedure ConstructSolution
2:     i ← 1
3:     while 0 < i do
4:         r ← ⌈2·i/n⌉
5:         team1_i ← ChooseTeam1(r)
6:         team2_i ← ChooseTeam2(team1_i, D_{team1_i,r})
7:         if LB < UB ∧ PropagateConstraints = valid then
8:             i ← i + 1
9:             if i = n · (n − 1) then
10:                UpdateBestSolution
11:                i ← i − 1
12:                UndoPairing(team1_i, team2_i, i)
13:        else
14:            UndoPairing(team1_i, team2_i, i)
15:            if D_{team1_i,r} = ∅ then
16:                i ← i − 1
17:                UndoPairing(team1_i, team2_i, i)
```

If pairing $team1$ with $team2$ does not cause the lower bound estimate, LB, to be greater than the upper bound, UB, nor does propagating constraints cause any domains to become empty, DFS* will then attempt to pair up another set of teams. Otherwise, it continues to try the rest of the available teams in $team1$'s domain. If it cannot find any teams that can be paired with $team1$, it backtracks to the previous pairing and undoes it. When it undoes a pairing, it also undoes any constraint propagations that it had done when pairing the two teams.

DFS* will have found a solution when i is equal to $n{\cdot}(n{-}1)$, the largest possible index for this problem. Once this is true, it updates the best solution, backtracks, and then continues to search the rest of the solution space. As described earlier, DFS* acts like a DFB&B approach after it has found a solution, which is why it does not stop searching after finding a solution.

The reason we perform the depth-first search and construct solutions in this manner is two-fold. The first is that it is easy to verify that constraints are not being violated and to do constraint propagation for the following round. Second, we can easily calculate the total distance seen so far for a partial solution, which is necessary for estimating the lower bounds.

3.2 Heuristics and Lower Bounds

The independent lower bound[3] is used as the admissible heuristic to calculate an estimated cost for the lower bound. For each individual team, it estimates the optimal tour that can be obtained based on what it has constructed so far. These estimates are independent of the other teams' schedules.

The heuristic estimates are found by using a general DFB&B approach for a single team. All characteristics, such as the number of consecutive away games already played, the number of away games remaining, the last team played, and the remaining teams it still needs to play away at, are taken into consideration along with the *at_most* constraint. What is not taken into consideration is the team's domain in the future rounds nor the domains of other teams.

The heuristic estimates also do not take into consideration the number of remaining home games, but instead try to find the best possible schedule with as many home games as needed. The reason for this is to reduce the size of the memory footprint, as described later in Section 3.3. But in order to reduce the running time of DFB&B, the maximum number of home games is set to half the number of remaining away games, rounded up, for each heuristic estimate. The heuristic estimate will still be admissible since there cannot be a shorter trip requiring more home games due to the following theorem.

Lemma 1. *Given a team, t, and two teams it has to play away, a_1 and a_2, the shortest tour visits a_1 and a_2 consecutively prior to returning home if the distances satisfy the triangle inequality.*

Proof. Let d_1 be the distance between t and a_1, let d_2 be the distance between t and a_2, let d_3 be the distance between a_1 and a_2, and let d_4 be the distance for returning home between visits to a_1 and a_2. Form the triangle $\triangle a_1 a_2 t$. If all angles of the triangle are less than $180\,^\circ$, then by definition of the triangle inequality, $d_3 < d_1 + d_2$. If one of the angles of the triangle is $180\,^\circ$, then t, a_1 and a_2 form a straight line and $d_3 = d_1 + d_2$. In either case, returning home between visits to a_1 and a_2 results in $d_4 = d_1 + d_2$. Taking into account both cases, $d_3 \leq d_4$. Therefore, the shortest tour visits a_1 and a_2 consecutively prior to returning home. □

Theorem 1. *Assume 'a' is the remaining number of away games a team has to play. Then the shortest tour uses no more than $h = \lceil \frac{a}{2} \rceil$ home games if the distances satisfy the triangle inequality.*

Proof. Let $h = \lceil \frac{a}{2} \rceil$. When a is even, partition the set of remaining away teams into subsets of two. Create a tour that returns home after visiting every subset of two, which is possible since there are h home games to use. When a is odd, partition the set of remaining away teams into $\lfloor \frac{a}{2} \rfloor$ subsets of two and one subset of the remaining team, which is possible since there are h home games to use. Create a tour that returns home after visiting every subset of two and then visit the final, single team. In both cases, when a is even or odd, using an additional home game within a subset of two will not result in a shorter tour because of Lemma 1. Thus if a is the remaining number of away games a team has to play, then the shortest tour uses no more than $h = \lceil \frac{a}{2} \rceil$ home games. □

To calculate the actual lower bound, the distances traveled so far are summed up with each team's estimates. Since each team's estimate is admissible, the sum of their estimated distances will also be admissible, thus it does not violate the requirements of an admissible lower bound for DFB&B.

3.3 Memory

The heuristic estimates used for this problem are expensive to calculate. To mitigate this, they are kept in memory. This is possible since DFS*, like other depth-first algorithms, uses a minimal amount of memory. By keeping the heuristic estimates in memory, they only need to be calculated once each and can then be reused. The calculations are all done at the start of the algorithm. Calculating all of the heuristic estimates at the start of the algorithm instead of the first time each is needed makes it easier to parallelize DFS*, as described later in Section 3.7. If DFS* were not being parallelized, then it would be possible to calculate and store the heuristic estimates when they are first seen. The advantage of this would be that unused estimations would not be calculated.

The heuristic estimates are stored with a minimal amount of memory in a multi-dimensional matrix. All that is stored at each index is the estimated distance. The index is composed of five aspects of the heuristic estimate: which team the estimate belongs to, the number of away games, which teams left to play away, the previous number of consecutive away games, and the last team played if away. As stated earlier, the heuristic estimates are calculated assuming as many home games as needed, thus the number of home games remaining is not used. This reduces the number of dimensions in the matrix by one.

The number of heuristic estimates needed is as follows. For each team's set of estimates, there are two cases: when the previous game was at home or away. When the previous game played was at home, there are $(n-1)$ possible number of remaining away games. Related to this is the number of combinations of teams it can play away. With the combinations, order and duplicates are ignored. This results in:

$$\sum_{i=1}^{n-1} \sum_{j=1}^{i} \binom{n-1}{j} \tag{1}$$

When the last game was not played at home, there are $(n-1)$ possible teams it could have last played. There are three possibilities of the length of the number of previous consecutive away games: [1-3]. Related to this is the number of remaining away games and the combinations. These last two values will be smaller due to the number of previous away games being greater than 0. This results in:

$$(n-1) \cdot \sum_{k=1}^{3} \sum_{i=1}^{n-1-k} \sum_{j=1}^{i} \binom{n-2}{j} \tag{2}$$

Together, those two values are the number of estimates needed for one team. As each team needs its own set of estimates, the number of heuristic estimates is:

$$n \cdot \left(\sum_{i=1}^{n-1} \sum_{j=1}^{i} \binom{n-1}{j} + (n-1) \cdot \sum_{k=1}^{3} \sum_{i=1}^{n-1-k} \sum_{j=1}^{i} \binom{n-2}{j} \right) \tag{3}$$

or, in more general terms, $\mathcal{O}(n^3 n!)$.

Each checkup into the matrix requires the index to be calculated, which is $\mathcal{O}(n)$. The $\mathcal{O}(n)$ is from having to go through a team's remaining away games and calculating the index based on what teams it has and has not played using binomial coefficients. This allows a mapping of combinations to indices. To minimize this, caching is used in a similar manner to how modern machines cache data. To do so, a two-level approach for the caching is used. At the upper level is the matrix with all the estimates. At the lower level, the heuristic estimates are cached for the next round for a team, since the values will not change until the team's previous round pairing has been changed.

3.4 Symmetrical Schedules

In the TTP, half of all the solutions can be discarded due to symmetry [5]. Symmetry in the TTP means that all schedules can be flipped across their rounds and still have the same distance. Thus, half of the schedules in the solution space are symmetrical images of the other half. An example of this can be seen in Figure 1.

The symmetry described here is the same general idea as described in the work by Irnich and Schrempp [5]. What differs is only how the idea is implemented. We have adapted their idea for a depth-first search approach instead of a column generation approach. Our approach also differs in that they customized the implementation for each of the NL and CIRC sets, while our general approach can be applied to any TTP instance.

The reason there is symmetry is because each team's schedule is treated independently for calculating distances, and the total distance is the sum of each individual schedule. All distances are symmetrical, resulting in a symmetrical traveling salesman problem for each team. In the symmetrical traveling salesman problem, a circular path can be traversed either way and will result in the same distance. Since a team's schedule starts and ends at home, it can be viewed as a circular path. Thus, each individual path can be reversed and made symmetrical. Two complete schedules are considered symmetric when each can be obtained by reversing the individual team schedules of the other.

To break this symmetry, DFS* begins with Team 1 as a "pivot". Team 1 is used since it is the first team to be paired for each new round. Once half

Team	1	2	3	4	5	6
1	3	2	4	@3	@2	@4
2	4	@1	@3	@4	1	3
3	@1	4	2	1	@4	@2
4	@2	@3	@1	2	3	1

Team	1	2	3	4	5	6
1	@4	@2	@3	4	2	3
2	3	1	@4	@3	@1	4
3	@2	@4	1	2	4	@1
4	1	3	2	@1	@3	@2

Fig. 1. Symmetric schedules with equivalent total distances. Both are optimal for NL4.

of a solution is constructed, meaning all games up to round $n-1$ have been scheduled, it checks that either the number of remaining home or away games is greater than the other. If the check fails, DFS* backtracks.

This check will discard half of the schedules for the following reason. When it has come to the midpoint, the algorithm will have seen $(n-1)$ rounds so far. This will be an odd number since n is always even. With $(n-1)$ being odd, then either the number of remaining home games or away games will be greater than the other. By restricting to one of the two symmetrical checks, half of the solutions are removed from the solution space.

When checking that the number of remaining away games is greater than the number of remaining home games, this is called symmetry-A, while checking that the number of remaining home games is greater is called symmetry-H. How this is done can influence the performance of DFS*, as will be later seen in Section 4.1. It is important to note that only one of these checks can be used during the running of DFS*, not both. If both were applied, then the algorithm would never be able to construct a complete solution.

In relation to propagating constraints, this check can be done before the midpoint of solution construction. In the earlier rounds, after each time Team 1 is paired with another team, DFS* will ensure that the symmetry check for Team 1 will not be violated once it gets to the solution construction midpoint.

3.5 Subtrees

Instead of performing one depth first search through the solution tree during each iteration of DFS*, we use a concept we call *subtrees*, in which DFS* performs multiple depth first searches from different starting points during each iteration. A subtree has its initial four pairings already fixed, or the initial two or three pairings in the cases of problem sets with four or six teams respectively.

To form the list of subtrees, DFS* creates all permutations of the initial, feasible pairings of teams in the same manner as the depth first search process. The first pairing will always involve Team 1, the second pairing will always involve either Team 2 or 3, mattering which team is paired with Team 1, and so forth. A subtree will have associated with it one of these permutations. This list of subtrees is created at the beginning and is reused for every iteration of DFS*.

The maximal number of fixed pairings is limited to four due to the branching factor of the search tree. For the first pairing, after choosing the first team, there is a possible $2 \cdot (n-1)$ teams it can be paired with in terms of both playing at home or away. The second pairing will have $2 \cdot (n-3)$, the third pairing will have $2 \cdot (n-5)$, and the final pairing $2 \cdot (n-7)$. Limiting the number of pairings to four helps keep the list of subtrees from growing too big for larger problem sets.

There are three reasons for using subtrees. The first reason is for calculating upper bound thresholds. When working with DFS*, additional information is stored with each subtree. This consists of a subtree's maximal depth that it constructed a solution to in the last iteration of DFS* and the minimal lower bound for that depth. This will then be used for calculating new upper bounds, which is explained later in Section 3.6.

The second reason deals with the order upon which subtrees are worked. Before the start of every iteration after the first, the subtrees are sorted by two criteria. They are first sorted in descending order according to the deepest depth they were able to construct a solution. Within each depth level, they are then sorted in ascending order by the minimal lower bounds. The reason they are sorted by this criteria is that the subtrees which will be seen first are most likely to be able to complete a solution once the upper bound is past the optimal solution. Secondly, this complete solution will hopefully be either the optimal or near-optimal solution, reducing the size of the search tree.

The third reason is that they are used in the parallelization of DFS*. In this case, they are used to help distribute the work amongst the various processors. This concept will be further explained later in Section 3.7.

3.6 Upper Bound Thresholds

Our approach differs from the original DFS* algorithm for calculating upper bound thresholds. The calculations of our new upper bound thresholds are tailored for this problem. While our approach differs, we have kept true to the spirit of the original papers, making sure the upper bound thresholds grow at a fast-enough pace to reduce the number of iterations needed.

DFS* begins with an upper bound cost of 0. After every iteration, a new upper bound is created based on the information gained from the subtrees. The new upper bound, UB, is the sum of two values, LB_m and D. The first value, LB_m, is the minimal lower bound of all subtrees that constructed solutions to the furthest depth. The second value, D, is a value which combines the deepest depth used and the average distance in the distance matrix of the problem.

To find the first part of the new upper bound, LB_m, DFS* goes through the sorted list of subtrees and finds the first subtree with a minimal lower bound that is greater than the previous iteration's upper bound. It then sets LB_m to the subtree's minimal lower bound value. In all cases that we have seen, the algorithm ends with a new LB_m that is greater than the previous upper bound. This is due to the large number of subtrees. The only possible way that none of the subtrees would have a distance value greater than the previous upper bound would be if all of their deepest depths were caused by constraint violations and not because they had lower bound violations. If this were to happen, LB_m would be set to the previous upper bound value. The new upper bound is still increased by adding the value D, but grows slowly in this rare case.

The reason we use only the first minimal lower bound instead of finding a LB_m value of all the subtrees is that this allows the algorithm to base the upper bound on promising subtrees which are closer to building a complete solution. These promising subtrees have a greater chance of containing the optimal solution than a subtree which cannot construct a solution past a much lower depth but having a LB_m value smaller than the promising subtrees.

After calculating LB_m, DFS* takes another step to help increase the rate of the upper bound's growth and reduce the number of iterations DFS* needs. For each new upper bound created, it adds to it the value D, which is defined as:

$$D = \left\lceil avg \cdot \frac{(n \cdot (n-1)) - i_m}{n \cdot (n-1)} \right\rceil \tag{4}$$

with avg being the average of the non-zero distances in the problem instance's distance matrix, n the number of teams in the problem, and i_m the depth of LB_m. As stated earlier, $n \cdot (n-1)$ is the total number of pairings in the problem, thus the largest possible depth. What this formula does is that the smaller of the value of i_m, the greater of the value added to the new upper bound. In the beginning, it increases the upper bound by a large amount. As the algorithm is able to build the search tree to a deeper depth, it decreases the amount being added to the upper bound. This makes sure the new upper bound does not overshoot the optimal solution by too much.

