Z. Kiziltan and B. Hnich (Eds.)

Proceedings of the CP 2006 Doctoral Programme

The Doctoral Programme of the Twelfth International Conference on Principles and Practice of Constraint Programming, CP 2006, Nantes, France, September 24–29, 2006.

Preface

The Doctoral Programme is a forum held during the Constraint Programming conference which provides an opportunity for a group of Ph.D. students to achieve visibility and discuss their research interests and career objectives with each other and established researchers in Constraint Programming and its related fields. After successful Doctoral Programmes in previous years, it is being run again this year for the sixth time on September 24, 2006 in Nantes, France.

The aims of the Doctoral Programme are the following:

- to provide a forum for Ph.D. students to present their current research, and receive feedback from other students and senior researchers;
- to promote contacts among Ph.D. students and senior researchers working in the same area;
- to exchange research experience;
- to support Ph.D. students with information and advice on academic, research and industrial careers;
- and to financially support its participants.

This year, 38 students are participating to the Doctoral Programme 30 of which also receive complimentary conference registration and accommodation.

The programme consist of presentations of the students who do not have any paper/poster accepted at the main technical programme, and tutorials given by senior researchers in the field. In addition, each student is matched to a mentor who is a senior researcher with similar research interests and who can advise the student on his/her research progress. Finally, a doctoral dinner is organised to bring together the students in an informal gathering.

The editors would like to take the opportunity and thank all the authors who submitted a paper to this volume, the members of the organizing committee, the CP 2006 conference chairs Frédéric Benhamou and Narendra Jussien for their help in the organization of the Doctoral Programme, as well as our generous sponsors for their financial support.

We hope that the present volume is useful for anyone interested in the current PhD topics and new trends in Constraint Programming and its related fields.

July 2006 Z. Kiziltan and B. Hnich

Organization

Doctoral Programme Chairs

Zeynep Kiziltan	D.E.I.S. Universita di Bologna, Italy
Brahim Hnich	Faculty of Computer Science, Izmir University of Economics, Turkey

Organizing Committee

Roman Barták	Charles University, Czech Republic
Hadrien Cambazard	Université de Nantes, France
Mats Carlsson	SICS, Sweden
Ian Miguel	University of St. Andrews, U.K.
Javier Larrosa	Universitat Politecnica de Catalunya, Spain
Gilles Pesant	École Polytechnique de Montréal, Canada
Steven Prestwich	University College Cork, Ireland
Francesca Rossi	Universitá di Padova, Italy
Meinolf Sellmann	Brown University, U.S.A.
Peter van Beek	University of Waterloo, Canada
Pascal van Hentenryck	Brown University, U.S.A.
Toby Walsh	University of New South Wales, Australia

Sponsoring Institutions

ACP (Association of Constraint Programming)
Nantes Métropole
Ilog
Ecole des Mines de Nantes
Région Pays de Loire
LINA (Laboratoire d'Informatique de Nantes-Atlantique)
Université de Nantes
Conseil Général de Loire Atlantique
IISI
4C (Cork Constraint Computation Centre)
Association Française de Programmation par Contraintes

Table of Contents

Resolution-based rules for the Weighted Max-SAT problem

Student: Federico Heras
Advisor: Javier Larrosa
fheras@lsi.upc.edu, larrosa@lsi.upc.es

UPC

Abstract. In this paper we present a set of useful inference rules for solving the weighted Max-SAT problem. They are based on the resolution rule for Max-SAT described in [9]. All of them are special cases of Max-SAT resolution that allows a DPLL-based solver to compute better lower bounds and to prune more values. We study the relation of our work with previous approaches and finally we perform an empirical comparison. Our results on several domains show that the resulting algorithm can be orders of magnitude faster than state-of-the-art Max-SAT and Weighted CSP solvers.

1 Introduction

Max-SAT is an optimization version of the SAT problem and it is known that many important problems can be naturally expressed as Max-SAT such as Maximum cut, maximum clique, combinatorial auctions, bayesian networks, etc. In recent years, there has been a considerable effort in finding efficient solving techniques [6, 12, 15, 11]. In all these works the core algorithm is a simple depth-first branch-and-bound and their contributions are good quality lower bounds. In [6] the Max-SAT problem, modelled as a WCSP ([8, 10]), is transformed and simplified at every node of the search tree. The other approaches compute a lower bound by counting inconsistencies between clauses without transforming the problem. In [9, 7], the WCSP framework is presented as a logical framework for Max-SAT. In previous works, the authors described the lower bounds in a procedural way. Now, the lower bounds and transformations can be explained with transformation rules. They are easier to understand than procedures. In this paper we present a set of inference rules that have been shown to be effective empirically and we study their relation to previous works. In particular, we introduce three new inference rules, all of them expressable as particular cases of resolution or hyper-resolution. We provide experimental results in different domains. The experiments indicate that our algorithm is orders of magnitude faster than any competitor.

2 Preliminaries

In the sequel $X = \{x, y, z, ...\}$ is a set of boolean variables taking values over the set $\{\mathbf{t}, \mathbf{f}\}$, which stands for *true* and *false*, respectively. A *literal* is either a variable (e.g. x) or its negation (e.g. $\bar{x}$). We will use $l, h, q, ...$ to denote literals and $var(l)$ to denote the variable related to l (namely, $var(x) = var(\bar{x}) = x$). If variable x is instantiated to $\mathbf{t}$, literal x is satisfied and literal $\bar{x}$ is falsified. Similarly, if variable x is instantiated to $\mathbf{f}$, literal $\bar{x}$ is satisfied and literal x is falsified. An *assignment* is an instantiation of a subset of X. The assignment is *complete* if it instantiates all the variables (otherwise it is partial). A *clause* $C = l_1 \vee l_2 \vee \ldots \vee l_k$ is a disjunction of literals such that $\forall_{1 \leq i,j \leq k,\ i \neq j} \quad var(l_i) \neq var(l_j)$. The size of a clause, noted $|C|$, is the number of literals that it has. $var(C)$ is the set of variables that appear in C (namely, $var(C) = \{var(l) | l \in C\}$). An assignment satisfies a clause iff it satisfies one or more of its literals. A formula in *conjunctive normal form* (CNF) is a conjuction of different clauses, normally expressed as a set. A satisfying complete assignment is called a *model*. Given a CNF formula, the SAT problem consists in determining whether there is any model for it or not.

The *empty clause*, noted $\Box$, cannot be satisfied. Consequently, when a formula contains $\Box$ it does not have any model and we say that it contains an *explicit contradiction*. Sometimes it is convenient to think of clause C as its equivalent $C \vee \Box$.

A *weighted clause* is a pair (C, w) such that C is a classical clause and w is the cost of its falsification. In this paper we assume costs being natural numbers. A *weighted* formula in conjunctive normal form (CNF) is a set of weighted clauses. The cost of an assignment is the sum of weights of all the clauses that it falsifies.

Consider a weighted clause $(A \vee \overline{l \vee C}, w)$ where $|A| >= 0$ and $|C| >= 1$. Negated weighted clauses are transformed into clausal form with the following recursion:

$$(A \vee \overline{l \vee C}, w) \equiv \{(A \vee \bar{C}, w), (A \vee \bar{l} \vee C, w)\}$$

We assume without loss of generality the existence of a known upper bound $\top$ of the optimal solution ($\top$ is a strictly positive natural number). A *model* is a complete assignment with cost less than $\top$. A Max-SAT instance is a pair $(\mathcal{F}, \top)$ and the task of interest is to find a model of minimum cost, if there is any. Observe that any weight $w \geq \top$ indicates that the associated clause *must be necessarily satisfied*. Thus, we can replace w by $\top$ without changing the problem. Consequently, we can assume all costs in the interval $[0..\top]$. The sum of costs is defined as,

$$a \oplus b = min\{a + b, \top\}$$

in order to keep the result within the interval $[0..\top]$. Let u and w two costs such that $u \geq w$. Their subtraction is defined as,

$$u \ominus w = \begin{cases} u - w & : \quad u \neq \top \\ \top & : \quad u = \top \end{cases}$$

Essentially, $\ominus$ behaves like the usual subtraction except in that $\top$ is an absorbing element. A clause with weight $\top$ is called *mandatory* (or *hard*). A weighted CNF formula may contain $(\Box, w)$. Since $\Box$ cannot be satisfied, w is added to the cost of any assignment. Therefore, w is an explicit *lower bound* of the optimal model. When the lower bound and the upper bound have the same value (i.e., $(\Box, \top) \in \mathcal{F}$) the formula does not have any model and we call this situation an *explicit contradiction*. In [9] is presented the Max-DPLL algorithm that is an extension of the DPLL [4].

2.1 Lower bounds based on counting inconsistencies and simplification rules

An inconsistency is a set of clauses such that at least one of them is always violated by any assignment. Various authors presented lower bounds assuming all clauses with cost one but they can be easily generalized: the weight associated to the inconsistency is the minimum weight of all the clauses involved in it. For each inconsistency, the involved clauses are marked as *visited* to avoid reusing clauses to detect other inconsistencies. It ensures an admissible lower bound. Lower bounds based on counting inconsistencies are calculated from scratch at each node of the search tree. The following three lower bound functions were proposed:

- LB_1 is the number of unsatisfied clauses by the current partial assignment (see [3]).
- LB_2 computes LB_1 and it increases by 1 the lower bound for every pair of clauses $(l, 1), (\bar{l}, 1)$ (see [13]).
- LB_3 computes LB_2 and it increases by 1 the lower bound for every three clauses $\{(l \vee h, 1), (\bar{l}, 1), (\bar{h}, 1)\}$ (see [1, 12]).
- LB_4 computes LB_1 and for each unit clause $(\bar{l}, 1)$, apply unit propagation until an inconsistency is detected (see [11]). Then, the clauses involved in the inconsistency are marked as visited and the lower bound can be increased by 1 safely.

Current solvers apply some simplifications rules as a preprocessing step. The most widely used is the *almost equivalent clause rule* presented in [2]. Given two clauses $(x \vee C, 1)$ and $(\bar{x} \vee C, 1)$, it replaces them by $(C, 1)$. It is used as a preprocessing in [1, 12, 11].

Let $LB_i(\mathcal{F})$ denote the result of computing LB_i in formula $\mathcal{F}$. It will be used for comparison purposes.

3 Max-RES: The Resolution rule for Max-SAT

As shown by [9], the notion of resolution can be extended to weighted formulas as follows,

$$\{(x \vee A, u), (\bar{x} \vee B, w)\} \equiv \left\{ \begin{array}{l} (A \vee B, m), \\ (x \vee A, u \ominus m), \\ (\bar{x} \vee B, w \ominus m), \\ (x \vee A \vee \bar{B}, m), \\ (\bar{x} \vee \bar{A} \vee B, m) \end{array} \right\}$$

where A and B are arbitrary disjunctions of literals and $m = \min\{u, w\}$. The effect of Max-RES, as in classical resolution, is to infer (namely, make explicit) a connection between A and B. However, there is an important difference between classical resolution and Max-RES. While the former yields the *addition* of a new clause, Max-RES is a transformation rule. Namely, it requires the *replacement* of the left-hand clauses by the right-hand clauses. The reason is that some cost of the left hand clauses must be subtracted in order to *compensate* the new inferred information. Consequently, Max-RES is better understood as a *movement* of knowledge in the formula.

Consider TRANSFORM($\mathcal{F}$,$r_1, r_2, \ldots, r_k$), where $r_1, r_2, \ldots, r_k$ are transformation rules based on Max-RES. It denotes the lower bound computed by the algorithm that transform the formula $\mathcal{F}$ by applying the rules $r_1, r_2, \ldots, r_k$ until a fix point is reached.

4 Max-DPLL with Inference

Similarly to what happens to plain DPLL [4], Max-DPLL does not seem to be very effective in practice. However, its performance can be improved dramatically if it is armed with more sophisticated solving techniques. One possibility is to let Max-DPLL perform a limited form of resolution at every search. Such process will presumably facilitate the posterior task of Max-DPLL.

4.1 Neighbourhood Resolution

An example suggested in [9] is Neighbourhood Resolution (NRES), which is Max-RES restricted to the $A = B$ case,

$$\{(x \vee A, u), (\bar{x} \vee A, w)\} \equiv \{(A, m), (x \vee A, u \ominus m), (\bar{x} \vee A, w \ominus m)\}$$

with $m = \min\{u, w\}$. Let NRES$_k$ be NRES restricted to $|A| = k$.

Remark 1 *$NRES$ is equivalent to the* 'almost equivalent clause rule' *presented in [2].*

Remark 2 *Given a formula $\mathcal{F}$, the lower bound computed by* TRANSFORM($\mathcal{F}$,NRES$_0$) *is the same as $LB_2(\mathcal{F})$.*

4.2 Hyper Resolution

In SAT, hyper resolution is a well known concept that refers to the compression of several resolution steps into one single step. We denote k-RES the compression of k resolution steps. NRES is beneficial because it derives smaller clauses. Pushing further this idea, we identify two situations where a small number of resolution steps derive smaller clauses. The first case corresponds to Triangle-Resolution (Tri-RES with 2 steps of resolution, similar to 2-RES in [7]):

$$\{(l \vee h \vee A, u), (\bar{l} \vee q \vee A, v), (\bar{h} \vee q \vee A, w)\} \equiv$$

$$\left\{ \begin{array}{l} (q \vee A, m), (h \vee q \vee A, m_1 \ominus m), (\bar{h} \vee q \vee A, w \ominus m), (l \vee h \vee A, u \ominus m_1) \\ (\bar{l} \vee q \vee A, v \ominus m_1), (l \vee h \vee \bar{q} \vee A, m_1), (\bar{l} \vee \bar{h} \vee q \vee A, m_1) \end{array} \right\}$$

where $m = min\{u, v, w\}$. Note that $(q \vee A, m)$ is smaller than any original clause.

The second case corresponds to 3-RES presented in [7]. We have extended it to a $(k+1)$-ary case ($k+1$ steps of resolution) and we restrict it to binary clauses due to space limitations. We will refer to it as *Chain-RES*:

$$\{(\overline{h_0} \vee h_1, w_1), (\overline{h_1} \vee h_2, w_2), (\overline{h_2} \vee h_3, w_3), \ldots, (\overline{h_{k-1}} \vee h_k, w_k), (h_0, v_1), (\overline{h_k}, v_2)\} \equiv$$

$$\left\{ \begin{array}{l} (\overline{h_0} \vee h_1, w_1 \ominus m), (\overline{h_1} \vee h_2, w_2 \ominus m), (\overline{h_2} \vee h_3, w_3 \ominus m), \ldots, (\overline{h_{k-1}} \vee h_k, w_k \ominus m), \\ (h_0 \vee \overline{h_1}, m), (h_1 \vee \overline{h_2}, m), (h_2 \vee \overline{h_3}, m), \ldots, (h_{k-1} \vee \overline{h_k}, m), \\ (h_0, v_1 \ominus m), (\overline{h_k}, v_2 \ominus m), (\Box, m) \end{array} \right\}$$

where $m = min\{w_1, w_2, w_3, \ldots, w_{k-1}, v_1, v_2\}$ and $k \geq 1$.

Remark 3 *Given a formula $\mathcal{F}$, lower bounds computed by* TRANSFORM($\mathcal{F}$,NRES$_0$,CHAIN-RES) *is the same as $LB_3(\mathcal{F})$ if $k = 1$.*

4.3 Hyper Resolution with Hard clauses

When clauses are hard, it may be advantageous applying hyper resolution because it does not generate new clauses. In fact, they are subsumed by the hard clauses. Consider:

$$Bin(l_1, l_2, \ldots, l_k) \equiv \left\{ \begin{array}{l} (\overline{l_1} \vee \overline{l_2}, \top), (\overline{l_1} \vee \overline{l_3}, \top), \ldots, (\overline{l_1} \vee \overline{l_k}, \top), \\ (\overline{l_2} \vee \overline{l_3}, \top), (\overline{l_2} \vee \overline{l_4}, \top), \ldots, (\overline{l_2} \vee \overline{l_k}, \top), \\ , \ldots, \\ (\overline{l_{k-1}} \vee \overline{l_k}, \top) \end{array} \right\}$$

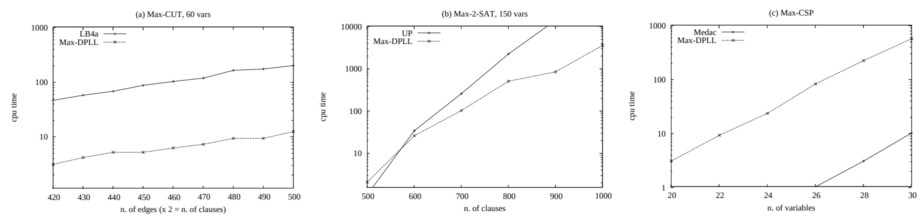


Fig. 1. Results on (a) Max-CUT , (b) Max-2-SAT and (c) Max-CSP. Note the logarithmic scale.

We have detected two typical cases where this situation appears: The first one is called *Hard-RES$_a$*:

$$\{(l_1 \vee l_2 \vee \ldots \vee l_k, \top), Bin(l_1, l_2, \ldots, l_k), (l_1, u_1), (l_2, u_2), \ldots (l_k, u_k)\} \equiv$$

$$\{(\Box, m), (l_1 \vee l_2 \vee \ldots \vee l_k, \top), Bin(l_1, l_2, \ldots, l_k), (l_1, u_1 \ominus m), (l_2, u_2 \ominus m), \ldots (l_k, u_k \ominus m)\}$$

where $m = min\{u_1, \ldots, u_k\}$ and $k \geq 2$.

The second one is called *Hard-RES$_b$*:

$$\{(l_1 \vee l_2 \vee \ldots \vee l_k, \top), Bin(l_1, l_2, \ldots, l_k), (h \vee l_1, u_1), (h \vee l_2, u_2), \ldots, (h \vee l_k, u_k)\} \equiv$$

$$\left\{ \begin{array}{l} (h, m), (l_1 \vee l_2 \vee \ldots \vee l_k, \top), Bin(l_1, l_2, \ldots, l_k), \\ (h \vee l_1, u_1 \ominus m), (h \vee l_2, u_2), \ldots, (h \vee l_k, u_k \ominus m) \end{array} \right\}$$

where $m = min\{u_1, u_2, \ldots, u_k\}$ and $k \geq 2$.

We noticed that Hard-RES$_a$ and Hard-RES$_b$ are very effective in WCSP problems with non-binary domains modelled as Max-SAT problems using direct encoding [14]. Note that in both cases the number of resolution steps is k.

5 Experimental Results

In this Section we evaluate the performance of Max-DPLL enhanced with the new inference rules against some state-of-the-art Max-SAT solvers. Our current implementation only takes into account clauses of arity less than or equal to two (the number of such clauses is $O(n^2)$ where n is the number of variables) for rules $NRES_0$, $NRES_1$, Tri-RES and Chain-RES. Finally, $NRES_k$, Hard-RES$_a$ and Hard-RES$_b$ with $k > 1$ are applied as a preprocessing step. We compared our implementation, that we call MAX-DPLL, with the following solvers: MAXSOLVER [15], LB4A [12], UP [11], MEDAC [5]. Note that MEDAC and Max-DPLL enhanced with NRES$_0$, NRES$_1$, Chain-RES rule (with $k = 1, 2$) introduced in this paper are deeply related. Hence, we will focus our experiments in the novel rules binary Tri-RES, Chain-RES (with $k > 2$), Hard-RES$_a$ and Hard-RES$_b$.

Remark 4 *Given a formula $\mathcal{F}$, the lower bound computed by* TRANSFORM($\mathcal{F}$,NRES$_0$, NRES$_1$,CHAIN-RES,TRI-RES,HARD-RES$_a$,HARD-RES$_b$) *is better than the one computed by $LB_3(\mathcal{F})$.*

We show the results of Max-DPLL and the second best solver for each comparison:

- Figure 1.a: MAX-CUT problems with 60 variables and varying the number of edges. Max-DPLL is one order of magnitude faster than the second best solver LB4a. In these problems, Tri-RES improves a lot the performance of our solver.
- Figure 1.b: Random MAX-2-SAT problems with 150 variables and varying the number of clauses. Max-DPLL is two orders of magnitude faster than the second best solver UP. The reason for the speed up are both Tri-RES and binary k-RES.
- Figure 1.c: Random Max-CSP sparse-tight problems modelled as Max-SAT using direct encoding [14]. Hard-RES$_a$ and Hard-RES$_b$ improve a lot the performance of Max-DPLL. The second best solver is Medac two orders of magnitude slower.

Remark 5 *Given a formula $\mathcal{F}$, lower bounds computed by* TRANSFORM($\mathcal{F}$, NRES$_0$, NRES$_1$,CHAIN-RES,TRI-RES, HARD-RES$_a$,HARD-RES$_b$) *and $LB_4(\mathcal{F})$ are incomparable.*

In spite of the previous remark, we noticed that the first lower bound was usually better than the second one empirically or that the overhead to compute the second one was not worthwhile. However, we know that both techniques are powerful and complementary. Hence, these two methods can be applied together: Given a Formula $\mathcal{F}_1$ the new hybrid applies TRANSFORM($\mathcal{F}_1$,NRES$_0$,NRES$_1$,CHAIN-RES,TRI-RES,HARD-RES$_a$,HARD-RES$_b$) until a fix point is reached. Let be $\mathcal{F}_2$ the transformed formula. Then it applies $LB_4(\mathcal{F}_2)$. We have not tested it because a new implementation with efficient structures for both methods is needed.

6 Conclusions and Future Work

In this paper we have introduced inference rules based on the resolution rule for Max-SAT and we have studied their relation to previous works. The performance of a DPLL-like algorithm has been dramatically improved by applying them at each node of the search tree. While the current work was focussed in detecting specific cases where resolution should be applied, we plan to investigate sophisticated methods for detecting by themselves when applying resolution is profitable.

References

1. T. Alsinet, F. Manyà, and J.Planes. A max-sat solver with lazy data structures. In *IBERAMIA*, pages 334–342, 2004.
2. N. Bansal and V. Raman. Upper bounds for maxsat: Further improved. In *ISAAC*, pages 247–258, 1999.
3. B. Borchers and J. Furman. A two-phase exact algorithm for max-sat and weighted max-sat problems. *J. Comb. Optim.*, 2(4):299–306, 1998.
4. M. Davis, G. Logemann, and G. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5:394–397, 1962.
5. S. de Givry, F. Heras, J. Larrosa, and M. Zytnicki. Existential arc consistency: getting closer to full arc consistency in weighted csps. In *Proc. of the 19th IJCAI*, 2005.
6. S. de Givry, J. Larrosa, P. Meseguer, and T. Schiex. Solving max-sat as weighted csp. In *Proc. of the 9th CP*, pages 363–376, Kinsale, Ireland, 2003. LNCS 2833. Springer Verlag.
7. F. Heras and J. Larrosa. New inference rules for effi cient max-sat solving. In *Proc. of the 21th AAAI*, 2006.
8. J. Larrosa. Node and arc consistency in weighted csp. In *Proceedings of the 18th AAAI*, pages 48–53, 2002.
9. J. Larrosa and F. Heras. Resolution in max-sat and its relation to local consistency for weighted csps. In *Proc. of the 19th IJCAI*, 2005.
10. J. Larrosa and T. Schiex. Solving weighted csp by maintaining arc-consistency. *Artificial Intelligence*, 159(1-2):1–26, 2004.
11. Chu Li, Felip Manyà, and Jordi Planes. Exploiting unit propagation to compute lower bounds in branch and bound max-sat solvers. In *CP*, pages 403–414, 2005.
12. H. Shen and H. Zhang. Study of lower bounds for max-2-sat. In *Proceedings of the 19th AAAI*, 2004.
13. F. Manya T. Alsinet and J. Planes. Improved branch and bound algorithms for max-sat. In *SAT03*, pages 408–415, 2003.
14. T. Walsh. Sat v csp. In *CP-2000*, pages 441–456, 2000.
15. Z. Xing and W. Zhang. Maxsolver: An effi cient exact algorithm for (weighted) maximum satisfi ability. *Artificial Intelligence*, 164(1-2):47–80, 2005.

Suffix arrays and weighted CSPs

Student: Matthias Zytnicki
Supervisors: Christine Gaspin, Thomas Schiex

INRA Toulouse – BIA

Abstract. In this paper, we describe a new constraint that uses some interesting data structure, the suffix array, well-known in pattern matching. We show how it helps answering the question of non-coding RNA detection in bio-informatics, and more precisely, finding the best hybrid with a duplex constraint.

1 Introduction

Thanks to the recent major advances in molecular biology, the problem of the detection of non-coding RNA (ncRNA) is now a hot topic in bio-informatics (cf. [1] for review). A ncRNA is usually represented by a sequel of letters, or *nucleotides*: A, C, G and T. An ncRNA also contains *interactions* —mainly A–T and C–G— that are essential to its biological function. In this paper, we will suppose we know the *signature* of a ncRNA family. The signature is the general shape of all the ncRNAs that share a common biological function. Our aim is the following: how can I get all the candidates matching a given signature, in a sequence that may contains several billions of nucleotides?

Among the proposed formalisms used to solve this problem, one of the most famous one uses statistical information in a context-free grammar that describes this signature [5]. However, some complex ncRNA families cannot be described within this formalism and [6] showed that only NP-complete formalisms may correctly describe them. This favors a CSP model of the problem and such a work has been been done in [7].

However, usual queries give hundred of thousands of solutions and, in practice, it is impossible to exploit this huge amount of solutions. Obviously, by looking more carefully at the solutions, some are better than others and it would be useful to give only the best ones to the user. This is why we used the weighted CSP formalism to solve the ncRNA detection problem.

One interesting element of signature that we would like to model is the duplex. It is the ability from the ncRNA to *hybridize*, i.e. to develop a stretch of interactions with a DNA strand, or another RNA. We would like to embed this element of signature into a global constraint. Many formalisms have been proposed (cf. [2] for review) to compute the underlying algorithm. Most of them are based on a dynamic programming algorithm that computes a kind of *edit distance* between a word and the factors of a long sequence. To save time and space, these algorithms have been ported to different structures, such as the suffix tree. This structure makes it possible to focus the search on the most promising regions and dramatically speeds up the search. Recently, some papers [3, 4] also proposed the suffix array to solve this kind of problem, with an enhancement that provides several advantages compared with the suffix tree, with no drawback.

In this paper, we present a new global constraints that checks with a suffix array whether there exists a word that matches (with possible errors) a factor in a given long sequence.

2 The WCSP model

The weighted CSP (WCSP) [8] framework is an extension of the CSP, that makes it possible to express *preferences* amoung solutions thanks to *soft constraints*. It has already been applied in resource allocation, scheduling, combinatorial auction, CP networks and probabilistic reasoning.

The *valuation structures* $\mathcal{S} = \langle E, \oplus, \leq \rangle$ is the algebraic object that specifies costs, where: $E = [0..k] \subseteq \mathbb{N}$ is the *set of costs*, k, which is the highest cost, can possibly be ∞, and it represents an *inconstency*; $\leq$ is the usual operator on $\mathbb{N}$, $\oplus$, the *addition* on E, is defined by $\forall (a, b) \in \mathbb{N}^2, a \oplus b = \min\{a + b, k\}$. A WCSP is a tuple $\mathcal{P} = \langle \mathcal{S}, \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where: $\mathcal{S}$ is the valuation structure, $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a set of n *variables*, $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is the set possible *values* of each variable, or *domains*, and the size of the largest one is d, $\mathcal{C} = \{c_1, \ldots, c_e\}$ is the set of e soft constraints.

An assignment t on the set of variables $Y \subseteq \mathcal{X}$ is a function that associates to each variable of Y one of its possible value: $t = (y_1 \leftarrow v_1, \ldots, y_m \leftarrow v_m)$. A soft constraint c_i involves a list of r_i variables $var(c_i) = (y_1, \ldots, y_{r_i})$ (r_i is the *arity* of c_i), and it associates to every assignment t of the involved variables a cost $c_i(t)$ in E. Given a constraint c_i and an assignment t of $var(c_i)$, $c_i(t) = k$ means that the constraint forbids the corresponding assignment. Another cost means the assignment is permitted by the constraint with the corresponding cost. Given an assignment t on the set Y of variables and another set X of variables such that $X \subseteq Y$, the projection of t w.r.t. X, noted $t|_X$, represents the assignment of the variables of X to the values given by t. The cost of a total assignment (i.e. of all the variables) $t = (v_1 \leftarrow x_1, \ldots, v_n \leftarrow x_n)$, noted $\mathcal{V}(t)$, is the sum over all the cost functions: $\mathcal{V}(t) = \bigoplus_{c_i \in \mathcal{C}} c_i(t|_{var(c_i)})$. A total assignment t is *consistent* if $\mathcal{V}(t) < k$. Finding a total consistent assignment with minimum cost is a NP-hard problem. Observe that, if $k = 1$, a WCSP reduces to classic CSP.

In our model, as described in [7], the variables represent the *positions* on the sequence of the elements of signature. The initial domain of the variables, unless otherwise stated, will therefore be equal to the size of the sequence. The constraints enforce the presence of the wished elements of signature between the specified variables. Within this model, a solution is a position for each variable, such that all the elements of signature specified by the constraints can be found. Our aim is to find all the solutions of the problem, given a maximum cost k.

We will not describe here all the constraints used, and we will focus on the duplex constraint. This constraints ensures that there exists a set of interactions between our sequence (the *main sequence*) and another given sequence (the *target sequence*). It has four parameters: the target sequence, the minimum and maximum sizes of the duplex, and the maximum number of errors in the interaction set. Similarly to the edit distance, the number of errors of a hybridization is the number of nucleotides that does not interact with any other nucleotide, plus the number of pair of nucleotides associated through a non-allowed interaction. The duplex constraint involves four variables: x_i, x_j, y_k and y_l. The x variables represent positions on the main sequence, and the y variables, positions on the target sequence. To solve the problem, we use a depth-first branch-and-bound algorithm that maintains a extension of 2B-consistency adapted to soft constraints, called *bound arc consistency* (BAC*, [9]).

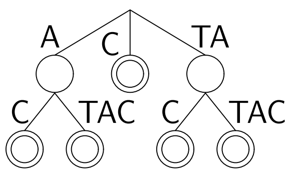
In our implementation, the duplex constraint remains idle until the x variables are assigned, and BAC* is only enforced on the y variables. In other words, when the word on the main sequence is known, we have to find the minimum cost of the constraint, when the y variables are assigned to their minimum and maximum values of their domains. Thus, our aim will be to design a algorithm that efficiently finds the minimum number of errors between a word and the factors of a given subsequence bounded by the y variables.

3 Suffix arrays

The suffix tree is a tree with edges labeled with words and this data structure has been widely used in pattern matching algorithms. Given a text, the paths from the root node of its suffix tree and its terminal nodes enumerate all the suffixes of this text (cf. **Fig.** 1(a)). It is a particularly convenient data structure, since it requires linear space w.r.t. the size of the text, takes linear time to build, and searching whether a word is contained in this text requires time proportionnal to this word (and is independent in the size of the text).

However, given a sequence T of size m, its suffix array S also requires a linear time to build, takes as much time to find a word, but requires a bit less space, and lead to less cache misses, thanks to the array structure [4]. Basically, a suffix array is an array where all the suffixes of a text are sorted through lexicographic order (cf. **Fig.** 1(b)). Of course, only the position $suf[i]$ of the first letter of each suffix i is stored. Additional information is also stored on each line of the array. First, the size of the longest common prefix (noted *lcp*) between the suffix of line i and line $i-1$ is inserted on line i (by convention, $lcp[0] = 0$).

(i, j) is called a *l-interval* iff: $lcp[i] < l, \forall k \in (i, j], lcp[k] \geq l, \exists k \in (i, j], lcp[k] = l$, and $lcp[j+1] < l$. These l-intervals can be compared with the nodes of the suffix tree. It is an interior node if $i \neq j$, and it is a leaf if $i = j$. Using linear space, we can build in linear time a function that, given a l-interval, gets their child l'-intervals (i', j'). With this function in hand, we can simulate a suffix tree with our suffix array. Using the same notations, we will note $letters(i, j)$ the factor $T[suf[i]..suf[i]+l]$, and $letters((i, j) \rightarrow (i', j'))$ the factor $T[suf[i]+l..suf[i]+l']$.



(a) the suffix tree

	i	suf	lcp	$text$
1	1	3	0	AC
1	2	1	1	ATAC
	3	4	0	C
2	4	2	0	TAC
2	5	0	2	TATAC

(b) the suffix array

Fig. 1. Two representations of TATAC

4 An algorithm for approximate matching

4.1 First algorithm

This algorithm takes as an input a word w of size n and a maximum edit distance $maxErr$. It returns the minimum distance between w and the set of factors of T, or $maxErr + 1$ if this distance is greater than $maxErr$. It uses a hybridization cost matrix c_{hyb}, that, given two nucleotides, returns the hybridization penalty (0 being a perfect hybridization). c_{ins} is the penalty cost for a non-hybridized nucleotide.

Let us explain first the function getCandidates(). It gets two strings, w and b, of size s and t respectively, and try to hybridize them. It also takes $maxErr$ as a parameter, which gives the maximum allowed distance between w and b. Basically, it is a simple Needleman-Wunsch dynamic programming algorithm. The only difference is that it returns a list containing all the solutions with a cost less than $nbErr$ that are located on the last row or on the last column of the dynamic programming matrix. If it is on the last row, then b has been totally matched with a prefix of w; if it is on the last column, w has been totally matched with a prefix of b. Each element of the solution list contains the number of matched letters of the prefix, the score of the match, and a boolean that states whether the solution is on the last row or on the last column.

The main function, getApproximateWord(), works as follows. On line **1**, we consider a l-interval, between the lines i and j in the array. We suppose that we have matched $pref_w$ letters of w so far, and we have encountered $nbErr$ errors.

The function getChildren() returns in constant time all the child intervals of (i, j). The line **3** checks whether the considered child l'-interval is an interior node or a leaf. In the former case, we try to match $letters((i, j) \rightarrow (i', j'))$ with the remaining unmatched letters of w through the function getApproximateWord() on line **4**. If the flag f of an element returned by this function is set to true (line **5**), then all the letters of w have been matched and we may have a solution. If not (line **6**), then we have to continue the exploration. For that, we store the current configuration (including the bounds and the lcp of the current interval, and the

Algorithm 1: Functions used for approximate search

```
Function getCandidates(string $w$, string $b$, int $maxErr$): list (int, int, bool)
    for $i \in [0..s]$ do $mat[i][0] = i$ ; for $j \in [1..t]$ do $mat[0][j] = j$ ;
    for $i \in [1..s]$ do for $j \in [1..t]$ do
        $mat[i][j] \leftarrow \min \begin{cases} mat[i-1][j-1] + c_{\text{hyb}}(w[i-1], b[j-1]), \\ mat[i-1][j] + c_{\text{ins}}, \\ mat[i][j-1] + c_{\text{ins}} \end{cases}$ ;
    for $j \in [0..t]$ do if $(mat[s][j] \leq maxErr)$ then $list.add(j, mat[s][j], \mathsf{true})$ ;
    for $i \in [0..s]$ do if $(mat[i][t] \leq maxErr)$ then $list.add(i, mat[i][t], \mathsf{false})$ ;
    return $list$ ;

Function getApproximateWord(string $w$, int $maxErr$): list(int, int, int)
    $stack.push(0, n-1, 0, 0)$ ; $min \leftarrow maxErr + 1$ ;
    while $(\neg stack.empty())$ do
1       $(i, j, pref_w, nbErr) \leftarrow stack.pop()$ ;
2       for $(i', j') \in$ getChildren$(i, j)$ do
3           if $(i' \neq j')$ then
4               $list \leftarrow$ getCandidates$(letters((i, j) \rightarrow$
                $(i', j')), w[pref_w..m-1], maxErr - nbErr)$ ;
                while $(\neg list.empty())$ do
                    $(len, score, f) \leftarrow list.pop()$ ;
5                   if $(f)$ then $min \leftarrow \min\{nbErr + score, min\}$ ;
6                   else $stack.push(i', j', pref_w + len, nbErr + score)$ ;
            else
7               $list \leftarrow$ getCandidates$(letters((i, j) \rightarrow$
                $(i', i')), w[pref_w..m-1], maxErr - nbErr)$;
                while $(\neg list.empty())$ do
                    $(len, score, f) \leftarrow list.pop()$ ;
8                   if $(f)$ then $min \leftarrow \min\{nbErr + score, min\}$ ;
    return $min$ ;
```

number of errors found so far) in a stack, that we will examine afterwards. If the current child interval is a leaf (line **7**), then there is only one possibility to match the remaining unmatched letters of w. Thus, we simply call getCandidates() and only keep the solutions that match all the letters of w on line **8**.

4.2 Optimizations

Furthermore, we have implemented several optimizations. First, we observed that the exploration often visits several times the same l-intervals with exactly the same configuration, or even with less interesting configurations (they contain more errors, with the same number of matched letters). Obviously, some work is unnecessarily done. To avoid it, without using too much space, we store at each node the last configuration that visited it.

Second, we propagate information between the y variables. For example, if we have some information about the y_k variable, then we may shrink the domain of the y_l variable, knowing the size of w, and the number of allowed errors.

Then, we tried to take advantage of the information given by the WCSP. For instance, the solver might have reduced the bounds of the y_k variable (which states the beginning of the duplex in the target sequence), and this information should be used to prune some branches of the suffix array. To achieve this dynamical pruning, we added on each l-interval (i, j) the smallest interval of S that contains the factor $letters(i, j)$, so that the values of y_k that have been deleted by the WCSP solver will never be explored by the suffix array.

A first, rough, evaluation of the worst time complexity of our algorithm is $(2m + maxErr)^{m+maxErr+1}\sigma^{maxErr}m^{maxErr+2}$, where σ is the size of the alphabet. On real life examples, where the main and the target sequences contain several millions of nucleotides, enforcing this constraint usually takes not more than a few seconds in the whole execution of the program. This is all the more encouraging as our program finds *all* the solutions of the problem.

5 Conclusions and future work

In this paper we have presented a new constraint, dedicated to bio-informatics problems (or, more generally, to text-based problems), that uses suffix arrays, in an attempt of combining constraints with pattern matching algorithms. In the future, we would like to compare our method with other existing ones, and provide for an empirical evaluation of our approach.

References

1. Eddy, S.: Non-coding RNA genes and the modern RNA world. Nature Reviews **2** (2001)
2. Gusfield, D.: Algorithms On Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge Univ. Press (1997)
3. Abouelhoda, M., Kurtz, S., Ohlebusch, E.: The enhanced suffix array and its application to genome analysis. In: Second Workshop in Algorithms in Bioinformatics. (2002)
4. Abouelhoda, M., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms (2004) 53–86
5. Eddy, S., Durbin, R.: RNA sequence analysis using covariance models. Nucleic Acids Research **22** (1994) 2079–88
6. Vialette, S.: On the computational complexity of 2-interval pattern matching problems. Theoretical Computer Science **312** (2004) 223–249
7. Thébault, P., de Givry, S., Schiex, T., Gaspin, C.: Combining constraint processing and pattern matching to describe and locate structured motifs in genomic sequences. In: 5th Workshop On Modelling and Solving Problems With Constraints. (2005)
8. Larrosa, J., Schiex, T.: Solving weighted CSP by maintaining arc-consistency. Artificial Intelligence **159** (2004) 1–26
9. Zytnicki, M., Schiex, T., Gaspin, C.: A new local consistency for weighted CSP dedicated to long domains. In: SAC 2006. (2006)

Symmetric Component Caching

Student:Matthew Kitching
Supervisor: Fahiem Bacchus

Department of Computer Science, University of Toronto, Canada.
[kitching|fbacchus]@cs.toronto.edu

1 Introduction

Caching is a powerful technique for improving the efficiency of backtracking search. It involves remembering (caching) information at some previously visited nodes in the search tree and using that information to prune nodes from the remaining search space. In the context of CSPs formula caching corresponds to remembering the subproblem that was solved at each previously visited node, and then using the cache to avoid solving the same subproblem more than once.

In [1] it is proved that formula caching can improve the worst case behavior of backtracking from $2^{O(n)}$ to $2^{O(w)\log(n)} = n^{O(w)}$ where n is the number of variables and w is the *treewidth* of the CSP graph. This can be a significant improvement in practice. An alternate way of achieving an $n^{O(w)}$ worst case bound is to perform search with decomposition.

Caching can be easily combined with decomposition, we simply cache the components encountered and solved during search. With caching we can improve the worst case bound from $n^{O(w)}$ to $n^{O(1)}2^{O(w)}$ [2, 1].

In this paper we explore the use of *component caching*, i.e., search with decomposition and caching, for solving constraint optimization problems (COPs). A number of features make our approach an innovation over previous works. First, we develop a technique that imposes no restriction on the variable ordering strategy used during search. Second, we develop some new techniques and data structures for minimizing the overhead of utilizing dynamic variable orderings during search. Third, we provide a deeper integration between caching and branch and bound, used when solving COP. And fourth, we show how the data structures we have developed for components allow us to also exploit symmetries. In particular, employing the techniques of [6] we automatically detect and exploit *component symmetries*.

2 Background

Constraint Optimization Problems (COPs). A COP $\mathcal{P}$ is specified by a tuple $\langle \mathcal{V}, Dom, \mathcal{C} \rangle$, where $\mathcal{V}$ is a set of variables, Dom is a domain of values for each variable ($Dom[V] : V \in \mathcal{V}$), and $\mathcal{C}$ is a set of constraints. Each constraint $C \in \mathcal{C}$ is a function defined over a subset of $\mathcal{V}$, called the $scope[C]$. C maps every instantiation of the variables in its scope to a real number. A solution to a COP is an assignment π to the variables in $\mathcal{V}$ such that $\sum_{C\in\mathcal{C}} C(\pi\downharpoonleft_{scope[C]})$ is maximized, where $\pi\downharpoonleft_{scope[C]}$ is the subset of π consisting of the assignments to the variables in $scope[C]$. Hard constraints are accommodated by assigning violating assignments $-\infty$. The *value* of a COP $\mathcal{P}$, $value(\mathcal{P})$ is simply the value that is achieved by a solution.

Reduced COPs. During backtracking we explore nodes that have been reached by making assignments to some set of variables. At node n where the set of assignments π has been made, the search must solve a sub-problem that is the reduction of the original COP by π. Let $\mathcal{P} = \langle \mathcal{V}, Dom, \mathcal{C} \rangle$ be the original COP, $varsOf[\pi]$ be the set of variables assigned in π, and $consOf[\pi]$ be the set of constraints that have been fully instantiated by π. The reduction of $\mathcal{P}$ by π, $\mathcal{P}\!\downharpoonright_{\pi}$, is a new COP $\langle \mathcal{V}', \mathcal{C}' \rangle$ with $\mathcal{V}' = \mathcal{V} - varsOf[\pi]$, i.e., the variables not assigned by π, and $\mathcal{C}' = \{C\!\downharpoonright_{\pi} \mid C \in \mathcal{C} - consOf[\pi]\}$, i.e., the reduction of the constraints not fully instantiated by π. The reduction of an individual constraint C by π, $C\!\downharpoonright_{\pi}$, is the new constraint whose scope is $scope(C) - varsOf[\pi]$ and whose value on a set of assignments τ is $C\!\downharpoonright_{\pi}(\tau) = C(\tau \cup \pi\!\downharpoonright_{scope(C)})$.

Components. A component of a COP $\mathcal{P} = \langle \mathcal{V}, Dom, \mathcal{C} \rangle$ is a COP $\rho = \langle \mathcal{V}', \mathcal{C}' \rangle$ formed from a subset of the variables and constraints of $\mathcal{P}$, i.e., $\mathcal{V}' \subseteq \mathcal{V}$ and $\mathcal{C}' \subseteq \mathcal{C}$. Its essential property is that it is disjoint from the rest of $\mathcal{P}$. In particular, $C \in \mathcal{C}' \rightarrow scope(C) \subseteq \mathcal{V}'$. A component ρ is itself a COP and thus it has a set of optimal solutions. Since ρ shares no constraints with the rest of $\mathcal{P}$, the optimal solutions of $\mathcal{P}$ consist of pairs of optimal solutions: an optimal solution of ρ and an optimal solution to the rest of $\mathcal{P}$. Thus we can solve ρ independently of the rest of $\mathcal{P}$.

3 Component Caching Search (CCS) with Branch and Bound

When we solve a COP $\mathcal{P}$ we provide a bound LB. We require that $value(\mathcal{P}) > LB$, so we can abort the computation of $value(\mathcal{P})$ whenever we determine that it is less than or equal to LB. For any branching variable V the value of $\mathcal{P}$ is the maximum value that we can achieve over all possible assignments to V. Each assignment $V = d$ yields some immediate value cv from the constraints that are fully instantiated by the assignment, along with some collection of components that have to be recursively solved. These components must yield more than $LB - cv$ in value for $V = d$ to be viable. Given upper bounds on the value that each component can achieve, we have that each individual component must yield at least $LB - cv -$ sum of the upper bounds of the other components. Hence, we can pass this new lower bound to the recursive computation the component. This computation might improve the bounds on the component's value, so after we return we can utilize these improved bounds in the bound we pass to the next component. Similarly, if the value we compute for $V = d$ is greater than LB, we can update LB to this higher value: there is no point in trying another assignment that yields a lower value than $V = d$.

Our innovation for achieving a full integration of branch and bound with caching is to cache two values with every component C, a lower bound, $C.lb$ and an upper bound $C.lb$ with $C.lb \leq value(C) \leq C.ub$. The first time we encounter C in the search we can initialize these bounds to problem specific values. If the computation of C's value is aborted by detecting that it cannot achieve the value we need in this context, we might still be able to improve the bounds on C's value. Our caching scheme allows us to retain this improved information so that the next time we encounter C, we can continue the computation of its value with its new better lower bound thus improving the efficiency of the new computation. The new computation might again be aborted, but we can again update the cached bound with information gathered from the new computation. This process stops if we eventually compute the component's exact value.

3.1 Component Templates

There are two operations in **`CCS+BB`** which have a non-trivial impact on the cost of processing each node of the search tree. These are breaking $\mathcal{P}$ up into components, which can be done with a depth-first search over its CSP graph, and cache, which naively requires searching among fairly complex reduced COP structures.

With unrestricted variable ordering, as employed here, we have to find alternate ways of implementing these operations efficiently. Here, we accomplish this by clustering related components into groups, and representing the entire group, along with a cache specific to the group, in a *component template*.

The key idea is that of a *dependency set*. Consider a component ρ of the original COP $\mathcal{P}$ created after some variable assignments have been made. ρ contains some variables and the set of reduced constraints. Each constraint of ρ is a constraint of $\mathcal{P}$ that has been reduced by some set of instantiated variables. We define the dependency set of ρ, $\mathit{dSet}[\rho]$, to be the set of *assigned* variables that reduced its constraints.

Assigning the dependency variables caused the component to be created. Once we have a component we can observe that *any* instantiation of its dependency variables will create a component over the same set of variables. All such components have the same dependency set, and they all contain the same set of variables.

A *component template* is used to store such groups of structurally similar components. We call the components covered by a template the *instances* of the template, and use $\rho \in \rho T$ to indicate that the component ρ is an instance of template ρT. A component template, ρT, consists of four items. (1) The set of component variables common to all of its instances, $\mathit{varsOf}[\rho T]$, (2) the set of dependency variables, $\mathit{dSet}[\rho T]$, common to all of its instances, (3) the set of constraints of the original COP that contain a variable from $\mathit{varsOf}[\rho T]$ in their scope, $\mathit{consOf}[\rho T]$, and (4) a cache used to store the upper and lower bounds of template's instances, $\mathit{cache}[\rho T]$.

Let the component ρ be an instance of the template ρT, $\rho \in \rho T$, and let π be the set of assignments to $\mathit{dSet}[\rho T]$ that uniquely determines ρ. We note two things. First, $\mathit{consOf}[\rho T]$ reduced by π are precisely the set of reduced components in ρ, and second once we have the template we can access ρ's bounds by using π to index into $\mathit{cache}[\rho T]$: $\mathit{cache}[\rho T][\pi]$. Thus cache lookup is reduced to array indexing, just as in the approaches that utilize restricted variable ordering. The only thing we need to do is first find the template ρT.

We say that a template is *triggered* when one of its instances appears in the reduced COP at a node of the search tree. The template cannot be triggered until all of its dependency variables are assigned, and this event can be efficiently detected using watch variable techniques that monitor the status of a single watch variable from the dependency set of each stored component template. When a variable is assigned, we check all of the templates it is watching, either updating their watches or testing to see if all of their component variables are unassigned. If by this check we efficiently detect that ρT has all of its dependency variables and none of its component variables assigned, then we can trigger ρT. Once a template is triggered we know that one of its instances is a component of the current reduced COP, and we can retrieve the bounds for this component by indexing $\mathit{cache}[\rho T]$ with the current assignments to $\mathit{dSet}[\rho T]$.

```
CCS+BB($\mathcal{PT}$,LB,$\pi$)   /* Template version */
1.  if $varsOf[\mathcal{PT}] = \emptyset$ return(0,0)
2.  $V$ := select variable from $varsOf[\mathcal{PT}]$ to branch on
3.  CompTs := all component templates triggered by assigning $V$
4.  rest := $\mathcal{PT}|_V$ with all variables and constraints in CompTs removed
              then remove $V$ from $varsOf$[rest] and add it to $dSet$[rest]
5.  NewCompTs := Break rest into new component templates, and
                 place each component into the permanent template store
6.  CompTs := NewCompTs $\cup$ CompTs
7.  foreach $d \in Dom[V]$
8.    $\pi$ := $\pi \cup \{V = d\}$
9.    Constraint_Propagation($\mathcal{PT}|_V, \pi$)
10.   cv := sum of the values of all constraints C of $\mathcal{PT}$
            that have become fully instantiated by line 9.
11.   foreach CompT$_i$ $\in$ CompsT $\{$(lb$_1$,ub$_i$) := cache[CompT$_i$] [$\pi|_{dSet[\mathtt{CompT_i}]}$] $\}$
12.   foreach CompT$_i$ $\in$ CompsT
13.     if($\sum_i$ ub$_i$ > LB-cv && lb$_i$ < ub$_i$)
14.       LB$_i$ := LB-cv-$\sum_{j \neq i}$ub$_j$
15.       (lb$_i$,ub$_i$) := CCS+BB(CompT$_i$, max(LB$_i$,lb$_i$),$\pi$)
16.   (lb$^d$,ub$^d$) = (cv,cv) + ($\sum_i$ lb$_i$, $\sum_i$ ub$_i$)
17.   LB := max(LB,lb$^d$)
18. (lb$^{\mathcal{P}}$,ub$^{\mathcal{P}}$) = (max$_d$(lb$^d$), max$_d$(ub$^d$))
19. cache[$\mathcal{PT}$] [$\pi|_{dSet[\mathcal{PT}]}$] := (lb$^{\mathcal{P}}$,ub$^{\mathcal{P}}$)
20. return(lb$^{\mathcal{P}}$,ub$^{\mathcal{P}}$)
```

Fig. 1. Component Caching Search with Branch and Bound and Templates

When search starts we detect components using depth-first search creating new templates for each component we detect. Each template can then be used during the rest of the search to efficiently detect the appearance of any of its instances. Thus, when faced with solving $\mathcal{P}$, we first find all triggered templates. Each triggered template disconnects its variables from $\mathcal{P}$. Now we only have to use depth-first search to detect components in the remaining part of $\mathcal{P}$, which is more efficient than searching all of $\mathcal{P}$.

Lines 3-6 realize the template triggering method outlined above. Given that we know we are about to instantiate V, we can find all triggered templates. The instances of these templates that will be created and processed in lines 7-17 will depend on the value assigned to V, but the triggering of the template is independent of V's assignment. Hence, we can find all triggered templates outside of the iteration over V's values. Each of these triggered templates will have component variables that are a subset of $varsOf[\mathcal{PT}] - V$, dependency sets that are a subset of $dSet[\mathcal{P}] \cup \{V\}$, and constraints that are a subset of $consOf[\mathcal{PT}]$: the instances of these templates are components of instances of $\mathcal{PT}$ reduced by assignments to V. We can then use depth-first search over rest, the remaining constraints of $\mathcal{PT}|_V$, where $\mathcal{PT}|_V$ is the reduction of $\mathcal{PT}$ by making V an assigned variable. That is $\mathcal{PT}\rfloor_V$ is $\mathcal{PT}$ with V removed from $\mathit{varsOf}[\mathcal{PT}]$ and added to $\mathit{dSet}[\mathcal{PT}]$. This search breaks `rest` into disjoint templates.

3.2 Symmetric Caching

The final part of our contribution is to develop a technique for utilizing symmetry in caching. Specifically, we develop a method where the cached bounds of a component (i.e., a template instance) can be reused as bounds for a symmetric version of the component. To accomplish this we adapt the techniques proposed in [6] to find symmetries between *templates*.