3.7 Parallelism

In order to further improve the performance of DFS*, we looked into parallelization [11]. We have designed a distributed approach to work across multiple processors with shared memory.

To distribute the work, the processors access the list of subtrees. Every processor will each have one subtree to work with at a time. It will expand the tree from that subtree, taking into account the best upper bound found so far or the maximum upper bound allowed. After a processor has expanded a subtree as far as it can, it will then place that subtree with the appropriate information into a new list of subtrees. These will be processed and sorted after the current iteration is over if no solution is found.

One of the advantages of our approach is that there are very few critical areas where the multiple processors share variables. The three key areas are the list of subtrees, the upper bound, and the heuristic estimate values. For the subtrees, locks are used when reading from the original list and writing to the second list which holds the finished subtrees. The upper bounds changes values within an iteration only during the final iteration, and this also requires a lock for race conditions. With the heuristic estimate values, these are calculated prior to running DFS*, thus there are no race conditions. The work for calculating the heuristic estimates is distributed across the processors, allowing the algorithm to calculate them as efficiently as possible.

4 Experiments

We ran two sets of experiments. The first set was to look at the performance of our approach in terms of the different improvements we have discussed in this paper. The second set was comparing our approach with other approaches and seeing how well it can find either new optimal solutions or new lower bounds to

the optimal solutions of different problem instances. For all of our experiments, they were run on a virtual machine utilizing between 1 to 4 processors running at 2.4Ghz with 3GB of shared memory. All code was written in C++.

4.1 Performance

Table 1 shows the results of looking at the different components. These tests were done sequentially on a single processor. All numbers represent the amount of time, in seconds, to finish. For comparison purposes, we worked with both DFS* and a standard DFB&B approach. In the cases of NL4 and CIRC4, we recorded a time of 0.0 seconds due to the algorithms being able to finish faster than the smallest resolution of time.

As we see in Table 1, using DFB&B and DFS* by themselves results in poor performance. More interestingly, DFB&B performs better than DFS* for the CIRC set. This may be due to the fact that there is little disparity between the distances for the CIRC instances, thus it does not benefit from using DFS* for the smaller problem sets.

Keeping heuristic estimates in memory had the largest impact in minimizing the running time of the approaches. It was able to reduce the time needed by over 2000% when applying DFS* to NL8 and over 1000% when applied to CIRC8. The other components tested, symmetry-A and symmetry-H, further reduced the running times. As can be seen, symmetry-A worked better than symmetry-H in 3 of the 4 instances. The reason symmetry-A works better in most cases is because of the way the depth first process picks values from the domains: it chooses home games before away games. It is important to note that symmetry breaking will not always cut the running time in half. The reason for this is that many of the solutions do not go past the midpoint of solution construction because of the strong lower bound heuristic estimations.

The size of the memory used when keeping heuristic estimates in memory was small for the tested problem instances. When looking at the larger problem sizes, the memory remained small up until 18 teams. Testing DFS* on 18 teams with the heuristic estimates in memory took up roughly 16.2% of the 3GB of RAM. For larger problem instances and problems other than the TTP, when the number of heuristic values is too great to store all in memory, then it may

Table 1. Comparison of the components used. "M" stands for keeping heuristic estimates in memory and "SA" and "SH" stands for using symmetry-A and symmetry-H respectively.

Instance	DFB&B	DFS*	DFB&B+M	DFS*+M	DFS*+M+SA	DFS*+M+SH
NL4	0.0	0.0	0.0	0.0	0.0	0.0
NL6	25.61	16.58	4.03	2.03	0.98	1.53
NL8	315 443.73	94 881.22	2 321.02	426.16	262.42	392.27
CIRC4	0.0	0.0	0.0	0.0	0.0	0.0
CIRC6	7.76	15.33	1.50	2.94	2.05	1.77
CIRC8	71 598.93	122 259.7	839.72	1 207.85	984.69	1 010.08

be beneficial to keep either the hardest to calculate estimates in memory or the most frequently used estimates in memory. In either case, the approach would result in better performance than not keeping any heuristic estimates in memory.

The amount of time needed to calculate all the heuristic estimates was minimal for most of the problem sets we worked with. When running in parallel across four processors, the wall time needed for problem sets of eight teams or less was under a second. The time increased quickly though for larger problem sets. When running DFS* on the NL set, NL10 took 6 seconds, NL12 took 1 317 seconds, and NL14 took 159 876 seconds. With the CIRC set, CIRC10 took 3 seconds, CIRC12 took 327 seconds, and CIRC14 took 40 714 seconds. There are two reasons for this sharp increase in time. The first is the larger number of estimates needed. The second is that we used a general DFB&B approach, which is not as efficient for calculating larger estimates. This was not an issue for the problem sets we mostly worked with, but if one wants to work with larger problem sets, then it is necessary to use more advanced techniques to calculate the heuristic estimates.

The number of iterations that DFS* went through were few: NL4 took 3 iterations, CIRC4 took 4 iterations, NL6 and CIRC6 took 5 iterations each, and NL8 and CIRC 8 took 6 iterations each. Generally the first few iterations required less than a second each, building up the upper bound towards a value near the optimal value. Then DFS* would take one or two additional longer iterations before the upper bound threshold had surpassed the optimal solution, allowing for the final iteration. While we have not solved NL10 or CIRC10, both problem sets have so far showed similar patterns for the iterations.

Table 2 looks at the parallelization of DFS*. As can be seen, parallelizing reduced the running time of DFS*. While it was not 100% efficient due to the overhead of parallelization, it was able to distribute the workload so processors were not being underused. We think what may be limiting the efficiency in these situations is the locks for the subtrees, as the processors can process a subtree very quickly for these smaller problem instances. We would expect our approach to reach higher efficiency with larger problem instances, as the processors would then be spending more time working on an individual subtree.

Table 2. Comparison of number of CPUs used. Times listed are wall times in seconds.

Instance	1	2	3	4
NL4	0.0	0.0	0.0	0.0
NL6	0.98	0.43	0.40	0.38
NL8	262.42	143.14	106.49	104.67
CIRC4	0.0	0.0	0.0	0.0
CIRC6	2.05	1.09	0.94	0.87
CIRC8	984.69	516.42	430.89	337.09

4.2 Comparison With Other Approaches

We compared the timings of our results with the current best approach at the time of the writing of this paper: Irnich and Schrempp's approach using a

Table 3. Comparison with Irnich and Schrempp's approach to the TTP

Instance	Irnich and Schrempp	Us
NL4	<0.3 secs	0.0 secs
NL6	<19 mins	0.98 secs
NL8	<18 hrs	262.42 secs
CIRC4	<0.2 secs	0.0 secs
CIRC6	<18 hrs	2.05 secs

branch-and-price algorithm with column generation[5]. We only compared the times on fully constrained problem instances that they were able to prove optimality. We did not run experiments on problem instances where some of the constraints were relaxed, making the problems easier. Their approach used a single processor of unknown speed, so we only list our timings which also used a single processor.

Table 3 shows the comparison between the two approaches. We list their timings in the same notation that they used. As can be seen by the results, our approach outperformed their approach by a large margin.

4.3 Final Results

Our final tests were applying DFS* to larger NL and CIRC instances. We also introduce a new problem set, the SUPER set, along with showing the results of applying DFS* to this new problem set.

NL and CIRC. We looked at applying our approach to a few of the larger NL and CIRC instances listed on the TTP website[7]. Table 4 displays the results of our approach. For these results, we used DFS* with heuristic estimates in memory, symmetry-A, and running across 4 processors. All times listed are total wall times for the approaches in seconds. We stopped each experiment when it became apparent that the time needed to finish the current iteration of DFS* was too long.

As we can see, our approach proved the previously known upper bound of CIRC8 is optimal while being able to find new lower bounds for three larger instances. Running DFS* on CIRC10 and NL10 took a long period of time, yet it was able to significantly reduce the gap between the previously best known lower bound and the current best known upper bound of the optimal solution.

Table 4. Final results. LB and UB are previously found lower and upper bounds of the optimal solution. Italicized lower bounds are improved lower bounds. Bold faced lower bounds are optimal solutions.

Instance	LB	UB	New LB	Decrease In Gap %	Time
NL10	57817	59436	*58831*	62.6	1 066 593
NL12	107548	110729	*108244*	21.9	102 806
CIRC8	130	132	**132**	100	337.09
CIRC10	228	242	*237*	64.3	2 723 466

Table 5. Results from the SUPER instances

Instance	LB	UB	Time
SUPER4	63405	63405	0.0
SUPER6	130365	130365	0.27
SUPER8	182409	182409	361.20
SUPER10	316329	316329	710 236
SUPER12	367812	-	637
SUPER14	467839	-	98 182

Super 14. Our final tests were done on a new set of instances we have created for the TTP. These instances are derived from the Super 14 rugby league. The teams in this league are from three countries: Australia with four teams, New Zealand with five teams and South Africa with five teams. This geography differs greatly from other problem sets, where the teams are generally evenly distributed. With the Super 14, we essentially have three clusters of teams, with two of the clusters close together and one very far from the other two.

To create these instances, we took the same approach as was used in creating the NL instances[3]. We found the distances between all 14 cities, and then took subsets to create SUPER4, SUPER6, SUPER8, SUPER10, SUPER12, and SUPER14. The number following "SUPER" refers to the number of teams in the instance. We made sure that the teams in each set are evenly distributed amongst the three countries.

We ran DFS* on all of the SUPER instances, using the same configuration as described for the NL and CIRC final experiments. As can be seen in Table 5, the SUPER set has shown varied difficulty. SUPER6 and SUPER10 both proved to be easier to solve than their NL and CIRC counterparts, but SUPER8 took longer. This may be a result of the distribution of teams amongst the three countries for each problem instance.

5 Conclusions and Discussion

In this paper, we have presented a new DFS* approach to the traveling tournament problem. It utilized memory to store heuristic estimates that are expensive to calculate. As we have shown, our approach is currently the best approach for this problem to date. We have been able to find previously-known optimal solutions in a far shorter amount of time than past approaches; we have found new optimal solutions to unsolved problem instances; and lastly, we have improved the lower bounds of optimal solutions for larger, unsolved problem instances.

One avenue of research to look at in the future is the constraint propagation used. As described in this paper, we only use forward checking for the constraint propagation. It is possible to apply more advanced techniques, though they usually come at a cost of additional overhead. We believe using these advanced techniques would slow down the depth first search for the smaller problem sets, but they may be beneficial for larger problem sets.

A further way of improving the performance is looking at the possibility of running our approach across multiple computers that may not have shared memory. Various trade-offs will have to come into consideration, especially how to handle the heuristic estimates. But by going this route, it could be possible to solve the much larger instances in a reasonable amount of wall time.

References

1. Benoist, T., Laburthe, F., Rottembourg, B.: Lagrange relaxation and constraint programming collaborative schemes for travelling tournament problems. In: Proceedings of CP-AI-OR 2001, Wye College, UK, pp. 15–26 (2001)
2. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers, San Francisco (2003)
3. Easton, K., Nemhauser, G., Trick, M.: The traveling tournament problem description and benchmarks. In: Walsh, T. (ed.) CP 2001. LNCS, vol. 2239, pp. 580–584. Springer, Heidelberg (2001)
4. Easton, K., Nemhauser, G., Trick, M.: Solving the travelling tournament problem: A combined integer programming and constraint programming approach. In: Burke, E.K., De Causmaecker, P. (eds.) PATAT 2002. LNCS, vol. 2740, pp. 100–109. Springer, Heidelberg (2003)
5. Irnich, S., Schrempp, U.: A new branch-and-price algorithm for the traveling tournament problem. Presented at Column Generation 2008, Aussois, France (June 17-20, 2008), http://www.gerad.ca/colloques/ColumnGeneration2008/slides/SIrnich.pdf (accessed March 07, 2009)
6. Sarkar, U.K., Chakrabarti, P.P., Ghose, S., De Sarkar, S.C.: Reducing reexpansions in iterative-deepening search by controlling cutoff bounds. Artificial Intelligence 50, 207–221 (1991)
7. Trick, M.: Challenge Traveling Tournament Problems, http://mat.gsia.cmu.edu/TOURN/ (accessed March 07, 2009)
8. Urrutia, S., Ribeiro, C.C., Melo, R.A.: A new lower bound to the traveling tournament problem. In: IEEE Symposium on Computational Intelligence in Scheduling, pp. 15–18 (2007)
9. Vempaty, N.R., Kumar, V., Korf, R.E.: Depth-First vs Best-First Search. In: Proc. National Conf. on Artificial Intelligence, AAAI 1991, Anaheim, CA, pp. 434–440 (1991)
10. Wah, B.W.: MIDA*: An IDA* search with dynamic control. Technical report, Coordinated Science Laboratoy, University of Illinois, Urbana, Illinois (1991)
11. Talbi, E.-G.: Parallel Combinatorial Optimization. John Wiley & Sons, Hoboken (2006)

Max Energy Filtering Algorithm for Discrete Cumulative Resources

Petr Vilím

ILOG, an IBM Company; 9, rue de Verdun, BP 85
F-94253 Gentilly Cedex, France
petr_vilim@cz.ibm.com

Abstract. In scheduling using constraint programming we usually reason only about possible start times and end times of activities and remove those which are recognized as unfeasible. However often in practice there are more variables in play: variable durations of activities and variable resource capacity requirements. This paper presents a new algorithm for filtering maximum durations and maximum capacity requirements for discrete cumulative resources. It is also able to handle optional interval variables introduced in IBM ILOG CP Optimizer 2.0. Time complexity of the algorithm is $\mathcal{O}(n \log n)$. The algorithm is based on never published algorithm by Wim Nuijten and a on slightly modified e-feasibility checking algorithm by Armin Wolf and Gunnar Schrader. The later algorithm is also described in the paper.

Keywords: Constraint Programming, Scheduling, Discrete Cumulative Resource, Propagation.

1 Introduction

Nowadays, constraint based scheduling engines like IBM ILOG CP Optimizer [1] allows to describe and solve very complex scheduling problems involving a variety of different constraints. This paper is focused on one of them – discrete cumulative resource for the case when durations and/or individual capacity requirements are not fixed. Traditionally we reason only about minimum start times and maximum end times using algorithms like Edge Finding [2], Not-First/Not-Last [5] or Energetic Reasoning [3]. This paper provides an algorithm for filtering of maximum activity durations and maximum capacity requirements.

The algorithm is not completely new. Although it was never published, Wim Nuijten implemented a similar algorithm for ILOG Scheduler several years ago. The old version of the algorithm has time complexity $\mathcal{O}(n^2)$, this paper presents a faster version with time complexity $\mathcal{O}(n \log n)$.

To demonstrate the problem on a simple example, lets consider the following subproblem: there is a pool of 10 workers (i.e., a discrete capacity resource with maximum capacity $C = 10$) who perform different tasks. Among these tasks there is a task to produce one particular product P. How many units of the product P is produced depends on how many workers are assigned to the tasks

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 294–308, 2009.

(i.e., how much capacity of the resource is used) and for how long (i.e., what is the duration of the task):

$$\text{nbP} = \text{workers} \times \text{duration}$$

If we do not produce at least 500 units of product P then we will have to buy the rest for the following cost (deduced from an initial budget):

$$\text{cost} = \max(0, 500 - \text{nbP}) \times 1\$$$

In this example, the budget and production of product P are tightly connected:

1. If we see that no more than 200$ can be invested into the purchase of product P (because the rest of the budget is needed for other things) then we need to allocate workers to produce at least 300 units of product P.
2. On the other hand if we see that there is no way to produce more than 100 units of product P (because the workers are needed for other tasks) we can immediately allocate 400$ from the budget to buy remaining products P. This is a critical propagation especially if the budget is short.

Both propagations above are very important for speeding up the search by better pruning the search tree. However for the propagation 2 it is necessary to be able to compute the maximum possible production of product P. And this is the topic of the paper.

The algorithm presented in the paper is also useful if there are optional activities – activities which may or may not be present in the solution (for example alternatives between several resources). In this case the algorithm can detect that there is no way to process an optional activity and therefore it cannot be present in the solution (and, in case of an alternative, another alternative must be chosen), see [4, 1].

2 Notation

Let us formalize the problem. There is a set T of $n = |T|$ non-preemptive non-optional activities. For the first part of the paper we assume that none of the activities in T is optional, that is, all activities in T are necessarily present in the solution. After we present Max Energy algorithm for non-optional activities we will show how to use it for optional activities.

Each activity $i \in T$ is described by the following attributes:

- the earliest possible starting time $\text{est}_i \in \mathbb{N}$,
- the latest possible completion time $\text{lct}_i \in \mathbb{N}$,
- the minimum processing time (duration) $\text{p}_i \in \mathbb{N}$,
- the maximum processing time (duration) $\overline{\text{p}}_i \in \mathbb{N}$.

Moreover, each activity $i \in T$ consumes during its processing some capacity of a resource. The capacity consumption during the whole processing of the activity is constant, however it may not be known in advance. In this case there is a range of possible capacity consumption:

– the minimum required capacity $\mathrm{c}_i \in \mathbb{N}$,
– the maximum required capacity $\overline{\mathrm{c}}_i \in \mathbb{N}$.

The resource can process several activities at the same time, however at any time the total used capacity cannot exceed the maximum resource capacity C. For an example see Figure 1.

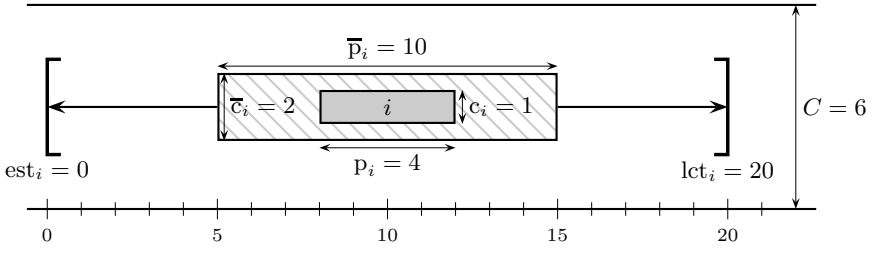


Fig. 1. An example of an activity i on a resource with capacity $C = 6$

Another way to characterize an activity is its energy. Informally, energy of an activity is:

$$\text{energy} = \text{processingTime} \times \text{capacity}$$

Because processing time and/or required capacity may be unbound, we characterize the energy of a task i by two numbers: minimum energy $\mathrm{e}_i = \mathrm{c}_i\,\mathrm{p}_i$ and maximum energy $\overline{\mathrm{e}}_i = \overline{\mathrm{c}}_i\overline{\mathrm{p}}_i$. The presented algorithm modifies maximum energy $\overline{\mathrm{e}}_i$ and this way also maximum capacity $\overline{\mathrm{c}}_i$ and maximum processing time $\overline{\mathrm{p}}_i$:

$$\overline{\mathrm{c}}_i := \min\left\{\lfloor \overline{\mathrm{e}}_i / \mathrm{p}_i \rfloor\,,\ \overline{\mathrm{c}}_i\right\} \tag{1}$$

$$\overline{\mathrm{p}}_i := \min\left\{\lfloor \overline{\mathrm{e}}_i / \mathrm{c}_i \rfloor\,,\ \overline{\mathrm{p}}_i\right\} \tag{2}$$

2.1 Earliest Completion Time, Energy Envelope

For the following algorithms we need a way to quickly estimate the earliest completion time of any set of activities $\Theta \subseteq T$. If Θ contains only one activity i then the computation of the earliest completion time is simple:

$$\mathrm{ect}_i = \mathrm{est}_i + \mathrm{p}_i$$

However in the general case it is much more complicated. Therefore we are looking for a good lower bound, traditionally defined as:

$$\mathrm{preEct}\,(\Theta) = \mathrm{est}_\Theta + \left\lceil \frac{\mathrm{e}_\Theta}{C} \right\rceil$$

where:

$$\mathrm{est}_\Theta = \min_{i \in \Theta}\{\mathrm{est}_i\}$$

$$\mathrm{e}_\Theta = \sum_{i \in \Theta} \mathrm{e}_i$$

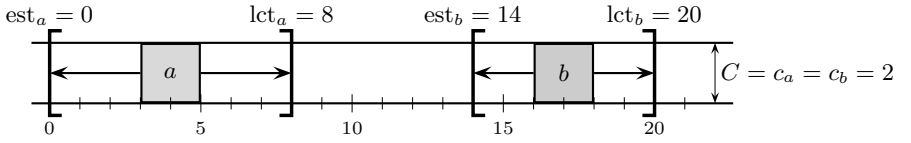


Fig. 2. An example: preEct $(\{a, b\}) = 4$ even though $\{a, b\}$ cannot end before $\text{ect}_b = 16$

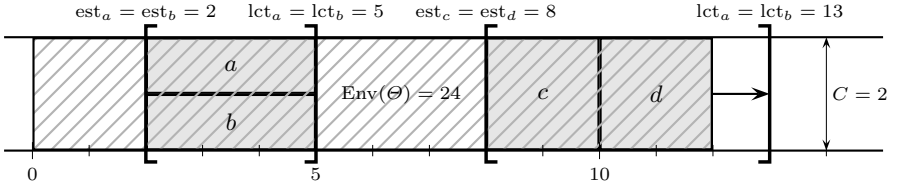


Fig. 3. Example of a set $\Theta = \{a, b, c, d\}$ with earliest completion time $\text{Ect}(\Theta) = 12$ and energy envelope $\text{Env}(\Theta) = 24$. Maximum envelope is achieved by the set $\Omega = \{c, d\}$. Energy envelope is depicted by gray lines.

Note that preEct is only a lower bound. For example:

(i) If we closely inspect all subsets $\Omega \subseteq \Theta$ we can achieve better estimation of earliest completion time of the set Θ. For an example Figure 2.

(ii) Since we take into account only total energy of the set Θ, we assume that all activities in Θ are fully "elastic". For example for $\Theta = \{i\}$ from Figure 1 $\text{preEct}(\Theta) = 1$ even though activity i cannot end before $\text{ect}_i = 4$. [1]

In this paper we will address only issue (i) by defining better estimation of earliest completion time:

$$\text{Ect}(\Theta) = \max_{\Omega \subseteq \Theta} \{\text{preEct}\,(\Omega)\}$$

What is algebraically equivalent to:

$$\text{Ect}(\Theta) = \max_{\Omega \subseteq \Theta} \left\{ \text{est}_\Omega + \left\lceil \frac{\text{e}_\Omega}{C} \right\rceil \right\} = \left\lceil \frac{\max_{\Omega \subseteq \Theta} \{C\,\text{est}_\Omega + \text{e}_\Omega\}}{C} \right\rceil$$

Lets call the numerator of the last fraction *energy envelope* of the set Θ:

$$\text{Env}(\Theta) = \max_{\Omega \subseteq \Theta} \{C\,\text{est}_\Omega + \text{e}_\Omega\} \tag{3}$$

Hence:

$$\text{Ect}(\Theta) = \left\lceil \frac{\text{Env}(\Theta)}{C} \right\rceil$$

For an example of $\text{Ect}(\Theta)$ and $\text{Env}(\Theta)$ see Figure 3.

The reason we defined energy envelope is that it is simpler to use in the algorithms than the earliest completion time.

[1] That is also the reason why earliest start time of an activity i is denoted by lowercase letters ect_i but earliest completion time of a set of activities Θ is denoted by $\text{Ect}(\Theta)$ with capital E.

3 Overload, E-Feasibility

This section provides a variation of the e-feasibility checking algorithm by Armin Wolf and Gunnar Schrader [7]. This algorithm is the basis of the Max Energy algorithm presented later in this paper.

Traditionally, we define an *overload* as a situation when a subset of activities $\Omega \subseteq T$ requires more resource energy than what is available between earliest possible start and latest possible end time of the set Ω (see for example [3]). If there is overload then no solution exists:

$$\forall \Omega \subseteq T: \quad (\mathrm{e}_\Omega > C(\mathrm{lct}_\Omega - \mathrm{est}_\Omega) \quad \Rightarrow \quad \text{fail}) \tag{OL}$$

where:

$$\mathrm{lct}_\Omega = \max\{\mathrm{lct}_i,\ i \in \Omega\}$$

If there is no overload then we say that the problem is *e-feasible*.

It would take too much time to check all subsets $\Omega \subseteq T$. Fortunately there is a faster way:

Proposition 1. *The problem is e-feasible if and only if*

$$\forall j \in T: \quad \mathrm{Env}(\mathrm{LCut}(T, j)) \leq C\,\mathrm{lct}_j$$

where $\mathrm{LCut}(T, j)$ *is a* left cut of T by activity j*:*

$$\mathrm{LCut}(T, j) = \{k,\ k \in T\ \&\ \mathrm{lct}_k \leq \mathrm{lct}_j\}$$

Proof. We will prove the equivalence by proving both implications:

1. If rule (OL) detects overload then there is a set Ω such that $C\,\mathrm{lct}_\Omega < C\,\mathrm{est}_\Omega + \mathrm{e}_\Omega$. In this case we define $j \in \Omega$ to be activity from set Ω such that $\mathrm{lct}_j = \mathrm{lct}_\Omega$ (if there are more activities with this property, we can choose arbitrarily). Thanks to the definition of j it holds that $\Omega \subseteq \mathrm{LCut}(T, j)$ and therefore:

$$C\,\mathrm{lct}_j = C\,\mathrm{lct}_\Omega < C\,\mathrm{est}_\Omega + \mathrm{e}_\Omega \overset{(3)}{\leq} \mathrm{Env}(\mathrm{LCut}(T, j))$$

   Therefore the second rule also detects overload.
2. If $\mathrm{Env}(\mathrm{LCut}(T, j)) > C\,\mathrm{lct}_j$ then by (3) there is a set $\Omega \subseteq \mathrm{LCut}(T, j)$ such that $C\,\mathrm{est}_\Omega + \mathrm{e}_\Omega = \mathrm{Env}(\mathrm{LCut}(T, j))$. And for this set Ω:

$$C\,\mathrm{lct}_\Omega \leq C\,\mathrm{lct}_j < \mathrm{Env}(\mathrm{LCut}(T, j)) = C\,\mathrm{est}_\Omega + \mathrm{e}_\Omega$$

   And therefore rule (OL) also detects overload. □

The key idea of the algorithm is to organize set $\mathrm{LCut}(T, j) = \Theta$ in a balanced binary tree, which we call Θ-tree (it is an extension of Θ-tree structure for unary resources described for example in [6]). Activities are represented by leaf nodes[2]

[2] This is the main difference from the algorithm in [7], that algorithm represents activities also in the internal nodes of the tree.

and sorted by est_i from left to right. Each node v of the tree holds the following values:

$$\mathrm{e}_v = \mathrm{e}_{\mathrm{Leaves}(v)} \tag{4}$$

$$\mathrm{Env}_v = \mathrm{Env}\left(\mathrm{Leaves}\left(v\right)\right) \tag{5}$$

Where $\mathrm{Leaves}(v)$ is a set of all activities represented by leaves of the subtree rooted in v. Figure 4 shows a Θ-tree from an example from Figure 3. Notice that the energy envelope of the represented set Θ is equivalent to the value Env of the root node.

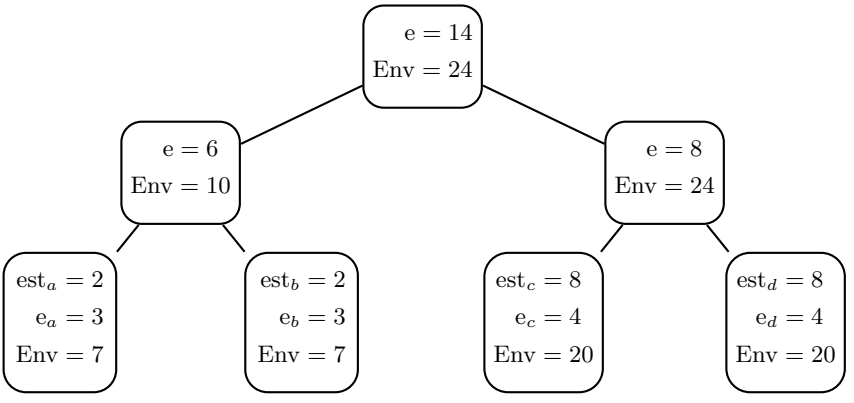


Fig. 4. An example of a Θ-tree for $\Theta = \{a, b, c, d\}$ from Figure 3

For a leaf node v representing an activity $i \in T$ the values in the tree are set to:

$$\mathrm{e}_v = \mathrm{e}_i$$

$$\mathrm{Env}_v = \mathrm{Env}\left(\{i\}\right) = C\,\mathrm{est}_i + \mathrm{e}_i$$

For internal nodes v these values can be computed recursively from their children nodes $\mathrm{left}(v)$ and $\mathrm{right}(v)$:

Proposition 2. *For an internal node v, values e_v and Env_v can be computed by the following recursive formula:*

$$\mathrm{e}_v = \mathrm{e}_{\mathrm{left}(v)} + \mathrm{e}_{\mathrm{right}(v)} \tag{6}$$

$$\mathrm{Env}_v = \max\left\{\mathrm{Env}_{\mathrm{left}(v)} + \mathrm{e}_{\mathrm{right}(v)},\ \mathrm{Env}_{\mathrm{right}(v)}\right\} \tag{7}$$

Proof. Formula (6) is trivial, we will prove only formula (7). From the definition (5), the value Env_v is:

$$\mathrm{Env}_v = \mathrm{Env}\left(\mathrm{Leaves}\left(v\right)\right) = \max\left\{C\,\mathrm{est}_\Omega + \mathrm{e}_\Omega,\ \Omega \subseteq \mathrm{Leaves}(v)\right\}$$

With respect to the node v we will split the sets Ω into the following two categories:

1. $\mathrm{Left}(v) \cap \Omega = \emptyset$, i.e., $\Omega \subseteq \mathrm{Right}(v)$. Clearly:

$$\max\{C\,\mathrm{est}_\Omega + \mathrm{e}_\Omega,\ \Omega \subseteq \mathrm{Right}(v)\} = \mathrm{Env}\,(\mathrm{Right}(v)) = \mathrm{Env}_{\mathrm{right}(v)}$$

2. $\mathrm{Left}(v) \cap \Omega \neq \emptyset$. Then $\mathrm{est}_\Omega = \mathrm{est}_{\Omega \cap \mathrm{Left}(v)}$ because leaf nodes are sorted by est_i. Let S be the set of all possible Ω considered in this part of the proof:

$$S = \{\Omega,\ \Omega \subseteq \Theta\ \&\ \Omega \cap \mathrm{Left}(v) \neq \emptyset\}$$

Then:

$$\begin{aligned}
&\max\{C\,\mathrm{est}_\Omega + \mathrm{e}_\Omega,\ \Omega \in S\} = \\
&\quad = \max\left\{C\,\mathrm{est}_{\Omega\cap\mathrm{Left}(v)} + \mathrm{e}_{\Omega\cap\mathrm{Left}(v)} + \mathrm{e}_{\Omega\cap\mathrm{Right}(v)},\ \Omega \in S\right\} = \\
&\quad = \max\left\{C\,\mathrm{est}_{\Omega\cap\mathrm{Left}(v)} + \mathrm{e}_{\Omega\cap\mathrm{Left}(v)},\ \Omega \in S\right\} + \mathrm{e}_{\mathrm{Right}(v)} = \\
&\quad = \mathrm{Env}_{\mathrm{left}(v)} + \mathrm{e}_{\mathrm{right}(v)}
\end{aligned}$$

We used the fact that the maximum is achieved only by such a set Ω for which $\mathrm{Right}(v) \subsetneq \Omega$. We also used the fact that $\Omega \cap \mathrm{Left}(v)$ enumerates all possible subsets of $\mathrm{Left}(v)$ and therefore:

$$\max\left\{C\,\mathrm{est}_{\Omega\cap\mathrm{Left}(v)} + \mathrm{e}_{\Omega\cap\mathrm{Left}(v)},\ \Omega \in S\right\} = \mathrm{Env}_{\mathrm{left}(v)}$$

Combining the results of parts 1 and 2 together we see that formula (7) is correct. □

Thanks to formulas (6) and (7), computation of values e_v and Env_v can be integrated within usual operations with balanced binary trees without changing their time complexity, see Table 1.