Each template ρT contains a subset of the constraints of the original COP. The variables in the scope of these constraints are either dependency variables or component variables of ρT. That is, $\bigcup_{c \in \mathit{consOf}[\rho T]} \mathit{scope}(c) = \mathit{dSet}[\rho T] \cup \mathit{varsOf}[\rho T]$. Using the techniques described in [6] we can, for each template, build a coloured graph representing the constraints, dependency variables, and component variables of the template, along with the original domains of these variables. Then using graph automorphism software (in our case NAUTY [5]) we can quite efficiently detect if two templates ρT_1 and ρT_2 are isomorphic when viewed as COPs. If they are then this means that every assignment of the variables in $\mathit{dSet}[\rho T_1] \cup \mathit{varsOf}[\rho T_1]$ can be mapped to an assignment of the variables in $\mathit{dSet}[\rho T_2] \cup \mathit{varsOf}[\rho T_2]$ such that these two assignments have identical values in the COPs specified by ρT_1 and ρT_2.

Once we have method for constructing a representation graph for each template, we further restrict the graph so that dependency variables are given a different colour from component variables. This forces any isomorphism between two templates to map dependency variables to dependency variables. With such a formulation, it can easily be shown that we can lookup bounds on instances of ρT_1 in ρT_2's symmetric cache: $\mathit{cache}[\rho T_1][\tau] = \mathit{cache}[\rho T_2][\sigma(\tau)]$.

To utilize this idea in **`CCS+BB`** we add some extra information to each template. At the time we build a new template we also construct its graph representation. We then search the previously stored templates to see if any of them are isomorphic. If an isomorphic template is found, we compute σ, the isomorphism between assignments to the dependency variables of the new template to assignments to the dependency variables of the old template. Finally, the cache interface of the new template is set so that it first applies σ then accesses the cache of the old template.

4 Empirical Results

We have implemented our approach and have tried it on the *Maximum Density Still Life* problem. For a full description of the problem, and the state of the art on this problem, see [4].

We solved Maximum Density Still Life using five different algorithms, all adapted from the EFC [3] solver, always using GAC as our constraint propagation algorithm. All experiments were run on a 2.2 GHz Pentium IV with 6GB of memory.

The algorithms we tested were. (1) a standard Branch and Bound algorithm (**BB**)). (2) Component Branch and Bound (**C+BB**). This version searches for components and

Size	**SCCS+BB**		**CCS+BB**		**T+BB**		**C+BB**		**BB**	
	nodes	*time*	*nodes*	*time*	*nodes*	*time*	*nodes*	*time*	*nodes*	*time*
4	1205	0	1739	0	1807	0	1806	0	899	0
5	7325	0	7555	0	8358	0	18632	0.24	6337	0
6	56044	0.5	105442	1.1	219546	2.0	189815	2.5	193757	1.5
7	739337	8.2	824354	9.2	$1.7*10^6$	16.9	$2.3*10^6$	34.33	$5.3*10^6$	42.1
8	$3.4*10^6$	39.4	$6.6*10^6$	73.8	$2.7*10^7$	254.6	$2.3*10^7$	332.2	$3.1*10^8$	2207
9	$7.1*10^7$	915.7	$9.7*10^7$	1234.0	$5.0*10^8$	5581.9	$4.2*10^8$	6951.2	NA	> 10000
10	$1.7*10^8$	2505.0	$5.6*10^8$	7845.4	NA	> 10000	NA	> 10000	NA	> 10000

Table 1. Nodes Searched and Time taken in CPU secs.

solves them separately. **C+BB** does not used templates, nor does it employ a cache. (3) Template Branch and Bound (**T+BB**) is a template version of **C+BB**. Its only improvement over **C+BB** is the use of templates to improve component detection. (4) Component Caching Search + Branch and Bound (**CCS+BB**) extends **T+BB** by activating the template cache. (5) Symmetric Component Caching Search + Branch and Bound (**SCCS+BB**) extends **CCS+BB** by allowing symmetric caching.

We see that decomposition (**C+BB**) yields significant improvements over standard branch and bound (**BB**) decreasing the size of the search tree. Since **T+BB** does not employ caching, it does not reduce the size of **C+BB**'s search tree (except for heuristic reasons), but **T+BB** is able to use template triggering to improve the efficiency of detecting components. **CCS+BB** makes a further improvement by activating the template caches, which results in a significant decrease in the size of the search space over **T+BB**. Finally, **SCCS+BB** makes an improvement by allowing symmetric caching which provides another significant decrease in the size of the search space.

5 Conclusions

We have presented an algorithm which incorporates search with decomposition, caching, and symmetric use of the cache, while mitigating much of the computational cost associated with such techniques.

References

1. F. Bacchus, S. Dalmao, and T. Pitassi. Algothims and complexity results for sat and bayesian inference. *FOCS 2003*, pages 340–351, 2003.
2. Adnan Darwiche. Recursive conditioning. *Artificial Intelligence*, 126:5–41, 2001.
3. G. Katsirelos. Efc constraint solver, 2004. http://www.cs.toronto.edu/gkatsi/efc.
4. J. Larrosa, E. Morancho, and D. Niso. On the practical applicability of bucket elimination: Still-life as a case study. *Journal of Artificial Intelligence Research*, 23:412–440, 2005. In *Workshop on Preferences and Soft Constraints*, 2005.
5. Brendan D. McKay. Nauty. available at http://cs.anu.edu.au/people/bdm/nauty/.
6. Jean-Francois Puget. Automatic detection of variable and value symmetries. In *International Conference on Principles and Practice of Constraint Programming*, pages 475–489, 2005.

Retroactive Ordering for Dynamic Backtracking

Students: Roie Zivan, Moshe Zazone, Uri Shapen
Supervisor: Amnon Meisels,
{zivanr,moshezaz,shapenko,am}@cs.bgu.ac.il

Department of Computer Science,
Ben-Gurion University of the Negev,
Beer-Sheva, 84-105, Israel

Abstract. Dynamic Backtracking (DBT) is a well known algorithm for solving Constraint Satisfaction Problems. In DBT, variables are allowed to keep their assignment during backjump, if they are compatible with the set of eliminating explanations. A previous study has shown that when DBT is combined with variable ordering heuristics it performs poorly compared to standard Conflict-directed Backjumping (CBJ) [1]. The special feature of DBT, keeping valid elimination explanations during backtracking, can be used for generating a new class of ordering heuristics. In the proposed algorithm, the order of already assigned variables can be changed. Consequently, the new class of algorithms is termed *Retroactive DBT*.
The proposed algorithm exploits the fact that a newly assigned variable can have a smaller current domain than variables which were assigned before it. The newly assigned variable can be moved to a position in front of assigned variables with larger domains and as a result prune the search space more effectively. The experimental results presented in this paper show an advantage of the new class of heuristics and algorithms over standard DBT and over CBJ. All algorithms tested were combined with forward-checking and used a *Min-Domain* heuristic.

1 Introduction

Conflict directed Backjumping (CBJ) is a technique which is known to improve the search of Constraint Satisfaction Problems (CSPs) by a large factor [4, 7]. Its efficiency increases when it is combined with forward checking [8]. The advantage of CBJ over standard backtracking algorithms lies in the use of conflict sets in order to prune unsolvable sub search spaces [8]. The down side of CBJ is that when such a backtrack (back-jump) is performed, assignments of variables which were assigned later than the culprit assignment are discarded.

Dynamic Backtracking [5] improves on standard CBJ by preserving assignments of non conflicting variables during back-jumps. In the original form of DBT, the culprit variable which replaces its assignment is moved to be the last among the assigned variables. In other words, the new assignment of the culprit variable must be consistent with all former assignments.

Although DBT saves unnecessary assignment attempts and therefore was proposed as an improvement to CBJ, a later study by Baker [1] has revealed a major drawback of DBT. According to Baker, when no specific ordering heuristic is used, DBT performs better than CBJ. However, when ordering heuristics which are known to improve the run-time of CSP search algorithms by a large factor are used [6, 2, 3], the performance of DBT is slower than the performance of CBJ. This phenomenon is easy to explain. Whenever the algorithm performs a back-jump it actually takes a variable which was placed according to the heuristic in a high position and moves it to a lower position. Thus, while in CBJ, the variables are ordered according to the specific heuristic, in DBT the order of variables becomes dependent on the algorithm's behavior [1].

In order to leave the assignments of non conflicting variables without a change on backjumps, DBT maintains a system of eliminating explanations ($Nogoods$) [5]. As a result, the DBT algorithm maintains dynamic domains for all variables and can potentially benefit from the *Min-Domain* (fail first) heuristic.

The present paper investigates a number of improvements to DBT that use radical versions of the *Min-Domain* heuristic. First, the algorithm avoids moving the culprit variable to the lowest position in the partial assignment. This alone can be enough to eliminate the phenomenon reported by Baker [1].

Second, the assigned variables which were originally ordered in a lower position than the culprit variable can be reordered according to their current domain size.

Third, a $retroactive$ ordering heuristic in which assigned variables are reordered is proposed. A *retroactive* heuristic allows an assigned variable to be moved upwards beyond assigned variables as far as the heuristic is justified.

If for example the variables are ordered according to the *Min-Domain* heuristic, the potential of each currently assigned variable to have a small domain is fully utilized. We note that although variables are chosen according to a *Min-Domain* heuristic, a newly assigned variable can have a smaller current domain than previously assigned variables. This can happen because of two reasons. First, as a result of *forward-checking* which might cause values from the current variables' domain to be eliminated due to conflicts with unassigned variables. Second, as a result of multiple backtracks to the same variable which eliminate at least one value each time. Therefore, the exploitation of the heuristic properties can be done, not only by choosing the next variable to be assigned, but by placing it in its *right* place among the assigned variables after it is assigned successfully.

The combination of the three ideas above was found to be successful in the empirical study presented in the present paper.

2 Retroactive Dynamic Backtracking

We assume in our presentation that the reader is familiar with both DBT following [1] and CBJ [8].

The first step in enhancing the desired heuristic (*Min-Domain* in our case) for DBT is to avoid the move forward in the resulting order, of variables that the algorithm back-tracks to (i.e. culprit variables). One way to do this is to try to replace the assignment of the culprit variable and *leave the variable in the same position*.

The second step is to reorder the assigned variables that have a lower order than the culprit assignment which was replaced. This step takes into consideration the possibility that the replaced assignment of a variable that lies higher in the order has the potential to change the size of the current domains of the already assigned variables that are ordered after it. The simplest way to perform this step is to reassign these variables using a *Min-Domain* heuristic.

```
Retroactive FC_DBT
1.  var_list ← variables;
2.  assigned_list ← $\phi$;
3.  pos ← 1;
4.  while (pos < N)
5.     next_var ← select_next_var(var_list);
6.     var_list.remove(next_var);
7.     assign(next_var);
8.  report solution;

procedure assign(var)
10. for each (value ∈ var.current_domain)
11.    var.assignment ← value;
12.    consistent ← true;
13.    forall (i ∈ var_list)
          and while consistent
14.       consistent ← check_forward(var, i);
15.    if not (consistent)
16.       nogood ← resolve_nogoods(i);
17.       store(var, nogood);
18.       undo_reductions(var, pos);
19.    else
20.       nogood ← resolve_nogoods(pos);
21.       lastVar ← $nogood.RHS\_variable$;
21.       newPos ← select_new_pos(var, lastVar);
22.       assigned_list.insert(var, newPos);
23.       forall (var_1 ∈ assigned_list) and
                (pos_var_1 > newPos)
24.          check_forward(var, var_1);
25.          update_nogoods(var, var_1);
26.       forall (var_2 ∈ var_list)
27.          update_nogoods(var, var_2);
28.       pos ← pos+1;
29.       return;
30. var.assignment ← $Nil$;
31. backtrack(var);
```

```
procedure backtrack($var$)
32. nogood ← resolve_nogoods(var);
33. if (nogood = $\phi$)
34.    report no solution;
35.    stop;
36. culprit ← $nogood.RHS\_variable$;
37. store(culprit, nogood);
38. culprit.assignment ← $Nil$;
39. undo_reductions(culprit, pos_culprit);
40. forall (var_1 ∈ assigned_list) and
          (pos_var_1 > newPos)
41.    undo_reductions(var_1, pos_var_1);
42.    var_1.assignment ← Nil;
43.    var_list.insert(var_1);
44.    assigned_list.remove(var_1);
45. pos ← pos_culprit;

procedure update_nogoods(var_1, var_2)
46. for each (val ∈
          {var_2.domain - var_2.current_domain})
47.    if not (check(var_2, val, var_1.assignment))
48.       nogood ← remove_eliminating_nogood
                (var_1, val);
49.       if not ($\exists var\_3$ ∈ nogood and
                pos_var_3 < pos_var_1)
50.          nogood ← ⟨var_1.assignment →
                var_2 ≠ val⟩;
51.       store(var_2, nogood);
```

Fig. 1. The Retroactive FC_DBT algorithm

The third step derives from the observation that in many cases the size of the current domain of a newly assigned variable is smaller than the current domains of variables which were assigned before it.

Allowing a reordering of assigned variables enables the use of heuristic information which was not available while the previous assignments have been performed. This takes the *Min-Domain* heuristic to a new level and generates a radical new approach. Variables can be moved up in the order, in front of assigned variables of the partial solution. As long as the new assignment is placed *after* the most recent assignment which is in conflict with one of the variable's values, the size of the domain of the assigned variable is not changed.

In the best ordering heuristic proposed by the present paper, the new position of the assigned variable in the order of the $partial\_solution$ is dependent on the size of its current domain. The heuristic checks all assignments from the last up to the first assignment which is included in the union of the newly assigned variable's eliminating Nogoods. The new assignment will be placed right after the first assigned variable with a smaller current domain.

Figures 1 presents the code of *Retroactive Forward Checking Dynamic Backtracking* ($Retro\_FC\_DBT$).

Introducing *Forward Checking* into *Retroactive* DBT is more complicated than in the case of standard DBT. After an assignment is performed all inconsistent values must be removed not only from the domains of unassigned variables but also from the domains of assigned variables with a lower priority than the new assignment.

2.1 Correctness of Retroactive DBT

For lack of space we only present an outline of the correctness proof of the *Retroactive DBT* algorithm. We first assume the correctness of the standard DBT algorithm (as proven in [5]) and prove that after the changes made for forward checking and for retroactive heuristics, it is still sound, complete and it terminates.

Soundness is immediate since after each successful assignment the $partial\_solution$ is consistent. Therefore, when the $partial\_solution$ includes an assignment for each variable the search is terminated and a consistent solution is reported. □

To prove the algorithm is complete we prove that the set of $Nogoods$ which can be generated by *Retroactive DBT* is included in the set of $Nogoods$ generated by *DBT*.

Last, we need to prove that the algorithm terminates. In order to do so, it is enough to show that the same $partial\_solution$ cannot be generated twice.

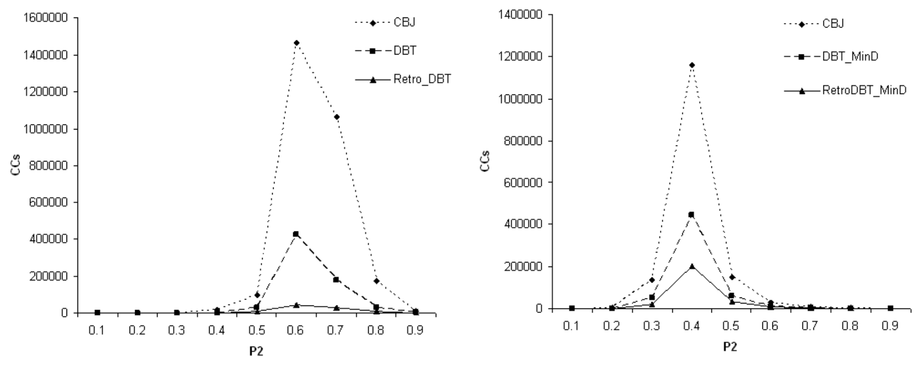
3 Experimental Evaluation

The common approach in evaluating the performance of *CSP* algorithms is to measure time in logical steps to eliminate implementation and technical parameters from affecting the results. The number of constraints checks serves as the measure in our experiments [9, 7].

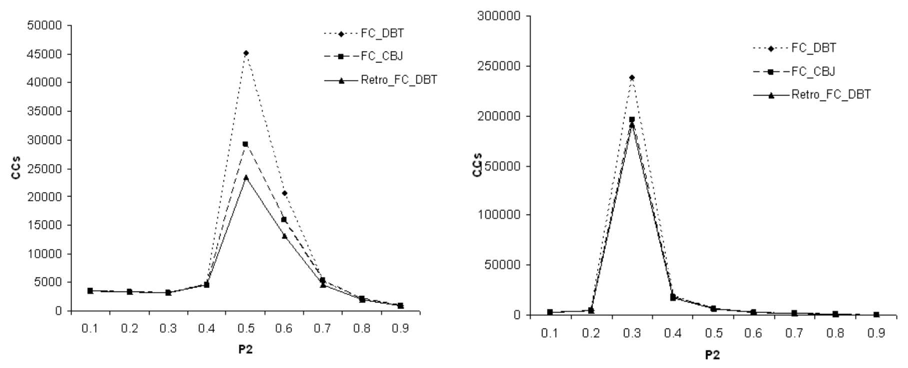
Experiments were conducted on random *CSPs* of n variables, k values in each domain, a constraints density of p_1 and tightness p_2 (which are commonly used in experimental evaluations of CSP algorithms [10]). Two sets of experiments were performed. In the first set the *CSPs* included 15 variables ($n = 15$) and in the second set the *CSPs* included 20 variables ($n = 20$). In all of our experiments the number of values for each variable was 10 ($k = 10$). Two values of constraints density were used, $p_1 = 0.3$ and $p_1 = 0.7$. The tightness value p_2, was varied between 0.1 and 0.9, in order to cover all ranges of problem difficulty. For each of the pairs of fixed density and tightness (p_1, p_2), 50 different random problems were solved by each algorithm and the results presented are an average of these 50 runs.

Three algorithms were compared, Conflict Based Backjumping (CBJ), Dynamic Backtracking (DBT) and Retroactive Dynamic Backtracking (*Retro* DBT). In all of our experiments all the algorithms use a *Min-Domain* heuristic for choosing the next variable to be assigned. In the first set of experiments, the three algorithms were implemented without forward-checking.

Figure 2 (a) presents the number of constraints checks performed by the three algorithms on low density CSPs ($p_1 = 0.3$). The CBJ algorithm does not benefit from the heuristic when it is not combined with forward-checking. The advantage of both



(a) (b)
Fig. 2. CCs performed by *DBT*, *CBJ* and *Retroactive DBT* (a) $p_1 = 0.3$, (b) $p_1 = 0.7$.



(a) (b)
Fig. 3. CCs performed by *FC_DBT*, *FC_CBJ* and *FC_Retroactive DBT* (a) $p_1 = 0.3$, (b) $p_1 = 0.7$.

versions of DBT over CBJ is therefore large. *Retroactive DBT* improves on standard *DBT* by a large factor as well. Figure 2 (b) present the results for high density CSPs ($p_1 = 0.7$). Although the results are similar, the differences between the algorithms are smaller for the case of higher density CSPs..

In our second set of experiments, each algorithm was combined with *Forward-Checking* [8]. This improvement enabled testing the algorithms on larger CSPs with 20 variables

Figure 3 (a) presents the number of constraints checks performed by each of the algorithms. It is very clear that the combination of *CBJ* with forward-checking improves the algorithm and makes it compatible with the others. This is easy to explain since the pruned domains as a result of forward-checking enable an effective use of the Min-Domain heuristic. Both *FC_CBJ* and *Retroactive FC_DBT* outperform *FC_DBT*. *Retroactive FC_DBT* performs better than *FC_CBJ*. Figures 3 (b) presents similar results for higher density CSPs. As before, the differences between the algorithms are smaller when solving CSPs with higher densities.

4 Discussion

Variable ordering heuristics such as *Min-Domain* are known to improve the performance of CSP algorithms $[6, 2, 3]$. This improvement results from a reduction in the search space explored by the algorithm. Previous studies have shown that DBT does not preserve the properties of variable ordering heuristics since it dynamically places variables during backtracking in a different position than the original position which was selected by the heuristic. As a result, DBT was found to perform poorly compared to CBJ [1]. The *Retroactive* DBT algorithm, presented in this paper, combines the advantages of both previous algorithms by preventing the placing of variables in a position which does not support the heuristic and allowing the reordering (or reassigning) of assigned variables with lower priority than the culprit assignment after a backtrack operation.

We have used the mechanism of *Dynamic Backtracking* which by maintaining eliminating $Nogoods$, allows variables with higher priority to be reassigned while lower priority variables keep their assignment. These dynamically maintained domains enable to take the *Min-Domain* heuristic to a new level. Standard backtracking algorithms use ordering heuristics only to decide on which variable is to be assigned next. *Retroactive* DBT enables the use of heuristics which reorder assigned variables. Since the sizes of the current domains of variables are dynamic during search, the flexibility of the heuristics which are possible in *Retroactive* DBT enables a dynamic enforcement of the *Min-Domain* property over assigned and unassigned variables.

The ordering of assigned variables requires some overhead in computation when the algorithm maintains consistency by using $Forward\ Checking$. This overhead was found by the experiments presented in this paper to be worth the effort since the overall computation effort is reduced.

References

[1] Andrew B. Baker. The hazards of fancy backtracking. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI '94), Volume 1*, pages 288–293, Seattle, WA, USA, July 31 - August 4 1994. AAAI Press.

[2] C. Bessiere and J.C. Regin. Using bidirectionality to speed up arc-consistency processing. *Constraint Processing (LNCS 923)*, pages 157–169, 1995.

[3] R. Dechter and D. Frost. Backjump-based backtracking for constraint satisfaction problems. *Artificial Intelligence*, 136:2:147–188, April 2002.

[4] Rina Dechter. *Constraint Processing*. Morgan Kaufman, 2003.

[5] M. L. Ginsberg. Dynamic backtracking. *J. of Artificial Intelligence Research*, 1:25–46, 1993.

[6] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[7] G. Kondrak and P. van Beek. A theoretical evaluation of selected backtracking algorithms. *Artificial Intelligence*, 21:365–387, 1997.

[8] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[9] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.

[10] B. M. Smith. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81:155 – 181, 1996.

A Simple Distribution-Free Approach to the Max k-Armed Bandit Problem

Student: Matthew J. Streeter[1]
Supervisor: Stephen F. Smith[2]

Computer Science Department and
Center for the Neural Basis of Cognition[1] and
The Robotics Institute[2]
Carnegie Mellon University
Pittsburgh, PA 15213
{matts,sfs}@cs.cmu.edu

Abstract. The max k-armed bandit problem is a recently-introduced online optimization problem with practical applications to heuristic search. Given a set of k slot machines, each yielding payoff from a fixed (but unknown) distribution, we wish to allocate trials to the machines so as to maximize the maximum payoff received over a series of n trials. Previous work on the max k-armed bandit problem has assumed that payoffs are drawn from *generalized extreme value* (GEV) distributions. In this paper we present a simple algorithm, based on an algorithm for the classical k-armed bandit problem, that solves the max k-armed bandit problem effectively without making strong distributional assumptions. We demonstrate the effectiveness of our approach by applying it to the task of selecting among priority dispatching rules for the resource-constrained project scheduling problem with maximal time lags (RCPSP/max).

1 Introduction

In the classical k-armed bandit problem one is faced with a set of k slot machines, each having an arm that, when pulled, yields a payoff drawn independently at random from a fixed (but unknown) distribution. The goal is to allocate trials to the arms so as to maximize the cumulative payoff received over a series of n trials. Solving the problem entails striking a balance between exploration (determining which arm yields the highest mean payoff) and exploitation (repeatedly pulling this arm).

In the max k-armed bandit problem, the goal is to maximize the *maximum* (rather than cumulative) payoff. This version of the problem arises in practice when tackling combinatorial optimization problems for which a number of randomized search heuristics exist: given k heuristics, each yielding a stochastic outcome when applied to some particular problem instance, we wish to allocate trials to the heuristics so as to maximize the maximum payoff (e.g., the maximum number of clauses satisfied by any sampled variable assignment, the minimum makespan of any sampled schedule). Cicirello and Smith (2005) show that a max

k-armed bandit approach yields good performance on the resource-constrained project scheduling problem with maximum time lags (RCPSP/max).

1.1 Motivations

All previous work on the max k-armed bandit problem has assumed that payoffs are drawn from *generalized extreme value* (GEV) distributions. A random variable Z has a GEV distribution if

$$\mathbb{P}[Z \leq z] = \exp\left(-\left(1 + \frac{\xi(z-\mu)}{\sigma}\right)^{-\frac{1}{\xi}}\right)$$

for some constants μ, $\sigma > 0$, and ξ. The assumption is justified by the Extremal Types Theorem [5], which singles out the GEV as the limiting distribution of the maximum of a large number of independent identically distributed (i.i.d.) random variables.

In this work, we do not assume that the payoff distributions belong to any specific parametric family. Roughly speaking, our approach will work best when the following two criteria are satisfied.

1. There is a (relatively low) threshold $t_{critical}$ such that, for all $t > t_{critical}$, the arm that is most likely to yield a payoff $> t$ is the same as the arm most likely to yield a payoff $> t_{critical}$. Call this arm i^*.
2. As t increases beyond $t_{critical}$, there is a growing gap between the probability that arm i^* yields a payoff $> t$ and the corresponding probability for other arms. Specifically, if we let $p_i(t)$ denote the probability that the i^{th} arm returns a payoff $> t$, the ratio $\frac{p_{i^*}(t)}{p_i(t)}$ should increase as a function of t for $t > t_{critical}$, for any $i \neq i^*$.

Figure 1 illustrates a set of two payoff distributions that satisfy these assumptions.

1.2 Related Work

The classical k-armed bandit problem was first studied by Robbins [10] and has since been the subject of numerous papers; see Berry and Fristedt [2] and Kaelbling [6] for overviews.

The max k-armed bandit problem was introduced by Cicirello and Smith [3, 4], whose experiments with randomized priority dispatching rules for the RCPSP/max form the basis of our experimental evaluation in §4.

2 Chernoff Interval Estimation

In this section we present and analyze a simple algorithm, Chernoff Interval Estimation, for the classical k-armed bandit problem. In §3 we use this approach as the basis for Threshold Ascent, an algorithm for the max k-armed bandit

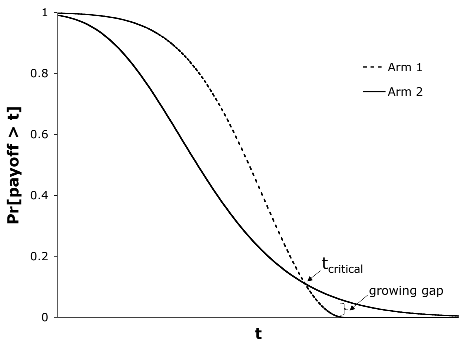


Fig. 1. A max k-armed bandit instance on which Threshold Ascent should perform well.

problem. Chernoff Interval Estimation is simply the well-known interval estimation algorithm [6, 7] with confidence intervals derived using Chernoff's inequality. Despite its simplicity, the algorithm's regret bound is state of the art. In particular, when the mean payoff returned by each arm is small (relative to the maximum possible payoff) our algorithm has much better performance than the recent algorithm of [1], which is identical to our algorithm except that confidence intervals are derived using Hoeffding's inequality.

We assume we are given a budget of n pulls and that there are k arms, each of which returns payoffs between 0 and 1. We denote by μ_i the (unknown) mean payoff returned by the i^{th} arm, and define $\mu^* = \max_{1 \leq i \leq k} \mu_i$.

```
Procedure **ChernoffIntervalEstimation**($n$, $\delta$):
  1. Initialize $x_i \leftarrow 0$, $n_i \leftarrow 0$ $\forall i \in \{1, 2, \ldots, k\}$.
  2. Repeat $n$ times:
     (a) $\hat{i} \leftarrow \arg\max_i U(\bar{\mu}_i, n_i)$, where $\bar{\mu}_i = \frac{x_i}{n_i}$ and
         $$U(\mu, n) = \begin{cases} \mu + \frac{\alpha + \sqrt{2n\mu\alpha + \alpha^2}}{n} & \text{if } n > 0 \\ \infty & \text{otherwise} \end{cases}$$
         where $\alpha = \ln\left(\frac{2nk}{\delta}\right)$.
     (b) Pull arm $\hat{i}$, receive payoff $R$, set $x_{\hat{i}} \leftarrow x_{\hat{i}} + R$, and set
         $n_i \leftarrow n_i + 1$.
```

Lemma 1. *During a run of ChernoffIntervalEstimation(n, δ) it holds with probability at least $1 - \frac{\delta}{2}$ that for all arms $i \in \{1, 2, \ldots, k\}$ and for all n repetitions of the loop, $U(\bar{\mu}_i, n_i) \geq \mu_i$.*

Proof. Omitted. □

Lemma 2. *During a run of ChernoffIntervalEstimation*(n, δ) *it holds with probability at least* $1-\delta$ *that each suboptimal arm* i *(i.e., each arm* i *with* $\mu_i < \mu^*$*) is pulled at most* $\frac{3\alpha}{\mu^*}\frac{1}{(1-\sqrt{y_i})^2}$ *times, where* $y_i = \frac{\mu_i}{\mu^*}$ *and* $\alpha = \ln\left(\frac{2nk}{\delta}\right)$*.*

Proof. Omitted. □

Theorem 1. *The expected regret incurred by ChernoffIntervalEstimation*(n, δ) *is at most*

$$2\sqrt{3\mu^* n(k-1)\alpha} + \delta\mu^* n$$

where $\alpha = \ln\left(\frac{2nk}{\delta}\right)$*.*

Proof. Omitted. □

3 Threshold Ascent

To solve the max k-armed bandit problem, we use Chernoff Interval Estimation to maximize the number of payoffs that exceed a threshold T that varies over time. Initially, we set T to zero. Whenever s or more payoffs $> T$ have been received so far, we increment T. We refer to the resulting algorithm as Threshold Ascent. The code for Threshold Ascent is given below. For simplicity, we assume that all payoffs are integer multiples of some known constant Δ.

```
Procedure **ThresholdAscent**($s$, $n$, $\delta$):
  1. Initialize $T \leftarrow 0$ and $n_i^R = 0$, $\forall i \in \{1, 2, \ldots, k\}, R \in$
     $\{0, \Delta, 2\Delta, \ldots, 1-\Delta, 1\}$.
  2. Repeat $n$ times:
     (a) While $\sum_{i=1}^{k} S_i(T) \geq s$ do:

                 $T \leftarrow T + \Delta$

         where $S_i(t) = \sum_{R>t} n_i^R$ is the number of payoffs $> t$
         received so far from arm $i$.
     (b) $\hat{\imath} \leftarrow \arg\max_i U\left(\frac{S_i(T)}{n_i}, n_i\right)$, where $n_i = \sum_R n_i^R$ and

         $U(\mu, n) = \begin{cases} \mu + \frac{\alpha + \sqrt{2n\mu\alpha + \alpha^2}}{n} & \text{if } n > 0 \\ \infty & \text{otherwise} \end{cases}$

         where $\alpha = \ln\left(\frac{2nk}{\delta}\right)$.
     (c) Pull arm $\hat{\imath}$, receive payoff $R$, and set $n_{\hat{\imath}}^R \leftarrow n_{\hat{\imath}}^R + 1$.
```

The parameter s controls the tradeoff between exploration and exploitation. To understand this tradeoff, it is helpful to consider two extreme cases.

Case $s = 1$. ThresholdAscent$(1, n, \delta)$ is equivalent to round-robin sampling. When $s = 1$, the threshold T is incremented whenever a payoff $> T$ is obtained. Thus the value $\frac{S_i(T)}{n_i}$ calculated in 2 (b) is always 0, so the value of

$U\left(\frac{I_i(T)}{n_i}, n_i\right)$ is determined strictly by n_i. Because U is a decreasing function of n_i, the algorithm simply samples whatever arm has been sampled the smallest number of times so far.

Case $s = \infty$. ThresholdAscent(∞, n, δ) is equivalent to ChernoffIntervalEstimation (n, δ) running on a k-armed bandit instance where payoffs $> T$ are mapped to 1 and payoffs $\leq T$ are mapped to 0.

4 Evaluation on the RCPSP/max

Following Cicirello and Smith [3, 4], we evaluate our algorithm for the max k-armed bandit problem by using it to select among randomized priority dispatching rules for the resource-constrained project scheduling problem with maximal time lags (RCPSP/max). We consider the five randomized priority dispatching rules in the set $\mathcal{H} = \{LPF, LST, MST, MTS, RSM\}$. See Cicirello and Smith [3, 4] for a complete description of these heuristics.

Briefly, in the RCPSP/max one must assign start times to each of a number of activities in such a way that certain temporal and resource constraints are satisfied. Such an assignment of start times is called a *feasible schedule*. The goal is to find a feasible schedule whose makespan is as small as possible, where makespan is defined as the maximum completion time of any activity. For a more complete description, see [9].

4.1 Results

We evaluate our approach on a set $\mathcal{I}$ of 169 RCPSP/max instances from the ProGen/max library [11]. For each instance $I \in \mathcal{I}$, we ran each heuristic $h \in \mathcal{H}$ 10,000 times, storing the results in a file. Using this data, we created a set $\mathcal{K}$ of 169 five-armed bandit problems (each of the five heuristics $h \in \mathcal{H}$ represents an arm). After the data were collected, makespans were converted to payoffs by multiplying each makespan by -1 and scaling them to lie in the interval $[0, 1]$.

For each instance $K \in \mathcal{K}$, we ran three max k-armed bandit algorithms, each with a budget of $n = 10,000$ pulls: Threshold Ascent with parameters $n = 10,000$, $s = 100$, and $\delta = 0.01$, the QD-BEACON algorithm of Cicirello and Smith [4], and an algorithm that simply sampled the arms in a round-robin fashion. Cicirello and Smith describe three versions of QD-BEACON; we use the one based on the GEV distribution. For each instance $K \in \mathcal{K}$, we define the *regret* of an algorithm as the difference between the minimum makespan (which corresponds to the maximum payoff) sampled by the algorithm and the minimum makespan sampled by any of the five heuristics (on any of the $10,000$ stored runs of each of the five heuristics). Table 1 summarizes our results.

Examining Table 1, we see that of the eight max k-armed bandit strategies we evaluated (Threshold Ascent, QD-BEACON, round-robin sampling, and the five pure strategies), Threshold Ascent has the least regret and achieves zero regret on the largest number of instances.

Table 1. Performance of eight heuristics on 169 RCPSP/max instances.

Heuristic	Σ **Regret**	$\mathbb{P}$[Regret = 0]
Threshold Ascent	188	0.722
Round-robin sampling	345	0.556
LPF	355	0.675
MTS	402	0.657
QD-BEACON	609	0.538
RSM	2130	0.166
LST	3199	0.095
MST	4509	0.107

Acknowledgment. The work reported in this paper was sponsored in part by the National Science Foundation under contract #9900298 and the CMU Robotics Institute.

References

1. Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47:235–256, 2002a.
2. Donald. A. Berry and Bert Fristedt. *Bandit Problems: Sequential Allocation of Experiments.* Chapman and Hall, London, 1986.
3. Vincent A. Cicirello and Stephen F. Smith. Heuristic selection for stochastic search optimization: Modeling solution quality by extreme value theory. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, pages 197–211, 2004.
4. Vincent A. Cicirello and Stephen F. Smith. The max k-armed bandit: A new model of exploration applied to search heuristic selection. In *Proceedings of AAAI 2005*, pages 1355–1361, 2005.
5. Stuart Coles. *An Introduction to Statistical Modeling of Extreme Values.* Springer-Verlag, London, 2001.
6. Leslie P. Kaelbling. *Learning in Embedded Systems.* The MIT Press, Cambridge, MA, 1993.
7. Tze Leung Lai. Adaptive treatment allocation and the multi-armed bandit problem. *The Annals of Statistics*, 15(3):1091–1114, 1987.
8. Rolf H. Möhring, Andreas S. Schulz, Frederik Stork, and Marc Uetz. Solving project scheduling problems by minimum cut computations. *Management Science*, 49(3):330–350, 2003.
9. Klaus Neumann, Christoph Schwindt, and Jürgen Zimmerman. *Project Scheduling with Time Windows and Scarce Resources.* Springer-Verlag, 2002.
10. Herbert Robbins. Some aspects of sequential design of experiments. *Bulletin of the American Mathematical Society*, 58:527–535, 1952.
11. C. Schwindt. Generation of resource–constrained project scheduling problems with minimal and maximal time lags. Technical Report WIOR-489, Universität Karlsruhe, 1996.

Improving the Performance of Ant Algorithms using Constraint Programming

Student: Broderick Crawford[1,2] and Supervisor: Carlos Castro[2]

[1] Engineering Informatic School, Pontifical Catholic University of Valparaíso, Chile
[2] Informatic Department, Federico Santa María Technical University, Chile
broderick.crawford@ucv.cl

Abstract. In this paper, we focus on the resolution of Crew Pairing Optimization formulated as Set Partitioning Problem. We try to solve it with Ant Colony Optimization algorithms and Hybridizations of Ant Colony Optimization with Constraint Programming techniques. We recognize the difficulties of pure Ant Algorithms solving strongly constrained problems. Therefore, we explore the addition of Constraint Programming mechanisms in the construction phase of the ants so they can complete their solutions. Computational results solving some test instances of the problem are presented showing the advantages to use this kind of hybridization.

Key words. Ant Colony Optimization (ACO), Constraint Programming (CP), Crew Pairing Optimization, Arc Consistency (AC), Set Partitioning Problem (SPP).

1 Introduction

The Crew Pairing Optimization has been investigated for many years and this problem continues challenging both scientists and software engineers. The basic problem is to partition a given schedule of airline flights into individual flight sequences called pairings. The pairing problem can be formulated as Set Partitioning Problem (SPP) or equality-constrained Set Covering Problem (SCP) [2]. In this work, we solve some test instances of Airline Flight Crew Scheduling with Ant Colony Optimization (ACO) algorithms and some hybridizations of ACO with Constraint Programming (CP). There exist problems for which the effectiveness of ACO is limited, among them the SPP [13]. The best performing metaheuristic for SPP is a genetic algorithm due to Chu and Beasley [4]. There already exists some first approaches applying ACO to the SCP [11]. More recent works [10, 12, 9] apply Ant Systems to the SCP and related problems. Trying to solve larger instances of SPP with ACO derives in a lot of unfeasible labeling of variables, and the ants can not obtain complete solutions using their classic transition rule. In this paper, we propose the addition of a lookahead mechanism in the construction phase of ACO thus only feasible partial solutions are generated. The lookahead mechanism allows the incorporation of information about the instantation of variables after the current decision. The idea differs from that proposed by [15] and [8], these authors propose a look ahead function evaluating the pheromone in the Shortest Common Supersequence Problem and estimating the quality of a partial solution of a Industrial Scheduling Problem respectively.

This paper is organised as follows: Section 2 is dedicated to the presentation of the problem. In Section 3, we describe the applicability of the ACO algorithms for SPP. In Section 4, we present the basic concepts to adding Constraint Programming techniques to ACO. In Section 5, we present results solving some benchmarks available in the OR-Library of Beasley [3]. Finally, in Section 6 we conclude the paper and give some perspectives for future research.

2 Problem Description

The resource planning in airlines is a very complex task, and without considering the fuel costs, the most important direct operating cost is the personnel. The planning and scheduling of crews is usually considered as two optimization problems: the crew pairing problem and the crew assignment problem (or rostering problem) [1]. In this paper we focus on the pairing problem modelled under the assumption that the set of feasible pairings and their costs are explicitly available, and can be expressed as a Set Partitioning Problem. SPP is the NP-complete problem of partitioning a given set into mutually independent subsets while minimizing a cost function defined as the sum of the costs associated to each of the eligible subsets. In the SPP matrix formulation we are given a $m \times n$ matrix $A = (a_{ij})$ in which all the matrix elements are either zero or one. Additionally, each column is given a non-negative cost c_j. We say that a column j can cover a row i if $a_{ij} = 1$. Let J denotes the set of the columns and x_j a binary variable which is one if column j is chosen and zero otherwise. The SPP can be defined formally as follows:

$$Minimize \qquad f(x) = \sum_{j=1}^{n} c_j \times x_j \tag{1}$$

$$Subject\ \ to \qquad \sum_{j=1}^{n} a_{ij} \times x_j = 1; \qquad \forall i = 1, \ldots, m \tag{2}$$

In this formulation, each row represents a flight leg that must be scheduled. The columns represent pairings. Each pairing x_j is a sequence of flights to be covered by a single crew over a 2 to 3 day period. It must begin and end in the base city where the crew resides. The optimization problem is to select the partition of the minimum cost from a pool of candidate pairings.

3 Ant Colony Optimization for Set Partitioning Problems

The basic idea of ACO algorithms comes from the capability of real ants to find shortest paths between the nest and food source. From a Combinatorial Optimization point of view the ants are looking for *good solutions*. Real ants cooperate in their search for food by depositing pheromone on the ground. An artificial ant colony simulates this behavior implementing artificial ants as parallel processes whose role is to build solutions using a randomized constructive search driven by pheromone trails and heuristic information of the problem. An important topic in ACO is the adaptation of the pheromone trails during algorithm execution to take into account the cumulated search experience: reinforcing the pheromone associated with components in good solutions and considering the *evaporation* of the pheromone over time in order to avoid premature convergence. ACO can be applied in a very straightforward way to SPP. The columns are chosen as the solution components and have associated a cost and a pheromone trail [6]. Each column can be visited by an ant only once and until a final solution has to cover all rows. A walk of an ant corresponds to the iterative addition of columns to the partial solution obtained so far. Each ant starts with an empty solution and adds columns until all rows are covered. A pheromone trail τ_j and a heuristic information η_j are associated to each eligible column j. A column to be added is chosen with a probability that depends of its pheromone trail and the heuristic information [6, 12]. The most common form of the ACO decision policy (*Transition Rule Probability*) when ants work with components is:

$$p_j^k(t) = \frac{\tau_j * \eta_j^\beta}{\sum\limits_{l \notin S^k} \tau_l [\eta_l]^\beta} \quad \text{if } j \notin S^k \tag{3}$$

where S^k is the partial solution of the ant k. The β parameter controls how important is η in the probabilistic decision. Setting good pheromone quantity is not a trivial task [11], iteratively the initial pheromone deposited in each component will be increased in relation to the frequency of the component in the ants solutions and decreased by evaporation over time. In this work we divided this frequency by the number of ants obtaining better results. In this paper we use a dynamic heuristic information that depends on the partial solution of an ant. It can be defined as $\eta_j = \frac{e_j}{c_j}$, where e_j is the so called cover value, that is, the number of additional rows covered when adding column j to the current partial solution, and c_j is the cost of column j [6]. In other words, the heuristic information measures the unit cost of covering one additional row.

But to determine if a column actually belongs or not to the partial solution (j $\notin S^k$) is not good enough. The traditional ACO decision policy, Equation 3, does not work for SPP because the ants, in this traditional selection process of the next columns, ignore the information of the problem constraints. And in the worst case, in the iterative steps is possible to assign values to some variable that will make impossible to obtain complete feasible solutions.

In this work, we use two instances of ACO: Ant System (AS) and Ant Colony System (ACS) algorithms, the original and the most famous algorithms in the ACO family [6]. In this paper we explore the addition of a lookahead mechanism in the construction phase of ACO, the technique that we used is one of the two basic techniques of Constraint Programming: Constraint Propagation. Also called Local Consistency, Consistency Enforcing, Filtering or Narrowing Domain Algorithms. Constraint Propagation is an efficient inference mechanism based on the use of the information in the constraints. The two basic techniques of Constraint Programming are *Constraint Propagation* and *Constraint Distribution.* The problem cannot be solved using constraint propagation alone, Constraint Distribution or Search is required to reduce the search space until constraint propagation is able to determine the solution. Constraint distribution splits a problem into complementary cases once constraint propagation cannot advance further. By iterating propagation and distribution, propagation will eventually determine the solutions of a problem.

4 ACO with Constraint Programming

Recently, some efforts have been done in order to integrate Constraint Programming techniques to ACO algorithms [14, 7]. An hibridization of ACO and CP can be approached from two directions: we can either take ACO or CP as the base algorithm and try to embed the respective other method into it. A form to integrate CP into ACO is to let it reduce the possible candidates of the not yet instantiated variables participating in the same constraints that the actual variable. A different approach would be to embed ACO within CP. The point at which ACO can interact with CP is during the labeling phase, using ACO to learn a value ordering that is more likely to produce good solutions. In this work, ACO uses CP in the variable selection (when adding columns to partial solution). The CP algorithm used in this paper is Forward Checking with Backtracking. The algorithm is a combination of Arc Consistency Technique and Chronological Backtracking [5]. It performs Arc Consistency between pairs of a not yet instantiated variable and an instantiated variable, i.e., when a value is assigned to the current variable, any value in the domain of a future variable which conflicts with this assignment is removed from the domain. Adding Forward Checking to ACO for SPP means that columns are chosen if they do not produce any conflict with the next column to be chosen. This reduces the search tree and the overall amount of computational work done. But it should be noted that in comparison with pure ACO algorithm, Forward Checking does additional work when each assignment is intended to be added to the current partial solution. Arc consistency enforcing always increases the information available on each variable labeling.

5 Experiments and Results

Table 1 presents results when adding Forward Checking to the basic ACO algorithms for solving test instances taken from the OR-Library [3]. The algorithms has been run with the following parameters setting: influence of pheromone (alpha)=1.0, influence of heuristic information (beta)=0.5 and evaporation rate (rho)=0.4 as suggested in [11, 12, 6]. The number of ants has been set to 120 and the maximum number of iterations to 160, so that the number of generated candidate solutions is limited to 19.200. For ACS the list size was 500 and

```
Procedure ACO+CP_for_SPP
 Begin
  InitParameters();
  While (remain iterations) do
     For k := 1 to nants do
       While (solution is not completed) and TabuList <> J do
         Choose next Column j with Transition Rule Probability
         For each Row i covered by j do                 /* constraints with j          */
           feasible(i):= Posting(j);                    /* Constraint Propagation      */
         EndFor
         If feasible(i) for all i then AddColumnToSolution(j)
                                  else Backtracking(j); /* set j uninstantiated        */
         AddColumnToTabuList(j); /* TabuList = columns in partial solution or conflict*/
       EndWhile
     EndFor
    UpdateOptimum();
    UpdatePheromone();
  EndWhile
  Return best_solution_founded
 End.
```

Fig. 1. ACO+CP ALGORITHM FOR SPP

Qo=0.5. Algorithms were implemented using ANSI C, GCC 3.3.6, under Microsoft Windows XP Professional version 2002. The effectiveness of Constraint Programming improving the performance of Ant Algorithms is shown to the SPP. Because the SPP is so strongly constrained the stochastic behavior of ACO was improved with lookahead techniques in the construction phase, so that almost only feasible partial solutions are induced.

Problem	Rows(Constraints)	Columns(Variables)	Optimum	Density	AS	ACS	AS+FC	ACS+FC
sppnw06	50	6774	7810	18.17	9200	9788	8160	8038
sppnw08	24	434	35894	22.39	X	X	35894	36682
sppnw09	40	3103	67760	16.20	70462	X	70222	69332
sppnw10	24	853	68271	21.18	X	X	X	X
sppnw12	27	626	14118	20.00	15406	16060	14466	14252
sppnw15	31	467	67743	19.55	67755	67746	67743	67743
sppnw19	40	2879	10898	21.88	11678	12350	11060	11858
sppnw23	19	711	12534	24.80	14304	14604	13932	12880
sppnw26	23	771	6796	23.77	6976	6956	6880	6880
sppnw32	19	294	14877	24.29	14877	14886	14877	14877
sppnw34	20	899	10488	28.06	13341	11289	10713	10797
sppnw39	25	677	10080	26.55	11670	10758	11322	10545
sppnw41	17	197	11307	22.10	11307	11307	11307	11307

Table 1. ACO WITH FORWARD CHECKING RESULTS. Table shows Problem code, Number of rows (constraints), Number of columns (decision variables), Best known solution, Density, and Costs obtained when applying AS and ACS with FC. An entry of "X" in the table means no feasible solution was found.

6 Conclusions and Future Directions

We have successfully combined Constraint Programming and ACO for the problem of set partitioning solving benchmarks of data sets. Our main conclusion from this work is that we can improve ACO with CP. We have shown that it is possible to add Arc Consistency to any ACO algorithms and the computational results confirm that the performance of ACO is possible to improve with this type of hibridization. Future versions of the algorithm will study the pheromone treatment representation and the incorporation of available techniques in order to reduce the input problem (Pre Processing) and improve the solutions given by the ants (Post Processing). We also plan to extend our hybridization to improve the ants solutions by other local search methods like Hill Climbing, Simulated Annealing or Tabu Search.

References

1. E. Andersson, E. Housos, N. Kohl and D. Wedelin. Crew Pairing Optimization. *In Yu G.(ed.) Operations Research in the Airline Industry*, Kluwer Academic Publishing, 1998.
2. E. Balas and M. Padberg. Set Partitioning: A Survey. *SIAM Review*, vol 18, pp 710–760, 1976.
3. J. E. Beasley. OR-Library:Distributing test problem by electronic mail. *Journal of Operational Research Society*, vol 41(11), pp 1069–1072, 1990.
4. P. C. Chu and J. E. Beasley. Constraint handling in genetic algorithms: the set partitoning problem. *Journal of Heuristics*, vol 4, pp 323–357, 1998.
5. R. Dechter and D. Frost. Backjump-based Backtracking for Constraint Satisfaction Problems. *Artificial Intelligence*, vol 136, pp 147–188, 2002.
6. M. Dorigo and T. Stutzle. *Ant Colony Optimization.* MIT Press, USA, 2004.
7. F. Focacci, F. Laburthe and A. Lodi. Local Search and Constraint Programming. In *Handbook of metaheuristics*, Kluwer, 2002.
8. C. Gagne, M. Gravel and W.L. Price. A Look-Ahead Addition to the Ant Colony Optimization Metaheuristic and its Application to an Industrial Scheduling Problem. In J.P. Sousa et al., eds., *Proceedings of the fourth Metaheuristics International Conference MIC'01*, pp 79–84, 2001.
9. X. Gandibleux, X. Delorme and V. T'Kindt. An Ant Colony Algorithm for the Set Packing Problem. In M. Dorigo et al., editor, *Proceedings of ANTS 2004*, vol 3172 of *LNCS*, pp 49–60. Springer, 2004.
10. R. Hadji, M. Rahoual, E. Talbi and V. Bachelet. Ant colonies for the set covering problem. In M. Dorigo et al., editor, *Proceedings of ANTS 2000*, vol 1838 of *LNCS*, pp 63–66. Springer, 2000.
11. G. Leguizamón and Z. Michalewicz. A new version of Ant System for subset problems. In *Proceedings of Congress on Evolutionary Computation CEC'99*, pp 1459–1464, Piscataway, NJ, USA, 1999. IEEE Press.
12. L. Lessing, I. Dumitrescu and T. Stutzle. A Comparison Between ACO Algorithms for the Set Covering Problem. In M. Dorigo et al., editor, *Proceedings of ANTS 2004*, vol 3172 of *LNCS*, pp 1–12. Springer, 2004.
13. V. Maniezzo and M. Milandri. An Ant-Based Framework for Very Strongly Constrained Problems. In M. Dorigo et al., editor, *Proceedings of ANTS 2002*, vol 2463 of *LNCS*, pp 222–227. Springer, 2002.
14. B. Meyer and A. Ernst. Integrating ACO and Constraint Propagation. In M. Dorigo et al., editor, *Proceedings of ANTS 2004*, vol 3172 of *LNCS*, pp 166–177. Springer, 2004.
15. R. Michel and M. Middendorf. An Island model based Ant system with lookahead for the shortest supersequence problem. In *Proceedings of PPSN 1998*, vol 1498 of *LNCS*, pp 692–701. Springer, 1998.

Inferring Variable Conflicts for Local Search[⋆]

Student: Magnus Ågren
Supervisors: Pierre Flener and Justin Pearson

Department of Information Technology, Uppsala University, Sweden
{agren,pierref,justin}@it.uu.se

Abstract. For efficiency reasons, neighbourhoods in local search are often shrunk by only considering moves modifying variables that actually contribute to the overall penalty. These are known as conflicting variables. We propose a new definition for measuring the conflict of a variable in a model and apply it to the set variables of models expressed in existential second-order logic extended with counting ($\exists\text{SOL}^+$). Such a variable conflict can be automatically and incrementally evaluated. Furthermore, this measure is lower-bounded by an intuitive conflict measure, and upper-bounded by the penalty of the model. We also demonstrate the usefulness of the approach by replacing a built-in global constraint by an $\exists\text{SOL}^+$ version thereof, while still obtaining competitive results.

1 Introduction

In local search, it is often important to limit the size of the neighbourhood by only considering moves modifying conflicting variables, i.e., variables that actually contribute to the overall penalty. See [4, 6, 8], for example.

We address the inference of variable conflicts from a formulation of a constraint. After giving necessary background information in Section 2, we propose in Section 3 a new definition for measuring the conflict of a variable and apply it to the *set* variables of models expressed in existential second-order logic extended with counting ($\exists\text{SOL}^+$) [5]. Such a variable conflict can be automatically and incrementally evaluated. The calculated value is lower-bounded by an intuitive target value, namely the maximum penalty decrease of the model that may be achieved by only changing the given variable, and upper-bounded by the penalty of the model. We demonstrate the usefulness of the approach in Section 4 by replacing a built-in constraint by an $\exists\text{SOL}^+$ version, while still obtaining competitive results.

2 Preliminaries

As usual, a *constraint satisfaction problem* (*CSP*) is a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X}$ is a finite set of variables, $\mathcal{D}$ is a finite set of domains, each $D_x \in \mathcal{D}$ containing the set of possible values for $x \in \mathcal{X}$, and $\mathcal{C}$ is a finite set of constraints, each being defined on a subset of $\mathcal{X}$ and specifying their valid combinations of values.

[⋆] This research was partially funded by EuroControl project C/1.246/HQ/JC/04.

A variable $S \in \mathcal{X}$ is a *set variable* if its corresponding domain D_S is $2^{\mathcal{U}}$, where $\mathcal{U}$ is a common finite set of values of some type, called the *universe*.

Local search iteratively makes a small change to a current assignment of values to *all* variables (configuration), upon examining the merits of many such changes, until a solution is found or allocated resources have been exhausted. The configurations examined constitute the neighbourhood of the current configuration, crucial guidance being provided by penalties and variable conflicts.

Definition 1. *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP. A* configuration *for P (or $\mathcal{X}$) is a total function $k : \mathcal{X} \to \bigcup_{D \in \mathcal{D}} D$. We use $\mathcal{K}$ to denote the* set of all configurations *for a given CSP or set of variables, depending on the context. A* neighbourhood function *for P is a function $n : \mathcal{K} \to 2^{\mathcal{K}}$. The* neighbourhood *of P with respect to (w.r.t.) a configuration $k \in \mathcal{K}$ and n is the set $n(k)$. The* variable neighbourhood *for $x \in \mathcal{X}$ w.r.t. k is the subset of $\mathcal{K}$ reachable from k by changing $k(x)$ only: $n_x(k) = \{\ell \in \mathcal{K} \mid \forall y \in \mathcal{X} : y \neq x \to k(y) = \ell(y)\}$. A* penalty function *of a constraint $c \in \mathcal{C}$ is a function $penalty(c) : \mathcal{K} \to \mathbb{N}$ such that (s.t.) $penalty(c)(k) = 0$ if and only if (iff) c is satisfied w.r.t. k. The* penalty *of c w.r.t. k is $penalty(c)(k)$. A* conflict function *of c is a function $conflict(c) : \mathcal{X} \times \mathcal{K} \to \mathbb{N}$ s.t. if $conflict(c)(x,k) = 0$ then $\forall \ell \in n_x(k) : penalty(c)(k) \leq penalty(c)(\ell)$. The* conflict *of x w.r.t. c and k is $conflict(c)(x,k)$.*

Example 1. Let $P = \langle \{S, T\}, \{D_S, D_T\}, \{S \subset T\} \rangle$ where $D_S = D_T = 2^{\mathcal{U}}$ and $\mathcal{U} = \{a, b, c\}$. A configuration for P is given by $k(S) = \{a, b\}$ and $k(T) = \emptyset$, or equivalently by $k = \{S \mapsto \{a,b\}, T \mapsto \emptyset\}$. The neighbourhood of P w.r.t. k and the neighbourhood function for P that moves an element from S to T is the set $\{k_a = \{S \mapsto \{b\}, T \mapsto \{a\}\}, k_b = \{S \mapsto \{a\}, T \mapsto \{b\}\}$. The variable neighbourhood for S w.r.t. k is the set $n_S(k) = \{k, k_1 = \{S \mapsto \emptyset, T \mapsto \emptyset\}, k_2 = \{S \mapsto \{a\}, T \mapsto \emptyset\}, k_3 = \{S \mapsto \{b\}, T \mapsto \emptyset\}, k_4 = \{S \mapsto \{c\}, T \mapsto \emptyset\}, k_5 = \{S \mapsto \{a,c\}, T \mapsto \emptyset\}, k_6 = \{S \mapsto \{b,c\}, T \mapsto \emptyset\}, k_7 = \{S \mapsto \{a,b,c\}, T \mapsto \emptyset\}\}$. Let the penalty and conflict functions of $S \subset T$ be defined by:

$$penalty(S \subset T)(k) = |k(S) \setminus k(T)| + \begin{cases} 1, & \text{if } k(T) \subseteq k(S) \\ 0, & \text{otherwise} \end{cases}$$

$$conflict(S \subset T)(Q,k) = |k(S) \setminus k(T)| + \begin{cases} 1, & \text{if } Q = T \text{ and } k(T) \subseteq k(S) \\ 1, & \text{if } Q = S \text{ and } k(T) \subseteq k(S) \text{ and } k(S) \neq \emptyset \\ 0, & \text{otherwise} \end{cases}$$

We have that $penalty(S \subset T)(k) = 3$. Indeed, we may satisfy P w.r.t. k by, e.g., adding the three values a, b, and c to T. We also have that $conflict(S \subset T)(S,k) = 2$ and $conflict(S \subset T)(T,k) = 3$. Indeed, by removing the values a and b from S, we may decrease the penalty of P by two. Similarly, by adding the values a, b, and c to T, we may decrease the penalty of P by three.

We use existential second-order logic extended with counting ($\exists$SOL$^+$) for modelling set constraints [1]. In the BNF below, the non-terminal symbol $\langle S \rangle$ denotes an identifier for a bound set variable S such that $S \subseteq \mathcal{U}$, while $\langle x \rangle$ and $\langle y \rangle$ denote identifiers for bound variables x and y such that $x, y \in \mathcal{U}$, and $\langle a \rangle$ denotes a natural number constant:

$$\begin{array}{lll}
\langle \mathit{Constraint} \rangle & ::= & (\underline{\exists}\ \langle S \rangle)^+\ \langle \mathit{Formula} \rangle \\
\langle \mathit{Formula} \rangle & ::= & \underline{(}\langle \mathit{Formula} \rangle\underline{)} \mid (\underline{\forall} \mid \underline{\exists})\langle x \rangle\ \langle \mathit{Formula} \rangle \\
 & \mid & \langle \mathit{Formula} \rangle\ (\underline{\wedge} \mid \underline{\vee})\ \langle \mathit{Formula} \rangle \mid \langle \mathit{Literal} \rangle \\
\langle \mathit{Literal} \rangle & ::= & \langle x \rangle\ (\underline{\in} \mid \underline{\notin})\ \langle S \rangle \\
 & \mid & \langle x \rangle\ (\underline{<} \mid \underline{\leq} \mid \underline{=} \mid \underline{\neq} \mid \underline{\geq} \mid \underline{>})\ \langle y \rangle \\
 & \mid & \underline{|}\langle S \rangle\underline{|}\ (\underline{<} \mid \underline{\leq} \mid \underline{=} \mid \underline{\neq} \mid \underline{\geq} \mid \underline{>})\ \langle a \rangle
\end{array}$$

As a running example, consider the constraint $S \subset T$ of Ex. 1. This may be expressed in $\exists\mathrm{SOL}^+$ by $\Omega = \exists S \exists T((\forall x(x \notin S \vee x \in T)) \wedge (\exists x(x \in T \wedge x \notin S)))$.

We proposed a penalty function for $\exists\mathrm{SOL}^+$ formulas in [1], which was inspired by [9]. For example, the penalty of a literal $x \in S$ w.r.t. a configuration k is 0 if $k(x) \in k(S)$ and 1, otherwise. The penalty of a conjunction (disjunction) is the sum (minimum) of the penalties of its conjuncts (disjuncts). The penalty of a universal (existential) quantification is the sum (minimum) of the penalties of the quantified formula where the occurrences of the bound variable are replaced by each value in the universe.

Example 2. Recall $k = \{S \mapsto \{a,b\}, T \mapsto \emptyset\}$ of Ex. 1. Then $\mathit{penalty}(\Omega)(k) = 3$.

3 Variable Conflicts of an $\exists\mathrm{SOL}^+$ Formula

The notion of abstract conflict measures the maximum possible penalty decrease obtainable by only changing the value of the given variable. It is uniquely determined by the chosen penalty function:

Definition 2. *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP and let $c \in \mathcal{C}$. The* abstract conflict function *of c w.r.t. $\mathit{penalty}(c)$ is the function $\mathit{abstractConflict}(c) : \mathcal{X} \times \mathcal{K} \to \mathbb{N}$ s.t. $\mathit{abstractConflict}(c)(x,k) = \max\{\mathit{penalty}(c)(k) - \mathit{penalty}(c)(\ell) \mid \ell \in n_x(k)\}$. The* abstract conflict *of $x \in \mathcal{X}$ w.r.t. c and $k \in \mathcal{K}$ is $\mathit{abstractConflict}(c)(x,k)$.*

Example 3. The function $\mathit{conflict}(S \subset T)$ of Ex. 1 gives abstract conflicts.

Similarly to our penalty function in [1], it is important to stress that the calculation of the variable conflict defined next is automatable and feasible incrementally [3], as it is based only on the syntax of the formula and the semantics of the quantifiers, connectives, and relational operators of $\exists\mathrm{SOL}^+$, but not on the intended semantics of the formula.

Definition 3. *Let $\mathcal{F} \in \exists\mathrm{SOL}^+$, let $S \in \mathit{vars}(\mathcal{F})$, and let k be a configuration for $\mathit{vars}(\mathcal{F})$. The* conflict *of S w.r.t. $\mathcal{F}$ and k is defined by:*

(a) $\mathit{conflict}(\exists S_1 \cdots \exists S_n \phi)(S,k) = \mathit{conflict}(\phi)(S,k)$

(b) $\mathit{conflict}(\forall x \phi)(S,k) = \sum\limits_{u \in \mathcal{U}} \mathit{conflict}(\phi)(S, k \cup \{x \mapsto u\})$

(c) $\mathit{conflict}(\exists x \phi)(S,k) =$
$\max\{0\} \cup \{\mathit{penalty}(\exists x \phi)(k) - (\mathit{penalty}(\phi)(k \cup \{x \mapsto u\}) - \mathit{conflict}(\phi)(S, k \cup \{x \mapsto u\})) \mid u \in \mathcal{U}\}$

(d) $\mathit{conflict}(\phi \wedge \psi)(S,k) = \sum\{\mathit{conflict}(\gamma)(S,k) \mid \gamma \in \{\phi,\psi\} \wedge S \in \mathit{vars}(\gamma)\}$

(e) $\mathit{conflict}(\phi \vee \psi)(S,k) = \max\{0\} \cup \{\mathit{penalty}(\phi \vee \psi)(k) - (\mathit{penalty}(\gamma)(k) - \mathit{conflict}(\gamma)(S,k)) \mid \gamma \in \{\phi,\psi\} \wedge S \in \mathit{vars}(\gamma)\}$

(f) $\mathit{conflict}(|S| \leq c)(S,k) = \mathit{penalty}(|S| \leq c)(k)$

(g) $\mathit{conflict}(x \in S)(S,k) = \mathit{penalty}(x \in S)(k)$

We only show cases for subformulas of the form $|S| \lozenge c$ and $x \bigtriangleup S$ where $\lozenge \in \{\leq\}$ and $\bigtriangleup \in \{\in\}$. The other cases are defined similarly.

Example 4. Recall once again $k = \{S \mapsto \{a, b\}, T \mapsto \emptyset\}$ of Ex. 1. According to Def. 3, we have that $\mathit{conflict}(\Omega)(S, k) = 2$ and $\mathit{conflict}(\Omega)(T, k) = 3$, i.e., the same values as obtained by the handcrafted $\mathit{conflict}(S \subset T)$ of Ex. 1.

The novelty of Def. 3 compared to the one in [8] lies in rules (c) and (e) for disjunctive formulas, due to the different abstract conflict that we target (see [3] for more details). The following example clarifies these rules in terms of (e).

Example 5. Consider $\mathcal{F} = (|S| = 5 \vee (|T| = 3 \wedge |S| = 6))$ and let k_1 be a configuration s.t. $|k_1(S)| = 6$ and $|k_1(T)| = 4$. Then $\mathit{penalty}(\mathcal{F})(k_1) = 1$ and we have $\mathit{conflict}(|S| = 5)(S, k_1) = 1$ and $\mathit{conflict}(|T| = 3 \ \wedge \ |S| = 6)(S, k_1) = 0$. Rule (e) applies for calculating $\mathit{conflict}(\mathcal{F})(S, k_1)$, which, for each disjunct, gives the *maximum possible penalty decrease* one may obtain by changing $k_1(S)$. This is 1 for the first disjunct since we may decrease $\mathit{penalty}(\mathcal{F})(k_1)$ by 1 by changing $k_1(S)$ as witnessed by $\mathit{penalty}(\mathcal{F})(k_1) - (\mathit{penalty}(|S| = 5)(k_1) - \mathit{conflict}(|S| = 5)(S, k_1)) = 1 - (1 - 1) = 1$. It is 0 for the second disjunct since we cannot decrease $\mathit{penalty}(\mathcal{F})(k_1)$ by changing $k_1(S)$ as witnessed by $\mathit{penalty}(\mathcal{F})(k_1) - (\mathit{penalty}(|T| = 3 \wedge |S| = 6)(k_1) - \mathit{conflict}(|T| = 3 \wedge |S| = 6)(S, k_1) = 1 - (1 - 0) = 0$. The maximum value of these is 1 and hence $\mathit{conflict}(\mathcal{F})(S, k_1) = 1$.

Consider now k_2 s.t. $|k_2(S)| = 4$ and $|k_2(T)| = 4$. Then $\mathit{penalty}(\mathcal{F})(k_2) = 1$ and $\mathit{conflict}(|T| = 3 \ \wedge \ |S| = 6)(T, k_2) = 1$. The maximum possible penalty decrease one may obtain by changing $k_2(T)$ in the only disjunct for T is -1 as witnessed by $\mathit{penalty}(\mathcal{F})(k_2) - (\mathit{penalty}(|T| = 3 \wedge |S| = 6)(k_2) - \mathit{conflict}(|T| = 3 \wedge |S| = 6)(T, k_2) = 1 - (3 - 1) = -1$. But we may not have a negative conflict, hence the union with $\{0\}$ in (e). Indeed, we cannot decrease $\mathit{penalty}(\mathcal{F})(k)$ by changing $k_2(T)$ since even if we satisfy $|k_2(T) = 3|$, the conjunct $|k_2(S) = 6|$ implies a penalty larger than 1 which is the minimum penalty of the two disjuncts.

We now state some properties of variable conflicts compared to the abstract conflict of Def. 2 and the formula penalty [1]. The proofs can be found in [3].

Proposition 1. *Let $\mathcal{F} \in \exists\mathrm{SOL}^+$, let k be a configuration for $\mathit{vars}(\mathcal{F})$, and let $S \in \mathit{vars}(\mathcal{F})$. Then $\mathit{abstractConflict}(\mathcal{F})(S, k) \leq \mathit{conflict}(\mathcal{F})(S, k) \leq \mathit{penalty}(\mathcal{F})(k)$.*

Corollary 1. *The function induced by Def. 3 is a conflict function w.r.t. Def. 1.*

4 Practical Results and Conclusion

The *progressive party problem* [7] is about timetabling a party at a yacht club, where the crews of certain boats (the guest boats) party at other boats (the host boats) over a number of periods. The crew of a guest boat must party at some host boat in each period. The spare capacity of a host boat is never to be exceeded. The crew of a guest boat may visit a particular host boat at most once. The crews of two distinct guest boats may meet at most once.

We use the same set-based model and local search algorithm as we did in [2]. The model includes *AllDisjoint*$(\mathcal{X})(k)$ constraints that hold iff no two distinct set variables in $\mathcal{X} = \{S_1, \ldots, S_n\}$ overlap. Assuming that this global constraint is not built-in, we may use the following $\exists$SOL$^+$ version instead:

$$\begin{aligned}\exists S_1 \cdots \exists S_n \forall x\ (\ &(x \notin S_1 \vee (x \notin S_2 \wedge \cdots \wedge x \notin S_n)) \wedge \\ &(x \notin S_2 \vee (x \notin S_3 \wedge \cdots \wedge x \notin S_n)) \wedge \cdots \wedge (x \notin S_{n-1} \vee x \notin S_n))\end{aligned}$$

We have run the same classical instances as we did in [2], on a 2.4GHz/512MB Linux machine. The following table shows the results for the $\exists$SOL$^+$ and built-in versions of the *AllDisjoint* constraint (mean run time in seconds of successful runs out of 100 and the number of unsuccessful runs, if any, in parentheses).

	$\exists$SOL$^+$ *AllDisjoint*					Built-in *AllDisjoint*				
H/periods (fails)	6	7	8	9	10	6	7	8	9	10
1-12,16			1.3	3.5	42.0			1.2	2.3	21.0
1-13			16.5	239.3				7.0	90.5	
1,3-13,19			18.9	273.2	(3)			7.2	128.4	(4)
3-13,25,26			36.5	405.5	(16)			13.9	170.0	(17)
1-11,19,21	19.8	186.7				10.3	83.0	(1)		
1-9,16-19	32.2	320.0	(12)			18.2	160.6	(22)		

The run times for the $\exists$SOL$^+$ version are only 2 to 3 times higher, though it must be noted that *efforts such as designing penalty and conflict functions as well as incremental maintenance algorithms for AllDisjoint were not necessary.* Note also that the robustness of the local search algorithm does not degrade for the $\exists$SOL$^+$ version, as witnessed by the number of solved instances.

To conclude, we proposed a new definition for inferring the conflict of a variable in a model and proved that any inferred variable conflict is lower-bounded by the targeted value, and upper-bounded by the inferred penalty. *The search is indeed directed towards interesting neighbourhoods, as a built-in constraint can be replaced without too high losses in run-time, nor any losses in robustness.*

References

1. M. Ågren, P. Flener, and J. Pearson. Incremental algorithms for local search from existential second-order logic. *Proceedings of CP'05.* Springer-Verlag, 2005.
2. M. Ågren, P. Flener, and J. Pearson. Set variables and local search. *Proceedings of CP-AI-OR'05.* Springer-Verlag, 2005.
3. M. Ågren, P. Flener, and J. Pearson. Inferring variable conflicts for local search. Tech. Rep. 2006-005, Dept. of Information Technology, Uppsala University, 2006.
4. P. Galinier and J.-K. Hao. A general approach for constraint solving by local search. *Proceedings of CP-AI-OR'00*, 2000.
5. N. Immerman. *Descriptive Complexity.* Springer-Verlag, 1998.
6. L. Michel and P. Van Hentenryck. A constraint-based architecture for local search. *Proceedings of OOPSLA'02*, 2002.
7. B. M. Smith *et al.* The progressive party problem: Integer linear programming and constraint programming compared. *Constraints*, 1:119–138, 1996.
8. P. Van Hentenryck and L. Michel. *Constraint-Based Local Search.* MIT Press, 2005.
9. P. Van Hentenryck, L. Michel, and L. Liu. Constraint-based combinators for local search. *Proceedings of CP'04.* Springer-Verlag, 2004.

Improvements on the Applicability of Nonlinear Constraint Solvers

Student: Leslie De Koninck*
Supervisor: Bart Demoen

Department of Computer Science, K.U.Leuven, Belgium

Abstract. Nonlinear constraints over the real numbers appear in many application domains, like chemistry, economics or computer graphics. Their use in Constraint Logic Programming environments have thusfar been quite limited, because of a combination of performance issues and commercial considerations. This research aims at improving the practical applicability of nonlinear constraints in a CLP environment. We focus in particular on interval-based constraint solving techniques. This paper presents our research goals, our current results and ideas for future work.

1 Introduction

Constraint (Logic) Programming is nowadays an established method for solving a wide variety of combinatorial problems. These are problems in which the problem variables take a value from a finite domain of possibilities. Solving constraints over other constraint domains is much less common practice. This is in particular true for constraints over the real numbers $\mathbb{R}$. For linear constraints over $\mathbb{R}$, algorithms from the Operations Research community are often used.

Nonlinear constraints over $\mathbb{R}$ appear in many practical applications, amongst others in chemical engineering [12, 8], economics [25] and computer graphics [16]. These constraints can be solved by methods like cylindrical algebraic decomposition [15], homotopy continuation [17] and most notably using techniques from interval analysis [1]. Only the latter is able to support the incremental nature of Constraint Logic Programming well.

Although much progress has been made in this area over the past decades, the constraint domain $\mathbb{R}$ has not reached the same general acceptance as do finite domain constraints. Reasons include the unpredictable nature of constraint solving in $\mathbb{R}$ (both with respect to time complexity and with respect to reachable solution precision) as well as the closed source nature of established nonlinear Constraint Programming systems like ILOG Solver [20] or ECL^{i}PS^e [26].

The aim of this research is to improve the usability of nonlinear constraints over a continuous domain, both by improving the solving techniques and by improving the public availability of nonlinear systems. We focus on the integration

* Research Funded by a Ph.D. grant of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)

of nonlinear solving techniques in a Constraint Logic Programming environment, in particular SWI-Prolog.

In Section 2 we present Constraint Handling Rules, a language designed for implementing Constraint Logic Programming systems. In Section 3, we briefly present interval-based techniques for nonlinear constraint solving. Section 4 gives an overview of our current results and finally, Section 5 gives some directions for future work.

2 Constraint Handling Rules

Constraint Handling Rules [7] is a rule-based language that was originally designed for the implementation of Constraint Logic Programming systems. Over the years, CHR has been used more and more as a general purpose programming language, supported by the result that every algorithm can be implemented in CHR in optimal time complexity [24]. CHR is now also available outside of its original Prolog context, in languages like Haskell, Java [27] or Curry [10].

CHR is particularly suitable for symbolic constraint processing, as it allows simplifying a conjunction of constraints and propagation of redundant constraints. The latter have proven useful in speeding up numerical constraint processing techniques. CHR takes care of bookkeeping issues like constraint store representation and marking which propagation rules have been tried already. Finally, Constraint Handling Rules have been successfully used to combine different constraint solvers into a more powerful system.

Because of these advantages, we have chosen to focus our research on the implementation of a nonlinear CLP system using CHR. This turns part of the research into a strong proof of concept for CHR. We also take a look at how certain disadvantages of CHR can be eliminated.

3 Interval-based Nonlinear Constraint Solving

In this section, we give a high level description of how interval arithmetic can be used to solve nonlinear Constraint Programming problems. An *interval* $\mathbf{x} = [\underline{x}, \overline{x}]$ is the closed set of reals $\{x \in \mathbb{R} \mid \underline{x} \leq x \leq \overline{x}\}$. We denote the set of all intervals by $\mathbb{IR}$. In practice, we only consider intervals with floating point bounds.

3.1 Interval Extensions

An interval extension of a function $f : \mathbb{R}^n \to \mathbb{R}$ is an interval function $F : \mathbb{IR}^n \to \mathbb{IR}$ satisfying $\forall x \in \mathbf{x} : f(x) \in F(\mathbf{x})$. In other words, it forms an outer approximation of the function. An interval extension of a constraint $c \subseteq \mathbb{R}^n$ is an interval constraint $C \subseteq \mathbb{IR}^n$ satisfying $c(x) \implies C(\mathbf{x})$ for all real vectors x and interval vectors $\mathbf{x}$ satisfying $x \in \mathbf{x}$.

The most basic type of interval extensions is the natural interval extension. It is formed by replacing each variable by its interval domain and replacing each primitive operator or function by its interval arithmetic equivalent.

In general, different arithmetic expressions that denote the same real function, do not necessarily denote the same interval function. This is because interval arithmetic does not have the same properties as real arithmetic. For instance, although $x - x = 0$ for all real x, in interval arithmetic $\mathbf{x} - \mathbf{x} = [0, 0]$ only if $\mathbf{x}$ is an interval of zero width. Therefore, it is sometimes useful to rewrite a real arithmetic expression into an equivalent one so as to get a more precise interval extension. Often, so-called center forms are used for this. For example, using the first order Taylor approximations gives us the Taylor interval extension.

Interval extensions are used to check constraint satisfaction for a whole range of points. In this way, we can create a discretization of the continuous search space into a finite number of intervals of a given precision. One can also use interval extensions to create safe versions of iterative root-finding algorithms like the Newton-Raphson method.

3.2 Consistency Techniques

Nonlinear constraint programming systems make use of local consistency techniques, similar to the well known arc consistency or bounds consistency for finite domain constraints. Constraint solving then consists of checking whether all constraints are consistent and if this is not the case, narrowing the domains of the constraint variables until consistency is reached. Two often used consistencies are box consistency and hull consistency [4]. They are described below.

Hull Consistency An n-ary constraint c is hull consistent with respect to variable $x_i \in \mathbf{x}_i$ if its domain $\mathbf{x}_i$ is the interval hull of the set

$$\{x_i \in \mathbf{x}_i \mid \exists x_1, \ldots, \exists x_{i-1}, \exists x_{i+1}, \ldots, \exists x_n c(x_1, \ldots, x_n)\}$$

Hull consistency can only be computed for constraints in which each variable occurs only once. Other constraints are first decomposed by introducing new variables for every variable occurrence and linking them by other constraints. This weakens the strength of hull consistency because it is a local consistency technique. We can use constraint inversion with respect to a variable that occurs only once in a constraint to create a weaker form of hull consistency that is more precise than using the decomposition [3].

Box consistency An n-ary constraint c is *box consistent* with respect to variable $x_i \in \mathbf{x}_i$ if its domain $\mathbf{x}_i = [\underline{x}_i, \overline{x}_i]$ satisfies the interval constraints

$$C(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \underline{x}_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n)$$
$$\text{and}$$
$$C(\mathbf{x}_1, \ldots, \mathbf{x}_{i-1}, \overline{x}_i, \mathbf{x}_{i+1}, \ldots, \mathbf{x}_n)$$

Box consistency is equivalent to hull consistency for constraints in which each variable occurs only once. It is stronger than hull consistency on the decomposition of constraints in which variables occur more than once. On the other hand, domain narrowing is considerably more expensive for box consistency compared to hull consistency.

4 Current Results and Work in Progress

In this section, we give an overview of our results so far and of the topics that we are currently working on.

4.1 INCLP($\mathbb{R}$)

Our first achievement is a new nonlinear CLP system called INCLP($\mathbb{R}$) [14], available at [13] and which will soon be incorporated into the popular open source Prolog distribution SWI-Prolog [28]. The main ideas on which the system is based, with amongst others box consistency and the Taylor interval extension, come from the Newton system [11] which also forms the basis for the nonlinear solver of ILOG.

The INCLP($\mathbb{R}$) system combines box consistency with a form of hull consistency based on constraint inversion, similar to the approach taken in [2]. Development versions offer different interval extensions. The system is the first nonlinear CLP system built using Constraint Handling Rules. The system scales well on typical benchmarks from the interval analysis community and improves on ECL^{i}PS^e on benchmarks for which the constraint decomposition created by hull consistency causes too little domain reduction.

4.2 Practical Implementation of Interval Extensions

There are many interval extensions which have nice theoretical properties, but are very expensive to compute. An example is the slope interval extension [22, 21, 18]. Its implementation by Rump [22] uses intersections with the natural interval extension of subterms to create an interval extension that is at least as good as the natural interval extension and often better. The main disadvantage is that it is computationally very expensive to calculate both forms for each subterm and this decreases its practical usability considerably.

We can already reduce the computational cost by using the natural interval extension alone for terms in which each variable occurs only once. In those terms the so-called dependency problem does not occur and the natural interval extension is optimal.[1] We currently investigate how we can reduce the number of interval evaluations even more without knowing the domains of the involved variables and without reducing the precision of the interval extension.

4.3 Search Strategies in CHR

Another aspect that we work on is the implementation of different search strategies using a Prolog CHR implementation. Prolog imposes its left to right depth first search order on CHR implementations based on it. We have designed a

[1] The dependency problem denotes the effect of overly wide interval extensions that is caused by treating dependent terms as independent

source to source transformation to create a different execution order, in particular breadth first. In this transformation, we use the nonbacktrackable global variables facility of SWI-Prolog to store changes to the CHR execution state so that expensive computations do not have to be redone when changing between branches. We have also extended the Refined Operational Semantics so that different search strategies are supported.

5 Future Work

We plan to investigate further improvements on amongst others interval extensions, other representations connected to interval arithmetic like affine arithmetic [5], scheduling algorithms and search heuristics. We have already started to work on the two aspects described below.

The INCLP($\mathbb{R}$) system uses an interval as the domain for its variables. An alternative is to use unions of intervals. This allows us to store the effects of domain splitting caused by division and even root extraction. The main difficulty here is to avoid an exponential increase in the number of intervals. It has been shown that this increase is not that drastic in practice because the constraint that all intervals should be mutually exclusive becomes more difficult to satisfy as the number of intervals increases [23].

Local consistency techniques like box consistency and hull consistency are often not able to do much domain pruning because they only look at one constraint at a time. Higher order consistencies like bound consistency [4] can overcome this problem, but are often computationally too expensive to be used in practice. We are working on a consistency technique whose strength lies in between box consistency and bound consistency and that can be made stronger or weaker by using a technique similar to weak box consistency [9].

References

1. Frédéric Benhamou. Interval constraint logic programming. In Podelski [19].
2. Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-Francois Puget. Revising hull and box consistency. In *ICLP*, pages 230–244, 1999.
3. Martine Ceberio and Laurent Granvilliers. Solving nonlinear systems by constraint inversion and interval arithmetic. In *AISC*, volume 1930 of *Lecture Notes in Computer Science*. Springer, 2000.
4. Hélène Collavizza, François Delobel, and Michel Rueher. Comparing partial consistencies. *Reliable Computing*, 5(3), 1999.
5. Luiz Henrique de Figueiredo and Jorge Stolfi. Affine arithmetic: concepts and applications. *Numerical Algorithms*, 37(1-4), Dec 2004.
6. Michael Fink, Hans Tompits, and Stefan Woltran, editors. *Proceedings of the 20th Workshop on Logic Programming.* INFSYS Research Report 1843-06-02 (TU Wien), 2006.
7. Thom W. Frühwirth. Constraint Handling Rules. In Podelski [19].
8. Chao-Yang Gau and Mark A. Stadtherr. Nonlinear parameter estimation using interval analysis. *AIChE Symp. Ser.*, 94(304), 1999.

9. Laurent Granvilliers, Frédéric Goualard, and Frédéric Benhamou. Box consistency through weak box consistency. In *ICTAI*, 1999.
10. Michael Hanus. Adding Constraint Handling Rules to Curry. In Fink et al. [6].
11. Pascal Van Hentenryck, Laurent Michel, and Frédéric Benhamou. Newton - constraint programming over nonlinear constraints. *Sci. Comput. Program.*, 30(1-2), 1998.
12. James Z. Hua, Joan F. Brennecke, and Mark A. Stadtherr. Reliable computation of phase stability using interval analysis: Cubic equation of state models. *Comput. Chem. Eng.*, 20, 1996.
13. Leslie De Koninck. The INCLP(R) website. http://www.cs.kuleuven.be/~leslie/INCLPR/.
14. Leslie De Koninck, Tom Schrijvers, and Bart Demoen. INCLP(R) - Interval-based nonlinear constraint logic programming over the reals. In Fink et al. [6].
15. Scott McCallum. Solving polynomial strict inequalities using cylindrical algebraic decomposition. *Comput. J.*, 36(5), 1993.
16. Don P. Mitchell. Robust ray intersection with interval arithmetic. In *Proceedings on Graphics interface*, 1990.
17. Alexander Morgan and Andrew Sommese. Computing all solutions to polynomial systems using homotopy continuation. *Appl. Math. Comput.*, 24(2), 1987.
18. Humberto Muñoz and Ralph Baker Kearfott. Slope intervals, generalized gradients, semigradients, slant derivatives, and csets. *Reliable Computing*, 10(3), 2004.
19. Andreas Podelski, editor. *Constraint Programming: Basics and Trends, Châtillon Spring School, Châtillon-sur-Seine, France, May 16 - 20, 1994, Selected Papers*, volume 910 of *Lecture Notes in Computer Science*. Springer, 1995.
20. Jean-François Puget. A C++ implementation of CLP. In *Proceedings of the 2nd Singapore Conference on Intelligent Systems*, 1994.
21. Dietmar Ratz. A nonsmooth global optimization technique using slopes — the one dimensional case. *Journal of Global Optimization*, 14(4), 1999.
22. Siegfried M. Rump. Expansion and estimation of the range of nonlinear functions. *Math. Comput.*, 65(216), 1996.
23. Rony Shapiro, Yishai A. Feldman, and Rina Dechter. On the complexity of interval-based constraint networks. In *MISC'99 Workshop on Applications of Interval Analysis to Systems and Control*, 1999.
24. Jon Sneyers, Tom Schrijvers, and Bart Demoen. The computational power and complexity of Constraint Handling Rules. In *Proceedings of the 2nd Workshop on Constraint Handling Rules*, October 2005.
25. John B. Taylor and Harald Uhlig. Solving nonlinear stochastic growth models: a comparison of alternative solution methods. NBER Working Papers 3117, National Bureau of Economic Research, Inc, 1990.
26. Mark Wallace, Stefano Novello, and Joachim Schimpf. ECLiPSe: A platform for constraint logic programming. *ICL Systems Journal*, 12(1), 1997.
27. Peter Van Weert, Tom Schrijvers, and Bart Demoen. K.U.Leuven JCHR: a user-friendly, flexible and efficient CHR system for Java. In *Proceedings of the 2nd Workshop on Constraint Handling Rules*, October 2005.
28. Jan Wielemaker. An overview of the SWI-Prolog programming environment. In *Proceedings of the 13th International Workshop on Logic Programming Environments*, December 2003.

Mini-bucket Elimination with Bucket Propagation

Student: Emma Rollon
Supervisor: Javier Larrosa

Universitat Politecnica de Catalunya,
Jordi Girona 1-3, 08034 Barcelona, Spain
erollon@lsi.upc.edu, larrosa@lsi.upc.edu

Abstract. *I*n this paper we introduce a new propagation phase that *Mini-bucket Elimination* (MBE) should execute at each bucket. The purpose of this propagation is to jointly process as much information as possible. As a consequence, the undesirable lose of accuracy caused by MBE when splitting functions into different mini-buckets is minimized. We demonstrate our approach in *scheduling* and *combinatorial auction*, where the resulting algorithm MBE^p gives important percentage increments of the lower bound (typically 50% and up to 1566%) with only doubling the cpu time.

1 Introduction

A *soft CSP* is a triplet $(\mathcal{X}, \mathcal{D}, \mathcal{F})$ where $\mathcal{X}$ is the set of variables, $\mathcal{D}$ is the set of finite domain values and $\mathcal{F}$ is the set of functions. Each function $f \in \mathcal{F}$ specifies how good is each different partial assignment of $var(f)$. The usual task of interest is to find a complete assignment t with minimum cost, if there is any.

Bucket elimination (BE) [1] is the reference algorithm to solve soft CSP by complete inference. When problems are too difficult to be solved exactly, approximation methods become the best option. *Mini-bucket elimination* (MBE) [2] is arguably one of the best-known general approximation algorithms for soft CSPs. It uses a control parameter z that allow us to trade time and space for accuracy. The time and space complexity of MBE is exponential in z and it is important to note that, with current computers, it is the space, rather than the time, what prohibits the execution of the algorithm beyond certain values of z.

In this paper, we introduce a new propagation phase that MBE must execute at each bucket. In this phase, mini-buckets are structured into a tree and costs are moved along branches from the leaves to the root. As a result, the mini-bucket root accumulates costs that will be processed together, while classical MBE would have processed them independently. It is important to note that the new propagation phase does not increase the complexity with respect classical MBE. Our experiments on *scheduling* and *combinatorial auctions* show that the addition of this propagation phase increases the quality of the lower bound provided by MBE quite significant.

2 Bucket and Mini-Bucket Elimination

Bucket Elimination(BE) [1] works in two phases. In the first phase, the algorithm selects a variable x_i, sums all the functions mentioning x_i (i.e., the bucket of x_i), and projects x_i from the resulting function, eliminating the variable from the problem. BE process each variable in turn, until no variable remains. The outcome of the first phase is the optimal cost of the problem. The second phase generates an optimal assignment of variables using the set of buckets that were computed in the first phase. The time and space complexity of *BE* is exponential in a structural parameter called *induced width*. In practice, it is the space and not the time what makes the algorithm unfeasible in many instances.

Mini-bucket elimination (MBE) [2] is the approximation version of BE. Given a control parameter z, MBE partitions buckets into smaller subsets called mini-buckets such that their arity is bounded by $z+1$. Each mini-bucket is processed independently. The outcome of the algorithm is a lower bound of the problem optimum. In the second phase MBE computes a (non-necessarily optimal) assignment t. The time and space complexity of MBE is exponential in z. Parameter z allows us to trade time and space for accuracy, because greater values of z increment the number of functions that can be included in each mini-bucket. Therefore, the bounds will be presumably tighter. MBE constitutes a powerful yet extremely general mechanism for lower bound computation.

3 Mini Buckets with Propagation

In this Section we define a refinement of MBE. It consists on performing an arrangement of costs in each bucket before processing it. It has been shown that in fair soft constraint frameworks, costs can be subtracted from one function if they are properly summed to another in order to preserve the equivalence of the problem [3, 4]. We incorporate this idea into MBE, at the bucket level. We propose to *transfer* costs between minibuckets in order to accumulate as much information as possible into one of them. This process is based on the concept of function subtraction: Let f and h be two functions such that $var(h) \subseteq var(f)$ and $\forall t, f(t) \geq h(t)$. Their *subtraction*, denoted $(f - h)$, is a new function with scope $var(f)$ defined as, $(f - h)(t) = f(t) - h(t)$.

Let f and g be two arbitrary functions. The *transfer of costs* from f to g, denoted $M(f, g)$, is done sequentially in three steps:

$$h := f[var(f) \cap var(g)]; \quad f := f - h; \quad g := g + h;$$

In words, function h contains costs in f that can be captured in terms of the common variables with g. Hence, they can be kept either in h or in f. Then, this costs are moved from f to g.

The following example illustrates and motivates the idea of *moving* costs inside MBE. Suppose that MBE is processing a bucket containing two functions f and g, each one forming a mini-bucket. Variable x_i is the one to be eliminated.

g: $x_i x_j$	
f f	5
f t	4
t f	1
t t	6

f: $x_i x_k$	
f f	2
f t	3
t f	4
t t	5

(a)

g: $x_i x_j$	
f f	7
f t	6
t f	5
t t	10

f: $x_i x_k$	
f f	0
f t	1
t f	0
t t	1

(b)

$g\downarrow x_i$ x_j	
f	1
t	4

$f\downarrow x_i$ x_k	
f	2
t	3

(c)

$g\downarrow x_i$ x_j	
f	5
t	6

$f\downarrow x_i$ x_k	
f	0
t	1

(d)

Fig. 1. Example of functions.

Standard MBE would process independently each minibucket, eliminating variable x_i in each function. Actually, that independent elimination of x_i from each mini-bucket causes the lose of accuracy from the lower bound of MBE. Ideally, f and g should be processed together, but their information is split into two pieces for complexity reasons. We propose to transfer costs from f to g (or conversely) before processing the mini-buckets. The purpose is to put as much information as possible in the same mini-bucket, so that all this information is jointly processed as BE would do. Consequently, the pernicious effect of splitting the bucket into mini-buckets will presumably be minimized.

Figure 1 depicts a numerical illustration. Consider functions f and g from Figure 1 (a). If variable x_i is eliminated independently, we obtain the functions in Figure 1 (c). If the problem contains no more functions, the final lower bound will be 3. Consider now the functions in Figure 1 (b) where costs have been moved from f to g. Note that as f and g only share variable x_i then $h = f \downarrow x_k$ is defined as $h(false) = 2$ and $h(true) = 4$. If variable x_i is eliminated independently, we obtain the functions in Figure 1 (d), with which the lower bound is 5.

The previous example was limited to two mini-buckets containing one function each. Nevertheless, the idea can be easily generalized to arbitrary mini-bucket arrangements. First, we extend the concept of movement of costs to deal with sets of functions. Let F and G be two sets of costs functions. Let $var(F) = \cup_{f\in F} var(f)$, $var(G) = \cup_{g\in G} var(g)$ and $Y = var(F) \cap var(G)$. The *movement of costs* from F to G is done sequentially in three steps:

$$h := (\sum_{f\in F} f)[Y]; \quad F := F \cup \{-h\}; \quad G := G \cup \{h\};$$

where $-h$ means that costs contained in h are to be subtracted instead of summed, when evaluating costs of tuples on F.

At each bucket $\mathcal{B}$, we construct a *propagation tree* $T = (V, E)$ where nodes are associated with mini-buckets and edges represent movement of costs along branches from the leaves to the root. Each node waits until receiving costs from all its children. Then, it sends costs to its parent. This flow of costs accumulates and propagates costs towards the root.

The refinement of MBE that incorporates this idea is called MBE^p. MBE^p (Figure 2) and MBE are very similar and, in the following, we discuss the main differences. After partitioning the bucket into mini-buckets (line 3), MBE^p con-

```
function $MBE^p(z)$
1.  for each $i = n..1$ do
2.      $\mathcal{B} := \{f \in F \mid x_i \in var(f)\}$;
3.      $\{\mathcal{P}_1, \ldots, \mathcal{P}_k\} :=$ Partition($\mathcal{B}, z$);
5.      $(V, E) :=$ PropTree($\{\mathcal{P}_1, \ldots, \mathcal{P}_k\}$);
6.      Propagation($(V, E)$);
7.      for each $j = 1..k$ do $g_j := ((\sum_{f \in \mathcal{P}_j} f) - h_j) \downarrow x_i$;
8.      $F := (F \cup \{g_1, \ldots, g_k\}) - \mathcal{B}$;
9.  endfor
10. return($g_1$);
endfunction
procedure Propagation($(V, E)$)
11. repeat
12.     select a node $j$ s.t it has received the messages from all its children;
13.     $h_j := (\sum_{f \in \mathcal{P}_j} f)[var(\mathcal{P}_j) \cap var(\mathcal{P}_{parent(j)})]$;
14.     $\mathcal{P}_j := \mathcal{P}_j \cup \{-h_j\}$;
15.     $\mathcal{P}_{parent(j)} := \mathcal{P}_{parent(j)} \cup \{h_j\}$;
16. until root has received all messages from its children;
endprocedure
```

Fig. 2. Mini-Bucket Elimination with Propagation. Given a WCSP $(\mathcal{X}, \mathcal{D}, \mathcal{F})$, the algorithm returns a zero-arity function g_1 with a lower bound of the optimum cost.

structs a propagation tree $T = (V, E)$ with one node j associated to each mini-bucket $\mathcal{P}_j$. Then, costs are propagated (lines 6, 11-16). Finally, line 7 sums the functions in the mini-buckets and eliminates variable x_i. The resulting functions are added to the problem in replacement of the bucket (line 8). Note that procedure `Propagation` moves costs between mini-buckets preserving the set of original functions.

Theorem 1. *The time and space complexity of MBE^p is $O(d^{z+1})$ and $O(d^z)$, respectively, where d is the largest domain size and z is the value of the control parameter.*

The success of the propagation phase of MBE^p greatly depends on the flow of information, which is captured in the propagation tree. Two ideas lead to heuristically good propagation trees:

First observation: it seems more appropriate to move costs to a mini-bucket where the costs go to a higher mini-bucket, so they have more chances to propagate useful information. We associate to each mini-bucket $\mathcal{P}_j$ a binary number $N_j = b_n b_{n-1} \ldots b_1$ where $b_i = 1$ iff $x_i \in \mathcal{P}_j$. We say that mini-bucket $\mathcal{P}_j$ is smaller than $\mathcal{P}_k$ (noted $\mathcal{P}_j < \mathcal{P}_k$) if $N_j < N_k$.

Second observation: the number of common variables determines the arity of the function that is used as a *bridge* in the cost transfer. The narrower the bridge, the less information can be captured.

In accordance with the two previous observations, we construct the propagation tree as follows: the parent of mini-bucket $\mathcal{P}_u$ will be a mini-bucket $\mathcal{P}_w$ such that $\mathcal{P}_u < \mathcal{P}_w$ and they share a maximum number of variables. This strategy combines the two criteria discussed above.

Instance	z	MBE(z)		MBE^p_r(z)		MBE^p_h(z)	
		Lb.	Time(sec.)	%	Time(sec.)	%	Time(sec.)
1506	20	184247	827.63	1.6	1628.93	29.8	1706.6
	15	163301	25.43	-5.5	51.48	30.6	51.39
	10	153274	1.33	-13.7	2.65	21.5	2.64
1403	20	181184	814.55	7.1	1702.82	59.6	1919.48
	15	162170	27.82	7.3	55.94	57.3	56.9
	10	146155	1.3	10.9	2.58	60.2	2.6
1405	20	191258	1197.06	0.5	2537.64	42.3	2622.88
	15	169233	33.88	-2.3	93.88	54.9	81.17
	10	142206	1.7	-25.3	3.51	64.7	3.5
1407	20	191342	1415.91	-4.0	2935.78	53.8	3008.78
	15	166298	47.44	3.5	94.17	60.1	102.78
	10	144264	2.03	13.8	4.19	68.6	4.23
408	20	5212	51.19	19.1	75.39	19.3	72.5
	15	5200	2.11	18.7	3.29	19.3	3.41
	10	2166	0.11	38.1	0.2	139.0	0.2

Fig. 3. Experimental results on Spot5 instances.

4 Experimental Results

The purpose of the experiments is to evaluate the effectiveness of the propagation phase and the impact of the propagation tree on that propagation. To that end, we compare the lower bound obtained with three algorithms: standard MBE, MBE with bucket propagation using as a propagation tree a chain of mini-buckets randomly ordered (i.e., MBE^p_r), and MBE with bucket propagation using a propagation tree heuristically built as shown in Section 3 (i.e., MBE^p_h).

For our first experiment, we consider instances from Spot5 satellite [5]. Some instances include in their original formulation an additional capacity constraint that we discard on this benchmark. Figure 3 shows the results[1]. Columns fifth and sixth indicates for MBE^p_r the percentage increment of the lower bound measured as $((Lb_{MBE^p_r} - Lb_{MBE})/Lb_{MBE}) * 100$ and its execution time. Columns seventh and eighth reports the same information for MBE^p_h. The first thing to be observed is that MBE^p_h increases the lower bound obtained with MBE for all the instances. Moreover, when both MBE^p_r and MBE^p_h increase the lower bound, MBE^p_h is always clearly superior. Therefore, it is clear that an adequate propagation tree impacts on the bounds obtained. Regarding MBE^p_h, its percentage increment is up to 139% (e.g. instance 408). The mean percentage increment is 54%, 38%, and 28% when the value of the control parameter z is 10, 15, and 20, respectively. Note that the effect of the propagation is higher for lower values of z because, as we increase the value of z, the propagated information decreases and the effect of the propagation is diminished. Moreover, the lower bounds obtained with $MBE^p_h(z = 10)$ outperforms the ones obtained with $MBE(z = 20)$ in almost all the instances. Regarding cpu time, MBE^p_h is from 2 to 3 times slower than MBE. The reason is that cost functions are evaluated twice. However, it is important to note that it is the space and not the time what bounds the maximum value of z that can be used in practice.

[1] For space reasons, we only report the results on some instances since the behavior of the others is the same.

Instance	z	MBE	MBE_r^p	MBE_h^p
		Lb.	%	%
brock200-1	18	66	30.3	48.4
	10	51	52.9	78.4
brock200-2	18	55	67.2	103.6
	10	29	200	268.9
brock200-4	18	63	36.5	65.0
	10	41	121.9	131.7
brock400-1	18	79	100	141.7
	10	46	256.5	273.9
brock400-2	18	75	114.6	157.3
	10	44	261.3	277.2
brock400-4	18	76	106.5	160.5
	10	47	248.9	289.3
brock800-1	18	71	336.6	454.9
	10	41	675.6	773.1
brock800-2	18	63	395.2	520.6
	10	37	748.6	875.6
brock800-3	18	68	352.9	483.8
	10	44	604.5	706.8
brock800-4	18	71	343.6	460.5
	10	36	758.3	902.7
c-fat200-1	18	71	32.3	78.8
	10	62	27.4	112.9

Instance	z	MBE	MBE_r^p	MBE_h^p
		Lb.	%	%
p-hat1000-1	15	85	380	654.1
	10	63	577.7	873.0
p-hat1000-2	15	57	589.4	821.0
	10	36	1013.8	1325
p-hat1500-1	15	69	802.8	1292.7
	10	82	686.5	1021.9
p-hat1500-2	15	64	812.5	1112.5
	10	45	1226.6	1566.6
p-hat1500-3	15	79	624.0	706.3
	10	54	924.0	1111.1
p-hat300-1	18	62	112.9	195.1
	10	48	187.5	306.2
p-hat300-2	18	61	121.3	168.8
	10	38	247.3	328.9
p-hat500-1	18	74	170.2	301.3
	10	50	330	524
p-hat500-2	18	75	178.6	248
	10	39	407.6	556.4
p-hat500-3	18	93	125.8	169.8
	10	50	300	338
p-hat700-1	15	66	340.9	581.8
	10	52	482.6	711.5

Fig. 4. Experimental results on maxclique instances.

Therefore, the constant increase in time is not that significant as the space complexity remains the same.

For our second experiment, we consider maxclique problems from the dimacs benchmark [6]. Figure 4 reports the results for some representative instances (the remaining instances follow the same pattern). As the behaviour of the cpu time is the same as for the previous benchmark, we do not report this information. The best results are obtained with MBE_h^p which obtains a percentage of increment of 1566% (see instance *p-hat1500-2*). In this case, the increase ranges from 14.6% to 1566% when z is set to 10, and from 17.6% to 1292% for the highest value of z. It is important to note that the bound obtained with MBE_h^p is always higher than that of MBE_r^p. For some instances, the percentage of increment of MBE_h^p is more than 4 times higher the one obtained with MBE_r^p (e.g. instance *c-fat200-1*). Therefore, it is clear that an adequate propagation tree impacts on the propagation phase and, as a consequence, on the bounds obtained.

References

1. Dechter, R.: Bucket elimination: A unifying framework for reasoning. Artificial Intelligence **113** (1999) 41–85
2. Dechter, R., Rish, I.: Mini-buckets: A general scheme for bounded inference. Journal of the ACM **50** (2003) 107–153
3. Larrosa, J., Schiex, T.: Solving weighted csp by maintaining arc-consistency. Artificial Intelligence **159** (2004) 1–26
4. Cooper, M.: High-order consistency in valued constraint satisfaction. Constraints **10** (2005) 283–305
5. Bensana, E., Lemaitre, M., Verfaillie, G.: Earth observation satellite management. Constraints **4(3)** (1999) 293–299
6. Johnson, D.S., Trick, M.: Second dimacs implementation challenge: cliques, coloring and satisfiability. DIMACS Series in Discrete Mathematics and Theoretical Computer Science. AMS **26** (1996)

Preprocessing QBF

Students: Jessica Davies and Horst Samulowitz
Supervisor: Fahiem Bacchus

Department of Computer Science, University of Toronto, Canada.
[jdavies| horst | fbacchus]@cs.toronto.edu

Abstract. In this paper we investigate the use of preprocessing when solving Quantified Boolean Formulas (QBF). Many different problems can be efficiently encoded as QBF instances, and there has been a great deal of recent interest and progress in solving such instances efficiently. Here we show that QBF instances can be simplified using techniques related to those used for preprocessing SAT. These simplifications can be performed in polynomial time, and are used to preprocess the instance prior to invoking a worst case exponential algorithm to solve it. We develop a method for preprocessing QBF instances that is empirically very effective. That is, the preprocessed formulas can be solved significantly faster, even when we account for the time required to perform the preprocessing. Our preprocessor, Prequel, significantly improves the efficiency of a range of state-of-the-art QBF solvers. Furthermore, Prequel is able to completely solve some instances just by preprocessing, including some instances that to our knowledge have never been solved before by any QBF solver.
An extended version of this paper was appears in the technical programme of CP 2006.

1 Introduction

QBF is a powerful generalization of SAT in which the variables can be universally or existentially quantified (in SAT all variables are implicitly existentially quantified). Current QBF solvers are typically limited to problems that are about 1-2 orders of magnitude smaller than the instances solvable by current SAT solvers. Nevertheless, QBF solvers continue to improve. Furthermore, many problems have a much more compact encoding when quantifiers are available, so a quantified solver can still be useful even if it can only deal with much smaller instances than a traditional solver.

In this paper we present a new technique for improving QBF solvers based on a modification of techniques already used in SAT. Namely we preprocess the input formula, without changing its meaning, so that it becomes easier to solve. As we demonstrate below our technique can be extremely effective, sometimes reducing the time it takes to solve a QBF instance by orders of magnitude.