Table 1. Worst-case time complexities of operations on Θ-tree

Operation	Time Complexity
$\Theta := \emptyset$	$\mathcal{O}(1)$
$\Theta := \Theta \cup \{i\}$	$\mathcal{O}(\log n)$
$\Theta := \Theta \setminus \{i\}$	$\mathcal{O}(\log n)$
$\mathrm{Env}(\Theta)$	$\mathcal{O}(1)$

The idea of the overload checking algorithm follows. We will iterate over all left cuts $\mathrm{LCut}(T, j)$ by non-decreasing lct_j. The cuts will be represented by Θ-tree what allows to quickly recompute $\mathrm{Env}\,(\mathrm{LCut}\,(T, j))$ each time when j is changed. For each set $\Theta = \mathrm{LCut}(T, j)$ we check e-feasibility using Proposition 1. The resulting Algorithm 1.1 has worst-case time complexity $\mathcal{O}(n \log n)$.

Algorithm 1.1. Overload Checking in $\mathcal{O}(n \log n)$

```
1  Θ := ∅;
2  for j ∈ T in non-decreasing order of lct_j do begin
3    Θ := Θ ∪ {j};
4    if Env(Θ) > C lct_j then
5      fail; {No solution exists}
6  end;
```

4 Max Energy Propagation

In this section we will extend the algorithm for overload detection to compute maximum energy of each activity $i \in T$. The idea of the propagation is to protect possible overload caused by increase of some energy demand e_i.

Consider for example situation on Figure 3. In this example, minimum required energy of activity c is $\mathrm{e}_c = 4$. Maximum required energy $\overline{\mathrm{e}}_c$ is not depicted on the figure, but lets say that $\overline{\mathrm{e}}_c = 10$. However considering also activity d (which requires at least $\mathrm{e}_d = 4$) the maximum feasible energy for activity c is 6, otherwise there would be an overload for $\Omega = \{c, d\}$. Therefore we can update $\overline{\mathrm{e}}_c := 6$, and according to formula (2) $\overline{\mathrm{p}}_c := 3$. What we just described on the example is the goal of the presented algorithm: for each activity $i \in T$, compute maximum feasible energy $\overline{\mathrm{e}}_i$ such that if e_i is increased above $\overline{\mathrm{e}}_i$ then there will be an overload.

In Proposition 1 we have learned that the resource is e-feasible iff:

$$\forall j \in T: \quad \mathrm{Env}\left(\mathrm{LCut}\left(T, j\right)\right) \leq C\,\mathrm{lct}_j$$

In other words we can assign to each set $\mathrm{LCut}\,(T, j)$ a maximum feasible envelope $\overline{\mathrm{Env}}$:

$$\overline{\mathrm{Env}}\left(\mathrm{LCut}\left(T, j\right)\right) := C\,\mathrm{lct}_j \tag{8}$$

The idea is to propagate maximum feasible envelope from the set $\mathrm{LCut}(T, j)$ into all its members and this way find maximum feasible energy of all activities.

Lets have have a look on the Θ-tree representing a particular set $\Theta = \mathrm{LCut}(T, j)$. For overload checking we compute recursively in each node the following values by formulas (6) and (7):

$$\mathrm{e}_v = \mathrm{e}_{\mathrm{left}(v)} + \mathrm{e}_{\mathrm{right}(v)} \tag{6}$$

$$\mathrm{Env}_v = \max\left\{\mathrm{Env}_{\mathrm{left}(v)} + \mathrm{e}_{\mathrm{right}(v)},\ \mathrm{Env}_{\mathrm{right}(v)}\right\} \tag{7}$$

The idea is to extend the tree by adding two more attributes into each node of the tree:

- maximum feasible energy envelope $\overline{\mathrm{Env}}_v$ of the set $\mathrm{Leaves}(v)$,
- and maximum feasible energy $\overline{\mathrm{e}}_v$ for the set $\mathrm{Leaves}(v)$.

The additional attributes can be also computed recursively, this time from root down to the leaves. It starts at the root node r (see (8)):

$$\overline{\mathrm{Env}}_r := C\,\mathrm{lct}_j \tag{9}$$

$$\overline{\mathrm{e}}_r := \infty \tag{10}$$

The recursive rules to propagate these values down the tree are:

$$\overline{\mathrm{Env}}_{\mathrm{right}(v)} := \overline{\mathrm{Env}}_v \tag{11}$$

$$\overline{\mathrm{Env}}_{\mathrm{left}(v)} := \overline{\mathrm{Env}}_v - \mathrm{e}_{\mathrm{right}(v)} \tag{12}$$

$$\overline{\mathrm{e}}_{\mathrm{right}(v)} := \min\left\{\overline{\mathrm{Env}}_v - \mathrm{Env}_{\mathrm{left}(v)}, \overline{\mathrm{e}}_v - \mathrm{e}_{\mathrm{left}(v)}\right\} \tag{13}$$

$$\overline{\mathrm{e}}_{\mathrm{left}(v)} := \overline{\mathrm{e}}_v - \mathrm{e}_{\mathrm{right}(v)} \tag{14}$$

For an example of computation of $\overline{\mathrm{e}}$ and $\overline{\mathrm{Env}}$ see Figure 5.

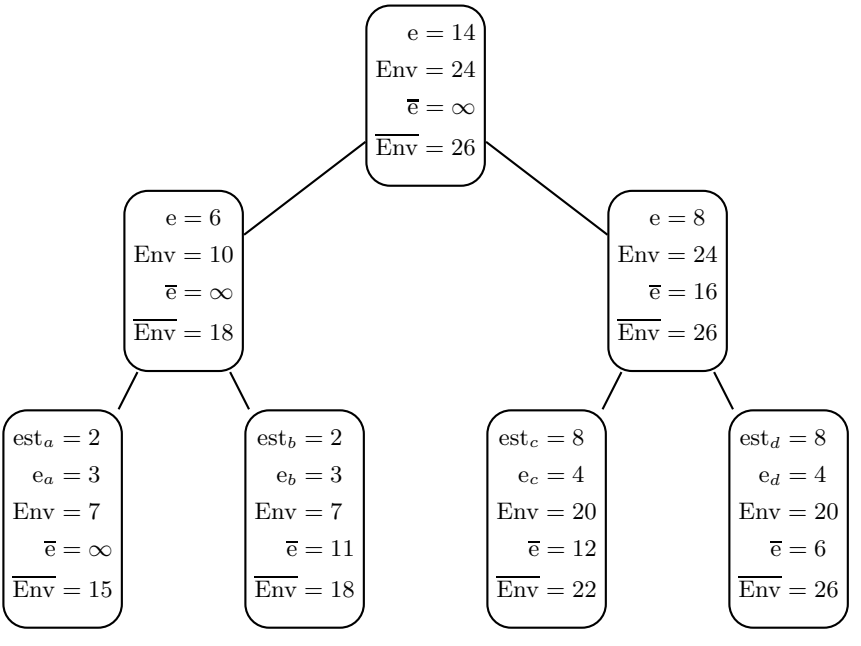


Fig. 5. Computation of $\overline{\mathrm{Env}}$ and $\overline{\mathrm{e}}$ for $\Theta = \{a, b, c, d\}$ from Figure 3. Notice that from the nodes representing activities c and d we can conclude that $\overline{\mathrm{e}}_c \leq 6$ and $\overline{\mathrm{e}}_d \leq 6$. In case of activity d because of $\overline{\mathrm{e}}$ in the node, in case of activity c because of $\overline{\mathrm{Env}}$ in the node as will be described by (15).

Formal proof of the recursive rules (11) – (14) will follow. But let us first explain for example construction of formula (13). If $\mathrm{e}_{\mathrm{right}(v)}$ is increased then it will cause also an increase of e_v by formula (6). However maximum feasible value

of e_v is $\overline{\mathrm{e}}_v$ and therefore the maximum feasible value of $\mathrm{e}_{\mathrm{right}(v)}$ has to fulfill the following formula:

$$\overline{\mathrm{e}}_{\mathrm{right}(v)} \leq \overline{\mathrm{e}}_v - \mathrm{e}_{\mathrm{left}(v)}$$

Similarly, increase of $\mathrm{e}_{\mathrm{right}(v)}$ can lead to the increase of Env_v by formula (7) but it cannot exceed the maximum feasible value $\overline{\mathrm{Env}}_v$. Therefore:

$$\overline{\mathrm{e}}_{\mathrm{right}(v)} \leq \overline{\mathrm{Env}}_v - \mathrm{Env}_{\mathrm{left}(v)}$$

Combining the these two formulas we get rule (13). The remaining three rules (11), (12) and (14) are constructed in a similar way.

Let us formally proof correctness of the rules (11) – (14). We start with the following lemma:

Lemma 1. *For a node w and its parent node v in a Θ-tree: if one of the values $\overline{\mathrm{Env}}_w$ and $\overline{\mathrm{e}}_w$ are not respected (that is $\mathrm{Env}_w > \overline{\mathrm{Env}}_w$ or $\mathrm{e}_w > \overline{\mathrm{e}}_w$) then at least one of the values $\overline{\mathrm{Env}}_v$, $\overline{\mathrm{e}}_v$ is not respected too.*

Proof. We will split the proof into two parts depending on whether w is left or right son of node v:

1. Case $w = \mathrm{left}(v)$. If $\overline{\mathrm{e}}_w$ is not respected then:

$$\begin{aligned} \mathrm{e}_w > \overline{\mathrm{e}}_w &\overset{(14)}{=} \overline{\mathrm{e}}_v - \mathrm{e}_{\mathrm{right}(v)} \\ \mathrm{e}_w + \mathrm{e}_{\mathrm{right}(v)} &> \overline{\mathrm{e}}_v \\ \mathrm{e}_v &> \overline{\mathrm{e}}_v \qquad \text{by (6)} \end{aligned}$$

   Therefore if $\overline{\mathrm{e}}_v$ is not respected then $\overline{\mathrm{e}}_v$ is not respected too.
   Similarly if $\overline{\mathrm{Env}}_w$ is not respected then:

$$\begin{aligned} \mathrm{Env}_w > \overline{\mathrm{Env}}_w &\overset{(12)}{=} \overline{\mathrm{Env}}_v - \mathrm{e}_{\mathrm{right}(v)} \\ \mathrm{Env}_w + \mathrm{e}_{\mathrm{right}(v)} &> \overline{\mathrm{Env}}_v \\ \max\{\mathrm{Env}_w + \mathrm{e}_{\mathrm{right}(v)},\ \mathrm{Env}_{\mathrm{right}(v)}\} &> \overline{\mathrm{Env}}_v \\ \mathrm{Env}_v &> \overline{\mathrm{Env}}_v \qquad \text{by (7)} \end{aligned}$$

   So if $\overline{\mathrm{Env}}_w$ is not respected then $\overline{\mathrm{Env}}_v$ is not respected too.
2. Case $w = \mathrm{right}(v)$. If $\overline{\mathrm{e}}_w$ is not respected then:

$$\mathrm{e}_w > \overline{\mathrm{e}}_w \overset{(13)}{=} \min\{\overline{\mathrm{Env}}_v - \mathrm{Env}_{\mathrm{left}(v)},\ \overline{\mathrm{e}}_v - \mathrm{e}_{\mathrm{left}(v)}\}$$

   Therefore
   (a) Either:

$$\begin{aligned} \mathrm{e}_w &> \overline{\mathrm{Env}}_v - \mathrm{Env}_{\mathrm{left}(v)} \\ \mathrm{Env}_{\mathrm{left}(v)} + \mathrm{e}_w &> \overline{\mathrm{Env}}_v \\ \mathrm{Env}_v &> \overline{\mathrm{Env}}_v \qquad \text{by (7)} \end{aligned}$$

   And so $\overline{\mathrm{Env}}_v$ is not respected.

(b) Or:

$$\begin{aligned} \mathrm{e}_w &> \overline{\mathrm{e}}_v - \mathrm{e}_{\mathrm{left}(v)} \\ \mathrm{e}_{\mathrm{left}(v)} + \mathrm{e}_w &> \overline{\mathrm{e}}_v \\ \mathrm{e}_v &> \overline{\mathrm{e}}_v \qquad \text{by (6)} \end{aligned}$$

And so $\overline{\mathrm{e}}_v$ is not respected.

Finally if $\overline{\mathrm{Env}}_w$ is not respected then:

$$\mathrm{Env}_w > \overline{\mathrm{Env}}_w \overset{(11)}{=} \overline{\mathrm{Env}}_v$$

Therefore

$$\mathrm{Env}_v \overset{(7)}{=} \max\{\mathrm{Env}_{\mathrm{left}(v)} + \mathrm{e}_w,\ \mathrm{Env}_w\} \geq \mathrm{Env}_w > \overline{\mathrm{Env}}_v$$

And thus $\overline{\mathrm{Env}}_v$ is not respected. □

A consequence of this lemma is:

Proposition 3. *Let $i \in \mathrm{LCut}(T, j)$ and let v be a leaf node representing activity i in Θ-tree for $\mathrm{LCut}(T, j)$. If $\mathrm{e}_i > \min\left\{\overline{\mathrm{e}}_v,\ \overline{\mathrm{Env}}_v - C\,\mathrm{est}_i\right\}$ then there is an overload and therefore the problem is unfeasible.*

Proof. If $\mathrm{e}_i > \min\left\{\overline{\mathrm{e}}_v,\ \overline{\mathrm{Env}}_v - C\,\mathrm{est}_i\right\}$ then it means that either $\overline{\mathrm{e}}_v$ or $\overline{\mathrm{Env}}_v$ in the node v is not respected. By the previous lemma it means that at least one of these values is not respected also in parent node of v. And so we continue this way to the root node r and prove that $\overline{\mathrm{e}}_r$ or $\overline{\mathrm{Env}}_r$ is not respected.

However for root node r, $\overline{\mathrm{e}}_r = \infty$ by (10) therefore $\overline{\mathrm{e}}_r$ has to be respected. The conclusion is that $\overline{\mathrm{Env}}_r$ is not respected and therefore:

$$\begin{aligned} \mathrm{Env}_r > \overline{\mathrm{Env}}_r &\overset{(9)}{=} C\,\mathrm{lct}_j \\ \mathrm{Env}\left(\mathrm{LCut}\left(T, j\right)\right) &> C\,\mathrm{lct}_j \end{aligned}$$

So there is overload by Proposition 1. □

The proposition above gives as an upper bound for maximum energy available for each activity $i \in \mathrm{LCut}(T, j)$:

$$\overline{\mathrm{e}}_i \leq \min\left\{\overline{\mathrm{e}}_v,\ \overline{\mathrm{Env}}_v - C\,\mathrm{est}_i\right\} \tag{15}$$

Notice that for example on Figures 3 and 5 the formula (15) gives $\overline{\mathrm{e}}_c = 6$ and $\overline{\mathrm{e}}_d = 6$ and therefore by (2) $\overline{\mathrm{p}}_c = 3$ and $\overline{\mathrm{p}}_d = 3$.

The basic idea of the algorithm follows: we iterate over all activities $j \in T$ and for each j we build Θ-tree representing $\mathrm{LCut}(T, j)$ by adding new nodes into the Θ-tree from the previous iteration. In each Θ-tree we propagate the maximum energy envelope $C\,\mathrm{lct}_j$ from the root to leave nodes and assign maximum energies to activities $i \in \mathrm{LCut}(T, j)$ according to formula (15). First version of the algorithm with time complexity $\mathcal{O}(n^2)$ is provided by Algorithm 1.2. Note that this is not the $\mathcal{O}(n^2)$ algorithm by Wim Nuijten, for better understanding we start with $\mathcal{O}(n^2)$ algorithm and then speed it up to $\mathcal{O}(n \log n)$.

The algorithm uses two procedures:

- `push_down(`v`)` pushes the values $\overline{\mathrm{Env}}_v$ and $\overline{\mathrm{e}}_v$ from the node v down the tree using the rules (11) – (14).
- `set_energy_max(`i`)` sets maximum energy $\overline{\mathrm{e}}_i$ of the activity i using formula (15).

Algorithm 1.2. Maximum energy propagation in $\mathcal{O}(n^2)$

```
1  Θ := ∅;
2  for j ∈ T in non-decreasing order of lct_j do begin
3    Θ := Θ ∪ {j};
4    if Env(Θ) > C lct_j then
5      fail; {No solution exists}
6    Env̄_Θ := C lct_j;
7    for nodes v in Θ-tree in non-decreasing order of their depth do
8      push_down(v);
9    for i ∈ Θ do
10     set_energy_max(i);
11 end;
```

Time complexity of this algorithm is $\mathcal{O}(n^2)$ because the inner cycles on lines 7 – 8 and 9 – 10 have time complexity $\mathcal{O}(n)$. In the following we will show how to improve the time complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$.

The key observation is that it is not necessary to push values $\overline{\mathrm{e}}_v$ and $\overline{\mathrm{Env}}_v$ down to leaves immediately. The values $\overline{\mathrm{e}}_v$ and $\overline{\mathrm{Env}}_v$ stays valid until new node is inserted into the subtree of v. Therefore it is possible to postpone `push_down(`v`)` until new node is inserted somewhere under the node v.

Current procedure `push_down(`v`)` simply overwrites values $\overline{\mathrm{e}}$ and $\overline{\mathrm{Env}}$ in children nodes of v. However that is no longer possible in the new algorithm because children nodes may contain information which was not pushed down yet. Therefore it is necessary create new procedure `push_down2(`v`)` which implements modified rules (11) – (14):

$$\overline{\mathrm{Env}}_{\mathrm{right}(v)} := \min\left\{\overline{\mathrm{Env}}_v,\ \overline{\mathrm{Env}}_{\mathrm{right}(v)}\right\} \tag{16}$$

$$\overline{\mathrm{Env}}_{\mathrm{left}(v)} := \min\left\{\overline{\mathrm{Env}}_v - \mathrm{e}_{\mathrm{right}(v)},\ \overline{\mathrm{Env}}_{\mathrm{left}(v)}\right\} \tag{17}$$

$$\overline{\mathrm{e}}_{\mathrm{right}(v)} := \min\left\{\overline{\mathrm{Env}}_v - \mathrm{Env}_{\mathrm{left}(v)},\ \overline{\mathrm{e}}_v - \mathrm{e}_{\mathrm{left}(v)},\ \overline{\mathrm{e}}_{\mathrm{right}(v)}\right\} \tag{18}$$

$$\overline{\mathrm{e}}_{\mathrm{left}(v)} := \min\left\{\overline{\mathrm{e}}_v - \mathrm{e}_{\mathrm{right}(v)},\ \overline{\mathrm{e}}_{\mathrm{left}(v)}\right\} \tag{19}$$

The values $\overline{\mathrm{e}}_v$ and $\overline{\mathrm{Env}}_v$ must be initialized to:

$$\overline{\mathrm{e}}_v := \infty$$

$$\overline{\mathrm{Env}}_v := \infty$$

The postponed calls of `push_down2(`v`)` must be executed just before new node is added into the tree. The most suitable place is to replace $\Theta := \Theta \cup \{j\}$ on line 3 by procedure `add(`Θ, j`)` which will make necessary postponed calls.

For simplicity, lets assume that the shape of the tree is fixed and no rebalancing occurs during the addition of new node into the tree[3]. The procedure `add(`Θ, j`)` has following steps:

1. Find a node w under which new node will be inserted.
2. Call `push_down2(`v`)` on all nodes v on the path from the root to w, and then reset these nodes to:

$$\overline{\mathrm{e}}_v := \infty$$
$$\overline{\mathrm{Env}}_v := \infty$$

3. Add new node in the tree, fill the leaf representing j by data about the activity j.
4. Recompute values e_v and Env_v on the path from this leaf to the root.

The resulting Algorithm 1.3 has worst case time complexity $\mathcal{O}(n \log n)$.

Algorithm 1.3. Maximum energy propagation in $\mathcal{O}(n \log n)$

```
1  Θ := ∅;
2  for j ∈ T in non-decreasing order of lct_j do begin
3    add(Θ, j);
4    if Env(Θ) > C lct_j then
5      fail; {No solution exists}
6    Env_Θ := C lct_j;
7  end;
8  for nodes v in Θ-tree in non-decreasing order of their depth do
9    push_down2(v);
10 for i ∈ T do
11   set_energy_max(i);
```

5 Optional Activities

As mentioned in the introduction, IBM ILOG CP Optimizer 2.0 [1] introduce a new variable type designed for scheduling – interval variable. The difference between activity (as we used this word in this paper) and an interval variable is that an interval variable by itself does not require any resource. An activity is created when an interval variable is constrained to require a resource. One interval variable may require more than one resource, in this case the interval variable is associated with several activities and all of them share their start times and end times.

[3] This can be achieved for example by computing perfectly balanced tree of all activities in advance.

Interval variable in IBM ILOG CP Optimizer has very important aspect: it can be declared as optional, that is, it may or may not be present in the solution. In this case all its activities are also optional.

Typical use of optional interval variables are alternative between two different actions. This can be easily modeled by two optional interval variables and a constraint that exactly one of them is present in the solution. Both optional intervals can require some discrete cumulative resource during their execution, however the interval which is not present in the solution does not affect any resource. For more details about optional interval variables see [4, 1].

From the point of view of the resource, there are some optional activities which may or may not be executed on the resource, this is yet to be decided. During the search an optional activity may become:

A) preset if:
   1) we decide to execute it as a search decision,
   2) or if we proved (by propagation) that no other alternative is possible,
B) absent if:
   1) we decided to not to execute it by a search decision,
   2) or if we proved (by propagation) that execution of this activity is not possible.

The algorithm Max Energy presented in this paper can do the propagation B2) above. This is very important propagation because usually any propagation on optionality status has a big impact on other variables.

How to use Algorithm 1.3 with optional activities? First observe that from the point of view of discrete cumulative resource, there is no difference between absent activity and activity with zero energy (that is, activity with zero duration or zero capacity requirement). Zero-energy activity can be processed anytime even though the resource is already full.

The idea is to use Algorithm 1.3 directly without any modification, but on modified input data. If an activity is optional then (just for the algorithm) its minimum energy is set to zero. This way, optional activity cannot influence any other activity, however upper bound $\overline{\mathrm{e}}_i$ for energy computed by the algorithm is valid also for optional activity i. Furthermore if $\overline{\mathrm{e}}_i < \mathrm{e}_i$ we can deduce that the activity i cannot be present in the solution.

6 Conclusions and Further Work

This paper presents a new Max Energy propagation algorithm which updates maximum durations and maximum capacity requirements on discrete cumulative resource with optional activities. The algorithm has better time complexity ($\mathcal{O}(n \log n)$ versus $\mathcal{O}(n^2)$) than old never published algorithm by Wim Nuijten. Experiments show that the new algorithm begin to be faster than the old one for n around 10. The algorithm is used by IBM ILOG CP Optimizer [1] since version 2.0.

References

[1] IBM ILOG CP Optimizer, http://www.ilog.com/products/cpoptimizer/
[2] Mercier, L., van Hentenryck, P.: Edge finding for cumulative scheduling. Informs Journal of Computing 20(1), 143–153 (2008)
[3] Philippe Baptiste, C.L.P., Nuijten, W.: Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems. Kluwer Academic Publishers, Dordrecht (2001)
[4] Philippe Laborie, J.R.: Reasoning with conditional time-intervals. In: Wilson, D., Lane, H.C. (eds.) Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference, Coconut Grove, Florida, USA, May 15-17, 2008, pp. 555–560. AAAI Press, Menlo Park (2008)
[5] Schutt, A., Wolf, A., Schrader, G.: Not-first and not-last detection for cumulative scheduling in $O(n^3 \log n)$. In: Umeda, M., Wolf, A., Bartenstein, O., Geske, U., Seipel, D., Takata, O. (eds.) INAP 2005. LNCS, vol. 4369, pp. 66–80. Springer, Heidelberg (2006)
[6] Vilím, P.: Global Constraints in Scheduling. PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, KTIML MFF, Universita Karlova, Malostranské náměstí 2/25, 118 00 Praha 1, Czech Republic (2007), http://vilim.eu/petr/disertace.pdf
[7] Wolf, A., Schrader, G.: $O(n \log n)$ overload checking for the cumulative constraint and its application. In: Umeda, M., Wolf, A., Bartenstein, O., Geske, U., Seipel, D., Takata, O. (eds.) INAP 2005. LNCS, vol. 4369, pp. 88–101. Springer, Heidelberg (2006)

Hybrid Branching

Tobias Achterberg[1] and Timo Berthold[2,⋆]

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

State-of-the-art solvers for Constraint Satisfaction Problems (CSP), Mixed Integer Programs (MIP), and satisfiability problems (SAT) are usually based on a branch-and-bound algorithm. The question how to split a problem into subproblems (*branching*) is in the core of any branch-and-bound algorithm. Branching on individual variables is very common in CSP, MIP, and SAT. The rules, however, which variable to choose for branching, differ significantly. In this paper, we present hybrid branching, which combines selection rules from all three fields.

Branching Rules. In MIP, *reliability pseudocost branching* [2] is the current state-of-the-art. This rule estimates the objective change in the LP relaxation when branching downwards and upwards. It uses the average objective gains per unit change, taken over all nodes, where this variable has been chosen for branching. The resulting two values are called the *pseudocosts* of a variable.

In CSP and SAT, where no objective function is available, one may better estimate the impact of a branching by taking the number of implied reductions of other variable domains into account [4]. In analogy to the pseudocosts, we call the estimated numbers of implied reductions the *inference values* of a variable.

In pure SAT solvers, learning short, valid conflict clauses from the analysis of infeasible subproblems is one of the key ingredients [5]. The *variable state independent decaying sum (VSIDS)* [6] branching strategy, which is a common rule in SAT solving, prefers variables that have been used to create recent conflict clauses. We call the VSIDS the *conflict values* of a variable.

The idea to use the average lengths of the conflict clauses a variable appears in for branching was recently suggested by Kilinc et. al. [3]. We call this the *conflict lengths* of a variable.

As noted above, all the described measures exist twice for each variable: for upwards and downwards branching. Therefore, one has to combine them into a single score value, which we do by multiplication [1].

Hybrid Branching. *Hybrid branching* combines all four selection criteria into a single one and additionally includes a score which is based on the number of subproblems that could be pruned due to branching on this variable, called the *cutoff values*. We first normalize all the five individual values by mapping them

⋆ Supported by the DFG Research Center Matheon *Mathematics for key technologies.*

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 309–311, 2009.

Table 1. Shifted geometric means of time (in sec.) and branch-and-bound nodes over four test sets

test set	MIPLIB2003		Cor@l		Cor@l-BP		Infeasible	
	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes
reliability	450.4	5091	803.6	4110	672.4	2145	290.7	5612
hybrid	445.6	5051	735.0	3575	577.2	1681	166.0	1998
ratio reli/hyb	1.01	1.01	1.09	1.15	1.16	1.28	1.75	2.81

into the interval $[0, 1)$. Afterwards, we take a weighted sum of them that puts a high weight on the pseudocosts, a medium weight on the conflict values and lengths, and a low weight on the inference and cutoff values.