2 QBF

A quantified boolean formula has the form $\boldsymbol{Q}.F$, where F is a propositional formula expressed in CNF and $\boldsymbol{Q}$ is a sequence of quantified variables ($\forall x$ or $\exists x$). We require that no variable appear twice in $\boldsymbol{Q}$ and that the set of variables in F and $\boldsymbol{Q}$ be identical (i.e., F contains no free variables, and $\boldsymbol{Q}$ contains no extra or redundant variables).

A **quantifier block** qb of $\boldsymbol{Q}$ is a maximal contiguous subsequence of $\boldsymbol{Q}$ where every variable in qb has the same quantifier type. We order the quantifier blocks by their sequence of appearance in $\boldsymbol{Q}$: $qb_1 \leq qb_2$ iff qb_1 is equal to or appears before qb_2 in $\boldsymbol{Q}$. Each variable x in F appears in some quantifier block $qb(x)$, and the ordering of the quantifier blocks imposes the following ordering on the variables. For two variables x and y we say that $x \leq_q y$ iff $qb(x) \leq qb(y)$. Note that the variables in the same quantifier block are unordered while the ordering with respect to the different quantifier blocks defines a partial order. We also say that x is **universal** (**existential**) if its quantifier in $\boldsymbol{Q}$ is ∀ (∃).

For example, $\exists e_1 e_2.\forall u_1 u_2.\exists e_3 e_4.(e_1, \neg e_2, u_2, e_4) \wedge (\neg u_1, \neg e_3)$ is a QBF with $\boldsymbol{Q} = \exists e_1 e_2.\forall u_1 u_2.\exists e_3 e_4$ and F equal to the two clauses $(e_1, \neg e_2, u_2, e_4)$ and $(\neg u_1, \neg e_3)$. The quantifier blocks in order are $\exists e_1 e_2$, $\forall u_1 u_2$, and $\exists e_3 e_4$, and we have, e.g., that, $e_1 <_q e_3$, $u_1 <_q e_4$, u_1 is universal, and e_4 is existential.

A QBF instance can be reduced by assigning values to some of its variables. The **reduction** of a formula $\boldsymbol{Q}.F$ by a literal ℓ (denoted by $\boldsymbol{Q}.F\big|_{\ell}$) is the new formula $\boldsymbol{Q}'.F'$ where F' is F with all clauses containing ℓ removed and the negation of ℓ, $\neg\ell$, removed from all remaining clauses, and $\boldsymbol{Q}'$ is $\boldsymbol{Q}$ with the variable of ℓ and its quantifier removed. For example, $\forall xz.\exists y.(\neg y, x, z) \wedge (\neg x, y)\big|_{\neg x} = \forall z.\exists y(\neg y, z)$.

Semantics. A SAT-model $\mathcal{M}_s$ of a CNF formula F is a truth assignment π to the variables of F that satisfies every clause in F. In contrast a QBF-model (**QBF-model**) $\mathcal{M}_q$ of a quantified formula $\boldsymbol{Q}.F$ is a **tree** of truth assignments in which the root is the empty truth assignment, and every node n assigns a truth value to a variable of F not yet assigned by one of n's ancestors. The tree $\mathcal{M}_q$ is subject to the following conditions:

1. For every node n in $\mathcal{M}_q$, n has a sibling if and only if it assigns a truth value to a universal variable x. In this case it has exactly one sibling that assigns the opposite truth value to x. Nodes assigning existentials have no siblings.
2. Every path π in $\mathcal{M}_q$ (π is the sequence of truth assignments made from the root to a leaf of $\mathcal{M}_q$) must assign the variables in an order that respects $\leq_q$. That is, if n assigns x and one of n's ancestors assigns y then we must have that $y \leq_q x$.
3. Every path π in $\mathcal{M}_q$ must be a SAT-model of F.

Thus a QBF-model has a path for every possible setting of the universal variables of $\boldsymbol{Q}$, and each of these paths is a SAT-model of F. We say that a QBF $\boldsymbol{Q}.F$ is QSAT iff it has a QBF-model. The QBF problem is to determine whether or not $\boldsymbol{Q}.F$ is QSAT.

The advantage of our "tree-of-models" definition is that it makes two key observations more apparent. These observations can be used to prove the correctness of our preprocessing technique.

A. If F' has the same SAT-models as F then $\boldsymbol{Q}.F$ will have the same QBF-models as $\boldsymbol{Q}.F'$.

B. A QBF-model preserving (but not SAT-model preserving) transformation that can be performed on $\boldsymbol{Q}.F$ is **universal reduction**. A universal variable u is called a *tailing universal* in a clause c if for every existential variable $e \in c$ we have that $e <_q u$. The universal reduction of a clause c is the process of removing all tailing universals from c [5]. Universal reduction preserves the set of QBF-models.

We call two QBF formulas **Q-equivalent** iff they have exactly the same QBF-models.

3 HyperBinary Resolution for SAT

The foundation of our polynomial time preprocessing technique is the SAT method of reasoning with binary clauses using hyper-resolution developed in [1, 2]. This method reasons with CNF SAT theories using the following "HypBinRes" rule of inference:

> Given a single n-ary clause $c = (l_1, l_2, ..., l_n)$, D a subset of c, and the set of binary clauses $\{(\ell, \neg l) | l \in D\}$, infer the new clause $b = (c - D) \cup \{\ell\}$ if b is either binary or unary.

For example, from (a, b, c, d), $(h, \neg a)$, $(h, \neg c)$ and $(h, \neg d)$, we infer the new binary clause (h, b). The advantage of HypBinRes inference is that it does not blow up the theory (it can only add binary or unary clauses to the theory) and it can discover a lot of new unit clauses. These unit clauses can then be used to simplify the formula by doing unit propagation which in turn might allow more applications of HypBinRes. Applying HypBinRes and unit propagation until closure (i.e., until nothing new can be inferred) uncovers *all* failed literals. That is, in the resulting reduced theory there will be no literal ℓ such that forcing ℓ to be true followed by unit propagation results in a contradiction. This and other results about HypBinRes are proved in the above references.

In addition to uncovering unit clauses we can use the binary clauses to perform equality reductions. In particular, if we have two clauses $(\neg x, y)$ and $(x, \neg y)$ we can replace all instances of y in the formula by x (and $\neg y$ by $\neg x$). This might result in some tautological clauses which can be removed, and some clauses which are reduced in length because of duplicate literals. This reduction might yield new binary or unary clauses which can then enable further HypBinRes inferences. Taken together HypBinRes and equality reduction can significantly reduce a SAT formula removing many of its variables and clauses [2].

4 Preprocessing QBF

Given a QBF $\boldsymbol{Q}.F$ we could apply HypBinRes, unit propagation, and equality reduction to F until closure. This would yield a new formula F', and the QBF $\boldsymbol{Q}'.F'$ where $\boldsymbol{Q}'$ is $\boldsymbol{Q}$ with all variables not in F' removed. Unfortunately, there are two problems with this approach. One is that the new QBF $\boldsymbol{Q}'.F'$ might not be Q-equivalent to $\boldsymbol{Q}.F$, so that this method of preprocessing is not sound. The other problem is that we miss out on some important additional inferences that can be achieved through universal reduction. We elaborate on these two issues and show how they can be overcome.

The reason why the straightforward application of HypBinRes, unit propagation and equality reduction to the body of a QBF is unsound, is that the resulting formula F' does not have exactly the same SAT-models as F, as is required by condition **A** above. In particular, the models of F' do not make assignments to variables that have been removed by unit propagation and equality reduction. Hence, a QBF-model of $\boldsymbol{Q}'.F'$ might not extendable to a QBF-model of $\boldsymbol{Q}.F$. For example, if unit propagation forced a universal variable in F, then $\boldsymbol{Q}'.F'$ might be QSAT, but $\boldsymbol{Q}.F$ is not (no QBF-model of $\boldsymbol{Q}.F$ can exist since the paths that set the forced universal to its opposite value will not be SAT-models of F). However, it is easy to fix this problem. Making unit propagation sound for QBF simply requires that we regard the unit propagation of a universal variable as equivalent to the derivation of the empty clause (i.e. false). This fact is well known and applied in all search-based QBF solvers.

Ensuring that equality reduction is sound for QBF is a bit more subtle. A sound version of equality reduction must respect the variable ordering. That is, if we detect that x and y are equivalent and $x <_q y$ then we always remove y from the theory replacing it by x. In addition, the case when y is universal must be considered equivalent to deriving the empty clause. We call this ($<_q$ preferred) equality reduction.

Although applying HypBinRes, unit propagation (where propagating a universal is like deriving False) and ($<_q$ preferred) equality reduction will be sound, this approach does not fully utilize the power of universal reduction (condition **B** above). So instead we use a more powerful approach that is based on the following modification of HypBinRes that "folds" universal reduction into the inference rule. We call this rule "HypBinRes+UR":

> Given a single n-ary clause $c = (l_1, l_2, ..., l_n)$, D a subset of c, and the set of binary clauses $\{(\ell, \neg l) | l \in D\}$, infer the universal reduction of the clause $(c \setminus D) \cup \{\ell\}$ if this reduction is either binary or unary.

For example, from $c = (u_1, e_3, u_4, e_5, u_6, e_7)$, $(e_2, \neg e_7)$, $(e_2, \neg e_5)$ and $(e_2, \neg e_3)$ we infer the new binary clause (u_1, e_2) when $u_1 \leq_q e_2 \leq_q e_3 \leq_q u_4 \leq_q e_5 \leq_q u_6 \leq_q e_7$. This example shows that HypBinRes+UR is able to derive clauses that HypBinRes cannot. Since clearly HypBinRes+UR can derive anything HypBinRes can, HypBinRes+UR is a more powerful rule of inference.

In addition to using universal reduction inside of HypBinRes we must also use it when unit propagation is used. For example, from the two clauses $(e_1, u_2, u_3, u_4, \neg e_5)$ and (e_5) (with $e_1 <_q u_i$) unit propagation by itself can only derive (e_1, u_2, u_3, u_4), but unit propagation with universal reduction can derive (e_1).

It turns out that in addition to gaining more inferential power, universal reduction also allows us to obtain the unconditionally sound preprocessing we would like to have.

Proposition 1 *Let F' be the result of applying HypBinRes+UR, unit propagation, universal reduction and ($<_q$ preferred) equality reduction to F until closure, where we always apply universal reduction before unit propagation. Then the QBF-models of $\boldsymbol{Q}'.F'$ are in 1-1 correspondence with the QBF-models of $\boldsymbol{Q}.F$.*

This result can be proved by showing that universal reduction generates the empty clause whenever a universal variable is to be unit propagated or removed via equality reduction.

Proposition 2 *Applying HypBinRes+UR, unit propagation, universal reduction and ($<_q$ preferred) equality reduction to $\boldsymbol{Q}.F$ until we reach closure can be done in time polynomial in the size of F.*

Prequel modifies $\boldsymbol{Q}.F$ exactly as described in Proposition 1. It applies HypBinRes+UR, unit propagation, universal reduction, and ($<_q$ preferred) equality reduction to F until it reaches closure. It then outputs the new formula $\boldsymbol{Q}'.F'$. Proposition 1 shows that this modification of the formula is sound. In particular, this preprocessing does not change the QSAT status of the formula.

To implement Prequel we adapted the algorithm presented in [2] which exploits a close connection between HypBinRes and unit propagation. In particular, this al-

Solver	*Skizzo*		*Quantor*		*Quaffle*		*Qube*		*SQBF*	
	no-pre	*pre*	*no-pre*	*pre*	*no-pre*	*pre*	*no-pre*	*pre*	*no-pre*	*pre*
# Instances	311	**351**	262	**312**	226	**238**	213	**243**	205	**239**
Time on common instances	9,748	**9,595**	10,384	**2,244**	36,382	**20,188**	41,107	**23,196**	46,147	**25,554**
Time on new instances	-	12,756	-	16,829	-	9,579	-	9,707	-	2,421

Table 1. For each solver we show its number of solved instances among all tested benchmark families with and without preprocessing, the total CPU time (in seconds) required to solve the preprocessed and un-preprocessed instances taken over the “common” instances (instances solved in both preprocessed and un-preprocessed form), and the total CPU time required by the solvers to solve the “new” instances (instances that can only be solved in preprocessed form).

gorithm uses trial unit propagations to detect new HypBinRes inferences. The main changes required to make this algorithm work for QBF were adding universal reduction, modifying the unit propagator so that it performs universal reduction prior to any unit propagation step, and modifying equality reduction to ensure it respects the quantifier ordering.

5 Empirical Results

We considered all of the non-random benchmark instances from QBFLib (2005) [6] (508 instances in total). We discarded the instances from the benchmark families von Neumann and Z since these are all very quickly solved by any state of the art QBF solver (less than 10 sec. for the entire suite of instances). We also discarded the benchmark families Jmc and Jmc-squaring. None of these instances (with or without preprocessing) can be solved within our time bounds by any of the QBF solvers we tested. This left us with 468 remaining instances from 19 different benchmark families. We tested our approach on all of these instances.

All tests were run on a Pentium 4 3.60GHz CPU with 6GB of memory. The time limit for each run of any of the solvers or the preprocessor was set to 5,000 seconds.

We studied the effect Prequel has on the performance of five state of the art QBF solvers **Quaffle** [9] (version as of Feb. 2005), **Quantor** [4] (version as of 2004), **Qube** (release 1.3) [7], **Skizzo** (v0.82, r355) [3] and **SQBF** [8]. Quaffle, Qube and SQBF are based on search, whereas Quantor is based on variable elimination. Skizzo uses mainly a combination of variable elimination and search, but it also applies a variety of other kinds of reasoning on the symbolic and the ground representations of the instances.

A summary of our results is presented in Table 1. The second row of the table shows the total time required by each solver to solve the instances that could be solved in both preprocessed and unpreprocessed form (the “common instances”). The data demonstrates that preprocessing provides a speedup for every solver. Note that the times for the preprocessed instances *include* the time taken by Prequel. On these common instances Quantor was 4.6 times faster with preprocessing, while Quaffle, Qube and SQBF were all approximately 1.8 times faster with preprocessing. Skizzo is only slightly faster on the preprocessed benchmarks (that it could already solve).

The first row of Table 1 shows the number of instances that can be solved within the 5000 sec. time bound. It demonstrates that in addition to speeding up the solvers on problems they can already solve, preprocessing also extends the reach of each solver, allowing it to solve problems that it could not solve before (within our time and memory bounds). In particular, the first row shows that the number of solved instances for each solver is significantly larger when Prequel is applied. The increase in the number of solved instances is 13% for Skizzo, 19% for Quantor, 5% for Quaffle, 14% for Qube and 17% for SQBF.

The time required by the solvers on these new instances is shown in row 3. For example, we see that SQBF was able to solve 34 new instances. None of these instances could previously be solved in 5,000 sec. each. That is, 170,000 CPU seconds were expended in 34 failed attempts. With Prequel all of these instances could be solved in 2,421 sec. Similarly, Skizzo expended 200,000 sec. in 40 failed attempts which with preprocessing could all be solved in 12,756 seconds.

In total, these results demonstrate that our preprocessing technique offers robust improvements to all of these different solvers, even though some of them are utilizing completely different solving techniques.

6 Conclusions

We have shown that preprocessing can be very effective for QBF and have presented substantial and significant empirical results to verify this claim. Nearly all of the publicly available instances are taken into account, and five different state of the art solvers are compared. Preprocessing with Prequel offers robust improvements across the different solvers among all tested benchmark families. The achieved improvement also includes almost 20 instances that to our knowledge have never been solved before.

References

1. Fahiem Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *Eighteenth national conference on Artificial intelligence*, pages 613–619, 2002.
2. Fahiem Bacchus and J. Winter. Effective preprocessing with hyper-resolution and equality reduction. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), Lecture Notes in Computer Science 2919*, pages 341–355, 2003.
3. M. Benedetti. skizzo: a QBF decision procedure based on propositional skolemization and symbolic reasoning. Technical Report TR04-11-03, 2004.
4. A. Biere. Resolve and expand. In *Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 238–246, 2004.
5. H. K. Büning, M. Karpinski, and A. Flügel. Resolution for quantified boolean formulas. *Inf. Comput.*, 117(1):12–18, 1995.
6. E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. http://www.qbflib.org/.
7. E. Giunchiglia, M. Narizzano, and A. Tacchella. QUBE: A system for deciding quantified boolean formulas satisfiability. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 364–369, 2001.
8. H. Samulowitz and F. Bacchus. Using SAT in QBF. In *Principles and Practice of Constraint Programming*, pages 578–592. Springer-Verlag, New York, 2005. available at http://www.cs.toronto.edu/˜fbacchus/sat.html.
9. L. Zhang and S. Malik. Towards symmetric treatment of conflicts and satisfaction in quantified boolean satisfiability solver. In *Principles and Practice of Constraint Programming (CP2002)*, pages 185–199, 2002.

Global Chance-Constraints: an Application to Stochastic Inventory Control

Student: Roberto Rossi
Supervisor: Steven Prestwich

Cork Constraint Computation Centre, University College, Cork, Ireland
{r.rossi,s.prestwich}@4c.ucc.ie

Abstract. Inventory theory provides methods for managing and controlling inventories under different constraints and environments. We consider a class of production/inventory control problems that has a single product and a single stocking location, when a stochastic demand with a known non-stationary probability distribution is given. A control policy for this type of inventory system is the one where the objective is to find the optimal number of replenishments, their timings and their respective order-up-to-levels that meet customer demands to a required service level. Two different models have been presented so far to solve this problem to optimality: a MIP model and an efficient CP model. In both these models negative orders are not allowed, so that if the actual stock exceeds the order-up-to-level for that review, this excess stock is carried forward and not returned to the supply source. Since this event is assumed to be rare, in both the models its effect is ignored. We present a *global chance-constraint* that lets us to compute exact buffer stock levels for the CP model by considering the effect that carrying excess stock has on the service level in each period of our planning horizon. An extended version of this paper hasn't been submitted to the technical programme.

1 Introduction

We consider the class of production/inventory control problems that refers to the single-location, single-product case under non-stationary stochastic demand. In this problem the following inputs are given: a planning horizon of N periods; and a demand d_t for each period $t \in \{1, \dots, N\}$, which is a random variable with probability density function $g_t(d_t)$. We will assume without loss of generality that these variables are normally distributed and that the demand occurs instantaneously at the beginning of each time period. The demand we consider is non-stationary, that is it can vary from period to period, and demands in different periods are assumed to be independent. A fixed delivery cost a is considered for each order and also a linear holding cost h is considered for each unit of product carried in stock from one period to the next. We assume that it is

[0] This work was supported by Science Foundation Ireland under Grant No. 03/CE3/I405 as part of the Centre for Telecommunications Value-Chain-Driven Research (CTVR) and Grant No. 00/PI.1/C075.

not possible to sell back excess items to the vendor at the end of a period. As a service level constraint we require the probability that at the end of each and every period the net inventory will not be negative set to be at least a given value α. Our aim is to minimize the expected total cost (ordering costs and holding costs) over the N-period planning horizon, satisfying the service level constraints.

Different inventory control policies [4] can be adopted to cope with the described problem. A policy states the rules to decide when orders have to be placed and how to compute the replenishment lot-size for each order. One of the possible policies that can be adopted is the replenishment cycle policy (R, S). Under the non-stationary demand assumption this policy takes the non-stationary form (R^n, S^n), where R^n denotes the length of the nth replenishment cycle, and S^n the order-up-to-level values for each replenishment. In order to provide a solution for our problem under the (R^n, S^n) policy we must populate both the sets R^n and S^n.

The first complete solution method for this problem was introduced by Tarim & Kingsman [2], who proposed a certainty-equivalent Mixed Integer Programming (MIP) formulation for computing (R^n, S^n) policy parameters. In [1] a more compact and efficient Constraint Programming (CP) formulation of the same problem has been introduced.

Both the MIP and the CP formulation assume that negative orders are not allowed, so that if the actual stock exceeds the order-up-to-level for that review, this excess stock is carried forward and not returned to the supply source, but since this event is assumed to be rare, in both the models its effects are ignored:

- The cost of carrying excess stock is ignored, therefore the actual cost of a policy can be higher than the one provided by the model
- The event of carrying excess stock can have a significant impact on the service level of the next periods, in particular it could be possible to exploit excess stock to provide the required service level keeping lower buffer stocks

This paper extends the CP model presented in [1] by expressing its service level constraint as a global chance-constraint able to dynamically compute exact buffer stock levels during the search. It has to be noticed that while in CP the former assumption can be relaxed using a dedicated global chance-constraint, this is not possible in MIP, where buffer stocks have to be pre-computed. CP is therefore not only a more efficient way than MIP for dealing with stochastic inventory control as shown in [1], but it is also a mandatory choice if we want to compute the optimal solution for the (R^n, S^n) policy. The paper is organized as follows. In Section 2 we describe the CP model for the (R^n, S^n) policy. In Section 3 we introduce our global chance-constraint. In Section 4 we show the effectiveness of our approach.

2 A CP model

In this section we review the CP formulation for the (R^n, S^n) policy proposed in [1]. For a detailed discussion of Constraint Programming see [5]. The CP formulation presented in [1] for the (R^n, S^n) policy is as follows:

$$\min \; E\{TC\} = \sum_{t=1}^{N} \left(a\delta_t + h\tilde{I}_t \right) \tag{1}$$

subject to, for $t = 1 \ldots N$

$$\tilde{I}_t + \tilde{d}_t - \tilde{I}_{t-1} \geq 0 \tag{2}$$

$$\tilde{I}_t + \tilde{d}_t - \tilde{I}_{t-1} > 0 \Rightarrow \delta_t = 1 \tag{3}$$

$$Pr\{\tilde{I}_t \geq 0\} \geq \alpha \tag{4}$$

$$\tilde{I}_t \in \mathbf{Z}^+ \cup \{0\}, \quad \delta_t \in \{0,1\} \tag{5}$$

Each decision variable $\tilde{I}_t$ represents the expected inventory level at the end of period t. The binary decision variables δ_t state whether a replenishment is fixed for period t ($\delta_t = 1$) or not ($\delta_t = 0$). The objective function (1) minimizes the total expected cost over the given planning horizon. Two terms contribute to the overall expected cost: ordering costs and inventory holding costs. Constraint (2) enforces a no-buy-back condition, which means that received goods cannot be returned to the supplier. As a consequence of this the expected net inventory at period t must be no less than the expected net inventory in period $t+1$ plus the expected demand in period t. Constraint (3) expresses the replenishment condition. We have a replenishment if the expected net inventory at period t is greater than the expected net inventory in period $t+1$ plus the expected demand in period t. This means that we received some extra goods as a consequence of an order. Chance-constraint (4) enforces the required service level α. In [1] such a constraint is expressed by means of pre-computed buffer stock levels for each possible replenishment cycle. The net inventory at the end of each replenishment cycle is therefore forced to be greater or equal to the respective minimum buffer stock. This means that the effect that excess stock from former periods has is not taken into account (Fig. 1). It is not possible to compute a-priori such

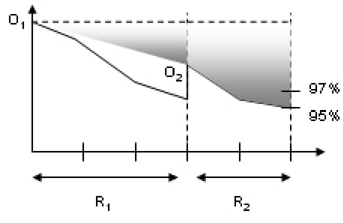


Fig. 1. Effect of excess stock: the buffer stock computed a-priori assures a 95% service level, but the combined effect of excess stocks from former periods produces a higher actual service level

an effect, because for each buffer stock it directly depends on the length of former replenishment cycles. In order to take this effect into account we can implement a global chance-constraint that dynamically computes buffer stock levels depending on the current partial assignment of the δ_i variables.

3 A new service level global chance-constraint

In this section, in order to describe our global chance-constraint, we will exploit the following property of buffer stocks.

Property 1. The buffer stock for any replenishment cycle depends only on the length of former replenishment cycles and not on subsequent cycle lengths.

We will consider now a two replenishment cycle case (Fig. 1) in an N period planning horizon, then we will extend the idea in a recursive fashion to the case of M subsequent replenishment cycles.

The planning horizon is made up of two consecutive replenishment cycles, let us call them R_1 and R_2. Let O_i be the opening inventory level for R_i. We assume that O_1 is known (Property 1). We define $P(H)$ as the probability of the event "observing a demand higher than $O_1 - O_2$ during R_1". $P(D_x)$ is the probability of the event "observing a demand less or equal to x, where $x \in \{0, ..., O_1 - O_2\}$, during R_1". S_y is the service level at the end of R_2 if a buffer stock y is hold. Then the correct buffer stock for R_2 can be computed as the minimum value b s.t.

$$P(H) \cdot S_b + \sum_{i=0}^{O_1-O_2} (P(D_i) - P(D_{i-1})) \cdot S_{b+O_1-O_2-i} \geq \alpha \tag{6}$$

where $O_2 = \tilde{d}_2 + b$. For the two replenishment cycles case, this can be rewritten using the following extended form

$$(1 - G^{-1}(\frac{Q}{\sigma_1})) \cdot G^{-1}(\frac{b}{\sigma_2}) + \sum_{i=-d_1}^{Q} (G^{-1}(\frac{i}{\sigma_1}) - G^{-1}(\frac{i-1}{\sigma_1})) \cdot G^{-1}(\frac{b-(i-Q)}{\sigma_2}) \geq \alpha \tag{7}$$

where $Q = O_1 - O_2 - \tilde{d}_1$, G^{-1} is the inverse normal cumulative distribution function, σ_i is the standard deviation of the demand for the replenishment cycle i. Notice that if the opening inventory level of R_1 is smaller than the opening inventory level of R_2, obviously the former cycle has no influence on the buffer stock and Condition 6 becomes $S_b \geq \alpha$. Furthermore, if the computed b is s.t. $R_2 \leq R_1 - \tilde{d}_1$, we just set the buffer stock to the minimum value allowed, that is $R_1 - \tilde{d}_1 - \tilde{d}_2$. Finally we should observe that, since we are using the standard normal distribution function and not the truncated one, we have to use the following normalized term in condition 6.

$$\frac{(1 - P(H))}{\sum_{i=0}^{O_1-O_2}(P(D_i) - P(D_{i-1}))} \cdot \sum_{i=0}^{O_1-O_2} (P(D_i) - P(D_{i-1})) \cdot S_{b+O_1-O_2-i}$$

We now define a **global chance-constraint** *serviceLevel*$(\tilde{I}, \delta, d, \alpha)$, where $\tilde{I}$ and δ are arrays of decision variables, d is an array of random variables with probability density function $g(d)$ and α is the required service level. This constraint assures for each replenishment period that, at the end of each and every time period, the probability the net inventory will not be negative is at least α. It is therefore semantically equivalent to Constraint 4 for $t = \{1, \dots, N\}$ and it can be used to express these constraints in the CP model. It has to be noticed that the *global view* provided by this constraint lets us to consider joint probabilities during the search, which are ignored instead when the same condition is expressed by means of many independent constraints as shown in [1]. In order to propagate this constraint at each node of the search tree if at least a decision variable δ_i, $i \in \{1, ..., N\}$ that has not been assigned yet exists we don't enforce any service level constraint; otherwise if $\exists \delta_i$ s.t. $\delta_i = 1$, we know that a replenishment cycle starts in period i and it covers subsequent periods till the minimum j, $j \geq i$ s.t. $\delta_{j+1} = 1$ or $j + 1 > N$. Property 1 assures that we can consider replenishment cycles in our planning horizon in a sequential fashion. Therefore we can generate the buffer stock for the first replenishment cycle, which is not affected by any other replenishment period, then we can generate the buffer stock for the second, which is only affected by the first one, etc. Each time we compute the buffer stock level b for a replenishment cycle we can remove from the domain of I_t, where t is the last period in the replenishment cycle, every value smaller than b.

In order to present the general case of M replenishment cycles it has to be noticed that the buffer stock of a replenishment cycle R_j is affected only by former replenishment cycles $\{R_i, \dots, R_{j-1}\}$, where i, $i \leq j$, is the max value s.t. $O_{i-1} < O_i$. If $i = j$ no former replenishment cycle affects R_j. Now $P(H)$ is the probability of the event "observing an inventory level that is less or equal to O_j in period $j - 1$", while $P(O_x)$ is the probability of the event "observing an inventory level that is equal to x in period $j - 1$", where $x \in \{O_j + 1, ..., O_i\}$. Since we know the distribution of the demand in periods $\{i, \dots, j\}$ and since former buffer stocks in periods $\{i, \dots, j - 1\}$ have been already set (Property 1), it is easy to recusively compute such probabilities by using a *scenario based approach*. We can therefore extend Condition 6 to compute the buffer stock b for R_j.

$$P(H) \cdot S_b + \frac{(1 - P(H))}{\sum_{x=O_j+1}^{O_i} P(O_x)} \cdot \sum_{x=O_j+1}^{O_i} P(O_x) \cdot S_{b+x-O_j} \geq \alpha \tag{8}$$

4 An example

Let us compare the solution provided when our global chance-constraint is used and the one provided by the original CP model. We assume an initial null inventory level and a normally distributed demand with a coefficient of variation $\sigma_t/\tilde{d}_t$ for each period $t \in \{1, \dots, 4\}$. The expected values for the demand in each period are $\{120, 70, 50, 40\}$. The other parameters are $a = 150$, $h = 1$, $\sigma_t/\tilde{d}_t = 0.4$, $\alpha = 0.8(z_{\alpha=0.8} = 0.8414)$. In Table 1 the optimal solution found when our global chance-constraint is used to dynamically generate buffer stock levels is compared with the one obtained by using a pre-computed matrix. It is

Original buffer stock computation			Dynamic buffer stock computation		
Policy cost: 548			Policy cost: 542		
I_1 117	δ_1 1	Service 99.2	I_1 117	δ_1 1	Service 99.2
I_2 47	δ_2 0	Service 80.4	I_2 47	δ_2 0	Service 80.4
I_3 62	δ_3 1	Service 100	I_3 59	δ_3 1	Service 99.8
I_4 22	δ_4 0	Service 82.8	I_4 19	δ_4 0	Service 80.1

Table 1. Optimal solution comparison

possible to see that by computing correct buffer stock levels we obtained a less costly policy, still meeting the required service level of 80%. In fact were able to keep lower stocks in the last two periods exploiting the effect of excess stocks carried on from former periods.

5 Conclusions

In [1] it has been shown that CP is a more natural and efficient way, compared to MIP, for expressing constraints for lot-sizing under the (R^n, S^n) policy. In this paper we showed that in CP it is also possible to dynamically consider during the search the effect of excess stocks from former replenishment cycles on the optimal buffer stock of a given period. When the service level constraint is expressed using the global chance-constraint we presented, the CP model can provide a better solution than the one produced by the MIP model or by the original CP model. As a future extension we aim to incorporate cost-based filtering methods in order to let our approach scale.

References

1. S. A. Tarim, B. Smith. Constraint Programming for Computing Dynamic (R,S) Inventory Policy With Non-Stationary Stochastic Demand Under Service Level Constraints. Cork Constraint Computation Center, UCC, Ireland, 2005.
2. S. A. Tarim, B. G. Kingsman. The Stochastic Dynamic Production/Inventory Lot-Sizing Problem With Service-Level Constraints. *International Journal of Production Economics* 88:105–119, 2004.
3. H. M. Wagner, T. M. Whitin. Dynamic Version of the Economic Lot Size Model. *Management Science* 5:89–96, 1958.
4. R. Peterson, E. Silver, D. F. Pyke. Inventory Management and Production Planning and Scheduling. John Wiley and Sons, New York, 1998.
5. K. Apt. Principles of Constraint Programming. Cambridge University Press, Cambridge, UK, 2003.
6. A. Charnes, W. W. Cooper. Chance-Constrainted Programming. *Management Science* 6(1):73–79, 1959.
7. L. Fortuin. Five Popular Probability Density Functions: a Comparison in the Field of Stock-Control Models. *Journal of the Operational Research Society* 31(10):937–942, 1980.
8. I. J. Lustig, J.-F. Puget. Program Does Not Equal Program: Constraint Programming and its Relationship to Mathematical Programming. *Interfaces* 31:29–53, 2001.

Mixed CSP Techniques Applied to Embodiment Design

Student: Raphaël Chenouard[2]
Advisors: Laurent Granvilliers[1] and Patrick Sebastian[2]
raphael.chenouard@bordeaux.ensam.fr, laurent.granvilliers@univ-nantes.fr,
patrick.sebastian@bordeaux.ensam.fr

[1] University of Nantes, Laboratoire d'Informatique de Nantes Atlantique, CNRS, BP 92208, F-44322 Nantes Cedex 3
[2] ENSAM Bordeaux, TRansferts Ecoulements FLuides Energétique, CNRS, F-33405 Talence Cedex

1 Embodiment Design

Design of new products in industry requires the investigation of product life cycle phases and the definition of product models corresponding to these different phases: needs, design requirement, product design, manufacturing, etc. Phases are identified by some authors [9, 12] within design processes: need and requirements definition, conceptual design, embodiment design and detailed design. More to the point, various design methods are used to optimize industrial products: design for X, ecodesign, robust design, etc. Robust design is based on modeling and used to assess product characteristics taking into account design process variabilities inherent to the system and its components. This paper is interested in modeling and numerical treatment in CSP based Robust Embodiment Design.

Embodiment design phase aims at choosing the main structuring characteristics (working structure, standard components, main dimensions, etc.) of the mechanical system being designed. The phase starts from knowledge established during the conceptual design phase and leads to feasible architectures (embodiment design solutions). Physics behavior and interactions of the components are considered to investigate the mechanical system's feasibility. Functional architectures describing the main chosen concepts are also considered through this design phase.

Embodiment design models consist of relations taking into account discrete variables (choices of standardized component, catalogs references) or continuous variables (component dimensions, physics phenomena). Physics behaviors, dimensions and architectures, costs, manufacturing constraints are expressed using the same formalism. The model is built from the conjunction of these constraints. Some of them may be related to logical formulae and are taken into account as conditional constraints.

2 Constraint Satisfaction Techniques

Design problems may be expressed as a set of constraints, defining relations between variables. The constraint satisfaction problem (CSP) formalism seems suitable for the modeling and solving of this type of problems. It is defined as a triple $\langle V, D, C\rangle$, where:

- $V = \{v_1, ..., v_n\}$ is the set of variables.
- $D = \{d_1, ..., d_n\}$ is the set of domains associated with the variables.
- $C = \{c_1, ..., c_k\}$ is the set of constraints restricting the variables' domain.

$S = \{s_1, ..., s_l\}$ is the set of a CSP solutions, such as for each s_i: each variable from V is affected to a value and all the constraints from C are satisfied.

Two types of CSP are identified: discrete and continuous CSP. The first type is concerned with domains expressed as finite sets of values, whereas the second one takes into account domains defining infinite sets of values conservatively discretized as intervals or union of intervals. The solving algorithms are in general specific to these two approaches. CSP's solutions are computed using mainly branch-and-prune algorithms [5] with two sorts of algorithms: consistency and search algorithms. Search algorithms explore variable's domains. Discrete domains are enumerated and continuous domains are splitted until a defined precision. All the real numbers cannot be represented and a set of finite approximation of floating point number enclosing the real solutions are computed [6]. As the search space may be huge, consitency algorithms [7] prune domains using the constraints set. Consistency techniques on continuous domains use labeling [3] or approximations and projections of domains. Consistency techniques on real constraints use the interval arithmetic to compute a complete set of approximated solutions [11, 1].

Dynamic CSP [8] or Conditional CSP (CondCSP) [4] appear to be more suitable than classical CSP approaches for supporting design and configuration problem solving. CondCSP only considers a relevant set of active variables, which determines the use of constraints. This formalism brings in the concept of active or inactive variable. The declaration of a CondCSP is therefore a quadruple. The three first sets are the same ones as within the classical CSP frame. The fourth set is the set of the initially active variables. Two types of constraints are defined in this formalism:

- Compatibility constraints define allowed value combinations for variables. They are only active if all their concerned variables are active.
- Activation constraints describe which variables to add to the active set. That is a logical implication, where a constraint is a condition under which a variable becomes active.

Dependency and causality of variables are clearly represented and more relevant search heuristics may be used [10].

Mixed constraints are often involved in configuration and design problems. These constraints are both applying on continuous and discrete variables [4].

Computation algorithms must be adapted to cope with these problem specificities. Currently, the computation algorithms are well suited for problems where discrete variables are prevailing. Indeed, these algorithms are based on bisections on intervals and on split at the midpoint (for discrete and continuous domains). Consistency check is performed at this point and the midpoint is removed from the solution set as soon as an inconsistency is detected. The two parts created are explored in the same way until the computation precision is reached for each domain. Continuous domains are discretized with each midpoint created. Moreover, the causality between variables expressed in this formalism is in conflict with some physics laws. Indeed, they are defined by several relations on the same variables and, therefore, all these constraints are active at the same time. Consequently the problem becomes inconsistent.

It may be noted that the formalisms presented do not fit completely with the design needs. Better algorithms for mixed constraints have to be used, because intervals are discretized into many midpoints. This approach does not take into account the real continuous meaning of such domains. Moreover, knowledge resulting from the design process can't be completely expressed and more specialized algorithms and formalism may be define to meet the design needs [14].

3 PhD Thesis

Our work takes place during the embodiment phase and aims to adapt CSP techniques to this application domain. Indeed, a new type of constraint is necessary to express directly conditional dependency between conditions and constraint activations. A condition is a logical expression of arithmetic relations expressed as constraints. Constraints are explicitly activated and this highlights dependency between variables from a condition and a constraint. The causality of variables is not taken into account and is not relevant in this design phase as component occurrences and their activations (design concepts) are not studied in embodiment design. The design concepts are chosen during the conceptual design phase. The components which have to be to chosen are often described using the same variables and criteria. These variables identify references (catalog of components) or key characteristics indicating which component is being used. Moreover, physics knowledge may be expressed using sets of relations, each relation being related to a specific behavior. The global behavior of the component emerges from the interaction between these specific behaviors.

Modeling in embodiment design requires the use of different types of variables. Design variables (DV) assess the main characteristics of a mechanical system. Their values define design alternatives. Auxiliary variables (AV) are introduced by designers during the modeling phase to link design variables to design criteria and assess the design alternative performances. These AV are redundant from the designer point of view, because they are not involved in the decision making process. However, they appear to be necessary in order to preserve the coherence and intelligibility of the model. Thus, two types of variables may be distinguished in CSP dedicated to embodiment design and have to be used to improve search heuristics.

Specific split strategies for bisections may also be used. Indeed, the use of conditional constraints allows us to know domains bounds, where constraints are activated through the expressed conditions. Splitting on these bounds activates more quickly the relevant constraints and consequently prunes also quickly the variables domains. Some problems appear as soon as continuous domains have to be split, because of the interval bounds, which must be integers or floating-point numbers. Otherwise, these bounds have to be approximated to the smallest hull of floating-point numbers, where constraints are not activated. On this interval, only consistency check can be done.

All these heuristics can be associated with other heuristics based on the CSP structure [2]. These approaches prove their efficiency, but, keeping the DV and AV heuristics requires a compromise between these two heuristics. Moreover, conditional constraints transform progressively the CSP structure throughout the solving process. Two approaches have to be considered: to compute dynamically, after the activation of each constraint, the new relevant order on the variables set, or to compute initially this order while taking into account the whole conditions.

Mechanical systems models are often composed of AV, which are explicitly defined. They are not essential for the model solving, but they enlarge the search space and decrease the solving algorithm efficiency. We choose to avoid splitting these variables and define them as arithmetic relations. The resolution algorithm doesn't lose time on splitting their domains, while the model's intelligibility is preserved. These variables are called aliases in the next paragraphs. For instance, the adimensional Reynolds number is used in fluid mechanics to link fluid viscosity and velocity to flowing dimensions. The use of Reynolds number by designers facilitates fluid mechanics modeling as well as model comprehension and reusability. It is expressed by the relation $Re = \frac{\rho \cdot V \cdot d}{\mu}$, and there are no need to explore its domain as soon as ρ, V, d and μ are known. Moreover aliases may be defined with a relation including other variables or aliases. However, aliases must not be linked through any dependency cycle.

4 First Results

Search heuristics have been implemented in a branch-and-prune framework taking into account the DV, AV and aliases properties. These algorithms are tested in the following paragraph on a small model of batch-exchanger, where a fluid is cooled down and is graded. The model takes into account 5 DV and a set of 10 AV including 6 aliases. There are also 6 conditional constraints that express choices in components catalogs of materials, fins and tubes. The first algorithm `solve_DV` computes all the solutions and starts to explore the search space with DV domains. Table 1 points out that the algorithm efficiency is quite better than the one of the classical B&P algorithm. 5282 solutions are computed in 7.06 s with 6509 splits on domains, whereas the classical algorithm computes 5376 and requires 7592 splits in 8.26 s. Choices in catalogs are performed by using DV and by starting to compute their values. The tube diameter characteristics, fin efficiencies and exchange surfaces are related to single values, whereas bisections are computed on their domains by the classical algorithms.

The second algorithm (`solve_DV_AV`) uses the AV to limit the search space exploration to the computation of embodiment design solutions. As mentioned earlier, designers have no interest in AV to make decisions. For each embodiment design solution, this algorithm only searches one value for each AV to validate the DV's values as values corresponding to a feasible solution. 56 solutions have been obtained. Computing time and number of bisections are quite lower than using a classical B&P algorithm or `solve_DV`. The completeness of the solution set is guarantied by restoring the domains of all AV to the state corresponding to the last DV bisection. For these solutions, AV precisions don't comply with their initial precision. However, this loss of precision is not significant from the designer point of view.

batch-exchanger	Solutions	Time (s)	Bisections
Classical B&P algorithm	5376	8.26	7592
Classical B&P algorithm + Alias	5395	8.24	7519
`solve_DV` algorithm	5282	7.06	6509
`solve_DV` algorithm + Alias	2701	5.72	2883
`solve_DV_AV` algorithm	56	0.63	677
`solve_DV_AV` algorithm + Alias	56	0.62	519

Fig. 1. Results of a batch-exchanger resolution, comparing classical solving algorithm with others that take into account of DV, AV and aliases

Figure 1 points out the impact of aliases on the solving time and on the number of solutions. Time appears to be quite better and the number of splits on domains are the lowest. These results are derived from the low number of AV and consequently, from less precision errors on DV. Indeed, AV splits induce reductions of the DV domains, and generate many solutions which are not required by designers. Aliases avoid the splitting of some domains and decrease AV impact on DV's pruning. It can be noted that in the classical B&P approach the number of solutions increase, but not the number of splits. In more complex models of multi-scale design problems, aliases appear to be very useful and increase the relevance of the approach proposed in this paper [13].

5 Prospects

The algorithms presented in this paper are based on variable characterization in robust embodiment design. To improve algorithm efficiency, we plan to implement automatic decomposition algorithms taking into account DV's priority. DV's precisions errors due to AV splits have been highlighted in this article.

However, some Auxiliary Variables are not aliases and some efforts have to be performed to better take into account the precision related to these remaining auxiliary variables.

In the long run, other significant aspects resulting from the embodiment design process have to be taken into account. For example, designers have to design product through different life cycle situations. The same product (characterized by the same Design Variables) has to be designed considering several environments. This difficulty may be related to Dynamic CSP or Conditional CSP domains. More to the point, new algorithms have to be developed to better take into account global constraints involved in the formulation of multi-scale design problems. These global constraints are mainly related to matrix algebra and to the catalogs of models used by designers to choose standard components.

The algorithms presented in this paper are currently tested on multi-scale embodiment design problems such as the design of aircraft air conditioning systems, wind turbines, etc. Our approach proves to broadly extend the scope of application of CSP solvers in these domains as decision support systems.

References

1. F. Benhamou and W.J. Older. Applying interval arithmetic to real, integer and boolean constraints. *Journal of Logic Programming*, 1(32):1–24, 1997.
2. C. Bliek, B. Neveu, and G. Trombettoni. Using graph decomposition for solving continuous csps. *Principles and Practice of Constraint Programming, CP'98*, Springer LNCS 1520:102–116, 1998.
3. Boi Faltings. Arc Consistency for Continuous Variables. *Artificial Intelligence*, 65(2):363–376, 1994.
4. Esther Gelle and Boi Faltings. Solving mixed and conditional constraint satisfaction problems. *Constraints*, 8(2):107–141, 2003.
5. E. Hyvönen. Constraint Reasoning Based on Interval Arithmetic. In *IJCAI'89*, Detroit, USA, 1989.
6. O. Lhomme. Consistency Techniques for Numeric CSPs. In *IJCAI'93*, Chambéry, France, 1993.
7. A.K. Mackworth. Consistency in networks of relations. *Journal of Artificial Intelligence*, 8:99–118, 1977.
8. Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *AAAI*, pages 25–32, 1990.
9. G. Pahl and W. Beitz. *Engineering design: A systematic approach.* Springer-Verlag Berlin Heidelberg, 1996.
10. Mihaela Sabin, Eugene C. Freuder, and Richard J. Wallace. Greater efficiency for conditional constraint satisfaction. In *CP 2003*, pages 649–663, 2003.
11. D. Sam-Haroud and B. Faltings. Consistency techniques for continuous constraints. *Constraints*, 1(1&2):85–118, sep 1996.
12. Dominique Scaravetti. *Formalisation préalable d'un problème de conception, pour l'aide à la décision en conception préliminaire.* PhD thesis, ENSAM, 2004.
13. P. Sébastian, J.P. Nadeau, and S. Aso. Numeric-csp for air-conditionning in aeronautics. In *8th World Multi-Conference on SCI*, Orlando, USA, 18-21 July 2004.
14. Elise Vareilles. *Conception et approches par propagation de contraintes : contribution à la mise en oeuvre d'un outil d'aide interactif* . PhD thesis, Institut National Polytechnique de Toulouse, École des Mines d'Albi, 2005.

The Portfolio Selection Problem: Opportunities for constrained–based metaheuristics

Student: Giacomo di Tollo
Supervisor: Andrea Roli

Dipartimento di Scienze
Università "G.D'Annunzio" Chieti–Pescara
ditollo@sci.unich.it

Abstract. The Portfolio Selection Problem is a well-known area of application for metaheuristics, but its basic formulation fails in incorporating real-world features. In this work we discuss some issues about how to enrich the model by introducing features and constraints to obtain realistic results.

Introduction

Portfolio selection is one of the most studied topics in finance: the problem (referred to as PSP), in its basic formulation, is concerned with selecting the portfolio of assets which minimize the risk, given a certain level of returns. The basic model is formulated in the seminal work by Markowitz[6], in which the formulation of the problem is given by minimizing the variance (as a risk measure) for a given level of return:

$$\min \sum_{i=1}^{n} \sum_{j=1}^{n} \sigma_{ij} x_i x_j \tag{1}$$

$$\sum_{i=1}^{n} r_i x_i \geq r_p \quad \sum_{i=1}^{n} x_i = 1 \quad x_i \in [0,1] \quad i,j = 1 \ldots n \tag{2}$$

where σ_{ij} represents covariance between assets i and j, r_p is the expected return rate and r_i is the (actual or forecasted) return rate of asset i. Note that portfolios are modeled as sets of assets whose weight sum up to one and can assume any value in the range [0, 1].

In this formulation the problem is solvable with exact methods, but when adding additional features, it becomes untractable even for small instances. So metaheuristic approaches have been exploited to solve realistic instance of portfolio selection[2][7].

In the following we will discuss about features that improve PSP formulation by considering real world investor behavior. These features can be modeled as constraints in a CP framework and efficiently tackled by solution procedures as metaheuristics.

1 Constraints

A shortcoming of the introduced formulation is that it lacks incorporating many aspects of real-world trading: maximum size of portfolio, minimum lots, transaction costs, preferences of which assets to include in the portfolio and by how much, management costs, etc. These aspects can be formulated by introducing constraints and in the following we will introduce some of the most relevant ones.

Cardinality Constraints The number of assets in the portfolio is limited. Introducing for each asset a binary variable z ($z = 1$ if asset is in the portfolio and 0 otherwise), the constraint can be expressed as follows:

$$\sum_{i=1}^{n} z_i = k \tag{3}$$

This constraint can be defined also in inequality form, imposing that the portfolio must contain no more ($\leq$) than k assets, and can be, of course, expressed also as a global cardinality constraint in CP.

Floor and Ceiling Constraints With this constraint we impose a minimum and maximum proportion (ε_i and δ_i) allowed to be held for each asset in portfolio. In other words the portion of the portfolio for a specific asset (each asset or some of them) must be included in a fixed interval. Generally, floor constraint is used to avoid the cost of administrating tiny portions of assets, while ceiling constraints to avoid excessive exposure to a specific asset (in some case it is imposed by law).

$$\varepsilon_i z_i \leq x_i \leq \delta_i z_i \tag{4}$$

Minimum lots In the literature, investments are generally continuously divisible, so as to be represented by a real variable, while in real world securities are negotiated as multiples of minimum lots: for each asset there exists a minimum tradable lot, generally referred to as *round*. This constraint cannot be added in the continuous model since rounds are expressed in money, while in the continuous model assets-portions are chosen regardless of their absolute value. For these reasons minimum lots are generally encountered only when dealing with the integer formulation[5], in which assets are labeled by their actual value rather than their ratios to whole portfolio. In integer values, if p_j is the price of asset j and ml_j its minimum tradable quantity, the minimum lot expressed in money is given by $c_j = ml_j p_j$.

Adding those issues to the original formulation makes the problem very hard to be solved by exact methods. Hence the need for designing efficient approximate algorithms, such as metaheuristics[1].

2 Neighborhood and Repair mechanism

In order to develop and fathom powerful local search strategies a key point is to define and understand the neighborhood relationship. This is a crucial point in metaheuristics: Often in the literature the introduction of neighborhood is not grounded to explicit motivations; this can lead to a misunderstanding of the algorithm behavior or to wrong conclusion referring to the applicability of such algorithm to specific problems (or instances). The problem formulation we are discussing requires the definition of constraints of various nature to model real-world features, and when local search is dealing with constraints, the neighborhood can be implemented in the following ways:

- *all feasible* approach: each candidate solution $s\prime$, belonging to the neighborhood of a current solution s, must satisfy the constraints at any step of the search process;
- *repair* approach, in which if a non-feasible solution is found, this is suddenly forced to satisfy constraints by an embedded repair-mechanism;
- *penalties* approach: we allow moving toward non-feasible solutions, but those will be assigned a penalty in the objective function, depending on the amount of violation.

Sometimes it turns out to be difficult determining which class a search method belongs to, as can be difficult to determine if a search trajectory moves only in feasible areas because of its formulation or because an implicit repair mechanism is embedded. Repair-mechanism has the effect to consider a large number of candidate solutions, but, in our opinion, it can cause loss of information and waste good partial solution features, even if it has the effect of reducing execution time.

A typical repair mechanism is explained in Streicher et al.[8], referring to a formulation with cardinality and minimum lots. This procedure takes as input a non normalized weight-vector, in which each weight represents the portion of portfolio held by an asset, and operates as follows:

1. all weights are normalized so as to sum to one. This is done by setting weights $x_i\prime = x_i / \sum_j x_j$;
2. the obtained vector is normalized so as to meet cardinality constraint: Only the k assets with largest value of $x_i\prime$ are held and then normalized to sum up to one;
3. a further normalization is required to meet minimum lots constraints: Weights of assets are forced to the previous roundlot level $x_i\prime\prime = x_i\prime - (x_i\prime \bmod c_i)$. The free portfolio amount is redistributed so as to meet minimum lots constraint buying quantities of c_is on assets with biggest $(x_i\prime \bmod c)$ until all the remainder is spent.

In this mechanism, at point 1), the repair mechanism operates normalizing all assets in the portfolio. Nevertheless, there is evidence that investors choose, for their portfolio, one highly risky asset (or a few ones) with high weight, while the remainder is partitioned in lots of small weights used to reduce risk.

In this situation, the former repair mechanism would loose important informations about the structure of portfolio. For this reason a new repair mechanism able to return a feasible solution composed of a vector $\overline{feas}$ can be defined just replacing the point 1) of the previous mechanism with the following routine:

1. Order assets in non-increasing-weights. Let o be the resulting vector[1];
2. Compute the vector $d = (o_1 - o_2), (o_2 - o_3), \dots (o_{n-1} - o_n)$; let d_i be the i-th component in the sequence;
3. Let m be the index of the maximum element in d: This represents the maximum distance between weights of adjacent assets in the ordered array o;
4. **if** $\sum_{a=1}^{m} o_a \geq 1$ return the vector

$$\overline{feas}_i = \frac{o_i}{\sum_{j=1}^{n} o_j}$$

5. **if** $\sum_{a=1}^{m} o_a < 1$ return the vector

$$\overline{feas}_i = \begin{cases} o_i & i = 1 \dots m \\ o_i \cdot \frac{1 - \sum_{j=1}^{m} o_j}{\sum_{l=m+1}^{n} o_l} & i = (m+1) \dots n \end{cases}$$

3 Integer versus Continuous Formulation

The formulation we introduced (continuous fractional formulation in which weights must sum up to one) is universally used as standard approach in metaheuristics formulation: Modern Portfolio Theory relies on this formulation since it was introduced by Markowitz[6], but it represents a simplified model of real–world situations. We introduce now two issues difficult to handle with the continuous formulation.

Transaction costs Transaction costs are difficult to manage for the peculiar type of their function. As stated in Konno and Wijayanayake[3], the total costs follow a non-convex function on the size of the transaction: at the beginning it is concave up to a certain point (unit-transaction cost gradually decrease as size increase), then increases linearly to another certain point (unit-transaction costs are here constant) and then becomes convex due to the illiquidity premium (unit prices increases due to the shortage of supply). The transaction cost function is not easy to determine, but it appears to be discontinuous and it can be expressed as follows: $C = (1 + v)[f + \phi\big((b + p)s\big)] + ms$, where v is the VAT rate, f are fixed costs, b is the brokerage rate, p the illiquidity premium, s represents the transaction size, m the marketable securities tax rate and ϕ represents a subjective arbitrary function often difficult to interpolate and to define. Illiquidity premium plays an important role in this scheme and it can be introduced in different ways, but herein we consider it as an increment of the brokerage rate.

We must consider that, even if Modern Portfolio Theory states that diversified portfolio are preferable to undiversified ones, there is evidence that investors choose undiversified portfolios. This is due to the action of transaction costs, since they were not included in the original model. Considering all typologies, transaction costs tend to reduce portfolio-diversification: This is partially due

[1] For sake of readability the resulting vector will be composed of weight values and the label of the corresponding asset.

to the introduction of fixed costs, while proportional ones do not have effects because they generate only a decrease in returns rate. It is clear however that only proportional costs are suitable to be included in the continuous model, as the remainder is sensitive to the invested amount.

Solution methods and discretization When applying solution methods (e.g. heuristics and metaheuristic strategies) to the PSP, the implementation has to be studied with particular care: Some metaheuristics for instance are designed to work in a discrete search space, while in the problem we are considering variables can assume any continuous value belonging to [0, 1] range. The basic idea for adapting the model to these techniques would be to discretize the space; this operation might not be conceptually sound since we only consider that assets weights must sum up to one, regardless of the total amount to be invested. We might decide to apply a discretization at 0.000005 intervals, without any trouble for the formulation, but it is clear that if we have to invest 1,000,000,000 euros the discretized minimum admissible lot will be 5,000 euros, while if we consider 10 euros to be invested it will amount to 0.00005 euros. This will not turn into errors or warnings, but it is clear that the meaning of the efficient frontier could be strongly misleading depending on the invested amount.

These issues lead us to face the dilemma of the integer formulation (in which assets weights are labeled by their money-value) versus the standard continuous one: After the latter was introduced, no extension was developed in order to include transaction costs and to manage ambiguity arising from discretization and total-investment. These issues seem to suggest the use of integer formulation, easily obtained by replacing equation (2) with the following:

$$\sum_{i=1}^{n} r_i x_i \geq r_p C \quad \sum_{i=1}^{n} x_i = C \quad x_i \geq 0, integer \quad i, j = 1 \ldots n \tag{5}$$

where C is the invested amount. The local search approach is, in our opinion, robust w.r.t the formulation, so it is able to handle the integer version too, ensuring important advantages: Lack of necessity of discretisation, correctness of meanings of formulation, easiness in including transaction costs and rounds. The diffusion of software tools such as *Comet*[4] that enable to implement metaheuristics while preserving a CP modelling approach help us include the issues discussed so far in the analysis and development of metaheuristic for the PSP. These packages enable us to define the model (so including various constraints in the formulation), and, separately, the search strategy, so that changing the first does not trigger bad or misleading behavior in the latter. In the PSP, this is of the most importance, as constraints *must* be added to the formulation in order to obtain satisfactory results. In real-world applications these constraints can be classified in two main types:

- Hard Constraints imposed in order to make the model the most realistic as possible (these constraints must be satisfied for each category of investor in each area we are taking into account);

– Soft Constraints, imposed in order to describe preferences and behaviors of investors to whom the analysis is directed (each of this constraint must be defined to describe a specific category or area).

The second class of constraints has often been under-considered. In our opinion, efforts in introducing soft constraints can help develop solutions for several class of investors, geographic areas, regulations and so on: Metaheuristics have already been tested on this problem showing satisfactory results on the basic formulation, and in our opinion the next advance has to be made on modelling. This can be achieved by embedding in the current formulation aspects already discussed in the economic and financial literature about portfolio selection, but only a few of them has been investigated empirically. For example, no comparison amongst the different risk measures has been made on the same instance.

4 Conclusions and future works

The portfolio selection problem has been proven to be suitable for a metaheuristic approach in which the formulation is enriched by constraints used both to define the problem formulation and explain investor behavior. The versatility of these strategies enables us to add and change various aspects of the formulation without affecting the search-process. Further research is aimed at formulating an integer model in which constraints can be easily defined and included, and to use it to obtain real-world oriented results. A comparison between integer and continuous formulation will be performed in order to show differences between the resulting portfolios (if any);furthermore a comparison of different risk measures and a formulation embedding illiquidity-premium-transaction cost will be studied.

References

1. C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, September 2003.
2. T.J. Chang, N. Meade, J.E. Beasley, and Y.M. Sharaiha. Heuristics for cardinality constrained portfolio optimisation. *Comput. Oper. Res.*, 27(13):1271–1302, 2000.
3. A. Wijayanayake H. Konno. Mean-absolute deviation portfolio optimization model under transaction costs. *Journal of the Operational Research Society of Japan*, 42(4), 1999.
4. P. Van Hentenryck and L. Michel. *Constraint-Based Local Search.* The MIT Press, 2005.
5. H. Kellerer, R. Mansini, and M.G. Speranza. Selecting portfolios with fixed costs and minimum transaction lots. *Annals of Operational Research*, 99(4), 2000.
6. H. Markowitz. Portfolio selection. *Journal of Finance*, 7(1):77–91, 1952.
7. A. Schaerf. Local search techniques for constrained portfolio selectionproblems. *Comput. Econ.*, 20(3):177–190, 2002.
8. F. Streichert, H. Ulmer, and A. Zell. Evaluating a hybrid encoding and three crossover operators on the constrained portfolio selection problem. In *Proceedings of Congress on Evolutionary Computation (CEC 2004)*, 2004.

The Art and Virtue of Symbolic Constraint Propagation

Student: Björn Hägglund[1], Supervisor: Anders Haraldsson[1]

[1] Linköping University, 581 83 Linköping, Sweden
{bjoha, andha}@ida.liu.se

Abstract. Constraint stores supporting new kinds of stored constraints can potentially increase the power of search and propagation solvers by several orders of magnitude, but may on the other hand destroy the ease with which propagators interact with each other. We define the problem and report about work in progress. A solution to the problem has been applied to our constraint solving environment, Angelica, used in the examples throughout the paper.

1 The Problem

This paper primarily concerns *S&P solvers*, that is, solvers based on search and propagation with computation spaces (called *universes* in this paper). See [8] in conjunction with [3] and/or [7]. The strength of such solvers lies in their extensibility and the wide range of problems they can deal with. Propagators using all sorts of propagation algorithms for propagating all sorts of constraints belonging to all sorts of problem domains can interoperate concurrently with each other, almost without knowing anything about what the others are doing. This makes it easy to add new propagation algorithms to the system. However, this strength is also a source of weakness. All communication between propagators goes through a store, and the kind of constraints allowed in that store is typically restricted to an extent ruling out more powerful propagation techniques. How can we redesign stores to remedy this? More precisely, we are interested in enabling more symbolic propagation. This section explains what that means, why it deserves further investigation, and why it is a non-trivial problem.

1.1 What is Symbolic Propagation?

Symbolic propagation is the activity of constraining a variable to a partial type or binding it to a partial term. A term is *partial* iff it contains unbound variables[1]. To make this more precise, we need to put it in some context. We assume there is a sound term rewriting system used throughout the solver to represent and reduce stored

[1] Note that we consider local names introduced by "λ", "∀" and other quantifiers to be parameters rather than variables.

constraints. [5] is a good introduction to such systems. We further assume that every stored constraint is either a binding or an ascription.

A *binding* is an equation whose lhs is a single variable. If the binding is stored (in the store of some universe), we say that the variable is *bound to* the rhs of the binding (*in* that particular universe). Variables that occurs in the lhs of some stored binding are said to be *bound*. Remaining variables are *unbound*. Examples from Angelica:

```
x = 2a + 7b  - 12c
y = 1, 2 » s » 4               % "»" is the sequence concatenation operator.
z = Bool
```

Assuming that a, b and c are constrained to numbers and s to sequences, storing these bindings would bind x to an arithmetic expression, y to a sequence, and z to a type.

An *ascription* is a type constraint whose lhs is a single variable and whose rhs is a type. If the ascription is stored, we say that the variable is *constrained to* the type in the rhs. Variables that occurs in the lhs of some stored ascription are said to be *constrained*. Remaining variables are *unconstrained*. Examples from Angelica:

```
x : Type
a : 5..9
f : 0..a -> 6..()
```

Storing these ascriptions will constrain x to be a type, a to be an integer between 5 and 9 (inclusive), and f to be a lambda term that when applied to an integer between 0 and a returns an integer greater than or equal to 6.

Restricting the lhs of each stored constraint to be a single variable enables the store to use some sort of dictionary where variables can be looked up in constant time. Note that we want the contract of a binding to be strong enough to allow propagators and store reducers to substitute a bound variable with the term it is bound to and to do so in any context and without further considerations. This need not be true for an unbound variable, even when constrained to be equal to some term.

1.2 Why It Deserves Further Investigation

Symbolic propagation has the potential to increase the power of S&P solvers by several orders of magnitude. To see this, let us look at an example. Suppose we want to solve the following problem from the Angelica constraint specification language:

```
x,y,z :: 0..()!
30z - 46y - 16x = 4712!
44x + 73y - 52z = 1936!
abs(x - y) ≤ 100!
```

0..() is the domain of all natural numbers. The exclamation marks indicates that the Boolean terms should be asserted rather than just evaluated. Remaining constructs have their usual mathematical meaning. Since Angelica does not come with abs, we need to define it:

```
abs = λx:Int. if x < 0 then -x else x!
```

Submitting this problem results in the following behavior: Each of the five submitted constraints is propagated by its own propagator. Nevertheless, the two only possible solutions are found immediately, with a minimal amount of search: one universe where $x < y$ and consequently x, y, z = 1093, 1140, 2488 and one universe where $y \le x$ and consequently x, y, z = 992, 896, 2060. If we replace $30z - 46y - 16x$ with $29z - 46y - 13x$, the problem has no solutions. Angelica recognizes this immediately, without search. If the difference constraint is omitted the problem has an infinite number of solutions. All of them are found within a single universe. We can see what they are by submitting the query x, y, z, which is reduced to -18 + 101[2], -1544 + 244[2], -2220 + 428[2], where [2] is a generated variable. By switching to visible variable annotations we see that [2] is constrained to 7..() or in other words, to an integer greater than or equal to 7.

Now compare this behavior with the behavior of an ordinary S&P solver, represented here by Oz (downloadable from [9]):

```
proc {Problem Solution}
   X Y Z
in
   Solution = X#Y#Z                          [X Y Z] ::: 0#FD.sup
   {FD.sumC [30 ~46 ~16] [Z Y X] '=:' 4712}  {FD.sumC [44 73 ~52] [X Y Z] '=:' 1936}
   {FD.distance X Y '=<:' 100}               FD.distribute ff [X Y Z]}
end
```

Sending the above problem to SearchAll leads to the following observations:

1. The most natural way to describe the problem does not work at all, because the call to SearchAll suspends unless all involved variables are constrained to finite domains. So we have to constrain the variables to the largest domain available and hope that the solutions are in this domain.
2. After 10 minutes, we were still waiting for the solutions. In fact, waiting a year would probably not be enough. Replacing FD.sup (albeit already a low number) with lower numbers significantly reduces search time, because the latter seems to grow almost quadraticly with the former. Using the number 20,000 yields a search time around 10 seconds on the laptop used to write this paper. But again, how can we know that that all solutions are found? Requiring the user to resort to this kind of manual guessing is not good.
3. Possibly, there are all sorts of tricks an experienced constraint programmer could use to reduce the search time, but this is beside the point. The user should only have to specify the problem in the most natural way. The rest should be automatic.
4. If we replace $30z - 46y - 16x$ with $29z - 46y - 13x$, Oz goes ahead searching for solutions anyway, despite the fact there are no solutions. This is discovered only after a search needing about as much time as if there were solutions.
5. Removing the call to FD.distance, we see that Oz naively enumerates all solutions. This is an unattractive way to present a large solution space.

Oz provides brilliant abstractions to program with concurrency. This is why we currently use it to implement Angelica. But despite being implemented on top of Oz, Angelica outperforms Oz (at least in some respects) when it comes to solving, and the main reason for this is symbolic propagation. The point with the above example is not to demonstrate that Angelica is good at solving linear Diophantine equations. The way to solve such equations can easily be figured out by anyone acquainted with elementary number theory (e.g. after having read [6]), and has also been studied extensively in the constraint community (see e.g. [1] and [2]). The point is that propagators using such algorithms are possible in Angelica, whereas they do not fit into solvers not allowing symbolic propagation. Angelica propagates linear Diophantine equations by binding variables to additions of unbound integer variables multiplied by integer constants. This is not allowed in Oz. The only symbolic propagation allowed in Oz is binding variables to partial records or other variables, but apart from that, no relationships between variables can be expressed directly in the stores. As a consequence, the way propagators can interact with each other is very limited. This cripples propagation and leaves most of the work to extensional search, whose complexity is hopelessly exponential. This limitation is not in any way specific to Diophantine equations. It affects virtually all problems involving large variable domains and constraints for which there are clever intensional (algebraic) propagation algorithms. Where only small domains are involved, symbolic propagation is probably pointless, but if symbolic propagation can be enabled without excluding any possibilities to do ordinary propagation, propagators would be free to use it without being required to do so.

1.3 Why It Is Not Trivial

At this point, it might seem tempting to allow all sorts of stored constraints, thereby, abracadabra, making all the symbolic propagation of our dreams possible. However, this is a bad idea as it would violate one or more of the following store requirements (known as *critical store requirements*):

1. The store of any solved universe must be constructively consistent.
2. Stores should never get jammed.
3. Designing well-behaved propagators should be reasonably easy and impose minimal restrictions on the choice of propagation algorithms.
4. The computational complexity of a critical transaction between a store and a propagator should be finite and not grow with the size of the store. The critical transactions are telling constraints, looking up variables, triggering suspended propagators, and firing triggers (notifying suspended propagators).
5. The computational complexity of keeping the store irreducible should be finite and grow reasonably with the size of the store. (A store S is *irreducible* iff no rhs of some constraint stored in S is reducible under S.)

Constructive Consistency. A store is *constructively consistent* iff it is irreducible and all and only valuations constructible from S are solutions to S. A *solution to* (*explicit model of*) a store S is a valuation satisfying all constraints stored in S. A *valuation* assigns exactly one value to every variable. A *value* is a closed term that cannot be further reduced because it is not supposed to be further reduced. A valuation is *constructible from* a store S iff it can be constructed as follows:

1. Assign some arbitrary value to every variable not occurring in the lhs of some constraint stored in S.
2. Let C be a constraint stored in S with a lhs not yet assigned and a rhs containing only assigned variables. If there is no such constraint, stop. Otherwise, repeat this step after having done the following: Let C2 be result of replacing all variables in

C with the values assigned to them. Reduce C2 until its rhs is a value. If C2 is a binding, assign the value in the rhs to the variable in the lhs. If C2 is an ascription, assign a value having the type in the rhs to the variable in the lhs.

Constructive consistency is necessary not only to ensure that found solutions can be trusted, but also to give the user a reasonable chance to understand the solution space defined by the store of a solved universe.

Avoiding Jammed Stores. A store is *jammed* when and only when there is a propagator wanting to store a constraint that would have been accepted if the store were empty, but is now impossible to make the store entail. It is typically the restrictions introduced to ensure store consistency that open the possibility to jam stores. Jamming does not affect solver soundness, but may severely cripple further propagation to an extent outweighing the advantages of using symbolic propagation.

Designing Well-behaved Propagators. Solver soundness relies on all propagators being sound. A propagator is *well-behaved* only if it is sound. But we also need to ensure universe stabilization: if all propagators of a universe are *well-behaved* and the store never notifies propagators unless new constraints (not previously entailed by the store) has been stored, then the universe should be guaranteed to eventually suspend, fail or succeed. It is important that the complexity of designing well-behaved propagators is minimized, and in particular that the well-behaved-ness of a propagator does not depend on what other propagators exist in the same universe. Otherwise, the point of using the S&P architecture is lost.

1.4 Problem Summary

How can we enable more symbolic propagation without violating any critical store requirement?

2 Solution Status

We have developed effective store design rules, compliance with which ensures constructive consistency, completely avoids jammed stores, and makes meeting the complexity requirements pretty straightforward. We are building a system complying with these rules without disabling any ordinary propagation; a system allowing far more symbolic propagation than what is typically the case in S&P solvers. This work is completed to an extent leaving no doubt that such systems are possible. However, more experiments are needed to get a more complete picture of the solving power enabled by symbolic propagation, especially when it comes to problems using several profoundly different kinds of constraints.

Symbolic propagation inevitably makes it trickier to find effective criteria for what makes propagators well-behaved. So far, the increased complexity seems to be acceptable, but further investigations are needed.

Unfortunately, the 6 page limit made it impossible to include even an outline of the design rules and an example of how to comply with them. An outline will be given in the presentation, should the paper be accepted.

3 Related Work

Despite talking to experts in the field and browsing through online proceedings of PPDP, ILCP, CP, RTA, PADL back to the early ninetieth, we found nothing resembling a general treatise on symbolic propagation in S&P solvers. Maybe we did not look in the right places using the right keywords. Or maybe the silence is caused by a general assumption that the whole issue was settled back in the early days. It would be surprising if discussions about the nature of constraint stores, and in particular what kind of constraints should be storable, were not common in those days. But the exact contents of these discussions does not really matter, as their outcome was that the "wrong" path was taken: symbolic propagation was dismissed from mainstream constraint research, probably because it was considered to be too problematic and/or making solvers too complicated.

However, we did find [4], an interesting paper on a related subject: first class propagators. They provide a way to get around restricted stores. Although an ad hoc way to improve things, the paper presents impressive experimental results indicating the solving power enabled by symbolic propagation.

References

1. Farid Ajili, Evelyn Contejean: *Complete Solving of Linear Diophantine Equations and Inequations without Adding Variables*. In Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP'95) in Cassis, France, September 1995. LNCS vol. 976, Springer-Verlag, Berlin/Heidelberg (1995)
2. Farid Ajili, Hendrik C.R. Lock: *Integrating Constraint Propagation in Complete Solving of Linear Diophantine Systems*. In Proceedings of the Tenth International Symposium on Principles of Declarative Programming (PLILP'98) in Pisa, Italy, September 1998. LNCS vol. 1490, Springer-Verlag, Berlin/Heidelberg (1998)
3. Krzysztof R. Apt: *Principles of Constraint Programming*. ISBN 0-521-82583-0, Cambridge University Press, Cambridge (2003)
4. Tobias Müller: *Promoting Constraints to First Class Status*. In Proceedings of the First International Conference on Computational Logic (CL'00) in London, UK, July 2000. LNCS vol. 1861, Springer-Verlag, Berlin/Heidelberg (2000)
5. Benjamin C. Pierce: *Types and Programming Languages*. ISBN 0-262-16209-1, MIT Press, London (2002)
6. Kenneth H. Rosen: *Elementary Number Theory and its Applications*. ISBN 0-201-87073-8, 4th edition, Addison-Wesley (2000)
7. Peter Van Roy, Seif Haridi: *Concepts, Techniques, and Models of Computer Programming*. ISBN 0-262-22069-5, MIT Press, London (2004)
8. Christian Schulte: *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*. LNCS vol. 2302, Springer-Verlag, Berlin/Heidelberg (2002)
9. *http://www.mozart-oz.org* (2006-04-24). Home of Mozart/Oz.

The Modelling Language Zinc

Student: Reza Rafeh
Supervisors: Maria Gärcia de la Banda, Kim Marriott, and Mark Wallace

Clayton School of IT, Monash University, Australia
{reza.rafeh,mbanda,marriott,wallace}@mail.csse.monash.edu.au

Abstract. We describe the Zinc modelling language. Zinc provides set constraints, constrained and user defined types, and polymorphic predicates and functions. The last allow Zinc to be readily extended to different application domains by user-defined libraries. Zinc is designed to support a modelling methodology in which the same conceptual model can be automatically mapped into different design models, thus allowing modellers to easily "plug and play" with different solving techniques and so choose the most appropriate for that problem. In our prototype implementation the Zinc model can be mapped into one of two design models, both of which are implemented in ECLiPSe. The first design model uses local search while the second uses a complete tree search with propagation based finite domain and set solvers. A core feature of the Zinc implementation supporting such solver and technique-independent modelling is the use of an intermediate language called FZM. This is designed to be simple and low-level enough to be significantly closer to the decision model, but sufficiently high-level to be a suitable intermediate model for all solvers.

1 Introduction

Solving combinatorial problems is a remarkably difficult task which requires the problem to be precisely formulated and efficiently solved. Even formulating the problem precisely is surprisingly difficult and typically requires many cycles of formulation and solving, while efficient solving often requires development of tailored algorithms which exploit the structure of the problem. Such algorithms achieve scalability and performance by first isolating the core combinatorial problem, and then exploiting the algorithms that have been proved to work best for the (sub)problem at hand.

Reflecting this discussion, modern approaches to solving combinatorial problems divide the task into two (hopefully simpler) steps. The first step is to develop the *conceptual model* of the problem which specifies the problem without consideration as to how to actually solve it. The second step is to *solve* the problem by mapping the conceptual model into an executable program called the *design model*. Ideally, the same conceptual model can be transformed into different design models, thus allowing modellers to easily "plug and play" with different solving techniques [4, 3].

Here we introduce a new modelling language, Zinc, specifically designed to support this methodology. There has been a considerable body of research into problem modelling which has resulted in a progression of modelling languages including MOLGEN [7], AMPL [2], Localizer [6], OPL [8], ESRA [1]. We gladly

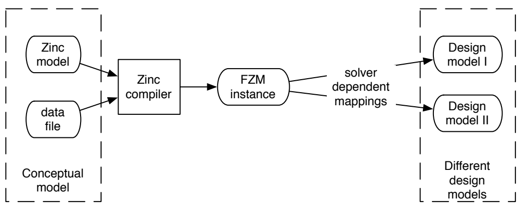


Fig. 1. Mapping a Zinc conceptual model to different decision models

acknowledge the strong influence these languages have had on our design. Our reasons to develop yet another modelling language are threefold.

First, we want the modelling language to be solver and technique independent, allowing the same conceptual model to be mapped to different solving techniques and solvers, i.e., be mapped to design models that use the most appropriate technique, be it local search, mathematical modelling, constraint programming, or a combination of the above. To date the implemented languages have been tied to specific underlying platforms. For example, AMPL was designed to interface to MIP packages such as Cplex and Xpress-MP, MOLGEN was interfaced to a propagation-based solver, and Localizer was designed to map down to a local search engine.

Second, we want to provide high-level modelling features but still ensure that the models are executable. Zinc offers structured types, sets, and user defined predicates and functions which allow a Zinc model to be encapsulated as a predicate. It also allows users to define “constrained objects” i.e., to associate constraints to a particular type thus specifying the common characteristics that a class of items are expected to have [5]. It supports polymorphism, overloading and type coercion which make the language comfortable and natural to use.

And third, we want Zinc to have a simple, concise core but allow it to be extended to different application areas. This is achieved by allowing Zinc users to define their own application specific library predicates, functions and types. This contrasts with, say, OPL which provides seemingly ad-hoc built-in types and predicates for resource allocation and cannot be extended to model new application areas without redefining OPL itself.

Of course there is considerable tension between these aims, since the higher-level the modelling language, the greater the gap between the conceptual model and the design model. Critical to the success of Zinc is the design of the intermediate modelling language, Flattened Zinc Model (FZM), which bridges this gap. FZM is designed to be simple and low-level enough to be significantly closer to the decision model, but sufficiently high-level to be a suitable intermediate model for all solvers. The translation process from the conceptual model consisting of a Zinc model and instance specific data (optionally given in separate datafiles), to FZM, to different technique specific design models is shown in Figure 1.

```
type PosInt = (int:x where x>0);
PosInt: sizeBase;
record Square=(var 1..sizeBase: x, y; PosInt: size);
list of  Square:squares;
constraint forall(s in squares)
              s.x + s.size =< sizeBase+1 /\
              s.y + s.size =< sizeBase+1;
predicate nonOverlap(Square: s,t) =
              s.x+s.size =< t.x  \/  t.x+t.size =< s.x \/
              s.y+s.size =< t.y  \/  t.y+t.size =< s.y;
constraint forall(i,j in 1..length(squares) where i<j)
              nonOverlap(squares[i], squares[j]);
predicate onRow(Square:s, int: r) =
              s.x =< r /\ r < s.x + s.size;
predicate onCol(Square:s, int: c) =
             s.y =< c /\ c < s.y + s.size;
assert sum(s in squares) (s.size^2) == sizeBase^2;
constraint forall(p in 1..sizeBase)
             sum(s in Squares) (s.size*holds(onRow(s,p))) == sizeBase /\
             sum(s in Squares) (s.size*holds(onCol(s,p))) == sizeBase;
output(squares);
```

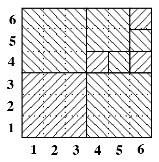


Fig. 2. Perfect Squares model

2 Zinc

Zinc is a first-order functional language with simple, declarative semantics. It provides: mathematical notation-like syntax; expressive constraints (finite domain and integer, set and linear arithmetic); separation of data from model; high-level data structures and data encapsulation including constrained types; user defined functions and constraints. Let us illustrate some of these features by means of an example.

Consider the model in Figure 2 for the perfect squares problem [9]. This consists of a base square of size `sizeBase` (6 in the figure) and a list of squares of various sizes `squares` (three of size 3, one of size 2 and five of size 1 in the figure). The aim is to place all squares into the base without overlapping each other.

The model defines a constrained type `PosInt` as a positive integer and declares the parameter `sizeBase` to be of this type. A record type `Square` is used to model each of the squares. It has three fields `x`, `y` and `size` where (x, y) is the (unknown) position of the lower left corner of the square and `size` is the size of its sides. The first constraint in the model ensures each square is inside the base (note that `\/` and `/\` denote disjunction and conjunction, respectively). The model contains three user-defined predicates: `nonOverlap` ensures two squares do not overlap, and `onRow` and `onCol` that a square is, respectively, on a particular row or column in the base.

The squares provided as input data are assumed to be such that they fit in the base exactly. To check this assumption, the model includes an assertion that equates their total areas.

The last constraint in the model is redundant since it is derived from the assumption that the squares exactly fill the base: the constraint simply enforces each row and column in the base to be completely full.

Data for the model can be given in a separate data file as, for example:

```
sizeBase=6;
squares = [ (x:_,y:_,size:s) | s in [3,3,3,2,1,1,1,1,1]];
```

3 Implementation

We have finished the first prototype of Zinc which implements the full syntax of it. It is written in Mercury with a Yacc generated parser and flex generated lexical analyser. It is about twelve thousand lines of Mercury code, and five thousand lines of C.

After doing syntax and semantic checking, Zinc adds the information in the associated data file to the compiled model and generates the Flattened Zinc Model (FZM) instance. The FZM language is an intermediate representation that is closer to the design model while is still solver independent. The FZM generation process eliminates features of the Zinc model that serve only to make it user friendly and includes: evaluation of all ground expressions, flattening high-level structures such as user-defined functions and iteration, simplifying complex data types and constraints and run-time checking such as assertion checking.

At the final stage, the FZM instance is mapped to the design model. Currently, we have developed two mappings to ECLiPSe. The first design model uses a complete tree search with propagation based finite domain and set solvers; and the second uses a form of local search, in which the local move automatically maintains hard constraints.

We wished to measure three factors about Zinc in our implementation: expressiveness, extensibility and performance.

In terms of expressiveness, we have performed a search of the literature and built models (see `www.csse.monash.edu.au/~rezar/Zinc/models`) for a variety of standard benchmark problems. As it can be gauged from these models, the expressiveness of Zinc results in very natural formalisations of the problems.

In terms of extensibility, perhaps the most clear example is the Zinc language itself and, in particular, its extensive use of library functions and predicates to implement Zinc's standard functions and predicates (instead of implementing them as built-ins).

In terms of performance, we compared our Zinc models with the equivalent ECLiPSe models using the same search strategy. The results are shown in the next two tables. All experiments were performed on a 3GHz Pentium 4 with 2Gb memory running Fedora.

Table 1 reports statistics on the mapping itself. We give the size of the original Zinc model; the size of the generated FZM; the size of the ECLiPSe program generated from the FZM; the size of the direct ECLiPSe program; and the total time taken to generate the ECLiPSe program from the Zinc model (this includes the time to generate the FZM). All sizes are given in “tokens” to abstract away from choice of identifier names etc.

Table 1. Zinc Mapping Statistics

Problem Name and parameters	Model Size (tokens) Zinc	Model Size (tokens) ECLiPSe	Mapping Time (sec)	Model Size (tokens) FZM	Model Size (tokens) Generated ECLiPSe
Golfers: arrays of int.sets 4 players, 2 gps, 3 wks	273 (5 cons)	1111	0.0540	678	2212 (65 cons)
Golfers: sets of sets 4 players, 2 gps, 3 wks	269 (5 cons)		0.323	10221	17233 (569 cons)
2X2 Job-Shop:	514 (3 cons)	1021	0.1560	3106	5122 (158 cons)
Knapsack: 30 Obj, 75% fit	326 (1 cons)	564	0.1280	964	1324 (31 cons)
Knapsack: 30 Obj, 50% fit	326 (1 cons)	564	0.1270	964	1324 (31 cons)
Stable-Marriage: 8 pairs	527 (4 cons)	955	0.4360	38624	40697 (822 cons)
8-Queens	88 (1 cons)	308	0.0650	1956	3221 (86 cons)
18-Queens	88 (1 cons)	308	0.5650	10251	16541 (461 cons)
28-Queens	88 (1 cons)	308	3.6140	25046	40261 (1136 cons)
Open-stacks: 7Cust, 5Prod	230 (2 cons)	736	0.0750	5199	6301 (109 cons)
Open-stacks: 9Cust, 7Prod	264 (2 cons)	723	0.1160	12681	13453 (193 cons)
Perfect-squares 5X5, 8 sq	300 (3 cons)	630	0.1920	6798	10638 (303 cons)

We do not give a model written directly in ECLiPSe for non-flat Social Golfers problem since this is quite unnatural to write in ECLiPSe.

We see that the Zinc model is consistently substantially smaller than the model written directly in ECLiPSe. The FZM and generated ECLiPSe code is orders of magnitude larger than the Zinc model and the direct ECLiPSe model. This is to be expected and reflects that the high-level iteration constraints have been flattened. The time to generate the ECLiPSe design model from the Zinc model is small, no more than a few seconds.

Table 2 shows the execution time of the ECLiPSe code generated from the Zinc model with the model written directly in ECLiPSe.

As we hoped we found that there was not a significant difference in execution time between the design model written directly in ECLiPSe and that generated from the Zinc model. This holds true for both the design model using complete tree search with propagation based solvers, and the one using local search.

4 Conclusion

We have presented a new modelling language Zinc designed to allow natural, high-level specification of a conceptual model. Unlike most other modelling languages, Zinc provides set constraints, constrained types and user defined types, and polymorphic predicates and functions. Unlike virtually all other modelling languages a Zinc model can be mapped into design models that utilize different solving techniques such as local search or tree-search with propagation based solvers. This comes from an appreciation of using an intermediate language called FZM which is a subset of Zinc and is significantly closer to the design model while is still solver independent.

We have compared a number of standard benchmarks written in Zinc and written in ECLiPSe. The Zinc models are considerably more concise and arguably more

Table 2. Comparing the direct and mapped programs

Problem Name	Tree search model		Local search model	
(cpu secs)	Direct	Generated	Direct	Generated
Golfers: arrays of int.sets 4 players, 2 gps, 3 wks	0.005	0.006	0.002	0.003
Golfers: sets of sets 4 players, 2 gps, 3 wks	-	0.008	-	0.003
2X2 Job-Shop:	0.00	0.00	0.002	0.003
Knapsack: 30 Obj, 75% fit	0.498	0.5	0.496	0.493
Knapsack: 30 Obj, 50% fit	3.465	3.449	0.734	0.728
Stable-Marriage: 8 pairs	0.131	0.075	-	-
8-Queens: all solutions	0.019	0.02	0.058	0.072
18-Queens: first solution	0.707	0.721	1.91	2.13
28-Queens: first solution	67.817	66.263	10.9	12.1
Open-stacks: 7Cust, 5Prod	0.002	0.001	0.273	0.157
Open-stacks: 9Cust, 7Prod	0.588	0.273	1.125	0.279
Perfect-squares 5X5, 8 sq	0.915	0.856	-	-

high-level and easier to understand. The ECLiPSe model automatically generated from Zinc (via FZM) has similar performance to the program written in ECLiPSe, assuming the same search method is used.

Currently our implementation uses a naive search procedure, but user-controlled search is vital for scalable performance on real problems. It seems sensible to allow the search component to be written in a Zinc-like language annotating the Zinc model. We are currently exploring this.

References

1. Pierre Flener, Justin Pearson, and Magnus Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *LOPSTR*, pages 214–232, 2003.
2. R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming.* Duxbury Press, 2002.
3. A.M. Frisch, C. Jefferson, B. Martinez-Hernandez, and I. Miguel. The rules of constraint modelling. In *Proc 19th IJCAI*, pages 109–116, 2005.
4. C. Gervet. *Large scale combinatorial optimization: A methodological viewpoint*, volume 57 of *Discrete Mathematics and Theoretical Computer Science*, page 151ff. DIMACS, 2001.
5. Bharat Jayaraman and Pallavi Tambay. Modeling engineering structures with constrained objects. In *PADL*, pages 28–46, 2002.
6. L. Michel and P. Van Hentenryck. Localizer: A modeling language for local search. In *Proc. Principles and Practice of Constraint Programming - CP97*, pages 237–251, 1997.
7. M. Stefik. Planning with constraints (molgen: Part 1). *Artificial Intelligence*, 16:111–139, 1981.
8. P. Van Hentenryck, I. Lustig, L.A. Michel, and J.-F. Puget. *The OPL Optimization Programming Language.* MIT Press, 1999.
9. E. W. Weisstein. Perfect square dissection. From MathWorld –A Wolfram Web Resource, http://mathworld.wolfram.com/PerfectSquareDissection.html, 1999.

Interleaved Search in DCOP for Complex Agents

Student: David A. Burke **
Supervisor: Kenneth N. Brown

Centre for Telecommunications Value-Chain Research
and Cork Constraint Computation Centre
Dept. of Computer Science, University College Cork, Ireland

1 Introduction

Many combinatorial problems are naturally distributed over a set of agents: e.g. coordinating vehicle schedules in a transport logistics problem [1], or scheduling meetings among a number of participants [2]. Distributed Constraint Reasoning (DCR) considers algorithms explicitly designed to handle such problems, searching for globally acceptable solutions while minimising communication between agents. Most of the algorithms, however, assume that each agent controls only a single variable. This assumption is justified by two standard reformulations [3], by which any DCR problem with complex local problems (i.e. multiple variables in each agent) can be transformed to give exactly one variable per agent: (i) *compilation*: for each agent, define a variable whose domain is the set of solutions to the original local problem; (ii) *decomposition*: for each variable in each local problem, create a unique agent to manage it. Other algorithms for handling multiple local variables in distributed constraint satisfaction (DisCSP) have been proposed [4][5][6]. These algorithms are specific to DisCSP, since they reason about violated constraints, and cannot be applied directly to distributed constraint optimisation problems (DCOP), which are concerned with costs. For DCOP, the two original reformulations remain the standard way of handling complex local problems [7].

Our research is focused on compilation, as this allows each agent to use state-of-the-art centralised solvers to model and solve their local problem. We have previously developed a new compilation method based on dominance and interchangeability, which increases the range of problems able to be solved efficiently using compilation [8]. More recent results show that our compilation method is more efficient than decomposition for agents with large and complex internal problems [9]. However, one drawback of compilation is that it requires local solutions to be found before the distributed search can begin. This can be expensive for very complex local problems and can result in wasteful search of areas that do not belong in the global solution.

We now propose an alternative compilation approach that does not require local solutions to be pre-determined. Our algorithm, *Interleaved-ADOPT* (I-ADOPT), consists of three key ideas: (i) during compilation, instead of finding

** This work is supported by Science Foundation Ireland under Grant No. 03/CE3/I405

optimal local solutions, we calculate lower and upper bounds of solution costs; (ii) we modify the ADOPT [7] algorithm to incorporate these solution bounds into its cost calculations; (iii) we interleave local and distributed search processes, using global cost information to direct the agent's local search. In this paper, we summarise our compilation method, and then describe I-ADOPT. We prove that the modifications to ADOPT retain correctness and completeness. This forms the basis for an empirical evaluation of I-ADOPT, which is now underway.

2 DCOP and Compilation

A Distributed Constraint Optimisation Problem consists of a set of *agents*, $A = \{a_1, a_2, ..., a_n\}$, and for each agent a_i, a set $X_i = \{x_{i1}, x_{i2}, \ldots, x_{im_i}\}$ of *variables* it controls, such that $\forall i \neq j \; X_i \cap X_j = \phi$. Each variable x_{ij} has a corresponding domain D_{ij}. $X = \bigcup X_i$ is the set of all variables in the problem. $C = \{c_1, c_2, \ldots, c_t\}$ is a set of *constraints*. Each c_k has a *scope* $s(c_k) \subseteq X$, and is a function $c_k : \prod_{ij:x_{ij} \in s(c_k)} D_{ij} \rightarrow I\!N$. The *agent scope*, $a(c_k)$, of c_k is the set of agents that c_k acts upon: $a(c_k) = \{a_i : X_i \cap s(c_k) \neq \phi\}$. An agent a_i is a *neighbour* of an agent a_j if $\exists c_k : a_i, a_j \in a(c_k)$. A *global assignment*, g, is the selection of one value for each variable in the problem: $g \in \prod_{ij} D_{ij}$. A *local assignment*, l_i, to an agent a_i, is an element of $\prod_j D_{ij}$. Let t be any assignment, and let Y be a set of variables, then $t_{\downarrow Y}$ is the projection of t over the variables in Y. The global objective function, F, assigns a cost to each global assignment: $F : \prod_{ij} D_{ij} \rightarrow I\!N :: g \mapsto \sum_k c_k(g_{\downarrow s(c_k)})$. An optimal solution is one which minimises F. The solution process, however, is restricted: each agent is responsible for the assignment of its own variables, and thus agents must communicate with each other, describing assignments and costs, in order to find a globally optimal solution.

Several algorithms for DCOP have been proposed, including ADOPT [7] – a complete algorithm that allows agents to work asynchronously. Agents are prioritised into a tree. Let H_i be the set of higher priority neighbours of a_i, and let L_i be the set of its children. During search, each agent a_i repeatedly and asynchronously performs a number of tasks:

1. Incoming values are received from higher priority agents and added to the current context CC_i, which is a record of higher priority neighbours' current variable assignments: $CC_i \in \prod_{j:a_j \in H_i} D_j$.
2. Costs are received from children and stored if they are valid for the current context – for each subtree, rooted by an agent $a_j \in L_i$, a_i maintains a lower bound, $lb(l_i, a_j)$, and an upper bound $ub(l_i, a_j)$ for each of its values l_i.
3. Costs are calculated for each of its possible values. Let C_{ij} be the constraint between z_i and z_j. The partial cost, $\delta(l)$, for an assignment of a particular value l_i to z_i is the sum of the agent's local cost $f_i(l_i)$, plus the costs of constraints between a_i and higher priority neighbours: $\delta(l_i) = f_i(l_i) + \sum_{j:a_j \in H_i} C_{ij}(l_i, CC_{i\downarrow z_j})$. The lower bound, $LB(l_i)$, for an assignment of a value l_i to z_i is the sum of $\delta(l_i)$ and the currently known lower bounds for all subtrees: $LB(l_i) = \delta(l_i) + \sum_{j:a_j \in L_i} lb(l_i, a_j)$. The upper bound, $UB(l_i)$,

is the sum of $\delta(l_i)$ and the currently known upper bounds for all subtrees: $UB(l_i) = \delta(l_i) + \sum_{j:a_j \in L_i} ub(l_i, a_j)$. The minimum lower bound over all value possibilities, LB_i, is the lower bound for the agent a_i: $LB_i = min_{l_i \in D_i} LB(l_i)$. Similarly, UB_i, is the upper bound for the agent a_i: $UB_i = min_{l_i \in D_i} UB(l_i)$.

4. The value that minimises the lower bound on the costs is chosen and sent to all neighbours in L_i.
5. LB_i and UB_i are passed as costs to the parent of a_i, along with the context to which they apply, CC_i.

As the search progresses, the bounds are tightened in each agent until the lower bound of the minimum cost solution is equal to its upper bound. If an agent detects this condition, and its parent has terminated, then an optimal solution is found and it may terminate also.

To handle multiple variables in each agent, we use the compilation approach we proposed in [8]. Let $T_i = \prod_j D_{ij}$ be the set of possible local assignments to the agent's internal problem. For each agent a_i, let $p_i = \{x_{ij} : \forall c \; x_{ij} \in s(c) \rightarrow s(c) \subseteq X_i\}$ be its *private* variables – the subset of its variables which are not directly constrained by other agents' variables – and let $e_i = X_i \backslash p_i$ be its *external* variables – the variables that do have direct constraints with other agents. To apply the *compilation* method: (i) for each agent a_i, create a new variable z_i with domain $D_i = \prod_{j:x_{ij} \in e_i} D_{ij}$; (ii) for each agent a_i, add a unary constraint function f_i, where $\forall l \in D_i$, $f_i(l) = \min\{f_i(t) : t \in T_i, t_{\downarrow e_i} = l\}$. That is, D_i contains all assignments to the external variables, and their cost is the minimum cost obtained when they are extended to a full local assignment for a_i[1]. (iii) for each set of agents $A_j = \{a_{j1}, a_{j2}, ..., a_{jp_j}\}$, let $R_j = \{c : a(c) = A_j\}$ be the set of constraints whose agent scope is A_j, and for each $R_j \neq \phi$, define a new constraint $C_j : D_{j1} \times D_{j2} \times \ldots \times D_{jp_j} \rightarrow I\!N :: l \mapsto \sum_{c \in R_j} c(l_{\downarrow s(c)})$), equal to the sum of the constraints in R_j (i.e. construct constraints between the agents' new variables, that are defined by referring back to the original variables in the problem). Once compiled, we can run single-variable DCOP algorithms using the new variables. We have shown that our approach gives orders of magnitude improvement over the basic compilation method for many parameter settings.

3 I-ADOPT: Interleaved Search in ADOPT

3.1 Compilation with Bound Estimates

Compilation requires local solutions to be pre-determined and the distributed search for a global solution cannot proceed until this is completed. To overcome this, we modify the compilation technique described in Section 2. For each a_i, create a new variable z_i with domain $D_i = \prod_{j:x_{ij} \in e_i} D_{ij}$. Then proceed to add a unary constraint function f_i, where $\forall l \in D_i$, $f'_i(l) = \min\{f_i(t) : t \in T_i, t_{\downarrow e_i} =$

[1] This differs from the basic compilation method in that we find only one optimal solution for each combination of assignments to external variables (instead of finding all local solutions), eliminating solutions that are either dominated or interchangeable.

$l\}$. However, during compilation we do not evaluate this function completely. Instead, we calculate a lower $lb(f_i(l))$, and upper $ub(f_i(l))$, bound $\forall l \in D_i$. To calculate these bounds, any suitable technique may be chosen (for example [10]).

3.2 Using Local Bounds in ADOPT

In I-ADOPT, definitive local costs for agents will not be known from the start. Therefore, we modify the ADOPT cost calculations such that $\delta(l)$ does not include the local cost. Instead, the lower and upper bounds for the local cost are included in the lower and upper bound calculations. The partial cost, $\delta'(l)$, for an assignment of a particular value l_i to z_i is the sum of the costs of constraints between a_i and higher priority neighbours (excluding the agent's local cost): $\delta'(l_i) = \sum_{j:a_j \in H_i} C_{ij}(l_i, CC_{i\downarrow z_j})$. The lower bound, $LB'(l_i)$, for an assignment of a value l_i to z_i is the sum of $\delta'(l_i)$ and the currently known lower bounds for all subtrees plus the lower bound on the local cost of l_i: $LB'(l_i) = \delta'(l_i) + \sum_{j:a_j \in L_i} lb(l_i, a_j) + lb(f_i(l_i))$. The upper bound, $UB'(l_i)$, is the sum of $\delta'(l_i)$ and the currently known upper bounds for all subtrees plus the upper bound on the local cost of l_i: $UB'(l_i) = \delta'(l_i) + \sum_{j:a_j \in L_i} ub(l_i, a_j) + ub(f_i(l_i))$.

3.3 Interleaving Local and Global Search

We coordinate the local and global search processes, tightening the bound on local solutions as we tighten the bounds in the global search. The agents local search process (Algorithm 1) runs on its own thread, thus interleaving with the global search process and making use of idle time that exists in the agent. Each loop of the algorithm attempts to tighten both the lower and upper bounds for its target value (lines 5, 10). It then checks for a new target value (3) and repeats. If the target value is changed, the algorithm will simply attempt to tighten the bounds for the new target value. A target value is solved when the lower bound

Algorithm 1: *local.start*

```
(1)   $stopped \leftarrow false$
(2)   while $!stopped$ and $!terminated$
(3)       $l_i \leftarrow getTarget()$
(4)       if $lb(f_i(l_l)) < ub(f_i(l_i))$
(5)           $lb(f_i(l_l)) \leftarrow lbSearch(l_i)$
(6)           $agent.notifyBoundUpdated()$
(7)       else
(8)           $stopped \leftarrow true$
(9)       if $lb(f_i(l_l)) < ub(f_i(l_i))$
(10)          $ub(f_i(l_i)) \leftarrow ubSearch(l_i)$
(11)          $agent.notifyBoundUpdated()$
(12)      else
(13)          $stopped \leftarrow true$
```

Algorithm 2: Modifications to ADOPT's *backtrack* procedure

```
(1)    ...
(2)    choose $l_i$
(3)    $local.setTarget(l_i)$
(4)    if $lb(f_i(l_l)) < ub(f_i(l_i))$ and $local.isStopped()$
(5)        $local.start()$
(6)    ...
(7)    if $terminateCondition()$
(8)        $local.terminate()$
```

equals the upper bound (4, 9). The methods for tightening the bounds, like the initial bound calculations, can be chosen independently of this algorithm.

To control execution of Algorithm 1, we modify the *backtrack* procedure in ADOPT. The backtrack procedure is activated when: (i) a new value is received from a parent; or (ii) if the costs of children have changed. We now add a third activation condition: (iii) if a bound in a local cost is updated (lines 6, 12 of Alg. 1). Two modifications are made to the backtrack algorithm (Alg. 2). First, the agent sets the target value of the local search to be its current value (3). If an optimal solution to the local problem for this value has not been found previously and the local search is currently not active, then the agent starts the local search (5). Thus, progress in the global search is directing the local search towards the most promising areas of the search space. Local values that do not have a possibility of being in the global solution will be ignored, reducing the local search required by each agent. Second, if the agent detects its termination condition (7), then it should notify the local search process to also terminate.

4 Algorithm correctness

We now prove that our algorithm, like ADOPT, is both correct and complete. Theorem 1 states that for an agent a_i, the lower bound calculated is guaranteed to be no greater than the lower bound calculated by the same agent in the standard ADOPT, and also that the upper bound calculated is no less than the upper bound calculated by the same agent in the standard ADOPT.

Theorem 1. $\forall l_i \in D_i, LB'(l_i) <= LB(l_i)$ *and* $UB'(l_i) >= UB(l_i)$

Proof Expanding LB_i gives: $f_i(l_i)+\sum_{j:a_j \in H_i} C_{ij}(l_i, CC_{i\downarrow z_j}) + \sum_{j:a_j \in L_i} lb(l_i, a_j)$. Expanding LB'_i gives: $\sum_{j:a_j \in H_i} C_{ij}(l_i, CC_{i\downarrow z_j}) + \sum_{j:a_j \in L_i} lb(l_i, a_j) + lb(f_i(l_i))$. The only difference is that LB_i uses $f_i(l_i)$ while LB'_i uses $lb(f_i(l_i))$. Since $lb(f_i(l_i))$ is a lower bound of $f_i(l_i)$, it follows that $LB'_i <= LB_i$. Similarly, the only difference between UB_i and UB'_i is that UB_i uses $f_i(l_i)$ while UB'_i uses $ub(f_i(l_i))$. Since $ub(f_i(l_i))$ is an upper bound of $f_i(l_i)$, it follows that $UB'_i >= UB_i$. □

Theorem 2. *If complete methods are used to tighten the bounds of local solutions, then the algorithm is complete.*

Proof $LB'_i = \sum_{j:a_j \in H_i} C_{ij}(l_i, CC_{i\downarrow z_j}) + \sum_{j:a_j \in L_i} lb(l_i, a_j) + lb(f_i(l_i))$. Using complete methods for the lower and upper bound searches for the local problem is guaranteed to find optimal local costs for each target value l_i. In the worst case all values $l_i \in D_i$ will be chosen as target values until their optimal solutions is found, thus reducing $lb(f_i(l_i))$ to be $f_i(l_i)$, and so LB'_i becomes equivalent to LB_i. Similarly, UB'_i becomes equivalent to UB_i. Since ADOPT is proven to be complete, it follows that I-ADOPT is also complete. □

5 Conclusion and Future Work

We have presented a modified version of the DCOP algorithm ADOPT for handling problems where each agent has multiple variables. Our algorithm, is based on a standard approach which compiles agents' local problems down to a single variable whose domain is the set of all local solutions. However, our approach eliminates the need to pre-determine local solutions. We use lower and upper bound estimates on the costs of local solutions, which allows the global search to progress without complete knowledge of the local solution costs of an agent. In turn, the progress of the global search provides direction for what areas in the local search space to explore, thus eliminating the need to search for local solutions that can never be part of the global solution. In addition, the agents can make use of naturally occurring idle time to interleave the local search process with the distributed search. This is currently work in progress, and future work will focus on performing a thorough experimental evaluation of the algorithm.

References

1. Calisti, M., Neagu, N.: Constraint satisfaction techniques and software agents. In: Proc. Agents and Constraints Workshop, AIIA. (2004)
2. Wallace, R., Freuder, E.: Constraint-based reasoning and privacy/efficiency tradeoffs in multi-agent problem solving. Artificial. Intelligence. **161** (2005) 209–227
3. Yokoo, M., Hirayama, K.: Algorithms for distributed constraint satisfaction: A review. Autonomous Agents and Multi-Agent Systems **3** (2000) 185–207
4. Armstrong, A., Durfee, E.: Dynamic prioritization of complex agents in distributed constraint satisfaction problems. In: Proc. IJCAI. (1997) 620–625
5. Yokoo, M., Hirayama, K.: Distributed constraint satisfaction algorithm for complex local problems. In: Proc. ICMAS. (1998) 372
6. Maestre, A., Bessière, C.: Improving asynchronous backtracking for dealing with complex local problems. In: Proc. ECAI. (2004) 206–210
7. Modi, P., Shen, W., Tambe, M., Yokoo, M.: Adopt: Asynchronous distributed constraint optimization with quality guarantees. Artificial Intelligence **161** (2005) 149–180
8. Burke, D., Brown, K.: Efficient handling of complex local problems in distributed constraint optimization. In: Proc. ECAI. (2006) To appear.
9. Burke, D., Brown, K.: A comparison of approaches to handling complex local problems in dcop. (2006) Submitted to DCSP workshop at ECAI.
10. Cabon, B., de Givry, S., Verfaillie, G.: Anytime lower bounds for constraint violation minimization problems. In: Proc. CP. (1998) 117–131

A New Algorithm for Sampling CSP Solutions Uniformly at Random

Student: Vibhav Gogate
Supervisor: Rina Dechter

Donald Bren School of Information and Computer Science
University of California, Irvine, CA 92697
{vgogate,dechter}@ics.uci.edu

Abstract. The paper presents a method for generating solutions of a constraint satisfaction problem (CSP) uniformly at random. Our method relies on expressing the constraint network as a uniform probability distribution over its solutions and then sampling from the distribution using state-of-the-art probabilistic sampling schemes. To speed up the rate at which random solutions are generated, we augment our sampling schemes with pruning techniques used successfully in the CSP literature such as conflict-directed back-jumping and no-good learning.