Computational Results. We tested our approach on a set of infeasible binary programs (BPs) and two libraries of general MIP instances which are publicly available: the MIPLIB2003 (`http://miplib.zib.de`) and the Cor@l collection (`http://coral.ie.lehigh.edu`). All tests were performed on a Intel Xeon 5150 2.66 GHz with 4 MB cache and 8 GB RAM. A time limit of one hour was imposed.

We incorporated hybrid branching into the constraint integer programming framework SCIP [1], version 1.1.0.5, using SoPlex 1.4.1 as underlying LP solver. We compared hybrid branching against reliability pseudocost branching, which is state-of-the-art in MIP solving. The shifted geometric means of the running times and the branch-and-bound nodes were used as performance measures.

It turns out that for the MIPLIB2003, both branching rules perform equally well, the difference is 1% in mean. For the Cor@l library, hybrid branching outperforms reliability branching, the running time increases by 9%, the number of nodes by 15%, when using reliability branching instead of hybrid branching. If we only regard binary programs, which are roughly a third of the Cor@l test set, the difference in performance is even larger. This meets our expectations, since the inference and conflict values are especially meaningful for 0-1 variables.

The results for BPs gave rise to the last experiment, which showed that for a set of infeasible binary programs, reliability branching is 75% slower and nearly triplicates the number of branch-and-bound nodes. The conflict values and conflict lengths arose from the analysis of infeasible subproblems, which explains that taking them into account is crucial for handling infeasible MIPs.

Overall, hybrid branching is a successful integration of CSP, SAT, and MIP technologies, which enables to solve standard MIP problems faster. By now, hybrid branching is used as default branching rule in SCIP.

References

1. Achterberg, T.: SCIP: solving constraint integer programs. Mathematical Programming Computation (1) (2008)
2. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Operations Research Letters 33, 42–54 (2005)
3. Karzan, F.K., Nemhauser, G., Savelsbergh, M.: Information based branching rules in integer programming. Presentation at INFORMS Annual Meeting (2008)

4. Li, C.M., Anbulagan: Look-ahead versus look-back for satisfiability problems. In: Smolka, G. (ed.) CP 1997. LNCS, vol. 1330, pp. 342–356. Springer, Heidelberg (1997)
5. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. of Comp. 48, 506–521 (1999)
6. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. of the DAC (July 2001)

Constraint Programming and Mixed Integer Linear Programming for Rescheduling Trains under Disrupted Operations

A Comparative Analysis of Models, Solution Methods, and Their Integration

Rodrigo Acuna-Agost[1], Philippe Michelon[1], Dominique Feillet[2], and Serigne Gueye[3]

[1] Université d'Avignon et des Pays de Vaucluse, Laboratoire d'Informatique d'Avignon (LIA), F-84911 Avignon, France
rodrigo.acuna-agost@univ-avignon.fr
[2] Ecole des Mines de Saint-Etienne, F-13541 Gardanne, France
[3] Laboratoire de Mathématiques Appliquées du Havre, F-76058 Le Havre, France

Extended Abstract

The problem of rescheduling trains under disrupted operations has received more attention in the last years because of the increasing demand in railway transportation. Railway networks are more and more saturated and the probability that delayed trains affect others is large. Therefore, it is very important to react after incidents by recalculating new arrival/departure times and reassigning tracks/platforms with the aim of minimizing the effect of propagation of incidents.

It is possible to define the rescheduling railway problem as follows: given a set of trains, a railway network, an original schedule (i.e., arrival/departure times of every train as well as the complete assignment of tracks and platforms), and known delays of one or several trains; the problem is to create a new provisional schedule by minimizing the difference with the original plan respecting several operational and commercial constraints. We propose to use an objective function that penalizes delays, changes of tracks/platforms, and unplanned stops; all of them referred to the original (unperturbed) schedule.

We present two different approaches: MIP (Mixed Integer Linear Programming) and CP (Constraint Programming). The solutions of both are comparable because they use the same objective function. Nevertheless, the definition of decision variables and constraints differs significantly.

On the one hand, the MIP uses an extension of a formulation originally presented in [5]. The more interested reader will find a full description of the model in our previous work [3]. This model includes many practical rules and constraints, which explains its relative complexity compared to other models presented in the literature. It supports allocation of tracks and platforms, connection of trains, bidirectional multi-track lines, and extra time for accelerating and braking.

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 312–313, 2009.

Three solution methods are proposed: right-shift rescheduling, a MIP-based local search method, and a new approach called SAPI (statistical analysis of propagation of incidents) [2] [3] [4]. Our computational experiments show that these method are very efficient but limited to little and medium size instances. The MIP model become unmanageable for big instances because of the exponential number of binary variables used to model the order of trains.

On the other hand, the CP model has an important advantage: it has fewer variables and constraints. As a consequence, the CP model requires only a fraction of the memory needed by the MIP model. Nevertheless, experimental results show that solution methods associated to this model are not as good as MIP methods.

In order to develop an efficient method for bigger instances, we combined both models considering their advantages. This methodology uses a MIP-based method (e.g., SAPI) in a little subproblem limiting the time horizon after the incidents where affected trains are concentrated. Then, the value of the objective function is used to estimate a lower and upper bounds of the complete, big size, problem. These values are then added as constraints in the CP model. The purpose of the added constraints is to remove portions of the search space which cannot lead to better solutions than the upper bound and worse than the lower bound, i.e., the remained search space is limited to all feasible solutions between both bounds. Thank to constraint propagation, the domain reduction of the objective function is reflected on the domain of the decision variables. It should be noted that the quality of the solution and the efficiency of the method depend on the quality of these bounds. Consequently, several procedures to estimate these bound are considered.

For evaluating the quality of these methods, some computational experiments have been executed in a real French network composed of 4454 arrival/departure events. The results show that MIP methods are the best for little and medium size instances while the combined methodology is better to solve big instances.

References

1. Acuna-Agost, R., Gueye, S.: Web MAGES (2006), http://awal.univ-lehavre.fr/~lmah/mages/
2. Acuna-Agost, R., Feillet, D., Gueye, S., Michelon, P.: SAPI - Statistical Analysis of Propagation of Incidents. In: A new approach applied to solve the railway rescheduling problem. NCP 2007, Rouen, France (2007)
3. Acuna-Agost, R., Feillet, D., Gueye, S., Michelon, P.: A MIP-based local search method for the railway rescheduling problem, Technical report, LIA (submitted for publication) (2009)
4. Acuna-Agost, R., Feillet, D., Gueye, S., Michelon, P.: Méthode PLNE pour le réordonnancement de plan de circulation ferroviaire en cas d'incident. In: ROADEF 2008, Clermont-Ferrand, France (2008)
5. Tornquist, J., Persson, J.A.: N-tracked railway traffic re-scheduling during disturbances. Transportation Research Part B (2006)

Constraint Models for Sequential Planning

Roman Barták and Daniel Toropila

Charles University in Prague, Faculty of Mathematics and Physics
Malostranské nám. 2/25, 118 00 Praha 1, Czech Repuplic
roman.bartak@mff.cuni.cz, daniel.toropila@matfyz.cz

1 Background

Planning and constraint satisfaction are important areas of Artificial Intelligence, but surprisingly constraint satisfaction techniques are not widely applied to planning problems where dedicated solving algorithms dominate. Classical AI planning deals with finding a sequence of actions that transfer the initial state of the world into a desired state. One of the main difficulties of planning is that the size of the solution – the length of the plan – is unknown in advance. On the other hand, a constraint satisfaction problem needs to be fully specified in advance before the constraint model goes to the solver. As Kautz and Selman showed, the problem of shortest-plan planning can be translated to a series of SAT problems, where each SAT instance encodes the problem of finding a plan of a given length. First, we start with finding a plan of length 1 and if it does not exist then we continue with a plan of length 2 etc. until the plan is found. The same approach can be applied when constraint satisfaction is used instead of SAT. There exists a straightforward constraint model of this type but more advanced constraint models exploit the structure of a so called planning graph, namely GP-CSP and CSP-PLAN. There also exist hand-crafted constraint models for particular planning problems such as CPlan. All these models share the logical representation of the problem where Boolean variables describe whether a given proposition holds in a given state and whether a given action is applied to a given state. Constraints are in the form of logical formulae (frequently implications). In our opinion, these models do not exploit fully the potential of constraint satisfaction, namely domain filtering techniques and global constraints. Therefore, we suggested to use multi-valued representation of states based on the state variable formalism SAS^+ that contributes to fewer variables with larger domains where domain filtering pays off. Also, we proposed to encapsulate the sets of logical constraints into combinatorial constraints with an extensionally defined set of admissible tuples. This extended abstract summarizes our recent results in the design of constraint models for planning problems presented in [1,2].

2 Reformulating Constraint Models

We first reformulated CSP-Plan, so far the fastest sequential constraint-based planner, to multi-valued representation. There is a set of variables V_i^s describing values of state

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 314 – 315, 2009.

variables for each state s and action variables A^s. These variables are connected by constraints describing preconditions of actions:

$$A^s = act \rightarrow \text{Pre}(act)^s\ ,\ \forall act \in \text{Dom}(A^s),$$

where *act* is a value representing particular action and Pre(*act*) is a formula describing precondition of the action. The second group of constraints describes how the state variable is changing between the states, a so called successor-state constraint:

$$V_i^s = val \leftrightarrow A^{s-1} \in C(i,val) \vee (V_i^{s-1} = val \wedge A^{s-1} \in N(i)),$$

where $C(i,val)$ denotes the set of actions containing $V_i = val$ among their effects, and $N(i)$ denotes the set of actions that do not affect V_i.

The main problems of the above model lay in the large number of constraints and weak domain filtering due to disjunctive character of constraints. To overcome these problems, we adopted the approach of substituting the above-mentioned propositional formulae using the constraints with extensionally defined set of admissible tuples (sometimes also called combinatorial constraints). The idea was to union the scope of "similar" constraints and to define the admissible tuples in extension rather than using a formula. In particular, we used one constraint to encapsulate all precondition constraints in each state and one constraint per state variable for each pair of subsequent states. The experimental results confirmed that the reformulated model leads to much better efficiency [1].

2.1 Enhancing Constraint Models

It is rare that a direct constraint model is enough to solve complex problems and frequently some extensions are necessary. We already proposed three extensions of the above base model. The first extension mimics the technique called lifting where instead of instantiating the action variables, we use domain splitting. This technique decreases the branching factor during backward (regression) planning. The second improvement adds dominance constraints to break plan permutation symmetries. The motivation is to forbid some subsequences of actions that lead to states reached by already explored plans. Finally, we used the idea of singleton consistency to validate whether an action that newly appeared in the last layer of the plan has some supporting action in the layer before. This technique decreases the size of the search space by eliminating some unreachable actions. All these extensions contributed to improved performance of the planner especially for harder problems [2].

Acknowledgments. The research is supported by the Czech Science Foundation under the contract no. 201/07/0205.

References

1. Barták, R., Toropila, D.: Reformulating Constraint Models for Classical Planning. In: Proceedings of FLAIRS 2008, pp. 525–530. AAAI Press, Menlo Park (2008)
2. Barták, R., Toropila, D.: Enhancing Constraint Models for Planning Problems. In: Proceedings of FLAIRS 2009. AAAI Press, Menlo Park (2009)

A Fast Algorithm to Solve the Frequency Assignment Problem

Mohammad Dib[1], Alexandre Caminada[1], and Hakim Mabed[2]

[1] UTBM, SET Lab, 90010 Belfort Cedex, France
{mohammad.dib,alexandre.caminada}@utbm.fr
[2] UFC, LIFC Lab, Numérica, 25200 Montbéliard, France
hakim.mabed@pu-pm.univ-fcomte.fr

Abstract. The problem considered in this paper consists in defining an assignment of frequencies to radio link between transmitters which minimize the number of frequency used to solve a CSP very well referenced as ROADEF'01 challenge. This problem is NP-hard and few results have been reported on techniques for solving it optimally. We applied to this version of the frequency assignment problem an original hybrid method which combines constraint propagation and Tabu search. Two Tabu lists are used to filter the search space and to avoid algorithm cycles. Computational results, obtained on a number of standard problem instances, show the efficiency of the proposed approach.

1 Tabu Search and Constraints Propagation Method

We propose here a constraint propagation algorithm using a dynamic backtrack process based on the concepts of *nogood* and Tabu list [1]. Search starts with a consistent partial configuration otherwise an empty solution. At each iteration the algorithm tries to extend the current partial solution in such manner to maintain the consistency. The instantiation of a new variable implies adding a new constraint, called decision constraint and noted $x_i = d_i$. When a decision constraint proves to be incompatible with the union of problem constraints C and the set of decision constraints, the algorithm reacts by repairing the incompatible decisions and cancels some instantiations already done. Constraints propagation procedure consists in filtering domains of the not yet affected variables in order to detect in advance the blocking situations called *deadend*. A *deadend* is detected when the domain of a given variable becomes empty. When a *deadend* is detected, the set of the incompatible decisions leading to the raised inconsistency is marked. These incompatible decisions are what we called a *nogood* [2]. When a *deadend* is reached, one of the decisions taking part in the *nogood* is cancelled. In our approach the storage of *nogoods* in a permanent Tabu list makes it possible to prevent the re-exploration of the blocking branches in the decision tree. It brings a lot of efficiency in the algorithm search. After *nogood* cancellation, the current partial solution is

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 316 – 318, 2009.

extended with the Dom/Deg heuristic to choose a variable. The choice of a value for the chosen variable must obey to two rules. Firstly, the partial solution resulting from the extension procedure should not contain any stored *nogood* in permanent Tabu list. Secondly the decision to be added should not be in a temporary Tabu list at the current iteration that is used to avoid short term cycles. The algorithm principle is described below.

```
Begin
1.  iter = 0;
2.  repeat
3.  ConstraintPropagation();      /*Propagation*/
4.  if isDeadEnd()
5.                  repeat
6.                         Nogood =getNogood();
7.                         AddNogoodInPermanentTabuList(Nogood);
8.                         CancelADecision ();
9.                         AddCancelledDecisionInTemporaryTabuList();
10.                        ConstraintPropagation();
11.                 until none isDeadEnd()
12. end if
13. iter = iter+1;
14. ExtendSolution();      /*extend the solution*/
15. until FindSolution() or iter >= iterMax
```

2 Tests and Results

Several methods are proposed in the literature to solve FAP. We used two test sets noted CELAR and GRAPH. Table 1 shows a comparison between the results of our method with those of the CN-Tabu [3]. CN-Tabu has proven its efficiency in the frame of ROADEF'01 challenge by obtaining the first rank among several concurrent teams. It was applied on the instances presented in Table 1 with Min-Span objective (minimizing the maximal frequency in the spectrum used by the solution). For each of these 12 instances, the table plots its name, the number of variables, the number of constraints to satisfy, the best result (the highest frequency used) obtained by CN-Tabu, the CPU time in seconds (same case for CN-Tabu and our method: P-IV, 3 GHz, 1GB RAM) and also the percentage of success rate (optimal solutions obtained on 20 executions). The character "-" is for absence of information. The results show the competitiveness of our hybrid method. We also did a performance comparison with ALS [4] which confirmed the good results (to be published soon).