1 Introduction

The paper presents a method for generating solutions to a constraint network uniformly at random. The idea is to express the uniform distribution over the set of solutions as a probability distribution and then generating samples from this distribution using monte-carlo sampling. We develop novel monte-carlo sampling algorithms that extend our previous work on monte-carlo sampling algorithms for probabilistic networks [4] in which the output of generalized belief propagation was used for sampling.

Our preliminary experiments revealed that our sampling schemes in [4] and its extensions proposed in this paper may fail to output even a single solution for constraint networks that admit few solutions. So we propose to enhance our sampling schemes with techniques like conflict directed back-jumping and no-good learning.

We demonstrate empirically the performance of our search+sampling schemes by comparing them with two previous schemes: (a) the WALKSAT algorithm [5] and (b) the mini-bucket approximation [2]. This work is motivated by a real-world application of generating test programs in the field of functional verification (see [2] for details).

2 Preliminaries

Definition 1 (constraint network). *A constraint network (CN) is defined by a 3-tuple,* $\langle \mathbf{X}, \mathbf{D}, \mathbf{C} \rangle$*, where* $\mathbf{X}$ *is a set of variables* $\mathbf{X} = \{X_1, \ldots, X_n\}$*, associated with a set of discrete-valued domains,* $\mathbf{D} = \{\mathbf{D_1}, \ldots, \mathbf{D_n}\}$*, and a set of constraints* $\mathbf{C} = \{C_1, \ldots, C_r\}$*. Each constraint* C_i *is a pair* $(\mathbf{S_i}, \mathbf{R_i})$*, where* $\mathbf{R_i}$ *is a relation* $\mathbf{R_i} \subseteq \mathbf{D_{S_i}}$ *defined on a subset of variables* $\mathbf{S_i} \subseteq \mathbf{X}$*.* $\mathbf{R_i}$ *contains the allowed tuples of* C_i*. A solution is an assignment of values to variables* $\mathbf{x} = (X_1 = x_1, \ldots, X_n = x_n)$*,* $X_i \in \mathbf{D}_i$*, such that* $\forall\, C_i \in \mathbf{C}$*,* $\mathbf{x}_{\mathbf{S_i}} \in \mathbf{R_i}$*.*

Definition 2 (Random Solution Generation Task). *Let* **sol** *be the set of solutions to a constraint network* $\mathcal{R} = (\mathbf{X}, \mathbf{D}, \mathbf{C})$. *We define the uniform probability distribution* $P_u(\mathbf{x})$ *over* $\mathcal{R}$ *such that for every assignment* $\mathbf{x} = (X_1 = x_1, \ldots, X_n = x_n)$ *to all the variables that is a solution, we have* $P_u(\mathbf{x} \in \mathbf{sol}) = \frac{1}{|\mathbf{sol}|}$ *while for non-solutions we have* $P_u(\mathbf{x} \notin \mathbf{sol}) = 0$. *The task of random solution generation is the task of generating each solution to* $\mathcal{R}$ *with probability* $\frac{1}{|\mathbf{sol}|}$.

3 Generating Solutions Uniformly at Random

In this section, we describe how to generate random solutions using monte-carlo (MC) sampling. We first express the constraint network $\mathcal{R}(\mathbf{X}, \mathbf{D}, \mathbf{C})$ as a uniform probability distribution $\mathcal{P}$ over the space of solutions: $\mathcal{P}(\mathbf{X}) = \alpha \prod_i C_i(\mathbf{S_i} = \mathbf{s_i})$. Here, $C_i(\mathbf{s_i}) = 1$ if $\mathbf{s_i} \in R_i$ and 0 otherwise. $\alpha = 1/\sum \prod_i f_i(\mathbf{S_i})$ is the normalization constant. Note that it is easy to prove that any algorithm that samples from $\mathcal{P}$ generates solutions to the constraint network uniformly at random. This allows us to use the following monte-carlo (MC) sampler to sample from $\mathcal{P}$.

```
Algorithm Monte-Carlo Sampling
Input: A factored distribution $\mathcal{P}$ and a time-bound ,Output: A collection of samples from $\mathcal{P}$.
Repeat until the time-bound expires

  1. FOR i = 1 to n
       (a) Sample $X_i = x_i$ from $P(X_i|X_1 = x_1, \ldots, X_{i-1} = x_{i-1})$
  2. End FOR
  3. If $x_1, \ldots, x_n$ is a solution output it.
```

Hence forth, we will use P to denote the conditional distribution $P(X_i|X_1, \ldots, X_{i-1})$. In [2], a method is presented to compute P in time exponential in tree-width. But tree-width is usually large for real-world networks and so we have to use approximations.

4 Approximating P using Iterative Join Graph Propagation

Because exact methods for computing P are impractical, we consider a generalized belief propagation algorithm [6] called Iterative Join Graph Propagation (IJGP) [3] to compute an approximation to P. IJGP is a form of sum-product belief propagation [6, 3] which takes as input a factored probability distribution $\mathcal{P}$ and a partial assignment $\mathbf{E} = \mathbf{e}$ and then performs message-passing on a special structure called the join-graph. A join-graph is a decomposition of functions in $\mathcal{P}$ into a collection of clusters labeled by variables. The output of IJGP can be used to compute an approximation $Q(X_i|\mathbf{e})$ of $P(X_i|\mathbf{e})$ [3]. If the number of variables in each cluster is bounded by i, we refer to IJGP as IJGP(i). The time and space complexity of one iteration of IJGP(i) is bounded exponentially by i (see [3] for details).

IJGP(i) can be used to compute an approximation to P by executing it with $\mathcal{P}$ and the partial assignment $X_1 = x_1, \ldots, X_{j-1} = x_{j-1}$ as input. Here, IJGP(i) should be executed n times, one for each instantiation of variable X_j in order to generate one full sample. This process may be slow because the complexity of generating N samples is $O(N * n * exp(i))$. To speed-up the sampling process, in [4] we pre-computed the approximation of P by executing IJGP(i) just once yielding a complexity of $O(N * n + exp(i))$.

Thus, at one end of the spectrum we have a method which executes IJGP at each instantiation and at the other end a method which executes IJGP just once before instantiating any variables. Therefore, we introduce a control parameter p which allows running IJGP(i) every p% of the possible n variable instantiations. This helps us analyze the spectrum in between as p changes. We call the resulting technique IJGP(i,p)-sampling.

The intuition behind using parameter p are the results of our preliminary empirical testing, in which we observed that changing only one (or a few) variables to become instantiated often does not impact the approximation Q computed by IJGP(i). Since these reruns of IJGP can present a significant overhead, it can be more cost-effective to rerun IJGP only periodically during sample generation. An important property of completeness is expressed in the following theorem.

Theorem 1 (Completeness). *IJGP(i,p)-sampling has a non-zero probability of generating any arbitrary solution to a constraint satisfaction problem.*

5 Improving the Approximate Sampling Algorithm

It is important to note that when all P's are exact in our monte-carlo sampler, all samples generated are guaranteed to be solutions to the constraint network. However, when we approximate P using IJGP such guarantees do not exist and our scheme will attempt to generate samples that are not consistent (rejection). Our preliminary experiments showed us that the rejection rate of IJGP(i,p)-sampling was quite high for constraint networks that admit few solutions. So in this section, we discuss how to decrease the rejection rate of IJGP(i,p)-sampling by utilizing constraint-based pruning schemes thereby speeding up the rate at which random solutions are generated.

5.1 Introducing Backjumping

Traditional sampling algorithms start sampling anew from the first variable in the ordering when an inconsistent sample is generated. This is clearly inefficient and instead we could backtrack to the previous variable, update the conditional distribution P to reflect the dead-end and re-sample the previous variable. In other words, we could perform backtracking search instead of pure sampling. We propose to use conflict-directed back-jumping for obvious efficiency reasons.

5.2 No-good learning

Conventional monte-carlo sampling methods do not learn no-goods from dead-ends once a sample is rejected. Thus they are subjected to thrashing as happens during systematic search. So we augment our sampling schemes that employ back-jumping search with no-good learning schemes as in [1]. Since each no-good can be considered as a constraint, they can be inserted into any cluster in the join graph that includes the scope of the no-good. So each time a no-good bounded by i is discovered, we check if the no-good can be added to a cluster in the join-graph and if so then subsequent runs of IJGP utilizes this no-good; thereby potentially improving its approximation.

We refer to the algorithm resulting from adding back-jumping search and no-good learning to IJGP(i,p)-sampling as IJGP(i,p)-search-sampling.

Problems (N,K,C,T)	Time	IJGP(3,p)-search-sampling No learning				IJGP(3,p)-search-sampling learning				MBE(3)
		p=0	p=10	p=50	p=100	p=0	p=10	p=50	p=100	p=0
		KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S
50,4,150,4	300s	0.0211 0.0031 138322	0.0284 0.0047 72744	0.0192 0.0027 47278	0.0076 0.0003 19002	0.0187 0.0028 162387	0.0102 0.0019 97560	0.0112 0.0021 38281	0.009 0.0003 15347	0.102 0.089 78218
50,4,180,4	300s	0.0278 0.0038 88329	0.0343 0.0047 59209	0.0221 0.0035 37012	0.0092 0.0006 23671	0.0285 0.0028 74126	0.0119 0.0024 61822	0.0185 0.0029 24102	0.0091 0.0006 11438	0.1116 0.0695 45829
100,4,350,4	1000s	0.0346 0.0074 82290	0.0319 0.0061 42398	0.0108 0.0028 19032	0.011 0.0017 11792	0.0403 0.0086 103690	0.0172 0.0048 37923	0.013 0.0026 25631	0.0053 0.0008 9872	0.134 0.073 93823
100,4,370,4	1000s	0.0249 0.0089 18894	0.0235 0.0062 17883	0.0267 0.0084 2983	0.0156 0.0037 1092	0.0167 0.0058 28346	0.0188 0.0061 14894	0.0143 0.0049 3329	0.0106 0.0019 1981	0.107 0.0332 33895

Table 1. Performance of IJGP(3,p)-sampling and MBE(3)-sampling on random binary CSPs.

Problems (N,K,C,T)	Time	IJGP(3,p)-search-sampling No learning				IJGP(3,p)-search-sampling learning				WALKSAT
		p=0	p=10	p=50	p=100	p=0	p=10	p=50	p=100	p=0
		KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S	KL MSE #S
50,150	20s	0.124 0.019 28002	0.092 0.007 18203	0.083 0.006 9032	0.088 0.006 1033	0.109 0.013 31093	0.089 0.006 11903	0.085 0.005 7392	0.073 0.005 1208	0.114 0.017 45838
50,200	20s	0.117 0.018 48917	0.087 0.0069 18772	0.086 0.0062 8944	0.0896 0.0052 1292	0.1068 0.0124 50962	0.085 0.0055 12636	0.084 0.005 7793	0.078 0.0057 1172	0.103 0.016 42950
100,350	100s	0.123 0.022 89029	0.089 0.009 54832	0.074 0.008 17945	0.082 0.008 1833	0.127 0.023 79293	0.088 0.009 42894	0.074 0.007 27983	0.068 0.005 2094	0.14 0.024 103934
100,400	100s	0.107 0.029 70298	0.077 0.0084 28901	0.049 0.0075 11309	0.024 0.0038 1003	0.128 0.039 60934	0.059 0.0081 39782	0.039 0.0077 9462	0.019 0.0023 1284	0.093 0.019 93024

Table 2. Performance of IJGP(i,p)-search-sampling and WALKSAT on randomly generated 3-SAT problems.

6 Experimental Evaluation

We experimented with randomly generated binary constraint networks, randomly generated 3-satisfiability (SAT) instances and SAT benchmarks available from satlib.org. For each network, we compute the fraction of solutions that each variable-value pair participates in i.e. $P_e(X_i = x_i)$. Our sampling algorithms output a set of solution samples $\mathbf{S}$ from which we compute the approximate marginal distribution: $P_a(X_i = x_i) = \frac{N_{\mathbf{S}(x_i)}}{|\mathbf{S}|}$ where $N_{\mathbf{S}(x_i)}$ is the number of solutions in the set $\mathbf{S}$ with X_i assigned the value x_i. We then compare the exact distribution with the approximate distribution using two error measures (accuracy): (a) *Mean Square error* - the square of the difference between the approximate and the exact, averaged over all values, all variables and all problems and (b) *KL distance* - $P_e(x_i) * log(P_e(x_i)/P_a(x_i))$ averaged over all values, all variables and all problems. Another important criteria is the number of solutions generated which we report in our results.

Note that we compare the performance of IJGP(i,p)-search-sampling with a WALKSAT based solution sampler [5] on all SAT instances and a mini-bucket approximation (MBE(i)) [2] based solution sampler on all CSP instances. Also note that the MBE(i)-based solution sampler used in [2] does not perform search while the one used in our experiments performs search (conflict-directed backjumping to be precise).

6.1 Results on randomly generated CSPs and 3-SAT problems

We experimented with 50 and 100 variable randomly generated constraint networks and 3-SAT problems. All problems are consistent. We had to stay with relatively small problems in order to apply exact techniques to count the solutions that each variable-value pair participates in. All approximate sampling algorithms were given the same amount of time to generate solution samples indicated by column *Time* in Tables 1 and 2. The results are averaged over 1000 instances each for 50-variable problems and 100 instances each for 100 variable problems. We used an i-bound of 3 in all experiments.

Note that the problems become harder as we increase the number of constraints for a fixed number of variables (phase transition). We can see that in the case of IJGP(i,p)-search-sampling, accuracy increases and the number of solutions generated decrease as we increase p (see Tables 1 and 2). Thus, we clearly have a trade-off between accuracy and the number of solutions generated as we change p. It is clear from Table 1 that our new scheme IJGP(i,0)-search-sampling is better than MBE(i) based solution sampler both in terms of accuracy and the number of solutions generated. Also, no-good learning improves the accuracy of IJGP(i,p)-search-sampling in most cases.

When we compare the results of IJGP(i,p)-search-sampling with WALKSAT (see Table 2), we see that the performance of WALKSAT is slightly better than IJGP(i,p)-search-sampling when $p = 0$ in terms of accuracy. However, as we increase $p \geq 10$, the performance of IJGP(i,p)-search-sampling is better than WALKSAT. It is easy to see that WALKSAT dominates IJGP(i,p)-search-sampling in terms of the number of solutions computed for $p = 50, 100$. However for $p = 0, 10$ the number of solutions generated by WALKSAT are comparable to IJGP(i,p)-search-sampling.

6.2 Results on SAT benchmarks from SATLIB

We also experimented with logistics and verification SAT benchmark problems available from satlib.org. On all the these benchmarks instances, we had to reduce the number of solutions that each problem admits by adding unary clauses in order to apply

	Logistics.a			Logistics.b			Logistics.d		
	N=828,Time=1000s			N=843,Time=1000s			N=4713,Time=1000s		
	IJGP(3,10)		WALK	IJGP(3,10)		WALK	IJGP(3,10)		WALK
	No Learn	Learn		No Learn	Learn		No Learn	Learn	
KL	0.00978	0.00193	0.01233	0.00393	0.00403	0.0102	0.0009	0.0003	0.0008
MSE	0.001167	0.00033	0.00622	0.00194	0.00243	0.0097	0.00073	0.00041	0.0002
#S	23763	32893	882	11220	21932	93932	10949	19203	28440

Table 3. KL distance, Mean-squared Error and number of solutions generated by IJGP(3,10)-sampling and Walksat on logistics benchmarks

	Verification1			Verification2		
	N=2654,Time=10000s			N=4713,Time=10000s		
	IJGP (3,10)		WALK	IJGP (3,10)		WALK
	No Learn	Learn		No Learn	Learn	
KL	0.0044	0.0037	0.003	0.0199	0.0154	0.01
MSE	0.0035	0.0021	0.0012	0.009	0.0088	0.0073
#S	1394	945	11342	1893	1038	8390

Table 4. KL distance, Mean-squared Error and number of solutions generated by IJGP(3,10)-sampling and Walksat on verification benchmarks

our exact algorithms. Here, we only experimented with our best performing algorithm IJGP(i,p)-search-sampling with i=3 and p=10. From Table 3 we can see that on the logistics benchmarks, IJGP(3,10)-search-sampling is slightly better than WALKSAT in terms of accuracy while on the verification benchmarks (see Table 4) WALKSAT is slightly better than IJGP(3,10)-search-sampling. WALKSAT however dominates our algorithms in terms of the number of solutions generated except for the Logistics.a instance.

7 Summary and Conclusion

The paper presents a new algorithm for generating random, uniformly distributed solutions for constraint satisfaction problems. The algorithms that we develop fall under the class of monte-carlo sampling algorithms that sample from the output of a generalized belief propagation algorithm and extend our previous work [4].

We show how to improve upon conventional monte-carlo sampling methods by integrating sampling with back-jumping search and no-good learning and is the main contribution of our work. This has the potential of improving the performance of monte-carlo sampling methods used in the belief network literature [4], especially on networks having large number of zero probabilities.

Our results look promising in that we are consistently able to generate near random solution samples. Our best-performing schemes are competitive with the state-of-the-art SAT solution samplers [5] in terms of accuracy and thus present a Monte-carlo (MC) style alternative to random walk solution samplers like WALKSAT [5].

References

1. R. Bayardo and D. Miranker. A complexity analysis of space-bound learning algorithms for the constraint satisfaction problem. In *AAAI*, 1996.
2. Rina Dechter, Kalev Kask, Eyal Bin, and Roy Emek. Generating random solutions for constraint satisfaction problems. In *AAAI*, 2002.
3. Rina Dechter, Kalev Kask, and Robert Mateescu. Iterative join graph propagation. In *UAI '02*, pages 128–136. Morgan Kaufmann, August 2002.
4. Vibhav Gogate and Rina Dechter. Approximate inference algorithms for hybrid bayesian networks with discrete constraints. *UAI-2005*, 2005.
5. Wei Wei, Jordan Erenrich, and Bart Selman. Towards efficient sampling: Exploiting random walk strategies. In *AAAI*, 2004.
6. Jonathan S. Yedidia, William T. Freeman, and Yair Weiss. Generalized belief propagation. In *NIPS*, pages 689–695, 2000.

Reasoning on bipolar preference problems

Student: Maria Silvia Pini
Supervisor: Francesca Rossi

Department of Pure and Applied Mathematics, University of Padova, Italy
E-mail: {mpini,frossi}@math.unipd.it

Abstract. Real-life problems present several kinds of preferences. In this paper we focus on *bipolar problems*, that are problems with both positive and negative preferences. Although seemingly specular notions, these two kinds of preferences should be dealt with differently to obtain the desired natural behaviour. We technically address this by generalizing the soft constraint formalism, and by considering the issue of the compensation between positive and negative preferences. In particular, we suggest how constraint propagation and branch and bound can be adapted to deal with bipolar problems.

1 Introduction

Real-life problems present several kinds of preferences. In this paper we focus on *bipolar problems*, i.e., problems with both positive and negative preferences and we present an algorithm based on branch and bound techniques for solving them. Parts of this paper concerning modelling of bipolar problems have appeared in [3].

Positive and negative preferences could be thought as two symmetric concepts, but this does not happen in real scenarios. In fact, assume, for example, to have a scenario with two objects A and B. If we like both A and B, i.e., if we give to A and B positive preferences, then the overall scenario should be more preferred than having just A or B alone, and so the combination of such a preferences should give an higher positive preference. Instead, if we dislike both A and B, i.e., if we give to A and B negative preferences, then the overall scenario should be less preferred than having just A or B alone and so the combination of such a negative preferences should give a lower negative preference. When dealing with both kinds of preferences, it is natural to express also indifference, which means that we express neither a positive nor a negative preference over an object. A desired behaviour of indifference is that, when combined with any preference, it should not influence the overall preference. Finally, besides combining positive preferences among themselves, and also negative preferences among themselves, we also want to be able to combine positive with negative preferences, allowing compensation, that must produce a positive or a negative preference. For example, if we have a meal with meat (which we like very much) and wine (which we don't like), then what should be the preference of the meal? To know that, we should be able to compensate the positive preference given to meat with the negative one given to wine.

In this paper we start from the soft constraint formalism [2] based on c-semirings, to model negative preferences. We then extend it via a new structure, that models positive preferences and then we define a combination operator between positive and negative preferences to model preference compensation. Finally, we propose how to adapt constraint propagation and branch and bound techniques for finding optimal solutions of bipolar problems.

2 Background: semiring-based soft constraints

A soft constraint [2] is a classical constraint [5] where each instantiation of its variables has an associated value from a (totally or partially ordered) set. This set has two operations, which makes it similar to a semiring, and is called a c-semiring. A c-semiring is a tuple $(A, +, \times, \mathbf{0}, \mathbf{1})$ where: A is a set and $\mathbf{0}, \mathbf{1} \in A$; $+$ is commutative, associative, idempotent, **0** is its unit element, and **1** is its absorbing element; $\times$ is associative, commutative, distributes over $+$, **1** is its unit element and **0** is its absorbing element. Consider the relation $\leq_S$ over A such that $a \leq_S b$ iff $a + b = b$. Then: $\leq_S$ is a partial order; $+$ and $\times$ are monotone on $\leq_S$; **0** is its minimum and **1** its maximum; $(A, \leq_S)$ is a lattice and, $\forall\ a, b \in A$, $a + b = lub(a, b)$. Moreover, if $\times$ is idempotent, then $(A, \leq_S)$ is a distributive lattice and $\times$ is its glb. Informally, the relation $\leq_S$ gives us a way to compare the tuples of values and constraints. In fact, when we have $a \leq_S b$, we will say that *b is better than a*. Given a c-semiring $S = (A, +, \times, \mathbf{0}, \mathbf{1})$, a finite set D (the domain of the variables), and an ordered set of variables V, a constraint is a pair $\langle def, con \rangle$ where $con \subseteq V$ and $def : D^{|con|} \rightarrow A$. Therefore, a constraint specifies a set of variables (the ones in con), and assigns to each tuple of values of D of these variables an element of A. A soft constraint satisfaction problem (SCSP) is just a set of soft constraints over a set of variables. For example, fuzzy CSPs [7] are SCSPs that can be modeled by choosing the c-semiring $S_{FCSP} = ([0, 1], max, min, 0, 1)$ and weighted CSPs [2] are SCSPs that can be modeled by using $S_{WCSP} = (\Re^+, min, +, +\infty, 0)$.

3 Negative preferences

The structure we use to model negative preferences is exactly a c-semiring [2] as described in the previous section. In fact, in a c-semiring there is an element which acts as indifference, that is $\mathbf{1}$, since $\forall a \in A$, $a \times \mathbf{1} = a$ and the combination between negative preferences goes down in the ordering (in fact, $a \times b \leq a, b$), that is a desired property. This interpretation is very natural when considering, for example, the weighted semiring $(R^+, min, +, +\infty, 0)$. In fact, in this case the real numbers are costs and thus negative preferences. The sum of different costs is worse in general w.r.t. the ordering induced by the additive operator (that is, min) of the semiring. From now on, we will use a standard c-semiring to model negative preferences, denoted as: $(N, +_n, \times_n, \bot_n, \top_n)$.

4 Positive preferences

When dealing with positive preferences, we want two main properties to hold: combination should bring to better preferences, and indifference should be lower than all the other positive preferences. These properties can be found in the following structure.

Definition 1. *A positive preference structure is a tuple $(P, +_p, \times_p, \bot_p, \top_p)$ such that P is a set and $\top_p, \bot_p \in P$; $+_p$, the additive operator, is commutative, associative, idempotent, with $\bot_p$ as its unit element ($\forall a \in P, a +_p \bot_p = a$) and $\top_p$ as its absorbing element ($\forall a \in P, a +_p \top_p = \top_p$); $\times_p$, the multiplicative operator, is associative, commutative and distributes over $+_p$ ($a \times_p (b +_p c) = (a \times_p b) +_p (a \times_p c)$), with $\bot_p$ as its unit element and $\top_p$ as its absorbing element*[1].

[1] In fact, the absorbing nature of $\top_p$ can be derived from the other properties.

The additive operator of this structure has the same properties as the corresponding one in c-semirings, and thus it induces a partial order over P in the usual way: $a \leq_p b$ iff $a +_p b = b$. This allows to prove that $+_p$ is monotone over $\leq_p$ (i.e., $\forall a, b \in P$ s. t. $a \leq_p b$ then $a \times_p d \leq_p b \times_p d$, $\forall d \in P$) and that it is the least upper bound in the lattice $(P, \leq_p)$ ($\forall a, b \in P$, $a \times_p b \geq_p a +_p b \geq_p a, b$.). An example of a positive preference structure is $P_1 = (R^+, max, sum, 0, +\infty)$, where preferences are positive reals aggregated with sum and compared with max.

5 Bipolar preference structures

For handing both positive and negative preferences we propose to combine the two structures described in sections 4 and 3 in what we call a *bipolar preference structure*.

Definition 2. *A bipolar preference structure is a tuple* $(N, P, +, \times, \bot, \Box, \top)$ *where*

- $(P, +_{|P}, \times_{|P}, \Box, \top)$ *is a positive preference structure;*
- $(N, +_{|N}, \times_{|N}, \bot, \Box)$ *is a c-semiring;*
- $+ : (N \cup P)^2 \longrightarrow (N \cup P)$ *is s. t.* $a_n + a_p = a_p$, $\forall a_n \in N$ *and* $a_p \in P$*; this operator induces a partial ordering on* $N \cup P$*:* $\forall a, b \in P \cup N$, $a \leq b$ *iff* $a + b = b$*;*
- $\times : (N \cup P)^2 \longrightarrow (N \cup P)$ *is an operator that,* $\forall\ a, b, c \in N \cup P$*, satisfies commutativity (*$a \times b = b \times a$*) and monotonicity property (if* $a \leq b$*,* $a \times c \leq b \times c$*).*

Bipolar preference structures generalize both c-semirings and positive structures. In fact, when in a bipolar structure $\Box = \top$, we have a c-semiring and, when $\Box = \bot$, we have a positive structure. In the following, we will write $+_n$ instead of $+_{|N}$ and $+_p$ instead of $+_{|P}$. Similarly for $\times_n$ and $\times_p$. When operator $\times$ will be applied to a pair in $(N \times P)$, we will sometimes write $\times_{np}$ and we will call it compensation operator. Given the way the ordering is induced by $+$ on $N \cup P$, easily, we have $\bot \leq \Box \leq \top$. Thus, there is a unique maximum element (that is, $\top$), a unique minimum element (that is, $\bot$); the element $\Box$ is smaller than any positive preference and greater than any negative preference, and it is used to model indifference. A bipolar preference structure allows us to have a richer structure for one kind of preference, that is common in real-life problems. In fact, we can have different lattices $(P, \leq_p)$ and $(N, \leq_n)$. For example, we could be satisfied with just two levels of negative preferences, while requiring several levels of positive preferences.

It is easy to show that the combination of a positive and a negative preference is a preference which is higher than, or equal to, the negative one and lower than, or equal to, the positive one. The following theorem holds when a bipolar preference structure $(N, P, +, \times, \bot, \Box, \top)$ is given.

Theorem 1. *For all* $p \in P$ *and* $n \in N$*,* $n \leq p \times n \leq p$*.*

This means that the compensation of positive and negative preferences must lie in one of the chains between the two combined preferences, that passes through the indifference element $\Box$. Possible choices for combining strictly positive with strictly negative preferences are thus the average, the median, the min or the max operator.

In general, the compensation operator $\times$ may be not associative. We have defined a list of sufficient conditions for non-associativity of $\times$. For example, if $\times_p$ or $\times_n$ is idempotent and if there are at least two elements $p \in P$ and $n \in N$, that are different from $\square$ s. t. $p \times n = \square$, then $\times$ is not associative. Since some of these conditions often occur naturally in practice, it is not reasonable to require associativity of $\times$.

In the following table each row corresponds to a bipolar preference structure.

N,P	$+_p$, $\times_p$	$+_n$, $\times_n$	$\times_{np}$	$\perp$, $\square$, $\top$
R^-, R^+	max, sum	max, sum	sum	$-\infty$, 0, $+\infty$
$[-1, 0]$, $[0, 1]$	max, max	max, min	sum	-1, 0, 1
$[0, 1]$, $[1, +\infty]$	max, prod	max, prod	prod	0, 1, $+\infty$

In the first structure positive preferences are positive real numbers and negative preferences are negative real numbers, the compensation is given by sum, while the ordering is given by max. In the second structure positive preferences are between 0 and 1 and negative preferences between -1 and 0. Again, compensation is sum, and the order is given by max. In the third structure positive preferences are between 1 and +∞ and negative preferences between 0 and 1. Compensation is obtained by multiplying the preferences and ordering is again via max.

6 Bipolar preference problems

Once we have defined bipolar preference structures, we can define a notion of bipolar constraint, which is just a constraint where each assignment of values to its variables is associated to one of the elements in a bipolar preference structure.

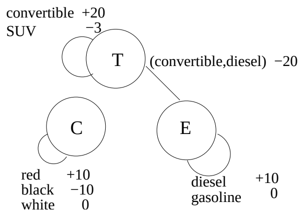
Definition 3. *Given a bipolar preference structure $(N, P, +, \times, \perp, \square, \top)$ a finite set D (the domain of the variables), and an ordered set of variables V, a constraint is a pair $\langle def, con \rangle$ where $con \subseteq V$ and $def : D^{|con|} \rightarrow (N \cup P)$.*

A bipolar CSP (V, C) is then just a set of variables V and a set of bipolar constraints C over V. We propose a way of defining the optimal solutions of a bipolar CSP that avoids problems due to the possible non-associativity. A solution of a bipolar CSP (V, C) is a complete assignment to all variables in V, with an associated preference that is computed by combining all the positive preferences associated to its projections over the constraints, combining all the negative preferences associated to its projections over the constraints, and then, combining the two preferences obtained so far. If $\times$ is associative, then other definitions of solution preference could be used while giving the same result. A solution s is an optimal solution if there is no other solution s' with $pref(s') > pref(s)$.

6.1 An example of bipolar CSP

Consider the scenario in which we want to buy a car. We have some preferences over the car's features. In terms of color, we like red, we are indifferent to white, and we hate black. Also, we like convertible cars a lot and we don't care much for SUVs. In terms of engines, we like diesel. However, we don't want a diesel convertible.

We may decide to represent positive preferences via positive integers and negative preferences via negative integers. Moreover, we may decide to maximize the sum of all kinds of preferences. This can be modelled by a preference structure where $N = [-\infty, 0]$, $P = [0, +\infty]$, $+ =$max, $\times$=sum, $\perp = -\infty$, $\square = 0$, $\top = +\infty$. We now model the example above over this bipolar preference structure. We have three variables: variable T (type) with domain $\{$convertible,SUV$\}$, variable E (engine) with domain $\{$diesel,gasoline$\}$, and variable C (color) with domain $\{$red,white,black$\}$. For the preferences over the colors, we define a constraint $c_1 = \langle def_1, \{C\} \rangle$ where, for example, we set def_1(red) $= +10$, def_1(black) $= -10$, and def_1(white) $= 0$. We also have a constraint over car types, say $c_2 = \langle def_2, \{T\} \rangle$ where we set def_2(convertible) $= +20$ and def_2(SUV) $= -3$. The constraint over engines can then be $c_3 = \langle def_3, \{E\} \rangle$, where we can set def_3(diesel) $= +10$ and def_3(gasoline) $= 0$. Finally, the last preference can be modelled by a constraint $c_4 = \langle def_4, \{T, E\} \rangle$, where we can set def_4(convertible, diesel) $= -20$ and $def_4(a, b) = 0$ for $(a, b) \neq$(convertible, diesel). The following figure shows the structure of such a bipolar CSP, where we use value 0 for modelling indifference.



Consider, now, solution s_1 =(red,convertible,diesel): we have $pref(s_1) = (def_1$(red)$+ def_2$(convertible)$ + def_3$(diesel)$) + def_4$(convertible, diesel) $= (10 + 20 + 10) + (-20) = 20$. We can compute the preference of all other solutions and we can see that the optimal solution is (red, convertible, gasoline) with global preference of 30.

6.2 Solving bipolar CSPs

Bipolar problems are NP-hard, since they generalise both classical and soft constraints, which are already difficult problems. However, we can devise algorithms and heuristics to solve them, hopefully efficiently in the average case. Preference problems based on c-semirings can be solved via a branch and bound technique, possibly augmented via soft constraint propagation, which may lower the preferences and thus allow for the computation of better bounds [2]. In bipolar CSPs, we have positive and negative preferences. Branch and bound techniques can be adapted to compute, at each search node k, an upper bound ub to the preferences of all the solutions in the k-rooted subtree.

If $\times$ is non-associative, then each node is associated to a positive and a negative preference, say p and n, which is obtained by aggregating all preferences of the same type in the instantiated part of the problem. An upper bound for the subtree can be computed, for example, by taking the aggregation of all the best positive and negative preferences in the non-instantiated part of the problem, say p' and n', and by aggregating them to the positive and negative preferences of the current node. This produces the upper bound $ub = (p \times_p p') \times (n \times_n n')$, where $p' = p_1 \times_p \ldots \times_p p_s$, $n' = n_1 \times_n \ldots \times_n n_w$, and $r = s + w$ is the number of non-instantiated variables/constraints. Thus ub can be computed via $r - 1$ aggregation steps and one compensation step.

If $\times$ is associative, however, we don't need to postpone compensation until all constraints have been considered, but we can interleave compensation and aggregation while searching for an optimal solution. This means that we can keep just one value $v = p \times n$ for each search node, that can be positive or negative, which is obtained by aggregating all preferences (both positive and negative) obtained in the instantiated part of the problem. The same can be done considering the best preferences in the uninstantiated part of the problem, obtaining a value v'. Thus, ub can now be written as $ub = v \times v'$, where $v' = a_1 \times \ldots \times a_r$, where $a_i \in N \cup P$ is the best preference found in a constraint of the uninstantiated part of the problem. Thus now ub can be computed via at most $r-1$ steps among which there can be many compensation steps. A compensation can generate the indifference $\Box$, which is the unit element for the compensation operator. Thus, when $\Box$ is generated, the successive computation step can be avoided.

If $ub \leq v$, where v is the preference of the best solution so far, we can prune the k-rooted subtree. To improve this upper bound, we can propagate negative preferences as it is done in soft constraints [2, 4]. In fact, such a propagation may lower the negative values while not changing the semantics of the problem. Due to the monotonicity of $\times$ and $\times_n$, the upper bound may thus become smaller and allow for more pruning. Positive preference can be propagated as well. However, since $\times_p$ returns higher positive preferences, their propagation produces higher values. This is not helpful in improving the upper bound, since monotonicity of $\times$ implies that a higher upper bound is obtained.

7 Related and future work

Bipolar reasoning and preferences have attracted some interest in the AI community. in [1], a bipolar preference model based on a fuzzy-possibilistic approach is described, but positive and negative preferences are kept separate and no compensation is allowed. In [6] totally ordered unipolar and bipolar preference scales are used, whereas we have presented a way to deal with partially ordered bipolar scales.

We plan to consider the presence of uncertainty in bipolar problems, possibly using possibility theory and to develop solving techniques for such scenarios. Another line of future research is the generalization of other preference formalisms, such as multi-criteria methods and CP-nets, to deal with bipolar preferences and to study the relation between bipolarization and importance tredeoffs.

Acknowledgements. This is a joint work with S. Bistarelli (University of Pescara, Italy) and K. Brent Venable (University of Padova, Italy).

References

1. S. Benferhat, D. Dubois, S. Kaci, and H. Prade. Bipolar representation and fusion of preferences in the possibilistic logic framework. In *KR 2002*. Morgan Kaufmann, 2002.
2. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint solving and optimization. *Journal of the ACM*, 44(2):201–236, mar 1997.
3. S. Bistarelli, M. S. Pini, F. Rossi, and K. B. Venable. Bipolar preference problems. In *ECAI-06 (poster)*, 2006.
4. M. Cooper and T. Schiex. Arc consistency for soft constraints. *AI Journal*, 154(1-2), 2004.
5. R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.
6. M. Grabisch, B. de Baets, and J. Fodor. The quest for rings on bipolar scales. *Int. Journ. of Uncertainty, Fuzziness and Knowledge-Based Systems*, 2003.
7. Zs. Ruttkay. Fuzzy constraint satisfaction. In *3rd IEEE International Conference on Fuzzy Systems*, pages 1263–1268, 1994.

A Case Study on Earliness/Tardiness Scheduling by Constraint Programming

Student: Jan Kelbel
Supervisor: Zdeněk Hanzálek

Department of Control Engineering, Faculty of Electrical Engineering
Czech Technical University in Prague
`kelbelj@fel.cvut.cz, hanzalek@fel.cvut.cz`

Abstract. The aim of this paper is to solve a problem on scheduling with earliness and tardiness costs using constraint programming approach, and to compare the results with the ones from the original timed automata approach. The case study, a production of lacquers, includes some real life features like operating hours or changeover times. Considering the earliness and tardiness costs of the case study, the problem was converted to the problem with objective of minimizing the total waiting time. Then a search heuristic was applied.

Keywords: scheduling, constraint programming, earliness and tardiness costs

The paper is submitted only to the Doctoral Programme.

1 Introduction

This paper deals with an application of constraint-based scheduling on a scheduling problem with earliness and tardiness costs. The case study, a production of lacquers, is originally from project AMETIST [1], where it was successfully solved using timed automata approach [2]. The primary motivation of this work is to compare the application of constraint programming with the timed automata solution (that was better than a MIP approach [1]).

The problem can be classified as the resource-constrained project scheduling problem, in structure similar to the job shop scheduling problem, with distinct due dates and release dates, and with job dependent earliness and tardiness costs. The scheduling problem with earliness and tardiness costs (E/T) and given due dates is NP-hard already in one-machine version [3]. There are OR methods specialized to solve some restricted E/T problems, e.g. [4]. More close to our problem is the E/T job shop scheduling problem in [5], where hybrid CP/MIP approach was used. In contrast, pure CP techniques are described in this paper.

Our method to solve the case study results from its feature that earliness costs are almost 50 times smaller than tardiness costs. The problem was converted to the problem with objective of minimizing the total waiting time subject to release times and due dates. Then a variation of time-directed labeling procedure was applied.

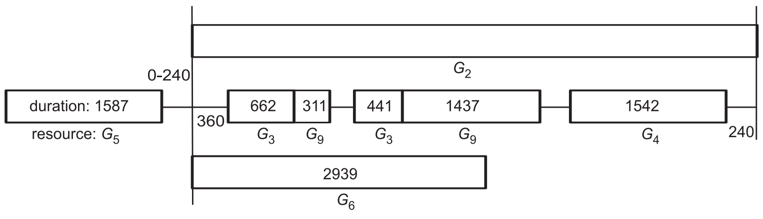


Fig. 1. The example recipe for uni lacquer.

The paper [2] introduces three versions of the lacquer production problem. Two of them, a *basic case study* and an *extended case study*, are in focus of our work. A *stochastic case study*, which deals with unavailability of resources due to breakdown or maintenance, is described in detail in [6].

The paper is organized as follows. The scheduling problem is described in Sect. 2, while Sect. 3 deals with the some aspects of constraint model and describes our approach to the search procedure. Results of experiments are presented in Sect. 4.

As a constraint programming environment we chose ILOG Solver and Scheduler via ILOG OPL Studio.

2 The Problem Statement

The production of a lacquer is described by a recipe. The recipe defines processing steps, called tasks according to the scheduling theory, to be performed. The definition includes processing times of the tasks, resources required for each task, and precedence constraints. Some of the precedences have delay constraints, i.e. maximal and minimal delays (time lags) between tasks are specified.

The case study contains 29 jobs, i.e. orders of lacquer to be produced. The job is specified by the recipe, quantity, release date (earliest start time) and due date. Processing times of tasks are dependent on the quantity of a lacquer. Three different recipes are included in the problem – for lacquer types uni, metallic and bronze.

The tasks of one recipe form a sequence, where each task requires one resource. Further, there is a special resource G_1 or G_2 (mixing vessels) that is needed for nearly the whole time of production of the job. In order to avoid multiprocessor tasks, a shadow task is created to model processing on the special resource. The example recipe is depicted in Fig. 1.

Case Studies

The paper introduces three instances of the two case studies. The goal of the *basic case study* (denoted as BF) is to find any feasible schedule using simplified model of the production. Each task of the recipe has processing time that is the same for all orders, and all resources are available continuously.

The *extended case study* (EO) is the cost optimization problem. Objective function introduced in [2] is a combination of total weighted earliness (a cost for storage of orders that are finished too early), total weighted tardiness (a penalty payment for delayed orders), and changeover cost. Due to the complexity of this objective function evaluation, a simpler objective function of total earliness of all jobs while tardy jobs are not allowed was introduced:

$$F = \sum_{j \in \mathcal{J}} d_j - C_j, \tag{1}$$

where $\mathcal{J}$ is set of jobs, d_j is due date and C_j is completion time of job j, and $d_j \geq C_j \quad \forall j \in \mathcal{J}$. The value of the production total cost, which is needed for comparison with other approaches, is computed subsequently from the results of the optimization. Next, some resources have more exact model of behaviour concerning changeover times and costs. Operating hours of resources have an exact model with breaks. Tasks cannot be scheduled during breaks, and since some tasks have processing time longer than the available time between breaks, these task are breakable, i.e. allowed to be interrupted by breaks.

The third case study instance is the *extended case study* with *performance factors* (EOP), where the performance factors model the unavailability of resources due to breakdown or maintenance [2]. The performance factor is assigned to each resource, and is used to extend the processing time of a task requiring the resource.

Resources

The resources are grouped according to their types. Some groups contain more than one machine, i.e. there are more machines of that type. Inside the group, the machines are parallel identical resources in terms of the scheduling theory [7], or in terms of constraint-based scheduling [8], the groups are discrete resources with capacity greater than one.

Groups with parallel identical resources are:

- Mixing vessel metal $G_1 = \{R_{11}, R_{12}, R_{13}\}$ is used in production of metallic and bronze lacquers.
- Mixing vessel uni $G_2 = \{R_{21}, R_{22}\}$ is used for production of uni lacquers.
- Dose spinner $G_3 = \{R_{31}, R_{32}\}$ is used in all recipes.
- Filling station $G_4 = \{R_{41}, R_{42}\}$ resource with changeover time and cost – cleaning is needed when two successive jobs are of different lacquer type. It is used in all recipes.

Dedicated resources are:

- Disperser $G_5 = \{R_5\}$ is used in uni lacquer recipe.
- Dispersing line $G_6 = \{R_6\}$ is used in uni lacquer recipe.
- Bronze mixer $G_7 = \{R_7\}$ is used in bronze lacquer recipe.
- Bronze dose spinner $G_8 = \{R_8\}$ is used in bronze lacquer recipe.

Finally, there is unrestricted resource laboratory G_9.

3 Modelling the Problem

With naive application of constraint programming, only solutions for smaller instances of the problem were found. So the state space of the problem was reduced, and a search heuristic was applied, as it is described in following paragraphs.

Constraints in the model

Heuristic constraint called *non-overtaking* [2] is used in order to reduce the search space of the problem. It ensures that job started earlier will be finished earlier. We used even more constraining version of this heuristic. Tasks of a job with earlier due date are constrained to start earlier. Jobs of the same lacquer recipe of the type r are indexed with $i \in \{1, \dots, n_r\}$ according to the increasing due date. $\mathcal{T}_r$ is the set of indices of tasks in recipe type r, and $S_{i,k}$ represents starting time of the task k of the job i. Then these constraints are declared:

$$S_{i,k} <= S_{i+1,k} \ , \quad \forall i \in \{1, \dots, n_r - 1\} \text{ and } \forall k \in \mathcal{T}_r .$$

Applying this heuristic on the model with constant processing times of tasks will remove symmetry from the search space. However, this heuristic was used also in the *extended case study* with task processing times dependent on the quantity of lacquer, leading to loss of some solutions.

Concerning the filtering algorithms on resources, the Edge-Finding algorithm was used on unary and also on discrete resources. Due to the changeover times, the discrete resource G_4 was modelled as a set of alternative unary resources.

Search Procedure

The search procedures that are available in OPL studio are written to minimize C_{max}. For example, the *SetTimes* search heuristic [9, 10] builds the schedule from earliest starting dates, and also default labeling procedure starts from the beginning of the domains. The objective of our problem is to schedule tasks as late as possible, but still not to exceed due date. The idea of our search procedure is to try to build the schedule backwards. At first, we try to place tasks as near as possible to the due dates.

One alternative of implementation of this approach is to create a new search procedure. The other one is that the time axis can be reversed and standard search procedures used, i.e. to convert the problem to the problem with objective of minimizing the total waiting time subject to release times and due dates.

Because *SetTimes* sometimes missed all solutions (it can miss a solution in some cases, which is also our case), we used a search heuristic, which is a simple version of time-directed labeling procedure [11]:

1. sort final tasks of jobs by latest possible completion time in decreasing order
2. for each task state two alternatives in the search tree: assign the latest possible completion time as the end time of the task, or decrease the latest possible completion time of the task.

Finally, the search heuristic was modified to be used with reversed time axis model, which was a chosen alternative.

The search heuristic defines the shape of the search tree. It was explored using two search strategies – Depth First Search (DFS) and Limited Discrepancy Search (LDS, denoted as Slice-Based Search in OPL).

4 Experimental Results

The subjects of the experiments were the three case study instances introduced in Sect. 2. The problem instance as is stated in [2] can be simplified. Some tasks can be eliminated, e.g. task requiring unrestricted resource laboratory can be replaced by a delay constraint, and some tasks can be put together. Finally, we end up with 139 tasks (of which 110 are breakable) in 29 jobs.

Time for the whole schedule is 9 weeks in 1 minute resolution (90720 minutes), but the time available for each job is about 2 weeks from release date to due date, that makes the search space smaller. Moreover, due dates of the jobs are almost evenly distributed from 3^{rd} to 9^{th} week.

The results are presented in Tab. 1, where column headers DFS-NO and LDS-NO determine the used search strategy with the *non-overtaking* constraint in the model, while LDS column corresponds to the model without the constraint. The “timed automata” column corresponds to results from [2].

The constraint programming experiments were run on Intel P4 3GHz CPU with 1GB of RAM, using OPL Studio 3.6 under Windows XP, timed automata experiments were run on Intel P4 Xeon 2.6GHz [2].

Table 1. Results for test instances

	DFS-NO		LDS-NO		LDS		timed automata	
	cost	CPU	cost	CPU	cost	CPU	cost	CPU
BF	–	0.09 s	–	0.09 s	–	–	–	0.45 s
EO	455,758	1374 s	455,758	18.4 s	454,246	181 s	*	*
EOP	1,234,959	1026 s	1,186,851	12.7 s	920,240	220 s	≈ 2,100,000	≈ 600 s

– . . . value not applicable
* . . . value not available

Comparing the results in columns DFS-NO and LDS-NO, wee can see big speed up in computation, i.e. the Limited Discrepancy Search helped our search heuristic a lot. In contrast, when using *SetTimes* search heuristic, no solution was found using either of the search strategies.

Concerning the impact of *non-overtaking* constraint, it leads to more or less significantly better results, but at the high expense of CPU time consumed by optimization.

5 Conclusion and Future Work

We showed a solution of a case study on resource-constrained scheduling problem with earliness/tardiness costs. With constraint programming approach, we found a solution of the EOP problem that is notably better than the solution found with timed automata. However, we tested our method only on three instances of the problem. There is also an expectation, that the 29 job instances are maybe one of the largest solvable in reasonable time. The plan for the future is at first to test the method on randomly generated data. Then our method could be improved, possibly by using approaches to solve similar problems.

References

1. AMETIST: European Community Project IST-2001-35304 (Advanced Methods for Timed Systems). http://ametist.cs.utwente.nl/ (2002)
2. Behrmann, G., Brinksma, E., Hendriks, M., Mader, A.: Production Scheduling by Reachability Analysis - A Case Study. In: Workshop on Parallel and Distributed Real-Time Systems (WPDRTS), published by IEEE Computer Society Press, Los Alamitos, California (2005)
3. Baker, K.R., Scudder, G.D.: Sequencing with earliness and tardiness penalties: A review. Operations Research **38**(1) (1990) 22–36
4. Sourd, F., Kedad-Sidhoum, S.: The one machine scheduling with earliness and tardiness penalties. Journal of Scheduling **6**(6) (2003) 533–549
5. Beck, J.C., Refalo, P.: A Hybrid Approach to Scheduling with Earliness and Tardiness Costs. Annals of Operations Research **118**(1–4) (2003) 49–71
6. Bohnenkamp, H.C., Hermanns, H., Klaren, R., Mader, A., Usenko, Y.S.: Synthesis and stochastic assessment of schedules for lacquer production. In: 1st Int. Conf. on Quantitative Evaluation of Systems (QEST), published by IEEE Computer Society Press, Los Alamitos, California (2004)
7. Blazewicz, J., Ecker, K.H., Pesch, E., Schmidt, G., Werglarz, J.: Scheduling Computer and Manufacturing Processes. Second edn. Springer-Verlag, Berlin (2001)
8. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-Based scheduling: Applying Constraint Programming to Scheduling Problems. Kluwer Academic Publishers (2001)
9. Le Pape, C., Couronne, P., Vergamini, D., Gosselin, V.: Time-versus-Capacity Compromises in Project Scheduling. In: Proceedings of the Thirteenth Workshop of the U.K. Planning Special Interest Group. (1994)
10. ILOG S.A.: ILOG OPL Studio 3.5 Language Manual. (2001)
11. van Hentenryck, P., Perron, L., Puget, J.F.: Search and strategies in OPL. ACM Transactions on Computational Logic **1**(2) (2000) 285–320

Limited Full Arc Consistency[⋆]

Student: Josef Zlomek
Supervisor: Roman Barták

Faculty of Mathematics and Physics
Charles University, Prague, Czech Republic
`zlomek@kti.mff.cuni.cz`

Abstract. Consistency techniques proved to be an efficient method for reducing the search space of a CSP. Several consistency techniques were proposed for soft constraints too. In this paper, we introduce a new stronger form of consistency called *limited full arc consistency* (LFAC*) for the Weighted CSP. We provide an algorithm for enforcing LFAC* and for maintaining it during Branch & Bound search. We also present some preliminary experimental results.

1 Introduction

Consistency techniques reduce the search space of a CSP [1] and thus make larger problems solvable. They reduce the variables' domains by removing values that can't be in any solution. A value can be pruned if there is no assignment including the value that satisfies all constraints.

Weighted CSP [2] is a well-known optimization version of CSP framework. WCSP extends classical CSP by assigning costs to tuples. The costs from different constraints are combined together. Solutions of WCSP are those complete assignments that have the lowest combined cost. Because of the costs, we can use the optimality reasoning in addition to the feasibility reasoning. We can prune a value if the value causes the cost of any complete assignment to be greater than the minimal cost found so far. Such a value obviously can't be in a solution.

Several arc-based consistency techniques have been introduced into WCSP recently: *arc consistency* (AC*) [6], *full directional arc consistency* (FDAC*) [5], and *existential directional arc consistency* (EDAC*) [3]. In order to detect too high combined costs, all these consistency techniques employ moving costs from one constraint to another. However, these consistency techniques restrict the movements of particular types to corresponding directions according to the order of variables.

The arc consistency techniques for a classical CSP do not restrict the pruning process to one direction. It is also quite limiting to restrict the direction in WCSP. In this paper, we introduce a new form of local consistency *limited full arc consistency* (LFAC*) that does not use fixed directions of cost moving.

The paper is structured as follows. Section 2 gives preliminary definitions. Section 3 defines limited full arc consistency and proposes its enforcing algorithm.

⋆ Supported by the Czech Science Foundation under the contract 201/05/H014 and by Grant Agency of Charles University under the contract 356/2006.

The correctness of the algorithm is shown too. Section 4 gives the experimental results of LFAC*. Section 5 gives conclusions and directions for future work.

2 Preliminaries

2.1 Frameworks

A *constraint satisfaction problem* (CSP) is a triple $P = (V, D, C)$. V is a set of variables. Each variable $v_i \in V$ has a finite domain $D_i \in D$ of possible values. C is a set of constraints. Constraint c is a relation defined on a subset of variables, denoted as $vars(c)$. A unary constraint is a constraint $C_i \subseteq D_i$, a binary constraint is a constraint $C_{ij} \subseteq D_i \times D_j$. A tuple is an assignment to a set of variables. Tuple t is consistent if all constraints referring only to variables assigned by t are satisfied. A solution of P is a consistent complete assignment.

Following the paper [5], a *weighted CSP* (WCSP) is a tuple $P = (k, V, D, C)$. $k \in \{1, \ldots, \infty\}$ defines the valuation structure $S(k) = (\{0, \ldots, k\}, \oplus, \geq)$, where $\oplus$ is defined as $a \oplus b = \min(k, a + b)$ and $\geq$ is the standard order on natural numbers. V is a set of variables, D is a set of variables' domains, and C is a set of constraints. Constraints assign costs to assignments to variables. For instance, a binary constraint is a function $C_{ij} : D_i \times D_j \to \{0, \ldots, k\}$. $C_\emptyset$ is a nullary constraint representing the lower bound of the cost of the solution. The cost of a tuple is the combined cost of the constraints. A solution is a tuple with the minimal cost.