Scenario	Variables	Constraints	[3]	Time	SR	Our results	Time	SR
SCEN 01	916	5548	680	1	-	680	1	100%
SCEN 02	200	1235	394	1	-	394	1	100%
SCEN 03	400	2760	666	1	-	**652**	**1**	**100%**
SCEN 05	400	2598	792	1	100%	792	1	100%
GRAPH 01	200	1134	408	1	-	408	7	100%
GRAPH 02	400	2245	394	1	-	394	1	100%
GRAPH 03	200	1134	380	1	100%	380	1	100%
GRAPH 04	400	2244	394	1	40%	**394**	1	**100%**
GRAPH 08	680	3757	652	15	**-**	652	**11**	**100%**
GRAPH 09	916	5246	666	664	**-**	666	**435**	**100%**
GRAPH 10	680	3907	394	17	25%	394	**10**	**100%**
GRAPH 14	916	4638	352	30	-	352	**22**	**100%**

References

[1] Dib, M., Caminada, A., Mabed, H.: Constraint Propagation with Tabu List for Min-Span Frequency Assignment Problem. In: Le Thi, H.A., Bouvry, P., Pham Dinh, T. (eds.) MCO 2008. CCIS, vol. 14, pp. 97–106. Springer, Heidelberg (2008)
[2] Jussien, N., Lhomme, O.: Local search with constraint propagation and conflict-based heuristics. Artificial Intelligence 139(1), 21–45 (2002)
[3] Dupont, A.: Étude d'une méta-heuristique hybride pour l'affectation de fréquences dans les réseaux tactiques évolutifs. PhD thesis, Université Montpellier II (October 2005)
[4] Devarenne, I., Mabed, H., Caminada, A.: Analysis of Adaptive Local Search for the Graph Coloring Problem. In: 6th Metaheuristic International Conference, MIC (2005)

A Hybrid LS/CP Approach to Solve the Weekly Log-Truck Scheduling Problem

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

We present a hybrid approach to solve the log-truck scheduling problem (LTSP), which combines routing and scheduling of forest vehicles and includes aspects such as multiple products and inventory management. The LTSP is closely related to some routing problems encountered in other industries, in particular, so called "pick-up and delivery problems". In general, the LTSP are more complex than classical PDP, since in the LTSP we must synchronise trucks and log-loaders to avoid as much as possible waiting times.

Several models and methods have been developped in the litterature to solve the LTSP. In Chile [4] proposed a heuristic-based model (ASICAM) to a simplified variant of the LTSP that does not consider log-loader waiting times, and produce only daily plan for trucks. [3] developped a hybrid CP / IP method, where the IP model generates optimal routes in term of deadhead, while CP deals with the scheduling part. The main two contributions of this paper are, first, to present a weekly version of the LTSP in which inventories have to be accounted for, and second, to propose a more efficient hybrid approach to this problem. A major interest for solving the weekly version of the LTSP lies in the fact that it allows to operate in a just-in-time mode, where saw mills receive what they need for production on a daily basis.

We proposed a two-phase method, in the first phase, we solve an IP formulation of a tactical problem, where we handle demand and stock constraints (per product and per day) at each woodmill. In order to ensure loading feasibility, we introduce daily capacity constraints at each supply point. At this level, restritions are on time availibity of trucks, the objective function is to minimize the cost of opening sites and the travel cost of full truckloads. This model can either be solved by an efficient MIP solver or through Local Search. We choose to use a tabu search algorithm by using the LS module of Comet1.1. In this case the search starts by supposing that all forest areas are opened all the week. Once the initial solution is generated, the local search is performed based on two neighborhoods structures. One which tries to close a single site, and one which flips the status of two sites having initially different status.

This first phase yields seven daily scheduling problems (from monday to sunday), in which, routing and scheduling decisions are integrated into a hybrid LS / CP method. This method is based on a local search approach operating on

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 319–320, 2009.

the trucks routes and two different scheduling models. The first one based on constraint-based local search (CBLS) and the programming language Comet1.1 (LS module). It takes advantage of invariants (one-way constraints) and a rich constraint language (see [1] and [2]). The second one is the constraint programming model used by [3]. Each time trucks routes are modified by the routing operator, the scheduling model is solved by a greedy algorithm (CBLS), and occasionnaly by the CP Comet solver. The CP search strategy consists on assigning times to all unloading activities (by using the SetTimes function) and generating values to all truck waiting time variables.

The algorithm structure of the second phase is organised as follow; at the beginning the search is focused only on the CBLS part. Once the improvement becomes less frequent, a hybridization between the CBLS and CP models is realized by imposing the best solution of the CBLS part as an upper bound in the CP model. The CP model is called every time, when the CBLS operator is unable to improve the best solution found so far, or when a promising solution with regards to routing decision yields cannot be efficiently scheduled by CBLS.

Experiments performed on real data provided by a large canadian timber company show a considerable gain comparing to previous work ([3]) both in term of problem size (weekly problem instead of daily problem) and a solution speed (minutes instead of hours). It also demonstrate that both CBLS and CP are essentiel ingredients to achieve good results. Research directions consist on developping new communication methods between hybrid CBLS and CP models, and how to controle when it is benefic to perform this communication.

References

1. Van Hentenryck, P., Michel, L.: Constraint Based Local Search. MIT Press, Cambridge (2001)
2. Van Hentenryck, P., Michel, L.: Differentiable invariants. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 604–619. Springer, Heidelberg (2006)
3. El Hachemi, N., Gendreau, M., Rousseau, L.-M.: Solving a log-ruck scheduling problem with constraint programming. In: Perron, L., Trick, M.A. (eds.) CPAIOR 2008. LNCS, vol. 5015, pp. 293–297. Springer, Heidelberg (2008)
4. 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)

Modelling Search Strategies in Rules2CP

François Fages and Julien Martin

INRIA Rocquencourt, BP105, 78153 Le Chesnay Cedex, France
http://contraintes.inria.fr

In this abstract, we present a rule-based modelling language for constraint programming, called Rules2CP [1], and a library PKML for modelling packing problems. Unlike other modelling languages, Rules2CP adopts a single knowledge representation paradigm based on logical rules without recursion, and a restricted set of data structures based on records and enumerated lists given with iterators. We show that this is sufficient to model constraint satisfaction problems together with search strategies, where *search trees* are expressed by *logical formulae*, and *heuristic choice criteria* are defined by *preference orderings on variables and formulae*. Rules2CP statements are compiled to constraint programs over finite domains (currently SICStus-prolog and soon Choco-Java) by term rewriting and partial evaluation.

The Packing Knowledge Modelling Language (PKML) is a Rules2CP library developed in the European project Net-WMS for dealing with real size non-pure bin packing problems in logistics and automotive industry. PKML refers to shapes in $\mathbb{Z}^K$. A *point* in this space is represented by the list of its K integer coordinates. A *shape* is a *rigid assembly of boxes*, represented by a record. A *box* is an orthotope in $\mathbb{Z}^K$. An *object* , such as a bin or an item, is a record containing one attribute `shapes` for the list of its *alternative shapes*, one *origin* point, and some optional attributes such as weight, virtual reality representations or others. The alternative shapes of an object may be the discrete rotations of a basic shape, or different object shapes in a configuration problem. The end in one dimension and the volume of an object with alternative shapes are defined with reified constraints.

PKML uses Allen's interval relations in one dimension, and the topological relations of the Region Connection Calculus in higher-dimensions, to express placement constraints. Pure *bin packing problems* can be defined as follows:

```
non_overlapping(Items,Dims) --> forall(O1,Item, forall(O2,Items,
        uid(O1)<uid(O2) implies notoverlap(O1,O2,Dims))).
containmentAE(Items,Bins,Dims) -->
        forall(I,Items, exists(B,Bins, contains_rcc(B,I,Dims))).
bin_packing(Items,Bins,Dims) --> containmentAE(Items,Bins,Dims)
        and non_overlapping(Items,Dims) and labeling(Items).
```

Other rules about weights, stability, as well as specific packing business rules can be expressed, e.g.

```
gravity(Items) --> forall(O1,Items, origin(O1,3)=0
        or exists(O2,Items, uid(O1)\#uid(O2) and on\_top(O1,O2))).
```

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 321–322, 2009.

The search is specified here with a simple labeling of the (coordinate) variables of the items. Heuristics can be defined as preference orderings criteria, using the expressive power of the language by pattern matching. For instance, the statement `variable_ordering([greatest(volume(^ )), is(z(^ ))])` expresses the choice of the variables belonging to the objects of greatest volume first, and among them, the z coordinate first.

On Korf's benchmarks of optimal rectangle packing problems [2] (i.e. finding the smallest rectangle containing n squares of sizes $S_i = i$ for $1 \leq i \leq n$), compared to the SICStus Prolog program of Simonis and O'Sullivan [3] which improved best known runtimes up to a factor of 300, the SICStus Prolog program generated by the Rules2CP compiler explores exactly the same search space, and is slower by a factor less than 3 due to the interpretation overhead for the dynamic search predicates. The following table shows the compilation and running times in seconds:

N	R2CP compilation	Rules2CP	Original
18	0.266	13	6
20	0.320	20	10
22	0.369	364	197
24	0.443	5230	1847
25	0.509	52909	17807

Rules2CP differs from OPL, Zinc and Essence modelling languages in several aspects among which: the use of logical rules, the absence of recursion, the restriction to simple data structures of records and enumerated lists, the specification of search by logical formulae, the specification of heuristics as preference orderings, and the absence of program annotations. The expression of complex search strategies and heuristics is currently not expressible in Zinc and Essence, and can be achieved in OPL in a less declarative manner by programming. On the other hand, we have not considered the compilation of Rules2CP to other solvers such as local search, or mixed integer linear programs, as has been done for OPL and Zinc systems.

As for future work, a large subset of PKML rules has been shown in [4] to be compilable with indexicals *within* the geometrical kernel of the global constraint `geost` providing better performance than by reification. The generality of this approach will be explored with greater generality for Rules2CP. The specification of search strategies in Rules2CP will also be explored more systematically, possibly with adaptive strategies in which the dynamic criteria depend on execution profiling criteria.

References

1. Fages, F., Martin, J.: From rules to constraint programs with the Rules2CP modeling language. In: Proc. 13th International Workshop on Constraint Solving and Constraint Programming CSCLP 2008. Recent Advances in Constraints, Roma, Italy. LNCS (LNAI). Springer, Heidelberg (to appear) (2008)
2. Korf, R.E.: Optimal rectangle packing: New results. In: ICAPS, pp. 142–149 (2004)
3. Simonis, H., O'Sullivan, B.: Using global constraints for rectangle packing. In: Proceedings of the first Workshop on Bin Packing and Placement Constraints BPPC 2008, associated to CPAIOR 2008 (2008)
4. Carlsson, M., Beldiceanu, N., Martin, J.: A geometric constraint over k-dimensional objects and shapes subject to business rules. In: Stuckey, P.J. (ed.) CP 2008. LNCS, vol. 5202, pp. 220–234. Springer, Heidelberg (2008)

CP-INSIDE: Embedding Constraint-Based Decision Engines in Business Applications

Jacob Feldman, Eugene Freuder, and James Little*

Cork Constraint Computation Centre, Department of Computer Science
University College Cork
Cork, Ireland
{j.feldman,e.freuder,j.little}@4c.ucc.ie

The CP-INSIDE project seeks to make constraint programming technology more accessible. In particular, it aims to facilitate the embedding of CP technology in business applications, and its integration with familiar software tools. A major objective of the CP-INSIDE project is to allow business developers to create constraint-based decision engines utilizing the power of CP technology in a vendor-neutral way. CP-INSIDE simplifies the development of specialized decision engines making them independent of the underlying generic CP solvers. There have already been several attempts to unify constraint programming languages (for instance, [1,2,3]) that directly or indirectly contributed to the same objective. CP-INSIDE is based on the following principles:

- Use main-stream programming languages like Java giving business application developers a simple CP API with no new languages to learn and with an easy access to already existing business objects and packages.
- Provide pre-built interfaces to commonly used software products such as MS Excel, Google Apps, and different rule engines. Application developers should be able to use familiar languages, tools, and a learn-by-example technique instead of becoming CP gurus.
- Not to develop another CP solver, but rather externalize commonly used features from the most popular CP solvers and provide a unified way to build adapters for different, already existing solvers.
- Allow CP researches to implement new algorithms/constraints in a unified way, test them with different solvers, and make them commonly available.

While we considered both Java and C# for CP-INSIDE, the initial release has been implemented in Java as a three-tier framework presented on Figure 1. The first tier "Business Interfaces" uses the second tier "Common CP API" to provide pre-built unified interfaces between different CP Solvers and popular tools such as MS Excel, Google Calendar, Facebook, MATLAB, or business rules management systems such as OpenRules. It also provides examples of how to build web interfaces for CP-based engines and to deploy them as web services. The Common CP API provides a Java

* This material is based upon works supported by Enterprise Ireland under Grant CFTD/06/209 and by the Science Foundation Ireland under Grant No. 05/IN/I886.

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 323–324, 2009.

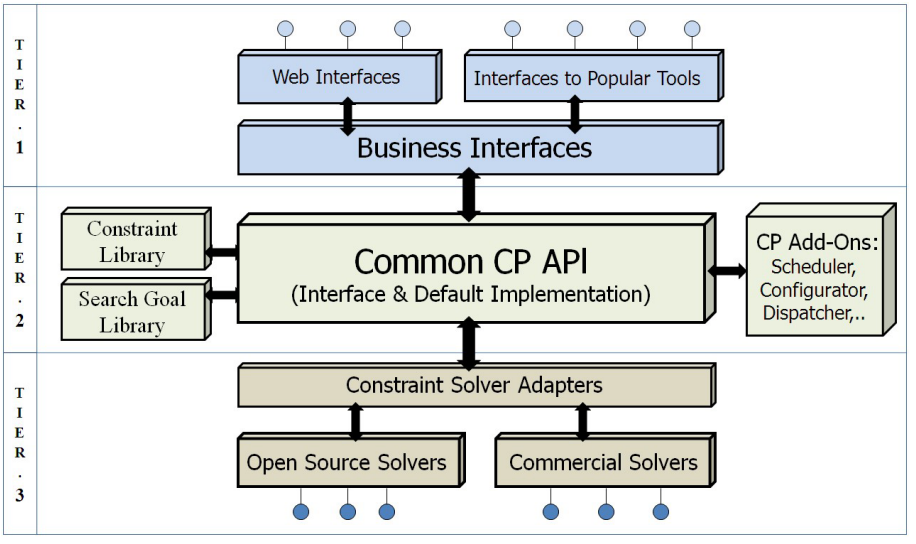


Fig. 1. CP-INSIDE Architecture

interface for major CP concepts and methods that can be utilized by business application developers to define and solve their own constraint satisfaction problems.