2.2 Some Local Consistencies in WCSP

Definition 1. *[5] Variable v_i is node consistent (NC*) if for all values $a \in D_i$, $C_\emptyset \oplus C_i(a) < k$, and there exists a value $a \in D_i$ such that $C_i(a) = 0$. P is NC* if every variable is NC*.*

Definition 2. *[5] Value $b \in D_j$ is a* simple support *for $a \in D_i$ if $C_{ij}(a, b) = 0$. Value $b \in D_j$ is a* full support *for $a \in D_i$ if $C_{ij}(a, b) \oplus C_j(b) = 0$.*

Definition 3. *[5] Constraint C_{ij} is arc consistent (AC) if every value $a \in D_i$ has a simple support in D_j. Variable v_i is AC if every constraint C_{ij} is AC. P is AC* if all variables are AC and NC*.*

Definition 4. *[3] Constraint C_{ij} is full arc consistent (FAC) if every value $a \in D_i$ has a full support in D_j. P is FAC* if every binary constraint is FAC and P is NC*.*

Definition 5. *[5] Variable v_i is full directional arc consistent (FDAC*) if every constraint C_{ij} such that $j > i$ is FAC, every constraint C_{ij} such that $j < i$ is AC, and v_i is NC*.*

Definition 6. *[3] Variable v_i is existential arc consistent (EAC) if there is at least one value $a \in D_i$ such that $C_i(a) = 0$ and it has a full support in every constraint C_{ij}. P is EAC* if every variable is EAC and NC*.*

Definition 7. *[3] P is EDAC* if it is FDAC* and EAC*.*

Definition 8. *[5] Subtraction $\ominus$ of b from a is defined as*

$$a \ominus b = \begin{cases} a - b & a \neq k \\ k & a = k \end{cases}$$

Definition 9. *[5] Projection of α cost units from $C_{ij} \in C$ to value $a \in D_i$ means subtracting α from $C_{ij}(a, b)$ $\forall b \in D_j$ and adding α to $C_i(a)$.*

Definition 10. *[5] Extension of β cost units from value $a \in D_i$ to $C_{ij} \in C$ means adding β to $C_{ij}(a, b)$ $\forall b \in D_j$ and subtracting β from $C_i(a)$.*

3 Limited Full Arc Consistency

The projection and extension operations, which are employed by the consistency algorithms like FDAC*[5] and EDAC*[3], move the costs from one constraint to another constraint. If the unary cost of a value reaches the upper bound, the value is pruned from the variable's domain. Because we want to prune as many values as possible, we would like to combine as much costs as possible. Stronger pruning will lead to smaller number of visited nodes during search, and hopefully to better running time.

However, finding a full support for every value is not possible in some cases. When we attempt to do so, we can enter an infinite loop (see figure 1). In order to avoid infinite loops, we detect them and do not enter them. For each value, we remember the maximum unary cost it had at some moment. If the propagation results in increasing the maximum known cost for some value, we have not been in this state yet and we allow the propagation. Otherwise, if no unary cost exceeds the maximum, we know we have been in the same or better state and thus we do not propagate. Therefore, we do not enter infinite loops.

Because of the loop-cutting, the consistency enforced using this methods is weaker than FAC* (which is impossible to enforce anyway). We call this form of consistency Limited Full Arc Consistency (LFAC*).

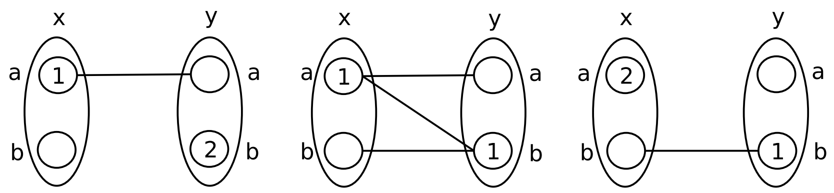


Fig. 1. There are variables x and y, each with values a and b. Unary costs are illustrated inside the domain value. Edges are the binary costs of value 1, zero costs are not shown. Value $a \in x$ does not have a full support. When we find the full support by extending 1 cost unit from $b \in y$ and projecting 1 cost unit to $a \in x$, we are in the similar situation as in the beginning, but reversed. Now $b \in y$ does not have a full support.

Definition 11. *Constraint C_{ij} is LFAC* if C_{ij} is FAC* and C_{ji} is AC*, or C_{ij} is AC* and C_{ji} is FAC*. Variable v_i is LFAC* if all its binary constraints are LFAC*.*

Remark 1. Constraint C_{ij} is LFAC* if it has full supports in one direction and simple supports in the other direction.

The algorithm for enforcing LFAC* employs two ideas mentioned in the beginning of this section. It concentrates as many costs as possible, and it uses the loop-cutting mechanism.

The algorithm maintains a set of variables to consider. Initially, the set contains all unassigned variables. In each step, one variable in the set is selected and removed from the set. The costs are concentrated to the selected variable from its neighbors, and the infeasible values are pruned. Because we want to concentrate as many costs as possible to the next variable too, we must ensure that there are costs to be moved to the next variable from its neighbors. It might not be possible to move all costs from the selected variable to the next variable directly. Therefore, we move the rest of costs from the selected variable back to its neighbors, because these neighbors may also be the neighbors of the next variable.

While the costs are moved from the selected variable, new variables are added into the set of variables to consider. A variable is added to the set if the maximum unary costs were increased, or when some values were pruned.

The algorithm enforces the LFAC* property, i.e. each binary constraint is AC* in one direction and FAC* in the other direction, and each unary constraint is NC*. The justification follows.

The procedure $Prune(v)$ removes all infeasible values of variable v and thus makes the variable NC*. The procedure is called whenever the costs are moved to v. Therefore, the problem remains NC* after moving the costs to any variable.

Each unassigned variable is examined in the inner loop at least once. Given a variable v_i, the procedure $ConcentrateCosts(v_i)$ makes all the constraints C_{ij} FAC*. The procedure $ExpandCosts(v_i)$ makes the constraints C_{ji} FAC* if the unary costs in v_j exceed the previous maximum unary costs.

Since both variables of the constraint C_{ij} are examined in the inner loop, the constraint is made FAC* in one direction and later in the other direction. We need to prove that if the constraint C_{ij} is FAC* and C_{ji} is made FAC* the constraint C_{ij} remains at least AC*.

When constraint C_{ij} is FAC*, for every $a \in D_i$ there is a full support $b \in D_j$, which implies that $C_{ij}(a, b) = 0$. When the constraint C_{ji} is made FAC*, costs are extended from C_i and projected to C_j. In order to make $b \in D_j$ FAC*, we need to project $m = \min_{a' \in D_i}\{C_{ji}(b, a') \oplus C_i(a')\}$ cost units to $b \in D_j$, and the $m - C_{ji}(b, a')$ cost units have to be extended first to $C_{ji}(b, a')$. Since $0 = C_{ji}(b, a) \leq m$, m cost units are extended from $a \in D_i$ and projected to $b \in D_j$. Therefore, $C_{ji}(b, a) = 0$ still holds. The value $a \in D_i$ has a simple support $b \in D_j$ and therefore $a \in D_i$ is AC*.

Algorithm 1 LFAC* enforcing algorithm

```
1: $Q \leftarrow$ unassigned variables
2: while $Q \neq \emptyset$ do
3:    InitBarriers()
4:    while $Q \neq \emptyset$ do
5:       $v$ = variable from $Q$ with the smallest domain
6:       $changed\_unary\_cost = ConcentrateCosts(v)$
7:       if $changed\_unary\_cost$ then
8:          $Prune(v)$
9:       end if
10:      $ExpandCosts(v)$
11:   end while
12:   $v$ = next variable chosen by heuristic
13:   $changed\_unary\_cost = ConcentrateCosts(v)$
14:   if $changed\_unary\_cost$ then
15:      if $Prune(v)$ then
16:         $Q \leftarrow Q \cup \{v\}$
17:      end if
18:   end if
19:   for all $v \in$ unassigned variables do
20:      if $Prune(v)$ then
21:         $Q \leftarrow Q \cup \{v\}$
22:      end if
23:   end for
24: end while
```

4 Experimental results

We have implemented the LFAC* enforcing algorithm to the ToolBar [4] solver. The experiments were performed on Pentium M running at 1.4 GHz with 512 MB of RAM. We have run various benchmarks that can be downloaded from the same site as ToolBar. The benchmarks included the academic problems and random generated problems. A subset of results is presented in Table 1.

The results show that the number of visited nodes is usually smaller, in many cases significantly smaller. However, there are problems for which the number of nodes is worse than in EDAC*, for example for few celar and jnh problems.

Unfortunately, the running time is often worse. The running time is better for some groups of problems, for example the large warehouse problems, and many jnh, dimacs and random k-SAT problems. We believe that the running time can be improved by better implementation and by avoiding useless computation.

The improvement of running time of LFAC* in comparison with EDAC* is often caused by the fact that LFAC* finds an initial feasible complete assignment with lower cost. The cost of the initial complete assignment is used as an upper bound of the cost of the solution. The lower upper bound, the more pruning can be done.

However, the work per node is greater for LFAC*. This is the reason why the running time is often worse for LFAC* even if the number of nodes is smaller.

	EDAC*		LFAC*		Ratio (%)	
Problem	**Nodes**	**Time**	**Nodes**	**Time**	**Nodes**	**Time**
sparse loose	132175	19.72	19208	20.76	14.53	105.29
sparse tight	7993	0.84	1973	1.46	24.69	175.11
dense loose	14706	1.48	5900	4.07	40.12	274.60
dense tight	22024	2.62	4740	4.81	21.52	183.90
complete loose	191739	25.94	124800	100.78	65.09	388.49
complete tight	156588	19.15	69976	56.58	44.69	295.49
capmo1	16760	125.63	6073	72.40	36.24	57.63
capmo2	11795	39.71	2662	23.43	22.57	59.00
capmo3	8357	51.18	3328	47.47	39.82	92.75
capmo4	6202	19.54	2388	20.15	38.50	103.12
capmo5	6148	20.05	3433	18.90	55.84	94.26

Table 1. Some results of LFAC* in comparison with EDAC* run until optimality is proven. Time is in seconds and Nodes illustrate the number of visited nodes during search. Ratio is the performance of LFAC* in comparison with EDAC* (EDAC* = 100%). The first group of results are the average times and the number of nodes for a group of random generated problems with a given density and tightness. The second group are the results of quite large warehouse problems.

Moreover, the value selection heuristic may choose different value because of different unary costs and thus different paths may be chosen during the search.

5 Conclusions

In this paper, we have introduced a new stronger form of local consistency for WCSP called limited full arc consistency (LFAC*). We provided an algorithm for enforcing LFAC* and proved its correctness.

The experimental results show that the number of nodes is usually smaller for LFAC*. Unfortunately, the running time of LFAC* algorithm is frequently worse than EDAC*. We believe that the running time can be improved by a better implementation. Still, the LFAC* algorithm is more effective than EDAC* for solving large warehouse problems and many other problems.

References

1. R. Dechter. *Constraint Processing.* Morgan Kaufmann Publishers, 2003.
2. E. C. Freuder, R. J. Wallace. *Partial Constraint Satisfaction.* Artificial Intelligence, Volume 58, pages 21–70, 1992.
3. S. de Givry, F. Heras, M. Zytnicki, J. Larrosa. *Existential arc consistency: Getting closer to full arc consistency in weighted CSPs.* In Proceedings of IJCAI–05, Edinburgh, Scotland, 2005.
4. F. Heras, J. Larrosa, S. de Givry, T. Schiex, E. Rollon. *ToolBar* http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/SoftCSP
5. J. Larrosa, T. Schiex. *In the quest of the best form of local consistency for Weighted CSP.* In Proceedings of IJCAI–03, Acapulco, Mexico, 2003.
6. J. Larrosa, T. Schiex. *Solving Weighted CSP by Maintaining Arc Consistency.* Artificial Intelligence, Volume 159 (1–2), 1–26, November 2004.

Overview of an Open Constraint Library

student: Marco Correia, supervisor: Pedro Barahona

Centro de Inteligência Artificial, Departamento de Informática,
Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
{mvc,pb}@di.fct.unl.pt

Abstract. Constraint programming is a flexible framework for addressing a broad class of problems [10]. Advances on this field include the introduction, improvement and specialization of inference and search algorithms for specific domains or applications. We argue that a software implementation that dynamically sustains the active development in this area is necessary but still lacking. Existing solutions [7,4] are confined to a (extensive) set of techniques and disallow or penalize extensions except for a limited set of parameters, often too restrictive in a research context. In this paper we point out some of these limitations and report ongoing work to circumvent them by using an emerging design paradigm named generic programming [11], which has been successfully integrated into the C++ programming language [17,9].

1 Introduction

From a CP researcher perspective, current available solvers allow a rather limited set of extensions. It is usually not allowed, or not an easy task, to introduce new domain type reasoning in an efficient way, using the available solver primitives. Recent developments in algorithm hybridization requires a level of control which is not present in most solvers. In this paper we try to identify some of the dependences that are the cause of such inflexibility, and simultaneously to point out possible solutions, by recurring to strong compile time parametrization, which is the basis of the generic programming paradigm. This work is partially related to [16], which is using generic programming to speed up constraint propagation, although diverging in some important design commitments.

This paper is organized as follows. In section 2 we justify the need for a highly customizable constraint library. In section 3 we present an outline of CaSPER, an architecture based on generic programming for achieving such goal. Finally section 4 concludes by pointing open problems and future directions.

2 Motivation

A constraint library is composed by a set of deeply interconnected components. Propagation, event communication and state management must cooperate very closely and therefore they are typically hard-coded together in monolithic kernels [4,7,14]. We identified at least four reasons that suggest the need for a modular architecture, in which the use of specialized versions of some of these components could potentially increase performance and extensibility.

1. Propagators acting on specific domains require particular information about domain updates. For example, most finite domain constraint propagators need only coarse or medium grained information (e.g. domain changed, bound changed), while finite set propagation is much more efficient with fine grained information (i.e. which values have been removed). The selection of the common denominator, a popular approach [2,7], penalizes performance in some domain specific models if too broad (in the example, finite domains), and forbids extension for other models if too restrictive.
2. Trailing is domain dependent. Common search algorithms need some sort of domain backup mechanism for backtracking purposes. Copying and restoring the entire domain at each update (element removal) should work for every domain type but it is not as efficient as recording only the modifications (trailing). Trailing requires support from the underlying engine, since it differs for domain representation and narrowing semantics. While only element removals are allowed in finite domains, updates in finite sets may involve also element insertions, while others require only bound update information [8].
3. A general propagator fixpoint mechanism is not optimal in all cases. Constraint propagation benefits from structured execution, where a preference function imposes an ordering in propagator execution. It is not clear if the same ordering suits all constraint domains and even all applications in the same domain (consider the case of modeling a global constraint with a set of local constraints for a domain where such global constraint does not exist, which actually occurs in [8]).
4. Distinct search methods require custom state handling. Search algorithms compromises important design decisions regarding the solver state handling method. State recomputation simplifies memory management but is less efficient for algorithms which explore solution space in depth-first order. State copying or trailing integrates well with this class of algorithms but might lead to memory explosion with other popular search methods (consider for example best first search). Recomputation, copying or a compromise between these two seems adequate for hybrid, parallel and modular search algorithms [15], while trailing is more suited for backtracking based methods.

3 Implementation

Creating a constraint library that is both modular and fast is an implementation challenge. We address this problem by using generic programming. Generic programming is about writing programs as generic as possible, without sacrificing efficiency. The key point in generic program design is the concept, a set of requirements that are general enough to cover a large family of abstractions but still allowing efficient processing of each member of the family. Software libraries naturally benefit from these techniques, providing them a set of software components that can be efficiently reused and extended in a wide variety of situations.

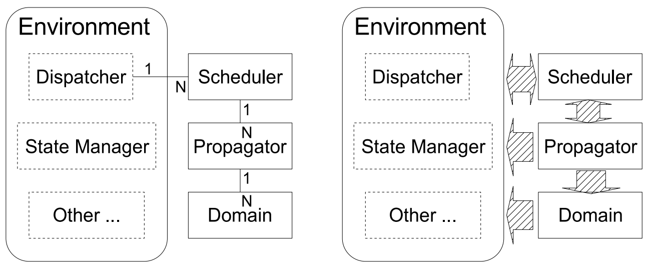


Fig. 1. CaSPER architecture: (left) entity-relationship diagram (right) component interaction.

Fig. 1 (left) shows CaSPER architecture. The several modules were carefully selected to address the issues enumerated in the previous section. In the rest of this section we present relevant implementation and communication details for each module, and cross-link them with the issues enumerated above.

3.1 Components revisited

The core of distinct state-of-the-art object oriented constraint libraries [3,6,12] typically agree on a set of objects, or components, which are the direct realization of the conceptual model. Variables, domains, constraints, predicates, etc, as first-class language entities in these libraries. These objects usually define an interface which may be exploited to introduce library extensions, or to solve specific problems. By using generic programming we are able to relax interfaces since most type checking is done at compile time. This new way of type resolution opens a new set of possibilities and simultaneously provides different perspectives on how these objects communicate.

Logical variables In CaSPER, logical variables are generic with respect to its domain type. More specifically, the type of the domain is statically available to the variable and in turn to all other objects that have access to it (instead of abstract pointers like [7]). This lets important functionality such as state-aware variable unification immediately available for any type. Interesting particular cases are logical variables with language built-in types such as int, float, etc, which are strongly typed cousins of prolog mutable terms.

Predicates Predicates state relationships among terms. As with variables, CaSPER predicates carry type information, in this case the type of the terms. This has several implications:

composition Static type nesting, allowing predicates such as $neg(fd = fd)$, and particularly useful for meta-predicates (e.g. $gac(fd+fd = fd)$, $bac(fd = fd)$), combining constraint declarative syntax with propagator selection (respectively generalized arc consistency and bounds arc consistency).

operator overloading Predicates may be overloaded like functions or methods. This avoids symbolic collision of predicates with the same name but with different semantics (e.g. finite domain addition $fd + fd = fd$ and finite set union $fdset + fdset = fdset$).

Propagators Propagators and goals specify how to enforce a given predicate. They are lazy-evaluated, which means that its execution is postponed from creation until some condition is met. In CaSPER, we followed the object oriented approach to address this problem [13,6,3] which consists in representing a propagator as a class. The propagator local data is stored as class members and a set of methods implement posting, executing and entailment checking.

Example 1. Unidirectional bound consistent propagator for equality constraint

```
bacd1(eq(var(fd),var(fd) ) ) : propagator {
  eq(var(fd),var(fd) ) p;
  void post(Sched) { Sched.suspend( chmin(p.p2) or chmax(p.p2), this); }
  bool entailed() {return p.p1.ground() and p.p2.ground() and p.p1 == p.p2;}
  bool execute() {
    p.p1.update_min(p.p2.min());
    p.p1.update_max(p.p2.max());
    return !p.p1.empty();
  }
};
```

In the example, the type of the predicate being enforced, and subsequently the type of variable domains, is known to the execute method (no casting is necessary). This allows aggressive compiler optimizations inside the most intensive (propagation) code. In this example propagation directly modifies variable domains, but the use of views (static indirection) such as proposed by [16] is also possible. The post method allows to reuse and combine existing propagators, an approach related to constraint handling rules [1]:

Example 2. Bound consistent propagator for equality constraint

```
bac(eq(var(fd),var(fd) ) ) : propagator {
  eq(var(fd),var(fd) ) p;
  void post(Sched)
  { Sched.post( bacd1(p.p1==p.p2) and bacd1(p.p2==p.p1) ); }
};
```

Goals Goals also enforce a given predicate, but relying on search rather than inference. Goals are the means to express non-determinism, currently by using Andorra style disjunction [5].

Schedulers Once created, propagators and goals are stored in objects called schedulers. Schedulers define an execution policy for their members, thus implementing common fixpoint algorithms. Schedulers are related to constraint combinators [15] since they are also propagators itself, and therefore suitable for nested composition. This feature may be used for implementing generalized negation and reification, partially completed in CaSPER.

Propagation schedulers implement batch and event-driven execution of suspended propagators. Combination of different propagation schedulers allow definition of complex fixpoint propagation trees (issue #3).

For general search, a stack based scheduler similar to [13,6] is available. A technique related to those used in [3,7] fits naturally to this scheduler for common search tree algorithms. The hybridization of more evolved search algorithms will hopefully benefit from composition of search schedulers, but this direction was not yet explored.

Environment The environment object connects all components of the constraint library. It is itself composed of an unlimited list of autonomous modules, namely an event dispatcher, a trail, a statistics module, garbage collection, a debugger, etc... Modules are independent from the rest of library objects, which permits different combinations to be created in a plugin-like architecture. Important components are:

dispatcher Responsible for storing, sorting and delivering events to suspended schedulers. Dispatchers widely used are LIFO, FIFO, and priority queues. The selection of the dispatcher will obviously interfere with the fixpoint propagation algorithm (issue #3).

state handler Manage choice points by recording and restoring changes to domains and other reversible data structures. Common state handlers are trailing and copying, and its selection per problem is possible (issues #2 and #4).

garbage collection Most generic programming based libraries [9,17] decouple memory model from algorithms. This permits the selection of the most efficient model for each situation, for example multi-threaded, memory limited (prolog like), traceable or dynamic memory based applications.

3.2 Operation

Fig. 1 (right) shows CaSPER operational diagram. Communication among propagators is done mostly through event passing, where an event is a user defined data structure with information about a domain update (issue #1). Distinct events may coexist in an environment but each scheduler only handles one event type.

4 Conclusion

The work described is still in progress. Current implementation solves the above mentioned problems, but it is rather incomplete for a more extensive benchmark (mostly because it lacks important global constraints and modelling primitives).

By taking advantage of individual search schedulers, high level predicates are being developed for modeling. These include imperative style constructs such as loops and conditionals. This was not discussed in this paper.

Open problems are the realization of efficient structured propagation (without unnecessary event duplication) and the composition of search schedulers. The latter might benefit from the notion of search space, an idea originally presented in [15]. Potential applications for this solver includes hybridization with other constraint paradigms, namely SAT.

References

1. Gregory J. Duck, Peter J. Stuckey, Maria J. García de la Banda, and Christian Holzbaur. Extending arbitrary solvers with constraint handling rules. In *PPDP*, pages 79–90. ACM, 2003.
2. Abderrahamane Aggoun et al. ECLiPSe 3.5 user manual, September 19 1995.
3. Simon De Givry. ToOLS: A library for partial and hybrid search methods, April 29 2003.
4. Laurent Granvilliers and Eric Monfroy. Composition operators for constraint propagation: An application to choco. *j-LECT-NOTES-COMP-SCI*, 2239:600–??, 2001.
5. Seif Haridi and Per Brand. ANDORRA prolog - an integration of prolog and committed choice languages. In *Proc. Int. Conf. on 5th gen. Computer Systems 1988*, 1988.
6. Martin Henz, Tobias Müller, and Ka Boon Ng. Figaro: Yet another constraint programming library. *Electr. Notes Theor. Comput. Sci*, 30(3), 1999.
7. ILOG. *ILOG Solver 5.1 User's Manual*. ILOG s.a. `http://www.ilog.com`, 2001.
8. Ludwig Krippahl and Pedro Barahona. Psico: Solving protein structures with constraint programming and optimization. *Constraints*, 7(3-4):317–331, 2002.
9. Lie-Quan Lee, Jeremy G. Siek, and Andrew Lumsdaine. The generic graph component library. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 399–414, 1999.
10. A. Mackworth and E. Freuder. The complexity of some polinomial network constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.
11. David R. Musser and Alexander A. Stepanov. Generic programming. In *ISSAC*, pages 13–25, 1988.
12. Jean-Francois Puget. A C++ implementation of CLP. In *Proceedings of the Second Singapore International Conference on Intelligent Systems*, Singapore, 1994.
13. Jean-Francois Puget and Michel Leconte. Beyond the glass box: Constraints as objects. In *ILPS*, pages 513–527, 1995.
14. Peter Van Roy. Announcing the mozart programming system. *SIGPLAN Notices*, 34(4):33–34, 1999.
15. Christian Schulte. *Programming Constraint Services: High-Level Programming of Standard and New Constraint Services*, volume 2302. Springer, 2002.
16. Christian Schulte and Peter J. Stuckey. Speeding up constraint propagation. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 619–633. Springer, 2004.
17. A. A. Stepanov and M. Lee. The Standard Template Library. Technical Report X3J16/94-0095, WG21/N0482, 1994.

A Hybrid Formulation for CSP

Student : Tony Lambert[1,2]
Supervisors: Eric Monfroy[1,3] and Frédéric Saubion[2]

[1] LINA, Université de Nantes, France (Tony.Lambert@lina.univ-nantes.fr)
[2] LERIA, Université d'Angers, France (Frederic.Saubion@univ-angers.fr)
[3] Universidad Santa María, Valparaíso, Chile (Eric.Monfroy@inf.utfsm.cl)

1 Introduction

Many problems in computer science can be formalized using the Constraint Satisfaction Problems (CSP) model. A CSP, usually defined by a set of variables associated to domains of possible values and by a set of constraints that limits the combinations of allowed values, can be solved by a large variety of algorithms basically design w.r.t. the problems parameters. Many of these problems are also associated to an optimization criterion.

To solve such problems, complete methods aim at exploring the whole search space in order to find all the solutions or to detect that the CSP is not consistent. We are here mainly concerned with constraint propagation with split of domains [3]. Other methods like Local search [1] or genetic algorithms aims at revealing patterns, amplifying, discovering hid informations. From a general point of view, the main idea is to control a set of individuals acting in concert, by containing fluctuations generated by both an exploration, scattering search trails and a restriction, breaking deadlocks.

Hybrid applications using combination of resolution paradigms and techniques have been studied (e.g., [4] presents an overview of possible uses of local search in constraint programming that presume endless possibilities for search hybridization). In this paper we are mainly concerned by a uniform formulation of such possibilities, realized thanks to an abstraction of the basic components involved.

2 Constraint Satisfaction Problems

We first recall basic notions related to CSP and their main resolution approaches.

A CSP is a tuple (X, D, C) where $X = \{x_1, \cdots, x_n\}$ is a set of variables taking their values in their respective domains $D = \{D_1, \cdots, D_n\}$. A constraint $c \in C$ is a relation $c \subseteq D_1 \times \cdots \times D_n$.

In order to simplify notations, D will also denote the Cartesian product of D_i and C the union of its constraints.

We describe now different approaches to solve CSP.

A tuple $d \in D$ is a solution of a CSP (X, D, C) if and only if $\forall c \in C, d \in c$.

A CSP (X, D, C) is solution if and only if $\forall d \in D, \forall c \in C, d \in c$.

A constraint optimization problem consists of an objective function f and a CSP (X, D, C). Solving such a problem consist in finding a feasible solution s (i.e. a solution of (X, D, C)) such that $f(s)$ is optimal.

Constraint propagation, one of the most famous techniques for solving CSP consists in iteratively reducing domains of variables by removing values that do not satisfy the constraints. However, reduction mechanisms use one or some of the constraints of the CSP. Thus, they enforce a local consistency property (such as arc consistency) but not a global consistency of the CSP. These reductions must be interleaved with a splitting mechanism (such as enumeration) in order to obtain a complete solver, (i.e., a solver which returns only solutions and does not loose any solution).

Given an optimization problem (which can be minimizing the number of violated constraints and thus trying to find a solution of the CSP), local search techniques [1] aim at exploring the search space, moving from a configuration to one of its neighbors. These moves are guided by a fitness function which evaluates the benefit of such a move in order to reach a local optimum. For the resolution of a CSP (X, D, C), the search space can be usually defined as the set of possible tuples of D, The fitness (or evaluation) function *eval* is related to the notion of solution and can be defined as the number of constraints c such that $t \notin c$ (t being a tuple from D). In this case, the problem to solve is indeed a minimization problem.

Evolutionary algorithms are mainly based on the notion of adaptation of a population of individuals to a criterion using evolution operators like crossover. Based on the principle of natural selection, *Genetic Algorithms* [6] have been quite successfully applied to combinatorial problems such as scheduling or transportation problems. The key principle of this approach states that, species evolve through adaptations to a changing environment and that the gained knowledge is embedded in the structure of the population and its members, encoded in their chromosomes. If individuals are considered as potential solutions to a given problem, applying a genetic algorithm consists in generating better and better individuals w.r.t. the problem by selecting, crossing, and mutating them. This approach reveals very useful for problems with huge search spaces. In the context of GA, for the resolution of a given CSP (X, D, C), the search space can be usually defined with the set of tuples D. We consider populations g of size i, $g \subseteq D$ such as $|g| = i$. An element $s \in g$ is an individual and represents a potential solution to the problem. Fitness functions provide information about the quality of an individual and so, of a population. Thus, these functions have to handle both the constraints of the problem and the optimization criterion.

3 A Hybrid System for CSP

K.R. Apt proposed in [2, 3] a general theoretical framework for modeling such reduction operators. In this context, domain reduction corresponds to the computation of a fixpoint of a set of functions over a partially ordered set. These functions, called *reduction functions*, abstract the notion of constraint. We have extended this framework in order to include local search and genetic algorithms components [7].

We propose here a new formulation of a general uniform framework, that makes the modelization of hybrid algorithm simpler. As mentioned above, the key idea of this system is to decompose each resolution process into basic function, extending the work of K.R. Apt to metaheuristics resolution techniques. Then, these function can be managed at the same level and the resolution process can be achieved by the generic algorithm proposed in [2]. From our system point of view, this resolution process can be described as sequence of transitions over a computational structure.

3.1 Construction of a partial ordering

By a partial ordering we mean a pair $(D, \sqsubseteq)$ consisting of a set D and a reflexive, antisymmetric and transitive relation $\sqsubseteq$ on D. Given a partial ordering $(D, \sqsubseteq)$ and element d of D is called the least element if $d \sqsubseteq e$ for all $e \in D$.

Given a set D, $\mathcal{P}(D)$ denotes the set of all subsets of D. $(\mathcal{P}(D), \supseteq)$ is a partial ordering, where $\supseteq$ is the reversed subset relation.

Definition 1. *Consider the Cartesian product $\mathcal{P}(D_1) \times \cdots \times \mathcal{P}(D_n)$ with elements ordered according to the reversed subset relation $\supseteq$. such that : $(X_1, \ldots, X_n) \supseteq (Y_1, \ldots, Y_n)$ iff $X_i \supseteq Y_i$ for all $i \in [1..n]$ s.t. $X_i, Y_i \in \mathcal{P}(D_i)$ This yields the partial ordering : $(\mathcal{P}(D_1) \times \cdots \times \mathcal{P}(D_n), \supseteq)$*

Definition 2. *To extend this relation to CSPs; we built an order on $\langle X, C, \mathcal{P}(D_1) \times \cdots \times \mathcal{P}(D_n)\rangle$ the set of CSPs with a set of variables X , a set of constraints C and $\mathcal{P}(D_1) \times \cdots \times \mathcal{P}(D_n)$ the search space , meaning all possible CSPs reachable for the initial CSP to solve $\langle X, C, D_1 \times \cdots \times D_n\rangle$.*

The pair $(\langle X, C, \mathcal{P}(D_1) \times \cdots \times \mathcal{P}(D_n)\rangle, \sqsubseteq)$ forms a partial ordering, relation $\sqsubseteq$ being defined on the reversed set relation by the last component of the CSP triplet (corresponding to domains).

Definition 3. *Consider the set $\mathcal{P}(\mathcal{P}(D_1) \times \cdots \times \mathcal{P}(D_n)$ of possible subset of $\mathcal{P}(D_1) \times \cdots \times \mathcal{P}(D_n)$. Consider relation $\sqsubseteq$ on it defined as :*

Given two set of CSPs Φ and Ψ members of the set $\mathcal{P}(\mathcal{P}(D_1) \times \cdots \times \mathcal{P}(D_n)$ with $\Phi = \{\phi_1, \ldots, \phi_k\}$ and $\Psi = \{\psi_1, \ldots, \psi_l\}$. The couple $(\Phi, \Psi) \in \sqsubseteq$ (i.e. $\Phi \sqsubseteq \Psi$) iff :

1. *$\forall \phi_i \in \Phi$:*
   - *(a1) either exists $\psi_j \in \Psi$ s.t. $\psi_j = \phi_i$*
   - *(a2) or exists $\psi_{j_1} \ldots, \psi_{j_h} \in \Psi$ s.t. $Sol(\phi_i) \subseteq \bigcup_{k=1..h} Sol(\psi_{j_k})$.*
2. *(b) and , $\forall \psi_j \in \Psi\ \exists \phi_i \in \Phi$ s.t. $\phi_i \sqsubseteq \psi_j$*

Property 1. This relation over CSPs forms a semi-ordering.

3.2 Domain reduction functions

The computation of the least common fixpoint of a set of functions F can be achieved by the Generic Iteration algorithm (GI) from [3], described in Figure 1. In the GI algorithm, G represents the current set of functions still to be applied ($G \subseteq F$), d is a partially ordered set (the domains in case of CSP).

GI: Generic Iteration Algorithm

```
$d := \perp$;
$G := F$;
While $G \neq \emptyset$ do
        choose $g \in G$;
        $G := G - \{g\}$;
        $G := G \cup update(G, g, d)$;
        $d := g(d)$;
endwhile
```

where for all G, g, d, the set of functions $update(G, g, d)$ from F is such that:

- $\{f \in F - G \mid f(d) = d \wedge f(g(d)) \neq g(d)\} \subseteq update(G, g, d)$.
- $g(d) = d$ implies that $update(G, g, d) = \emptyset$.
- $g(g(d)) \neq g(d)$ implies that $g \in update(G, g, d)$

Fig. 1. The Generic Iteration Algorithm

Suppose that all functions in F are inflationary ($x \sqsubseteq f(x)$ for all x) and monotonic ($x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$ for all x, y) and that $(D, \sqsubseteq)$ is finite. Then, every execution of the **GI** algorithm terminates and computes in d the least common fixpoint of the functions from F (see [2]).

Note that in the following we consider only partial orderings.

3.3 Sampling

Sampling consists in extracting from a CSP a complete or partial assignment. The sample is then considered as a new CSP added to the set :

$$\begin{aligned} \mathcal{S} : \mathcal{P}(\langle X, C, \mathcal{P}(D_1) \times \cdots \times \mathcal{P}(D_n)\rangle) &\to \mathcal{P}(\langle X, C, \mathcal{P}(D_1) \times \cdots \times \mathcal{P}(D_n)\rangle) \\ \{\phi_1, \ldots, \phi_n\} &\mapsto \{\phi_1, \ldots, \phi_n, \phi_{n+1}\} \end{aligned}$$

s.t. $\exists \phi_i$ with $\phi_i \sqsubseteq \phi_{n+1}$

We will consider a complete assignment if : $\phi_{n+1} \equiv \langle X_{n+1}; C_{n+1}; D_{n+1}\rangle$ with $D_{n+1} = D_{n+1_1}, \ldots, D_{n+1_k}$ and $\forall i \in [1..k], |D_{n+1_i}| = 1$. Or partial if only some variables are assigned : $\phi_{n+1} \equiv \langle X_{n+1}; C_{n+1}; D_{n+1}\rangle$ with $D_{n+1} = D_{n+1_1}, \ldots, D_{n+1_k}$ and $\exists i \in [1..k] tq |D_{n+1_i}| = 1$

Property 2. **Sampling is inflationary** $x \sqsubseteq \mathcal{S}(x)$
$\{\phi_1, \ldots, \phi_n\} \sqsubseteq \{\phi_1, \ldots, \phi_n, \phi_{n+1}\}$, the proof is straightforward.

Property 3. **Sampling is monotonic** $x \sqsubseteq y$ **implies** $\mathcal{S}(x) \sqsubseteq \mathcal{S}(y)$: $\Phi \sqsubseteq \Psi \longrightarrow \mathcal{S}(\Phi) \sqsubseteq \mathcal{S}(\Psi)$ with $\Phi = \{\phi_1, \ldots, \phi_n\}$ and $\Psi = \{\psi_1, \ldots, \psi_m\}$ we have : $\{\phi_1, \ldots, \phi_n\} \sqsubseteq \{\psi_1, \ldots, \psi_m\} \longrightarrow \{\phi_1, \ldots, \phi_n, \phi_{n+1}\} \sqsubseteq \{\psi_1, \ldots, \psi_m, \psi_{m+1}\}$, the proof is straightforward.

3.4 Reducing

Reduce the search space is essential to reach a solution with a complete approach, in the context of CSP it is translated in deleting values from domains and thus to be sure not losing solutions.

$$\begin{aligned} \mathcal{R} : \mathcal{P}(\langle X, C, \mathcal{P}(D_1) \times \cdots \times \mathcal{P}(D_n)\rangle) &\rightarrow \mathcal{P}(\langle X, C, \mathcal{P}(D_1) \times \cdots \times \mathcal{P}(D_n)\rangle) \\ \{\phi_1, \ldots, \phi_i, \ldots, \phi_n\} &\mapsto \{\phi_1, \ldots, \phi_i', \ldots, \phi_n\} \end{aligned}$$

Where $\phi_i' = \emptyset$ or $\phi = \langle X, C, D_i\rangle$ and $\phi' = \langle X, C, D_i'\rangle$ s.t. $D_i' \subseteq D_i$.

4 Reduction functions

Therefore, we consider a structure : distinguish several transition rules :

- **Domain reduction** (DR) correspond to node consistency , arc consistency and hyper-arc consistency.

  $$\{\phi_1, \ldots, \phi_i, \ldots, \phi_n\} \rightarrow^{DR} \{\phi_1, \ldots, \phi_i', \ldots, \phi_n\}$$

  Where $DR = \mathcal{R}^m$ with $m > 0$.
- **Split** (SP) is the result of a domain shattered, and is formalized for a domain of size m as m incomplete samples generated and by the reduction (deletion) of the original CSP, solutions being dealt.

  $$\{\phi_1, \ldots, \phi_i, \ldots, \phi_n\} \rightarrow^{SP} \{\phi_1, \ldots, \phi_i^1, \ldots, \phi_i^m, \ldots, \phi_n\}$$

  Where $SP = \mathcal{S}^m\mathcal{R}$.
- **Local Search** (LS) step aims at generating new samples and moving to a chosen neighbor and formally corresponds to m sample (generate neighborhood) followed by m reduction (sélect a neighbor).

  $$\{\phi_1, \ldots, \phi_i, \ldots, \phi_n\} \rightarrow^{LS} \{\phi_1, \ldots, \phi_i', \ldots, \phi_n\}$$

  Where $LS = \mathcal{S}^m\mathcal{R}^m$.

– **Evolution** can be divided into three movements : crossover CR is a simple sample, mutation MU is a sample followed by a reduction and selection SE (sub population) is a s times reduction.

$$\{\phi_1, \ldots, \phi_n\} \rightarrow^{CR} \{\phi_1, \ldots, \phi_n, \phi_{n+1}\}$$

Where $CR = \mathcal{S}$.

$$\{\phi_1, \ldots, \phi_i, \ldots, \phi_n\} \rightarrow^{MU} \{\phi_1, \ldots, \phi'_i, \ldots, \phi_n\}$$

Where $MU = \mathcal{SR}$.

$$\{\phi_1, \ldots, \phi_n\} \rightarrow^{SE} \{\phi_1, \ldots, \phi'_n\}$$

Where $SE = \mathcal{R}^s$.

Therefore, a resolution is a finite sequence starting from an initial problem $\langle X, C, D_0 \rangle$ and providing as final state $\langle X, C, D^1_n \rangle, \cdots, \langle X, C, D^k_n \rangle$ by applying previously described transition rules. . The selection corresponds to the usual selection process used in genetic algorithms to select from a given population the next generation. The evolution transition allow us to apply mutation and crossover operators. In this context, a strategy is a sequence $(t_i)_{1 \leq i \leq n}$ where $\forall 1 \leq i \leq n, t_i \in \{DR, SP, LS, SE, CR, MU\}$. Our purpose is to study, at a fine grain level, the hybridization of these transition rules over various CSP and optimization problems.

5 Conclusion

Hybrids techniques enable us to reach a high level of efficiency for solving complex combinatorial and optimisation problems, in this paper we present a suitable general framework to model hybrid solving algorithms. We have shown that this work can serve as a basis for formulation of an integration of GA, LS and CP methods with their main properties. This framework will provide a uniform background to classify, compare, analyse, describe and control hybrid algorithms at the lowest level.

References

1. E. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization.* John Wiley & Sons, Inc., 1997.
2. K. R. Apt. From chaotic iteration to constraint propagation. In *Proceedings of ICALP'97*, pages 36–55. Springer-Verlag, 1997.
3. K. R. Apt. *Principles of Constraint Programming.* Cambridge Univ. Press, 2003.
4. F. Focacci, F. Laburthe, and A. Lodi. Local search and constraint programming. In *Handbook of Metaheuristics*, Kluwer Academic, 2002.
5. I. Gent, T. Walsh, and B. Selman. http://www.4c.ucc.ie/ tw/csplib/, funded by the UK Network of Constraints.
6. J. H. Holland. *Adaptation in Natural and Artificial Systems.* 1975.
7. E. Monfroy, F. Saubion and T. Lambert. Hybrid CSP Solving. *In Proceedings of FROCOS 2005 , LNCS 3717, Springer Verlag, 2005.*

A CP-based column generation approach to scheduling for Wireless Mesh Networks

Student: Stefano Gualandi
Supervisor: Federico Malucelli

Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
{gualandi,malucell}@elet.polimi.it

Abstract. This paper presents ongoing work on the application of Constraint Programming based column generation to the optimum integrated link scheduling and power control problem arising in Wireless Mesh Networks. In a Wireless Mesh Network, all nodes are equipped with identical half-duplex transceiver and omni-directional antennas. Direct communication links can be established between couples of transmitter/receiver nodes if radio interference at the receiver is below a given threshold. Radio interference depends on the ratio between the power of the useful signal and the overall received signals. The integrated link scheduling and power control problem consists in assigning communication links to time slots in such a way to satisfy the traffic demands while avoiding radio interference. This scheduling problem has a Mixed Integer Programming formulation. We propose a hybrid column generation decomposition where the sub-problem (pricing) is solved using Constraint Programming. Numerical results demonstrate that the proposed approach outperforms a traditional column generation approach.

1 Introduction

Wireless Mesh Networks are composed of nodes communicating through omni-directional antennas which must share the spectrum. Each node can establish a communication link at a time where it can be either *transmitter* or *receiver*. Wireless Mesh Networks are more complicated than wired networks, since during radio communication every client may interfere with other communications. Several communication protocols exist trying to avoid interference while assuring the Quality of Service, but this paper focuses on the Time Division Multiple Access (TDMA) protocol. In this protocol, the time is discretized into fixed length slots that are organized cyclically. During each time slot, a transmitter node can establish a communication link with a receiver node, if the interference at the receiver is below a given threshold. The interference strongly depends on the power used by nodes transmitting simultaneously. Using high level of power makes the communication reliable, but increases both interference and energy consumption. Therefore, the goal of the TDMA protocol is to assign communication links to time slots while avoiding interference, and to minimize the number of time slots required to achieve all packet transmissions. Several heuristic algorithms exist in literature that solve this problem, but they either consider fixed power or use a simple model of interference. A formal definition of the problem using the *Signal-to-Interference and Noise Ratio* (SINR) model and considering variable levels of power is given in [1]. The authors call their formulation the Integrated Link Scheduling and Power control (ILSP) problem, and propose a Mixed Integer Programming (MIP) model with quadratic constraints. By reduction to the edge coloring problem, they prove ILSP problem to be NP-complete.

This paper presents a Constraint Programming-based column generation approach for the Integrated Link Scheduling and Power control problem. The CP-based column generation framework was introduced by [2], and it has found a number of successful applications in airline crew assignment [3] and employee time-tabling [4]. In our application, the use of Constraint Programming is motivated by the tightness of the constraints modeling the physical interference. This tightness was observed by tackling the problem with a traditional column generation approach. By solving the column generation sub-problems with a Constraint Programming Solver, we can better exploit the problem structure comparing with a traditional column generation approach which solves a complicated linearized sub-problem at each iteration. This allow us to improve the running performance of one order of magnitude.

The outline of this paper is as follow. Next section presents the formal model of the ILPS problem. Section 3 describes the CP-based column generation approach used to tackle this problem. Section 4 gives implementation details and presents numerical results. Finally, we conclude with discussions on future work.

2 Problem description

A Wireless Mesh Network topology with n clients is formalised with a directed graph (V, E), where for every client i there is a node in V, i.e. $|V| = n$, and for every communication link (i, j) there is an edge in E. The number of packets each transmitter i has to send to a receiver j is given by coefficients R_{ij} of an asymmetric matrix R. The physical model used in this paper for the interference is the *Signal-to-Interference and Noise Ratio* (SINR) [1]: a communication link (i, j) can be established only if the SINR is equal or greater than a given threshold γ:

$$\text{SINR}_j = \frac{G_{ij} p_{ij}}{\eta_j + \sum_{(l,m)\in E(i,j)} G_{lj} p_{lm}} \geq \gamma \qquad (1)$$

whereby, p_{ij} is the transmit power of link (i, j) and ranges in $[0..P_{max}]$; G_{ij} is the propagation gain which is inversely proportional to the distance between nodes i and j; η_j is the thermal noise at receiver j. The summation in the denominator models the aggregate interference at the receiver given by all the other links (l, m) transmitting at the same time with link (i, j). The Integrated Link Scheduling and Power control consists in finding the minimal number of time slots which satisfy the traffic demand given by matrix R, while considering that during a time slot can communicate only links which satisfy their SINR threshold.

3 CP-based column generation

The main idea of column generation approaches is to exploit problem sub-structures [5]: the original problem is decomposed into so called *master* and *pricing* problems which reflect different sub-structures (families of constraints). The *master* problem has usually a compact MILP formulation which can have an exponential number of variables. The variables of the master are associated to feasible solutions (also called patterns or *columns*) of the real problem. The trick is to start with an initial feasible sub-set of patterns and solve the associated continuous relaxation of the *restricted* master problem to optimality. The solution thus obtained is an upper bound (in case of a minimization problem; it is lower bound otherwise) of the object value of the master. To improve the objective value, it is necessary to introduce new variables. The question is: which and how many variables to introduce next? To answer this question, the *pricing* problem comes to play: it looks for a feasible pattern with negative reduced costs, that is, a pattern having the corresponding dual constraint violated. Therefore, the pricing is built using the dual values associated to the optimal solution of the continuous relaxation of the restricted master problem. Linear Programming (LP) duality ensures that the solution of the restricted master problem cannot be improved if no variables with strictly negative reduced cost exist.

For the ILSP control problem under investigation, we propose the following decomposition: each variable of the master problem denotes a feasible pattern, that is, a time slot where the active communication links do not interfere with each other. For each pattern s there is a non negative integer variable λ_s giving the number of times the corresponding patterns is used. The set of all the possible patterns is called S, while the set of patterns where a link (i,j) is active is denoted S_{ij}. The problem of finding which patterns and how many times are used to satisfy the traffic demand R is formalised as follow:

$$\min \sum_{s \in S} \lambda_s \tag{2}$$

$$\text{s.t.} \sum_{s \in S_{ij}} \lambda_s \geq R_{ij} \quad \forall (i,j) \in E \tag{3}$$

$$\lambda_s \in \mathbb{N}^+ \quad \forall s \in S \tag{4}$$

The objective function (2) minimizes the number of time slots; constraints (3) state the traffic demand R_{ij} of each link (i,j); constraints (4) impose integrality. By omitting integrality in constraints (4) we get the linear continuous relaxation, which is solved using an LP solver. The initial subset of feasible patterns defining the restricted master problem is obtained using a greedy algorithm developed in [6] and out of the scope of this paper. The values σ_{ij} of dual variables for constraints (3), obtained by solving the LP relaxation of the restricted master problem, are used to build the *pricing* problem formulated as a Constraint Optimization Problem as follows:

$$\min\ (1 - \sum_{(i,j)\in E} \sigma_{ij} z_{ij}) \tag{5}$$

$$\text{s.t.}\ \sum_{(i,j)\in E} z_{ij} + \sum_{(j,i)\in E} z_{ji} \le 1 \qquad \forall i \in V \tag{6}$$

$$z_{ij} \Rightarrow \frac{G_{ij} p_{ij}}{(\eta + \sum_{(l,m)\in E: l\neq i} G_{lj} p_{lm})} \ge \gamma \qquad \forall (i,j) \in E \tag{7}$$

$$\neg z_{ij} \Rightarrow p_{ij} = 0 \qquad \forall (i,j) \in E \tag{8}$$

$$z_{ij} \in \{0,1\} \qquad \forall (i,j) \in E \tag{9}$$

$$p_{ij} \in \{0,1,\dots,P_{max}\} \qquad \forall (i,j) \in E \tag{10}$$

The objective function (5) computes the reduced cost of the pattern associated to variables z; from LP duality, if the reduced cost (the objective value) is non-negative, the corresponding restricted master problem cannot be further improved. Constraints (6) formalise that all nodes can transmit in uni-casting and half-duplex, and are single receiving: they can participate to a single transmission at a time, and can be either transmitter or receiver. Constraints (7) are the SINR derived constraints: if node i is transmitting to node j ($z_{ij} = 1$), then the SINR at receiver j must be greater than a threshold γ; otherwise, ($z_{ij} = 0$), the power p_{ij} is set to zero (8). The binary decision variable z_{ij} (9) indicates if client i is transmitting to j, and the decision variable p_{ij} (10) gives the level of power used by transmitter i. Column generation approaches work as follows: (i) the continuous relaxation of the restricted master problem with an initial feasible subset of patterns is solved to optimality; (ii) by using the values σ_{ij} of constraints (3) the pricing is built and solved to optimality; (iii) if the objective function value (5) is strictly negative, the pattern defined by variables z_{ij} is added to the restricted master problem and the algorithm returns to step (i); otherwise, if objective value (5) is positive, column generation stops, since the restricted master problem cannot be further improved. In traditional column generation, the LP relaxation of the master is solved using a linear solver, while the pricing is solved with an MILP solver (applied to a linearization of the pricing) or with dynamic programming algorithms. In our approach, following the idea proposed by [2], we use Constraint Programming to solve the pricing to optimality. The following section gives details of our implementation, and shows a comparison with a traditional approach using an MILP solver for the pricing.

4 Implementation and numerical results

The bottleneck of solving the ILSP problem defined in the master (2)–(4) and the pricing (5)–(10) with column generation is the pricing. In [6], the authors present a column generation approach using a MILP solver (CPLEX) for solving both master and pricing; they observed that most of CPU time (almost 99%) is spent in solving the pricing. Moreover, they observed that SINR constraints (7) are strongly reducing the set of feasible solutions, i.e., they are *tight* constraints. This motivates the use of Constraint Programming to solve the pricing. In our implementation, the continuous relaxation of the master is solved with an LP solver. Using the duals values, the pricing is built as a Constraint Optimization Problem and is solved with Gecode (www.gecode.org). The objective function (5) is translated into the following linear constraint:

$$\sum_{(i,j)\in E} \sigma_{ij} z_{ij} > y \tag{11}$$

whereby $y \in \{1, \ldots, y^{ub}\}$ is a finite domain integer variable, and y^{ub} is an upper bound on its value. Constraints (7) are slightly modified: since denominators are always strictly positive, they become implication of linear constraints. All the linear constraints are propagated using bound consistency. The labeling heuristic of variables z and p are: (i) choose first the variable with the higher number of dependent constraints; (ii) assign to the selected variable first the bigger value. The pricing is solved with depth first (CP) branch-and-bound. We have tried three strategies for solving the pricing and adding columns to the restricted master: *CP-opt* solves the pricing to optimality and adds only the best column; *CP-all* adds all the columns found during the branch-and-bound search; *CP-first* stops the search after the first feasible solution and adds the corresponding column.

To evaluate and compare CP-based and MILP-based column generation, we use instances of Wireless Mesh Networks provided by the authors of [6]. Those instances have nodes located into a square area of $100m^2$ with randomly generated traffic demand R. For each couple of nodes the traffic demand is strictly positive (i.e., the corresponding graph (V, E) is complete). The constant parameters are set to: $\eta = 10^{-6}$ mW, $P_{max} = 30$ mW, and $\gamma = 10$. Table 1 shows the numerical results of instances with 10 nodes and 20 nodes, having respectively 90 and 380 communication links (at the time of writing we are running more tests). The comparison of CPU time between the CP-based and MILP-based column generation is striking: in all the instances the first approach outperforms the latter of one order of magnitude. The two methods generate different optimal patterns for the pricing instances, and this justifies the different number of generated patterns.