Along with an interface, the Common CP API includes default implementations for major constraints and search goals that do not depend on a specific underlying CP solver. However, the API also allows a user to utilize the power of a specific solver by providing an access to the native implementation. With the CP API a user may define new custom constraints and search goals without the necessity to go down to the CP solver level. CP-INSIDE includes a Scheduler built on top of the CP API, thus providing scheduling concepts, constraints, and goals even for solvers that do not support them. Similarly, we plan to build add-ons for other business verticals. The third tier provides a set of adapters for popular existing Java-based CP solvers. The initial set includes adapters for commercial ILOG JSolver (`www.ilog.com`) and open source CP solvers Choco (`www.choco.sourceforge.net`), Constrainer (`www.constrainer.sourceforge.net`). These adapters can be used as examples for integrating other Java-based CP solvers.

References

1. Van Hentenryck, P.: The OPL Optimization Programming Language. MIT Press, Cambridge (1999)
2. Schlenker, H., Ringwelski, G.: POOC - a platform for object-oriented constraint programming. In: O'Sullivan, B. (ed.) CologNet 2002. LNCS, vol. 2627, pp. 159–170. Springer, Heidelberg (2003)
3. Nethercote, N., Stuckey, P., Becket, R., Brand, S., Duck, D., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: Bessière, C. (ed.) CP 2007. LNCS, vol. 4741, pp. 529–543. Springer, Heidelberg (2007)

An Integrated Genetic Algorithm and Integer Programming Approach to the Network Design Problem with Relays

Abdullah Konak[1] and Sadan Kulturel-Konak[2]

[1] Penn State Berks, Information Sciences and Technology,
Tulpehocken Road P.O. Box 7009 Reading, PA 19610
konak@psu.edu
[2] Penn State Berks, Management Information Systems,
Tulpehocken Road P.O. Box 7009 Reading, PA 19610
sadan@psu.edu

Extended Abstract

The network design problem with relays (NDPR) arises in many real-life telecommunication and supply chain networks. Similar to the multi-commodity network design problem, a set of commodities is given, and each commodity k is to be routed through single path from the source node $s(k)$ to the sink node $t(k)$. However, an upper bound λ is imposed on the distance that a commodity k can travel on the path from the source node $s(k)$ to the sink node $t(k)$ without visiting special nodes, called relays. For example, in digital telecommunication networks, relays represent points where attenuated communication signals are regenerated. A fixed cost of f_i is incurred when a node i is dedicated as a relay, and each edge (i, j) has an installation cost of $c_{i,j}$ and a length of $d_{i,j}$. The NDPR is defined as selecting a set of edges from a given set of candidate edges and determining relay nodes to minimize the network design cost while making sure that each commodity k is routed through a single path on which the distances between the node $s(k)$ and the first relay, between any consecutive relays, and between the last relay and the node $t(k)$ are less than the upper bound λ.

This research first presents a new path-based formulation of the NDPR. Based on this formulation, it is shown that for a given set of commodity paths, the optimal location of relays can be determined by solving a set covering problem. Then, this approach is used in a genetic algorithm (GA) to solve the NDPR in two phases. Using a specialized crossover-mutation operator, the GA searches for a path for each commodity, and the relay locations are determined by solving the corresponding set covering problem. The proposed approach is extensively tested on several problems. Comparisons with the existing exact and heuristic approaches show that the proposed GA is very effective. Further research may focus on how to extend the proposed approach to solve the two-edge disjoint NDPR or the capacitated version of the problem.

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, p. 325, 2009.

A Benders' Approach to a Transportation Network Design Problem

Benjamin Peterson and Michael A. Trick

Tepper School of Business, Carnegie Mellon University, Pittsburgh PA 15213, USA

Benders' decomposition is an iterative technique for mathematical programming in which a problem is decomposed into two parts: a master problem which solves for only a subset of the variables, and a subproblem which solves for the remaining variables while treating the master variables as fixed. The technique uses information from the solution of the subproblem to design a constraint that is unsatisfied by the current master solution but necessary for the optimal solution of the full problem. The master problem and subproblem are solved iteratively as new constraints are added to the master problem until no such constraint exists and the last master solution therefore optimizes the full problem. In traditional Benders' decomposition [1], the subproblem is a linear program, and the Benders' constraint is based on the dual solution to that subproblem. More recently, logic-based [3] or combinatorial Benders' [2] have been developed that generate Benders constraints based on infeasibility or suboptimality-based constraints. By not requiring a linear programming subproblem, Benders approaches provide an excellent approach to integrating disparate optimization approaches.

We apply a form of this technique to a mixed-integer network optimization problem, in which a set of deliveries are required to move between pairs of points in a network, and a set of feasible routes are given to carry the deliveries from which a minimum-cost feasible set must be found. This model appeared in [4] in the context of choosing transportation contracts by the United States Postal Service. A contract corresponds to a particular transportation route at a particular time. Packages can be assigned to chosen routes that meet the service time requirements of the package.

Full Problem:

$$\begin{aligned} \min\ & \textstyle\sum_{r\in\mathcal{R}} C_r y_r \\ \text{s.t.}\ & \textstyle\sum_{r\in in\mathcal{R}} A_{rd} x_{rd} \geq 1, \forall d \in \mathcal{D} \\ & \textstyle\sum_{d\in\mathcal{D}} A_{rd} V_d x_{rd} \leq S_r y_r, \forall r \in \mathcal{R} \\ & y_r \in \{0,1\}, \forall r \in \mathcal{R} \\ & x_{rd} \geq 0, \forall d \in \mathcal{D}, r \in \mathcal{R} \end{aligned}$$

Variables:

$y_r = 1$ if route r is used, 0 otherwise

x_{rd} = proportion of delivery d carried by route r

Parameters:

$\mathcal{R}$ = set of routes

$\mathcal{D}$ = set of deliveries

C_r = cost of route r

S_r = capacity of route r

V_d = volume of delivery d

$A_{rd} = 1$ if route r can carry delivery d, 0 otherwise

We decompose the problem by assigning the route variables y to the master problem and the assignment variables x to the subproblem as follows. The goal

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 326–327, 2009.

is to find an optimal solution to the master problem that is also feasible in the subproblem using relatively few of the iteratively-generated constraints.

Master Problem:

$$\begin{aligned}\min\ & \textstyle\sum_{r\in\mathcal{R}} C_r y_r \\ \text{s.t. } & \textstyle\sum_{r\in\mathcal{R}} a_r^i y_r \geq b^i, \forall i \in I \\ & y_r \in \{0,1\}, \forall r \in \mathcal{R}\end{aligned}$$

where I is a set of iteratively added constraints

Subproblem:

$$\begin{aligned}\min\ & 0 \\ \text{s.t. } & \textstyle\sum_{r\in\mathcal{R}} A_{rd} x_{rd} \geq 1, \forall d \in \mathcal{D} \\ & \textstyle\sum_{d\in\mathcal{D}} A_{rd} V_d x_{rd} \leq S_r \bar{y}_r, \forall r \in \mathcal{R} \\ & x_{rd} \geq 0, \forall d \in \mathcal{D}, r \in \mathcal{R}\end{aligned}$$

where $\bar{y}$ is that last computed solution of the master problem.

The master problem begins with only its binary constraints and I empty, thus it returns the solution $\bar{y} = 0$. Each master solution $\bar{y}$ is then checked for feasibility in the subproblem. If infeasible, a cut $\sum_r \in \mathcal{R} a_r^i y_r \geq b^i$ is constructed and added to the master problem to be resolved. The cuts we construct have a natural interpretation. If we have some subset Z of the deliveries D, the total capacity of all routes used in the master solution $\bar{y}$ that are capable of carrying a delivery in Z must be at least the total volume of the deliveries in Z. The necessity of this condition is fairly obvious, but we also prove that the set of all such constraints for all possible subsets $Z \subseteq D$ is sufficient to describe the entire feasible region of the subproblem. That is, all solutions satisfying these constraints satisfy the constraints of the subproblem and vice versa. We can find sets Z with insufficient capacity relative to $\bar{y}$ by solving a relatively small mixed integer program.

It is our goal to iteratively generate a relatively small set of these constraints as quickly as possible such that the optimal solution to the master problem is also feasible in the subproblem. We found that if we initially add the constraint constructed from the full set $Z^0 = \mathcal{D}$ of deliveries, and then add a constraint in each iteration constructed from the delivery set Z^i of minimum cardinality such that cuts off the current master solution, then we arrive a full problem optimal solution quickly.

In our computational experiments, we discover that the most recent release of CPLEX, version 11.0, is able to solve the full network problem much faster than previous releases, suggesting that the latest version is very adept at finding good cuts for the full MIP on its own. However, in the current implementation of our algorithm using the Benders' approach, we are able to arrive at an optimal solution in about half the time required by CPLEX for most of our test instances.

References

1. Benders, J.F.: Partitioning Procedures for Solving Mixed Variables Programming Problems. Numerische Mathematik 4, 238–252 (1962)
2. Codato, G., Fischetti, M.: Combinatorial Benders' Cuts for Mixed-Integer Linear Programming. Operations Research 54(4), 756–766 (2006)
3. Hooker, J.N.: Integrated Methods for Optimization. Springer, Heidelberg (2006)
4. Pajunas, A., Matto, E.J., Trick, M.A., Zuluaga, L.F.: Optimizing Highway Transportation at the United States Postal Service. Interfaces 37(6), 515–525 (2007)

Progress on the Progressive Party Problem

Helmut Simonis*

Cork Constraint Computation Centre
University College Cork
h.simonis@4c.ucc.ie

We report new results for solving the progressive party problem with finite domain constraints, which are competitive with the best local search results in the literature. When introduced in [1], the progressive party problem was a show case for finite domain constraint programming, since a solution of the original problem for 6 time periods could be found in 26 minutes with Ilog Solver, while an integer programming model was not successful. Improved results using finite domains were reported in [2]. Since then, alternative solutions using MILP have been proposed, while local search methods have been significantly faster, as well as more stable for given problem variations. The best results (to my knowledge) are from [3].

Details of the problem can be found on its CSPLIB page (`http://www.csplib.org`). The problem consists of assigning guest teams of different sizes to host boats (different capacities) for multiple team periods. Each guest can visit a host atmost once, and two guest teams should also meet atmost once. We look for feasible solutions for different numbers of time periods and/or host/guest sizes.

We have explored two models for the problem. The first considers the complete problem by posting all variables for all time periods together. It creates `alldifferent` constraints for all variables of each guest team, expressing that a guest team can visit a host atmost once. For each time period we state a `bin-packing` constraint [4] to handle the capacity constraints for the host boats. A collection of reified equality constraints between pairs of guests handle the condition that two guests meet atmost once.

When analysing the behaviour of the first model, it was easy to see that there was very limited interaction between time periods, and no effective constraint propagation for one level until variables of that level were assigned. This suggested a decomposition model (already used in [1]) which sets up the problem for one time period at a time. If guests are assigned to the same host in a time period, then disequality constraints are imposed between them for future time periods. Each layer adds new disequality constraints, so that the problem for each following layer becomes more constrained. These disequalities can be expressed with a global `some-different` [5] constraint, although we use the binary decomposition. The model then consists of a single `bin-packing` constraint and a `some-different` constraint expressing all disequalities between the variables for a time period. Surprisingly, we do not loose propagation with this decomposed model compared to the first model, while speeding up execution significantly.

The key to solving this problem is a customized search routine, which combines five elements. We to solve the problem period by period, i.e. assigning all variables

* This work is supported by Science Foundation Ireland (Grant Number 05/IN/I886) and by Cisco Systems Inc. through the Cisco Collaborative Research Initiative.

W.-J. van Hoeve and J.N. Hooker (Eds.): CPAIOR 2009, LNCS 5547, pp. 328–329, 2009.

for one period before starting on the next. This provides a more stable method than using `first_fail` over all variables. We use a `first_fail` variable selection inside each time period. For each time period we use of a partial search strategy (credit-based search [2]) to avoid deep backtracking in dead parts of the search tree. A randomized value selection leads to a more equitable choice of values. Once this randomization element is in place, we can also use a restart strategy which cuts off search after all credit in some layer has been expended, and begins exploration from the top again.

We compare results obtained with our second model in ECLiPSe 6.0 with the results published in [3]. Except for a single problem instance (4/9), results are comparable. A full version of the paper can be found at `http://4c.ucc.ie/~hsimonis/party.pdf`

Table 1. Results on Problems from [3]

		ECLiPSe 6.0					Comet				
Problem	Size	Solved	Min	Max	Avg	σ	Solved	Min	Max	Avg	σ
1	6	100	0.187	0.343	0.226	0.027	100	0.33	0.38	0.35	0.01
1	7	100	0.218	0.515	0.271	0.044	100	0.39	0.49	0.44	0.02
1	8	100	0.250	2.469	0.382	0.258	100	0.50	0.72	0.57	0.04
1	9	100	0.266	9.906	1.253	1.734	100	0.74	1.46	1.01	0.15
1	10	100	0.375	136.828	23.314	21.738	100	1.47	41.72	4.68	5.80
2	6	100	0.218	2.375	0.624	0.484	100	0.37	0.52	0.43	0.04
2	7	100	0.266	3.453	1.117	0.873	100	0.47	1.64	0.73	0.18
2	8	100	0.297	15.421	2.348	2.263	100	0.75	7.16	2.69	1.26
2	9	100	0.469	107.187	20.719	21.162	99	4.41	162.96	33.54	34.10
3	6	100	0.219	3.266	0.551	0.504	100	0.37	0.56	0.43	0.04
3	7	100	0.250	3.734	0.889	0.705	100	0.49	1.45	0.74	0.18
3	8	100	0.296	21.360	2.005	2.417	100	0.84	11.64	2.85	1.55
3	9	100	1.078	173.266	34.774	32.731	96	4.41	164.44	40.10	40.14
4	6	100	0.219	9.922	2.443	2.146	100	0.39	0.72	0.47	0.05
4	7	100	0.360	25.297	3.531	3.421	100	0.55	2.33	0.87	0.26
4	8	100	0.438	53.547	8.848	9.193	100	1.23	11.38	3.68	1.91
4	9	63	3.062	494.109	206.324	161.250	94	8.35	166.90	59.55	44.44
5	6	100	0.203	7.922	1.498	1.441	100	0.53	5.29	1.67	0.75
5	7	100	0.266	28.000	5.889	5.463	100	1.77	132.82	29.72	28.76
6	6	100	0.219	15.219	2.147	2.661	100	0.58	31.84	2.74	3.31
6	7	100	0.407	64.312	11.328	12.290	88	3.24	152.37	56.92	48.91

References

1. Brailsford, S., Hubbard, P., Smith, B., Williams, H.: Organizing a social event - a difficult problem of combinatorial optimization. Computers Ops. Res. 23, 845–856 (1996)
2. Beldiceanu, N., Bourreau, E., Chan, P., Rivreau, D.: Partial search strategy in CHIP. In: 2nd International Conference on Metaheuristics MIC 1997, Sophia Antipolis, France (1997)
3. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. MIT Press, Boston (2005)
4. Shaw, P.: A constraint for bin packing. In: Wallace, M. (ed.) CP 2004. LNCS, vol. 3258, pp. 648–662. Springer, Heidelberg (2004)
5. 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)

Author Index