5 Future work

In this paper, we have presented ongoing work on the application of CP-based column generation to the ILPS problem arising in Wireless Mesh Networks. To our knowledge, it is the first time this approach is used to tackle the ILPS problem, and the performance compared to a traditional column generation approach are striking. Many directions of investigation exist, and, so far, we are

				MILP-based		CP-opt		CP-all		CP-first	
Instance	$\bar{S}$	Obj	Opt	NC	Time	NC	Time	NC	Time	NC	Time
p_10_1	129	590	568	16	29	15	4.9	14	4.93	14	1.63
p_10_3	121	692	602	33	63	42	16.4	50	11.1	54	6.98
p_10_8	121	554	536	17	51	22	9.73	24	8.86	29	2.82
p_20_2	533	2436	2326	-	-	68	11579	77	1383	76	1532
p_20_3	533	2576	2479	-	-	44	7251	58	2192	57	2370
p_20_5	576	2497	2412	-	-	66	14615	83	1754	82	1869

Table 1. Results for random instances with 10 and 20 nodes with a time limit of 28800 seconds. The initial set of columns is $\bar{S}$; the solution of the continuous relaxation of the restricted master problem with $\bar{S}$ is *Obj*, while after column generation is *Opt*. *NC* is the number of columns added during column generation, and *Time* is given in seconds.

investigating possible continuous relaxations of the pricing in order to further improve the resolution time, as suggested in [7, 8]. Other directions of research under investigation are: first, the algorithms presented in [1] may be used as a basis for designing specialized (cost-based?) filtering algorithms for the ILPS problem, as in [7]; secondly, since the pricing problem is solved several times with identical constraints apart from the cost function, it could be modeled as Dynamic-CSP, and constraints could embed basic *learning* mechanisms in order to save propagation time (which is the dominating resolution time for the CP-based approach).

References

1. Behzad, A., Rubin, I.: Optimum integrated link scheduling and power control for ad hoc wireless networks. to appear in IEEE Transactions on Vehicular Technology (2006)
2. Junker, U., Karisch, S.E., Kohl, N., Vaaben, B., Fahle, T., Sellmann, M.: A framework for constraint programming based column generation. In: Proceedings of CP. (1999)
3. Fahle, T., Junker, U., Karisch, S.E., Kohl, N., Sellmann, M., Vaaben, B.: Constraint programming based column generation for crew assignment. J. Heuristics **8**(1) (2002) 59–81
4. Gendron, B., Lebbah, H., Pesant, G.: Improving the cooperation between the master problem and the subproblem in constraint programming based column generation. In: Proceedings of CPAIOR. (2005)
5. Wolsey, L.A.: Integer Programming. John Wiley & Sons, New York (1998)
6. Capone, A., Carello, G.: Scheduling optimization in wireless mesh networks with power control and rate adaptation. In: [Submitted to] - IEEE Communications Society Conference on Sensor, Mesh, and Ad Hoc Communications and Networks. (2006)
7. Sellmann, M.: Reduction Techniques in Constraint Programming and Combinatorial Optimization. PhD thesis, University of Paderborn (2003)
8. van Hoeve, W.J.: Operations Research Techniques in Constraint Programming. PhD thesis, University of Amsterdam (2005)

BlockSolve: a Bottom-Up Approach for Solving Quantified CSPs

Student: Guillaume Verger
Supervisor: Christian Bessiere

LIRMM, CNRS/University of Montpellier, France
{verger,bessiere}@lirmm.fr

Abstract. Thanks to its extended expressiveness, the quantified constraint satisfaction problem (QCSP) can be used to model problems in model checking or planning under uncertainty that cannot be expressed in the standard CSP formalism. This is only recently that the constraint community got interested in QCSPs and that algorithms to solve it were proposed. In this paper we propose `BlockSolve`, an algorithm for solving QCSPs that factorizes computations made in branches of the search tree. Instead of following the order of the variables in the quantification sequence, our technique searches for combinations of values for existential variables at the bottom of the tree that will work for values of variables earlier in the sequence. An experimental study shows the good performance of `BlockSolve` compared to a state of the art QCSP solver.

1 Introduction

The quantified constraint satisfaction problem (QCSP) is an extension of the constraint satisfaction problem (CSP) in which variables are totally ordered and quantified either existentially or universally. This generalization provides a better expressiveness for modelling problems. Model Checking and planning under uncertainty are problems that can be modeled with QCSP. But such an expressiveness implies a high complexity. Whereas CSP is in NP, QCSP is PSPACE-complete, which is considered harder.

The SAT community has also done a similar generalization from the problem of satisfying a Boolean formula into the quantified Boolean formula problem (QBF). The most natural way to solve instances of QBF or QCSP is to instantiate variables from the outermost quantifier to the innermost. This approach is called *top-down*. Most QBF solvers implement top-down techniques. Those solvers lift SAT techniques to QBF. Nevertheless, Biere [1], or Pan and Vardi [2] proposed different techniques to solve QBF instances. Both try to eliminate variables from the innermost quantifier to the outermost quantifier, an approach called *bottom-up*. The bottom-up approach is motived by the fact that the efficiency of heuristics that are used in the SAT domain seems to be greatly reduced in QBF. The drawback of such approaches is the cost in space.

The problem of solving a QCSP is more recent than QBF, so there are few QCSP solvers. Gent, Nightingale and Stergiou [3] developed QCSP-Solve, a top-down solver that uses generalizations of well-known techniques in CSP. Repair-based methods seem to be quite helpful as well, as shown by Stergiou in [4].

In this paper we introduce `BlockSolve`, the first bottom-up algorithm to solve QCSPs. `BlockSolve` instantiates variables from the innermost to the outermost. On the one hand, this permits to factorize equivalent subtrees during the search. On the other hand, `BlockSolve` only uses classical CSP techniques, no need for generalizing them into QCSP techniques. The algorithm processes a problem as if it were composed of pieces of classical CSPs.

The rest of the paper is organized as follows. Section 2 defines the concepts that we will use during the paper. Section 3 describes `BlockSolve`. Finally, Section 4 experimentally compares `BlockSolve` to the state-of-the-art QCSP solver QCSP-Solve and Section 5 contains a summary of this work and details for future work.

2 Preliminaries

In this section we define the basic concepts that we will use.

Definition 1 (Quantified Constraint Network). *A* quantified constraint network *is a formula $\mathcal{QC}$ in which:*

- *$\mathcal{Q}$ is a* **sequence** *of quantified variables $Q_i x_i, i \in [1..n]$, with $Q_i \in \{\exists, \forall\}$ and x_i a variable with a domain of values $D(x_i)$,*
- *$\mathcal{C}$ is a conjunction of constraints $(c_1 \wedge ... \wedge c_m)$ where each c_i involves some variables among $x_1, \ldots, x_n$.*

Now we define what is a solution of a quantified constraint network.

Definition 2 (Solution). *The* solution *of a quantified constraint network $\mathcal{QC}$ is a tree such that:*

- *the root node r has no label,*
- *every node s at distance i $(1 \leq i \leq n)$ from the root r is labelled by an instantiation $(x_i \leftarrow v)$ where $v \in D(x_i)$,*
- *for every node s at depth i, the number of successors of s in the tree is $|D(x_{i+1})|$ if x_{i+1} is a universal variable or 1 if x_{i+1} is an existential variable. When x_{i+1} is universal, every value w in $D(x_{i+1})$ appears in the label of one of the successors of s,*
- *for any leaf, the instantiation on $x_1, \ldots, x_n$ defined by the labels of nodes from r to the leaf satisfies all constraints in $\mathcal{C}$.*

It is important to notice that contrary to classical CSPs, variables are ordered as an input of the network. A different order in the sequence $\mathcal{Q}$ gives a different network.

If all variables are existentially quantified, a solution to the quantified network is a classical instantiation, so the network is a classical constraint network.

Definition 2 leads to the concept of quantified constraint satisfaction problem.

Definition 3 (QCSP). *A* quantified constraint satisfaction problem *(QCSP) is the problem of the existence of a solution to a quantified constraint network.*

We point out that this original definition of QCSP, though different in presentation, is equivalent to previous recursive definitions. The advantage of ours is that it formally specifies what a solution of a QCSP is.

Example 1. Consider the quantified network $\exists x_1 \exists x_2 \forall x_3 \forall x_4 \exists x_5 \exists x_6$, $(x_1 \neq x_5) \wedge (x_1 \neq x_6) \wedge (x_2 \neq x_6) \wedge (x_3 \neq x_5) \wedge (x_4 \neq x_6) \wedge (x_3 \neq x_6), D(x_i) = \{0, 1, 2, 3\}, \forall i$. Figure 1 shows a solution of this network.

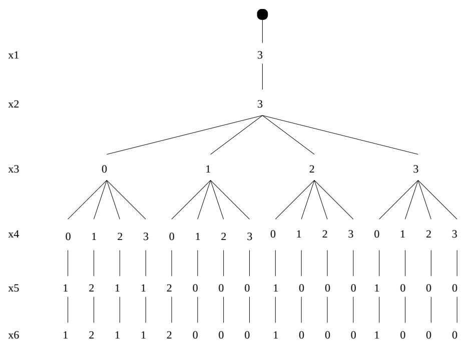


Fig. 1. A solution tree of Example 1

We define the concept of block, which is the main concept handled by our algorithm `BlockSolve`.

Definition 4 (Block). *A block in a network $\mathcal{QC}$ is a maximal subsequence of variables in $\mathcal{Q}$ that have the same quantifier. We call a block that contains universal variables a* universal block*, and a block that contains existential variables an* existential block*.*

If x and y are variables in two different blocks, with x earlier in the sequence than y, we say that x is the *outer* variable and y is the *inner* variable. In this paper, we limit ourselves to binary constraints for simplicity of presentation: A constraint involving x_i and x_j is noted c_{ij}.

The concept of block can be used to define a solution tree of a QCSP. This is a compressed version of the solution tree defined above.

Figure 2 is the block-based version of the solution tree in Fig. 1. The problem is divided in three blocks, the first and the third blocks are existential whereas the second block is universal. Existential nodes are completely instantiated, it means that all variables of those blocks have a single value. The universal block is in three nodes, each one composed of the name of the variables and a union

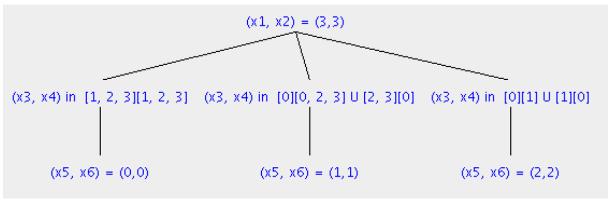

(x1, x2) = (3,3)
(x3, x4) in [1, 2, 3][1, 2, 3]
(x3, x4) in [0][0, 2, 3] U [2, 3][0]
(x3, x4) in [0][1] U [1][0]
(x5, x6) = (0,0)
(x5, x6) = (1,1)
(x5, x6) = (2,2)

Fig. 2. Solution of example 1

of Cartesian products of sub-domains. Each of the universal nodes represents as many nodes in the solution tree of Fig. 1 as there are tuples in the product.

`BlockSolve` will use this concept of blocks for generating a solution and for solving the problem.

Blocks will divide the problem in levels. Each couple (universal block, existential block) is a level. If the first block is existential, the first level is only composed by this block. We note p the number of levels in a problem.

We call $\mathcal{P}_k$ the subproblem that contains variables in levels k to p and constraints that are defined on those variables. The universal block at level k is noted $block_\forall(k)$ and the existential block is $block_\exists(k)$. $\mathcal{P}_1$ is the whole problem $\mathcal{P}$. The principle of `BlockSolve` is to solve $\mathcal{P}_p$ first, then using the result to solve $\mathcal{P}_{p-1}$, and so on until it solves $\mathcal{P}_1 = \mathcal{P}$.

3 The `BlockSolve` Algorithm

In this section we present the algorithm, and describe how it works on QCSP instances.

The main idea in `BlockSolve` is to instantiate existential variables of the last block, and to go up to the root instantiating all existential variables. Each assignment v_i of an existential variable x_i can lead to the deletion of inconsistent values of outer variables by propagation.

Removing a value of an outer *existential* variable is similar to the CSP case. While the domains of variables are non empty, it is possible to continue instantiating variables. But if a domain is reduced to the empty set, it will be necessary to backtrack on previous choices on inner variables and to restore domains.

Removing a value of an outer *universal* variable implies that we will have to find also another instantiation of inner variables that supports this value, because all tuples in universal blocks have to match to a partial solution of inner subproblem. But the instantiation that removes a value in the domain of an universal variable must not be rejected: it can be compatible with a subset of tuples of the universal block. The bigger the size of the subset, the better the grouping. Factorizing tuples of values for a universal block in large groups is a way for minimizing the number of times the algorithm has to solve subproblems. Each time an instantiation of inner variables is found consistent with a subset of tuples for a universal block, we must store this subset and solve again the inner subproblem wrt remaining tuples for the universal variables.

For a level k starting from level 1, we try to solve the subproblem $\mathcal{P}_{k+1}$, without forgetting that it must be compatible with all constraints in $\mathcal{P}$. If there is no solution for $\mathcal{P}_{k+1}$, we try to solve $\mathcal{P}_k$ with the tuples on $block_\exists(k)$ not yet tried when solving $\mathcal{P}_{k+1}$ (they were removed by instantiations in $\mathcal{P}_{k+1}$ that finally led to failure). If there exists a solution for $\mathcal{P}_{k+1}$, we try to instantiate $block_\exists(k)$ with values consistent with some of the tuples on $block_\forall(k)$. If success, we remove the tuples on $block_\forall(k)$ that are known to extend on inner variables, and we start again the process on the not yet supported tuples of $block_\forall(k)$.

`BlockSolve` needs more space than a top-down algorithm like QCSP-Solve. It keeps in memory all tuples of existential and universal blocks for which a solution has not yet been found. The size of such sets can be exponential in the number of variables of the block. `BlockSolve` keeps sets of tuples as unions of Cartesian products, which uses far less space than tuples in extension. In addition, computing the difference between two unions of Cartesian products is much faster than with tuples in extension.

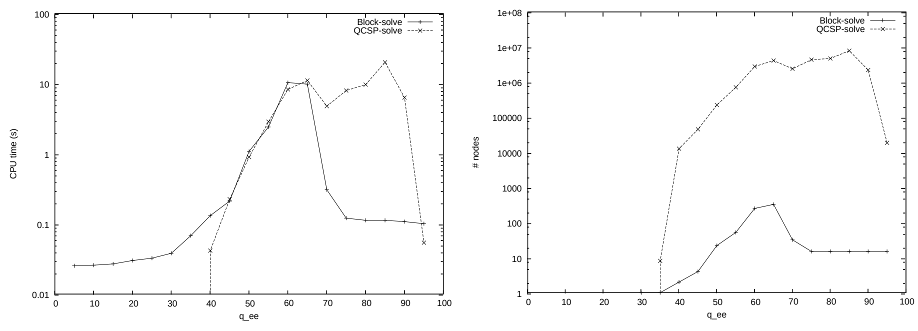
4 Experiments

In this section we compare QCSP-Solve and `BlockSolve` on random problems. The experiments show the differences between these two algorithms in CPU time and number of visited nodes.

`BlockSolve` is developed in Java using Choco as constraint library [5]. This library provides different propagation algorithms and a CSP solver. After loading the data of a problem, `BlockSolve` creates tables that fit to set of tuples of each block and finally launches the main function.

Instances of QCSP presented in these experiments have been created with a generator based on that used in [3]. The generator we use allow us to set the number of variables, the number of blocks, the number of variables in existential and universal blocks, the number of constraints, the looseness of binary constraints between a universal variable and an inner existential variable ($q_{\forall\exists}$), and the looseness of constraints between two existential variables($q_{\exists\exists}$). We fix all parameters except $q_{\exists\exists}$ during the experimentations. Results that are presented here in figure 3 are for instances of QCSP that have 25 variables with domains of 8 values, 5 variables by block. The first block of each instance is an existential one. For these instances, the transition phase is at $q_{\exists\exists} = 55$

We note that results are comparables for instances that are unsatisfiables (on the left of the transition phase), even if QCSP-Solve detects the inconsistency a little bit faster than `BlockSolve` when problems are far from the transistion phase. On the right, for problems that are satisfiables, `BlockSolve` is more efficient than QCSP-Solve. In almost all instances, `BlockSolve` visits far less nodes than QCSP-Solve.



CPU time Number of nodes explored

Fig. 3. Problems with 25 variables and 5 blocks.

5 Conclusions

In this paper we presented `BlockSolve`, a bottom-up QCSP solver that uses classical CSP techniques. Its specificity is that it treats variables from leaves to root in the search tree, and factorizes lower branches avoiding the search in subtrees that are equivalent. The larger this factorization, the better the algorithm, thus minimizing the number of nodes visited. Experiments show that grouping branches gives `BlockSolve` a great stability in time spent and in number of nodes visited. The number of nodes `BlockSolve` visits is much smaller than the number of nodes visited by QCSP-Solve in almost all instances.

Future work will focus on improving time efficiency of `BlockSolve`. Great improvements can probably be obtained by designing heuristics to efficiently prune subtrees that are inconsistent. Furthermore, most of the cpu time is spent updating and propagating tables of tuples on blocks. Finding better ways to represent them could significantly decrease the cpu time of `BlockSolve`.

We are very grateful to Kostas Stergiou, Peter Nightingale and Ian Gent, who kindly provided us with the code of QCSP-Solve.

References

1. Biere, A.: Resolve and expand. In: SAT. (2004)
2. Pan, G., Vardi, M.Y.: Symbolic decision procedures for qbf. In Wallace, M., ed.: CP. Volume 3258 of Lecture Notes in Computer Science., Springer (2004) 453–467
3. Gent, I., Nightingale, P., Stergiou, K.: QCSP-solve: A solver for quantified constraint satisfaction problems. In: Proceedings IJCAI'05, Edinburgh, Scotland (2005) 138–143
4. Stergiou, K.: Repair-based methods for quantified csps. In: Proceedings CP'05, Sitges, Spain (2005) 652–666
5. Choco: A Java library for constraint satis-faction problems, constraint programming and explanation-based constraint solving, http://choco.sourceforge.net. (2005)

A constraint programming application for allocating sensors and improving the diagnosability of a system

Student: R. Ceballos
Supervisors: R. M. Gasca and C. Del Valle

Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla

Abstract. Model-based diagnosis enables isolating the faults of a system. In this work, a new approach is proposed in order to improve the computational complexity of the diagnosis process. The key idea is the addition of a set of new sensors in order to improve the diagnosability of the system. The number and the allocation of the new sensors is obtained by solving a constraint satisfaction problem (CSP). In order to improve the computational complexity of the resolution of the CSP we propose a greedy method for determining the bottlenecks of the system.

1 Introduction

The objective of the Model-based diagnosis (MBD)[1][2] is the detection and identification of the reason for any unexpected behaviour, and the isolation of the components which fail. In MBD, the behaviour of components is simulated by using constraints. Inputs and outputs of components are represented as variables of the constraints. These variables can be observable and non-observable depending on the sensors allocation.

In this work, a new approach is proposed in order to improve the computational complexity for isolating faults in a system. Our approach is based on the addition of new sensors. A constraint satisfaction problem is obtained in order to select the necessary sensors to guarantee the problem specification. Our approach maintains the requirements of the user (detectability, diagnosability,...). In order to improve the computational complexity of the CSP problem, we propose an algorithm for determining the bottleneck sensors of the system.

The diagnosability of systems is a very active research area in the diagnosis community. A toolbox integrating model-based diagnosability analysis and automated generation of diagnostics is proposed in [3]. The proposed toolbox supports the automated selection of sensors based on the analysis of detectability and discriminability of faults. In this line, [4] proposed a methodology to obtain the diagnosability analysis using the analytical redundancy relations (ARR). This approach is based on an exhaustive analysis of the structural information. The objective is the addition of new sensors in order to increase the diagnosability.

Our paper has been organized as follows. First, definitions and notations are established in order to clarify MBD concepts. Section 3 introduces the basis of our approach. Section 4 describes the specification of the CSP. Section 5 shows the greedy method in order to improve the CSP resolution. Finally, conclusions are drawn and future work is outlined.

2 Notation and definitions

In order to explain our methodology, it is necessary to establish some concepts and definitions from the model-based diagnosis theories.

Definition 1. A *System Model* is a finite set of equality constraints which determine the system behaviour. This is done by means of the relations between non-observable and observable variables (sensors) of the system.

Definition 2. A *Diagnosis* is a particular hypothesis for how the system differs from its model. Any component could be working or faulty, thus the diagnosis space for the system initially consists of 2^{nComp} - 1 diagnoses [2], where *nComp* is the number of components of the system. The goal of diagnosis is to identify and refine the set of diagnoses.

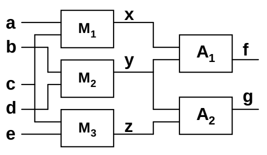
Definition 3. A set of components T is a *Cluster of components* [5], if it does not exist a common non-observable variable of any component of the cluster with any component outside the cluster, and if for all $Q \subset T$ then Q is not a cluster of components.

All common non-observable variables between components of the same cluster belong to the cluster; therefore, all the connections with components which are outside the cluster are monitored. A cluster of components is completely monitored, and for this reason the detection of faults inside the cluster is possible without any information from other components which do not belong to the cluster. A more detailed explanation appears in [5].

3 The basis of the algorithm

Our approach is based on the generation of new clusters of components by allocating sensors in some of the non-observable variables. These new clusters reduces the computational complexity of the diagnosis process since it enables the generation of the diagnosis of the whole system based on the diagnosis of the subsystems. Let C be a set of n components of a system, and C_1 and C_2 be clusters of n - m and m components such that $C_1 \cup C_2 = C$, then the computation complexity for detecting conflicts in C_1 and C_2 separately is lower than in the whole system C, since the number of possible diagnoses of the two clusters is $(2^{n\text{-}m}) + (2^{m})$ - 2 $\leq 2^{n\text{-}m} \quad 2^{m}$ - 2 which is less than 2^{n}-1.

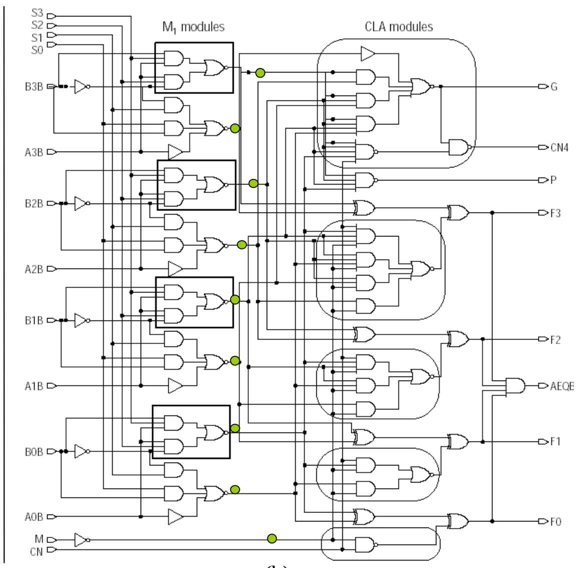
The clustering process enables isolating the faults of the original system, since the multiple faults which include components of different clusters are eliminated. These kinds of faults are transformed into single or multiple faults which belong to only one cluster. The computational complexity for discriminating faults in a system is always upper than in an equivalent system divided into clusters.



System Model

Constraints	V_o	V_{nob}
M_1: x=a*c	$a = a_{ob}$	x
M_2: y=b*d	$b = b_{ob}$	y
M_3: z=c*e	$c = c_{ob}$	z
A_1: f=x+y	$d = d_{ob}$	
A_2: g=y+z	$e = e_{ob}$	
	$f = f_{ob}$	
	$g = g_{ob}$	

(a)



(b)

Fig. 1. a) Polybox example b) 74181 ALU example

4 The CSP problem specification

The objective is to obtain the best allocation of sensors in order to generate new clusters. The allocation of the sensors will be formulated as a Constraint Satisfaction Problem (CSP). A CSP is a way for modelling and solving real problems as a set of constraints among variables.

Figure 1 shows two well-known examples in model-based diagnosis. Figure 1a shows the standard problem used in the diagnosis community [2], the polybox system. The system consists of five components: three multipliers, and two adders, as it appears in the system model (Figure 1a). Figure 1b shows the 74181 4-Bit ALU. It is one of the ISCAS-85 benchmarks [6]. It includes 61 components, 14 inputs and 8 outputs. Table 1 shows the set of constants, variables and constraints for determining the number and location of sensors for the polybox example. The components and non-observable variables are represented as enumerated variables. The following constants and variables are included:

– *cluster[p]*: This set of constants represents the set of possible clusters of the system. The maximal number of clusters is the number of components.
– *nSensors*: This variable stores the number of new sensors. It must be equal or smaller than the number of non-observable variables($nNonObsVar$).
– *sensor[k]*: This set of variables represents the possible new sensors of the system. They store a boolean value in the interval {true, false}, where *true* implies that there must be a sensor, and *false* the opposite.

Table 1. CSP for the polybox sensors allocation

Constants	**Value**
(1) Enumeration$\{M_1,M_2,M_3,A_1,A_2\}$	$\{1, \ldots, \text{nComp}\}$
(2) Enumeration$\{X,Y,Z\}$	$\{1, \ldots, \text{nNonObsVar}\}$
(3) cluster[1..nComp]	$\{1, \ldots, \text{nComp}\}$
Variable (= initial value)	**Domain**
(4) nSensors = {free}	$\mathcal{D} \in \{1, \ldots, \text{nNonObsVar}\}$
(5) sensor[1..nNonObsVar]	sensor[k] $\in$ {true, false}
(6) clusterOfComp[1..nComp]	clusterOfComp[i] $\in \{1, \ldots, \text{nComp}\}$
(7) clusterDist[1..nComp]	clusterDist[t] $\in \{1, \ldots, \text{nComp}\}$
(8) lim[1..nComp]	lim[s] $\in \{1, \ldots, \text{nComp}\}$
Constraints	
(9) $\neg$sensor[X] $\Rightarrow$ clusterOfComp[M_1] = clusterOfComp[A_1]	
(10) $\neg$sensor[Y] $\Rightarrow$ clusterOfComp[M_2] = clusterOfComp[A_1]	
(11) $\neg$sensor[Y] $\Rightarrow$ clusterOfComp[M_2] = clusterOfComp[A_2]	
(12) $\neg$sensor[Y] $\Rightarrow$ clusterOfComp[A_1] = clusterOfComp[A_2]	
(13) $\neg$sensor[Z] $\Rightarrow$ clusterOfComp[M_2] = clusterOfComp[A_2]	
(14) distribute(clusterDist, clusterOfComp, cluster)	
(15) $\forall$ j: 1$\leq$i$\leq$nComp: lim[j] $\leq$ cluster[j]	
(16) $\forall$ j: 1<j$\leq$nComp: (clusterOfComp[j-1] = lim[j-1]) $\Rightarrow$ lim[j-1]+1 $\leq$ lim[j]	
(17) $\forall$ j: 1<j$\leq$nComp: (clusterOfComp[j-1] < lim[j-1]) $\Rightarrow$ lim[j-1] $\leq$ lim[j]	
(18) nSensor = (N 1$\leq$j$\leq$nNonObsVar: sensor[j] = true)	

- *clusterOfComp[i]*: This set of variables represents the cluster associated to each component *i*.
- *clusterDist[t]*: This set of variables stores the number of components included in each cluster *t*.
- *lim[j]*: This set of variables are used in order to break the symmetries of the solutions of the sensors assignment. They hold the maximal number of the cluster that can be assigned to a component.

For each common non-observable variable between two components it is generated a constraint which guaranties that if there is not a sensor the two components must belong to the same cluster. Table 1 shows the constraints (9),(10),...(13) that store this kind of information, and it is based on Figure 1. The final sensors allocation is stored in $sensor_k$, and the distribution of the clusters is stored in $clusterDist_i$. The constraint (14) maintains the distribution of the components in the possible clusters. The constraints (15), (16) and (17) limit the number of possible solutions and eliminate the symmetries in the allocations of the components to a cluster. These constraints guarantee that for each sensors allocation, only one distribution of the components in the clusters is possible. The constraint (18) holds in the variable *nSensor* the total number of new sensors used in the solution. The optimization problem can have different objectives, depending on the user and the problem requirements. Two typical goals can be:

- To minimize the number of sensors (if the number of clusters is fixed).
- To minimize the maximal number of components in each cluster (if the maximal number of sensors is fixed).

It is possible to add other constraints in order to guarantee properties of the solution. For example in order to guarantee prices, to respect requirements of the customers, to store incompatibilities, to specify problems with the space, ...

```
sensorsOrder(P)
  componentVotes[nComp][nNonObsVar]
  sensorVotes[nNonObsVar]
  // All the components (1..nComp) votes the variables (sensors)
  // associated to the minimal paths
  forEach j between 1 to nComp
    forEach $P_k$ from component i to component j
      forEach q between 1 to length($P_k$)
        $\triangle$ = (voteValue / (length($P_{k,i,j}$) $\cdot$ length($P_k$))
        forEach v includes in path[q]
          componentVotes[i][$P_{k,i,j,q}$] += $\triangle$
        endForEach
      endForEach
    endForEach
  endForEach
  // Recounting of votes for each sensor
  forEach $\text{sensor}_j$ between 1 to nSensors
    sensorVotes[j] = 0
    forEach i between 1 to nComp
      sensorVotes[j] += componentVotes[i][j] / ( $\triangle$ $\cdot$ nComp )
    endForEach
  endForEach
  return sort(sensorVotes)
```

Fig. 2. Algorithm for obtaining the bottleneck sensors of the system ($O(\mathrm{n}^2 \cdot \mathrm{m}^2)$, where n is the number of components and m is the number of non observable variables)

5 Improving the algorithm: A greedy method

The computational complexity of a CSP is exponential in general. We propose a method for obtaining what are the most important sensors in order to generate new clusters, that is, the bottlenecks of the system. This method has two phases:

1. The calculation of the minimal paths: A graph where the nodes represent the components of the system, and the edges represent the connections between each two components (non-observable variables). Each edge has a weight calculated as the number of common non-observable variables between two components. By applying the Floyd's algorithm (dynamic programming), all the shortest paths between all pairs of nodes will be stored.
2. In order to determine which are bottlenecks of the system, each minimal path will vote which sensors are more important. Figure 2 shows this algorithm. Each minimal path will vote for the included non-observable variables of the minimal path. The number of votes is scaled in order to guarantee that each component generates the same total number of votes. These votes enable generating a sorted list of non-observable variables, which will be the most important sensors.

The bottleneck sensors of the system represent the best sensors in order to isolate components. The sorted list of sensors enables creating a CSP with fewer variables in order to find the solution of the problem in a limited time. Only the solutions included in the combinations of the m bottleneck sensors will be tested, and therefore the number of possible solutions will be lower than 2^m. The optimal solution is not guaranteed, but the reduction of computational complexity enables finding a solution in a limited time.

Example: In the *Alu74181* example the most important sensors are (based on the number of votes): E_{02}(930), E_{03}(878), X_{28}(773), E_{01}(737), E_{00}(583), D_{00}(514), D_{01}(463), D_{02}(326), D_{03}(301),... The other sensors have less than 166 votes. The first 9 sensors are represented by shaded circles in Figure 1b. The possible diagnoses in the system are 2^{61} - 1. By using the first 9 selected sensors the number of clusters is 17 (all with less than 6 components) and the computational complexity is reduced to less than 2^9 possible diagnoses.

6 Conclusions and future work

The methodology was applied to two standard examples, and the results are very promising. The computational complexity of the diagnosis process is improved due to the optimal allocation of the sensors. Our approach is only based on topological properties, this characteristic enables applying this approach to different types of systems. It is possible to add other constraints in order to guarantee other properties of the solution. As future work, we are working in new greedy methods in order to improve the votes counting.

7 Acknowledgements

This work has been funded by the M. de Ciencia y Tecnología of Spain (DPI2003-07146-C02-01) and the European Regional Development Fund.

References

1. Reiter, R.: A theory of diagnosis from first principles. Artificial Intelligence 32 **1** (1987) 57–96
2. de Kleer, J., Mackworth, A., Reiter, R.: Characterizing diagnoses and systems. Artificial Intelligence **2-3** (1992) 197–222
3. Dressler, O., Struss, P.: A toolbox integrating model-based diagnosability analysis and automated generation of diagnostics. In: DX03, 14th International Workshop on Principles of Diagnosis, Washington, D.C., USA (2003) 99–104
4. Travé-Massuyés, L., Escobet, T., Spanache, S.: Diagnosability analysis based on component supported analytical redundancy relations. In: 5th IFAC Symposium on Fault Detection, EEUU (2003)
5. Ceballos, R., Pozo, S., del Valle, C., Gasca, R.M.: An integration of FDI and DX techniques for determining the minimal diagnosis in an automatic way. MICAI, Lecture Notes in Artificial Intelligence, LNAI 3789 (2005) 1082–1092
6. Hansen, M.C., Yalcin, H., Hayes, J.P.: Unveiling the ISCAS-85 Benchmarks: A Case Study in Reverse Engineering. IEEE Design and Test of Computers **16** (1999) 72–80

Symmetry Breaking in Subgraph Pattern Matching

Student: Stéphane Zampelli
Supervisors: Yves Deville, Pierre Dupont

Université Catholique de Louvain,
Department of Computing Science and Engineering,
2, Place Sainte-Barbe
1348 Louvain-la-Neuve (Belgium)
{sz,yde,pdupont}@info.ucl.ac.be

1 Introduction

This work aims at applying and extending symmetry techniques for subgraph matching. Symmetries arise naturally in graphs since a permutation can be viewed as an automorphism of a graph. However, although a lot of graph problems have been tackled [1, 2], a computation domain for graphs has been defined [3], and despite the fact that symmetries and graphs are related, little has been done to investigate the use of symmetry breaking for graph problems in constraint programming.

2 Background

A **graph** $G = (N, E)$ consists of a **node set** N and an **edge set** $E \subseteq N \times N$, where an edge (u, v) is a pair of nodes. The nodes u and v are the endpoints of the edge (u, v). We consider directed and undirected graphs. A **subgraph** of a graph $G = (N, E)$ is a graph $S = (N', E')$ where N' is a subset of N and E' is a subset of E.

A **subgraph monomorphism** (or subgraph matching) between G_p and G_t is a total injective function $f : N_p \rightarrow N_t$ respecting the monomorphism constraint : $(u, v) \in E_p \Rightarrow (f(u), f(v)) \in E_t$.

The CSP model of graph matching should represent a total function $f : N_p \rightarrow N_t$. This total function can be modeled with $X = x_1, ..., x_n$ with x_i a FD variable representing the ith node of G_p and $D(x_i) = N_t$. The injective condition is modeled with the global constraint `alldiff`$(x_1, ...x_n)$. The monomorphism condition is translated into the global constraint `MC`$(x_1, ..., x_n) \equiv \bigwedge_{(i,j)\in E_p}(x_i, x_j) \in E_t$. Implementation, comparison with dedicated algorithms, and extension to subgraph isomorphism and to graph and function computation domain can be found in [4, 5].

A CSP instance is a triple $< X, D, C >$ where X is the set of variables, D is the universal domain specifying the possible values for those variables, and C is the set of constraints. In the rest of this document, $n = |N_p|$, $d = |D|$, and $D(x_i)$ is the domain of x_i. A symmetry over a CSP instance P is a bijection σ mapping solutions to solutions, and hence non solutions to non solutions [6]. Since a symmetry is a bijection where domain and target sets are the same, a symmetry is a permutation. A *variable symmetry* is a bijective function $\sigma : X \to X$ permuting a (non) solution $s = ((x_1, d_1), \ldots, (x_n, d_n))$ to a (non) solution $s' = ((\sigma(x_1), d_1), \ldots, (\sigma(x_n), d_n))$. A *value symmetry* is a bijective function $\sigma : D \to D$ permuting a (non) solution $s = ((x_1, d_1), \ldots, (x_n, d_n))$ to a (non) solution $s' = ((x_1, \sigma(d_1)), \ldots, (x_n, \sigma(d_n)))$. A *value and variable symmetry* is a bijective function $\sigma : X \times D \to X \times D$ permuting a (non) solution $s = ((x_1, d_1), \ldots, (x_n, d_n))$ to a (non) solution $s' = (\sigma(x_1, d_1), \ldots, \sigma(x_n, d_n))$. A *conditional symmetry* of a CSP P is a symmetry holding only in a sub-problem $P^{'}$ of P. The conditions of the symmetry are the constraints necessary to generate $P^{'}$ from P [7]. A *group* is a finite or infinite set of elements together with a binary operation (called the group operation) that satisfy the four fundamental properties of closure, associativity, the identity property, and the inverse property. An *automorphism of a graph* is a graph isomorphism with itself. The sets of automorphisms $Aut(G)$ define a finite permutation group.

3 Variable Symmetries

It has been shown that the set of variable symmetries of the CSP P is the automorphism group of a *symbolic graph* $S(P)$ [6] and this automorphism group can be computed by using tools such as NAUTY [8]. We show that $Aut(S(P))$ is equal to $Aut(G_p)$, and thus that the set of variable symmetries is $Aut(G_p)$.

Figure 1 shows an instance composed of two undirected triangles. The 6 solutions are : $\{(1,2,3), (1,3,2), (2,1,3), (2,3,1), (3,2,1), (3,1,2)\}$. The two generators of the pattern are (1 3) and (2 3). The size of its automorphism group is 6. Once the variable symmetries are broken, the unique solution is $\{(1,2,3)\}$.

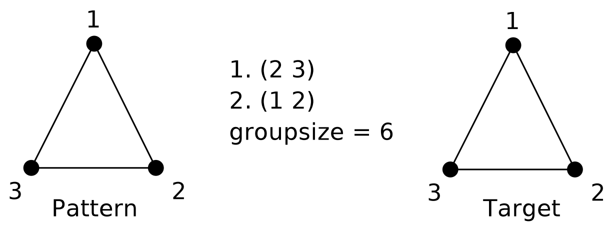


Fig. 1. Example of instance.

Two techniques were selected to break variable symmetries. The first technique is an approximation and breaks only the generators of symmetry group [9] computed by NAUTY. The second technique breaks all variable symmetries of an injective problem by using a SchreierSims algorithm, provided that the generators of the variable symmetry group are known [6].

4 Value Symmetries

In graph matching, value symmetries are the automorphisms of the target graph and do not depend on the pattern graph.

Figure 2 gives an example of a value symmetry on the target graph. There is only one generator for this graph : (1 2). Suppose the pattern graph is a path of length 2 : $x_1 \rightarrow x_2 \rightarrow x_3$. Suppose $(1,3,2)$ is a solution. Then $(2,3,1)$ is also a solution. Suppose $(1,3,4)$ is a solution. Then $(2,3,4)$ is also a solution.

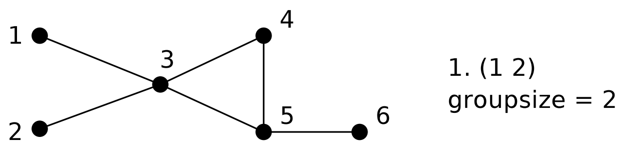


Fig. 2. Example of value symmetry on the target graph.

Breaking initial value symmetries can be done by using GE-Tree technique [10]. The idea is to modify the distribution by avoiding symmetrical value assignment [6].

5 Conditional Value Symmetries

In subgraph monomorphism, the relations between values are explicitly represented in the target graph. This allows the detection of conditional value symmetries. The induced graph upon the nodes $\cup_{i \in N_p} D(x_i)$ is the subgraph G_t^*

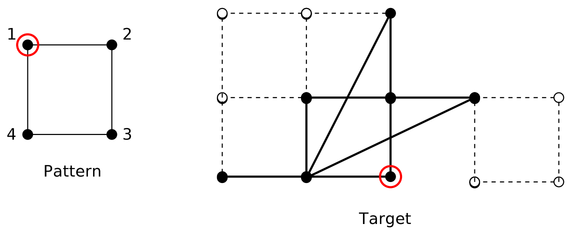


Fig. 3. Example of dynamic target subgraph.

of G_t in which a solution is searched. Figure 3 shows an example of dynamic target graph. In this figure, the circled nodes are assigned together. The blank nodes are the nodes excluded from $\cup_{i \in [1,\cdots,n]} D(x_i)$, and the black nodes are the nodes included in $\cup_{i \in [1,\cdots,n]} D(x_i)$. The plain edges are the selected edges for the dynamic target subgraph.

We show that each automorphism of G_t^* is a conditional value symmetry for the state S and that the conditions are the assignments corresponding to assigned variables. We show how the GE-Tree algorithm can be modified to handle conditional values symmetries.

6 Local Value Symmetries

In this section, we introduce the concept of local value symmetries, that is value symmetries on a subproblem. Such symmetries will be detected and exploited during the search. We first introduce the partial dynamic graph concept. Those graphs are associated to a state in the search and correspond to the unsolved part of the problem. This can be viewed as a new local problem to the current state. The **partial dynamic pattern graph** G_p^- is an induced subgraph of G_p on the set of non assigned variables. The **partial dynamic target graph** G_t^- is an induced subgraph of G_t on the union of the domains of the non assigned variables.

When forward checking (FC) is used during the search, in any state in the search tree, every constraint involving *one* uninstantiated variable is arc consistent. In other words, every value in the domain of an uninstantiated variable is consistent with the partial solution. This FC property on a binary CSP ensures that one can focus on the uninstantiated variables and their associated constraints without loosing or creating solutions to the initial problem. Such a property also holds when the search achieves stronger consistency in the search tree (Partial Look Ahead, Maintaining Arc Consistency, ...).

We show that value symmetries of the local CSP P' can be obtained by computing $Aut(G_t^-)$, and that these symmetries can be exploited without loosing or adding solutions to the initial matching problem. It is important to notice that the value symmetries of P' are *not* conditional symmetries of P. It is not possible to add constraints to P to generate P', since the P' is a new problem with less variables. As the CSP P' is a local CSP associated to a state S, these value symmetries are called *local value symmetries.*

Consider the subgraph monomorphism instance (G_p, G_t) in Figure 4. Nodes of the pattern graph are the variables of the corresponding CSP, i.e. node i of G_p corresponds to variable x_i. Suppose that x_1 has been assigned to value 1. Because of MC propagation, $D(x_3) = \{4, 6, 7\}$. Moreover, because of alldiff$(x_1, \cdots, x_n)$, value 1 is deleted from all domains $D(x_i)$ $(i \neq 1)$. The new CSP $P^{'}$ consists of the subgraph of $G_p^- = (\{2, 3, 4, 5\}, \{(2, 3), (3, 2), (3, 5), (5, 3), (4, 5), (5, 4), (2, 4), (4, 2)\})$ and $G_t^- = (\{2, 3, 4, 5, 6, 7\}, \{(2, 3), (3, 2), (3, 5), (5, 3), (4, 5), (5, 4), (2, 4), (4, 2), (6, 7), (7, 6)\})$. The domains of the variables of $P^{'}$ are : $D(x_3) = \{4, 6, 7\} = \{4\}$, $D(x_2) = \{2, 5, 6, 7\} = \{2, 5\}$, $D(x_5) = \{2, 5, 6, 7\} = \{2, 5\}$, $D(x_3) = \{3, 4, 6, 7\} = \{3, 4\}$. For the state S, $Sol(S) = \{(1, 5, 4, 3, 2), (1, 2, 4, 3, 5)\}$ and $BSol(S) = \{(1, 2, 4, 3, 5)\}$. For the subproblem $P^{'}$, $Sol(P^{'}) = \{(5, 4, 3, 2),(2, 4, 3, 5)\}$ and $BSol(P^{'}) = \{(2, 4, 3, 5)\}$. The partial assignment $(x_1, 1)$ in state S together with the solutions of $P^{'}$ equals $Sol(S)$.

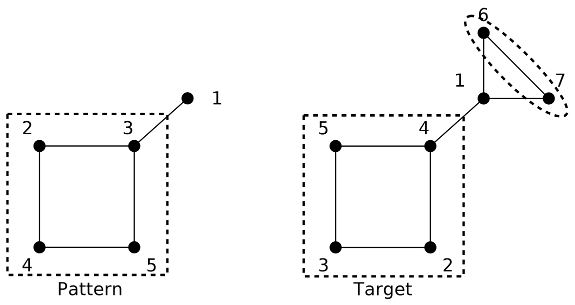


Fig. 4. Example of local value symmetry. The dashed squares show the new subgraph monomorphism instance for CSP P'.

Breaking local value symmetries is equivalent to breaking value symmetries on the subproblem P'. Puget's method [6] and the dynamic GE-Tree method [10] can thus be applied to the local CSP P'.

7 Experimental results

The data graphs used to generate instances are from the GraphBase database containing different topologies and has been used in [11]. The undirected set contains graphs ranging from 10 nodes to 138 nodes. Using those graphs, there are 1225 instances for undirected graphs. All runs were performed on a dual Intel(R) Xeon(TM) CPU 2.66GHz with 2 Go of RAM.

In our tests, we look for all solutions. A run is solved if it finishes under 5 minutes, unsolved otherwise. First we applied the basic CSP model; then the GE-Tree technique (value symmetries) and the full variable symmetry (FVS) technique that breaks all variable symmetries, and both techniques together. Results are shown in Table 1. In those runs, the preprocessing time has been considered. The total time column shows the total time needed for the solved instances. The mean time column shows the mean time for the solved instances. The mean time was reduced in both variable symmetry breaking and in value symmetry breaking. However the value symmetry breaking tends to be more effective regarding the mean time. This can be explained by the fact that symmetric solutions arise mainly when the target is highly symmetrical i.e. has a lot of edges. Both variables and value symmetry breaking slightly increase the percentage of the solved instances. Thanks to variable and value symmetry breaking, more instances are solved. Interestingly, the combination of GE-Tree and FSV does not outperform GE-Tree alone. This shows that symmetries on the target graph, in this data set, are more important regarding the number of solutions. However, the percentage of solved instances do not highly increase, calling for a dynamic symmetry breaking.

Table 1. Comparison over GraphBase undirected graphs for variable and value symmetries.

All solutions 5 min.				
	solved	unsol	total time	mean time
CSP	53,6%	46,3 %	31 min.	20.1 sec.
GE-Tree	55,3%	44,7 %	6 min.	3.21 sec.
FVS	54,9 %	45,1%	31 min.	19 sec.
GE-Tree and FVS	55,3 %	44,7%	26 min.	8.68 sec.

8 Conclusion

In this work, we developed value and variable symmetries for subgraph matching. We also showed how to detect dynamic symmetries. As future works, we would like to implement the conditional and local value symmetry detection, asses its performance on highly regular and symmetrical graphs, consider local variable symmetries and consider automatic detection of symmetries in graph computation domain.

References

1. Beldiceanu, N., Flener, P., Lorca, X.: The tree constraint. In: Proceedings of CP-AI-OR'05. Volume LNCS 3524., Springer-Verlag (2005)
2. Sellman, M.: Cost-based filtering for shorter path constraints. In: Proc. of the 9th International Conference on Principles and Pratice of Constraint Programming (CP). Volume LNCS 2833., Springer-Verlag (2003) 694–708
3. Dooms, G., Deville, Y., Dupont, P.: Cp(graph): Introducing a graph computation domain in constraint programming. In: Proc. of the 9th International Conference on Principles and Pratice of Constraint Programming (CP). Volume LNCS 3709., Springer-Verlag (2005)
4. Zampelli, S., Deville, Y., Dupont, P.: Approximate constrained subgraph matching. Principles and Pratice of Constraint Programming (2005)
5. Deville, Y., Dooms, G., Zampelli, S., Dupont, P.: Cp(graph+map) for approximate graph matching. 1st International Workshop on Constraint Programming Beyond Finite Integer Domains, CP2005 (2005)
6. Puget, J.F.: Automatic detection of variable and value symmetries. In: Proc. of the 9th International Conference on Principles and Pratice of Constraint Programming (CP). Volume LNCS 3709., Springer-Verlag (2005) 477–489
7. Gent, I.P., Kelsey, T., Linton, S.A., McDonald, I., Miguel, I., Smith, B.M.: Conditional symmetry breaking. In: Proc. of the 9th International Conference on Principles and Pratice of Constraint Programming (CP). Volume LNCS 3709., Springer-Verlag (2005) 256–270
8. McKay, B.D.: Practical graph isomorphism. Congressus Numerantium **30** (1981) 45–87
9. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry breaking predicates for search problem. In: Proceedings of KR'96. (1996)
10. C.M., R.D., Gent, I., T., K., S., L.: Tractable symmetry breaking in using restricted search trees. ECAI'04 (2004)
11. Larrosa, J., Valiente, G.: Constraint satisfaction algorithms for graph pattern matching. Mathematical. Structures in Comp. Sci. **12**(4) (2002) 403–422

Adversarial Constraint Satisfaction plays sHex

Student: David Stynes[1]
Supervisor: Ken Brown[1]

Cork Constraint Computation Centre,
Dept of Computer Science, UCC, Cork, Ireland
d.stynes@4c.ucc.ie, k.brown@cs.ucc.ie

Abstract. Adversarial Constraint Satisfaction models problems where self-motivated decision makers operate in a shared problem space. It combines techniques from constraint satisfaction and AI game playing. Here we apply it to a new domain, in which players attempt to generate winning paths in an extension of the game Hex.

1 Introduction

Adversarial Constraint Satisfaction[1] can be used to model problems in which intelligent self-motivated participants with different objectives must form a shared solution which satisfies all of the problem's constraints. The objectives of the participants can be conflicting, which results in an adversarial situation where each participant is attempting to find a solution to the problem which maximises their objective at the expense of the other participants' objectives. We aim to expand upon the system presented in [1] to allow for a greater variety of constraints within problems and for more complex objectives for participants, in order to extend the range of problems that can be tackled with Adversarial CSP. In this paper we consider a variation of Hex, which is inspired by a security system problem in which one participant's objective is defence of the system and the other participant's objective is to breach the system's security.

2 Background

Adversarial CSP was proposed in [1] to model problems with multiple decision makers with independent and possibly conflicting objectives. It proposed a protocol for coordination in which agents take turns to instantiate the variables in a CSP. The problem must end in a satisfactory solution, so a controller ensures that the agents backtrack out of situations which lead to an inconsistency. On a turn, an agent evaluates the current state of the problem and instantiates a variable, with the aim of guiding the solution towards its respective objective. It can assign any unassigned variable with any value that is not known to cause a conflict. To make this decision, an agent should reason about the likely moves of the other agents. The depth of moves the agent will look ahead to is limited by the time complexity of searching further; while in theory the agent could look ahead until a complete problem solution is formed, this would take an unreasonable amount of time and in general, the look ahead is limited to merely a few moves.

The agent then needs a strategy for choosing a move. Four different strategies were considered in [1]. *Minimax* chooses moves which minimise the maximum score its opponent can get. *Maximin* chooses moves to guard against an opponent trying to minimise the current participant's own score, and so selects moves that maximise the minimum score it can achieve. *Maximax* assumes that each participant will attempt to maximise their own objective. *MaxWS* is an algorithm designed to handle inaccuracies in the evaluation function. The evaluations can only estimate the score from the current position, since future moves and constraint propagation may reduce the score obtainable. Also, since on subsequent moves the opponent will be looking ahead to a deeper level, it will have more information on which to base the decision. Thus for moves under the opponent's control we return evaluations weighted by a probability that the opponent will make that move, while for the current participant's moves we choose the move that maximises a weighted sum of the evaluation. The probability of an opponent's move is calculated by dividing the payoff for that opponent by the sum of the payoffs over all moves.

The game of Hex is a two player game on a rhombic board with hexagonal cells. The classic board dimensions are 11x11, but it can be any size. The players, Black and White, take turns placing pieces of their own color on empty cells of the board. Black's objective is to connect the north-west edge of the board to the south-east edge with a chain of black pieces. White's objective is to connect the south-west edge of the board to the north-east edge with a chain of white pieces. Figure 1 shows a sample Hex position on a 4x4 board, and Fig. 2 shows black playing a winning move from that position.

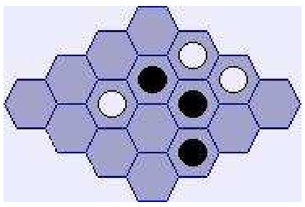


Fig. 1. A sample Hex position

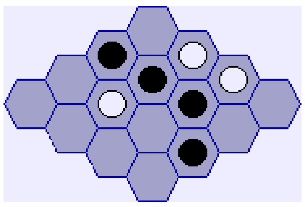


Fig. 2. Black plays a winning move

The game can never end in a draw. This follows from the fact that is all of the cells are occupied, then a winning chain for Black or White must exist. Anshelevich[2] uses virtual connections, which represent sub-games on the board which a player is guaranteed to win even if the opponent plays in the sub-game first, and an electrical network flow model to play computer Hex and is the current state of the art.

3 Extending Hex to sHex

The basic game of Hex can be modelled as a graph (see Fig. 3), where each node in the graph represents a cell of the board and the players are attempting to create a path of nodes of their own color from one side of the graph to the other. This graph has no constraints between the nodes.

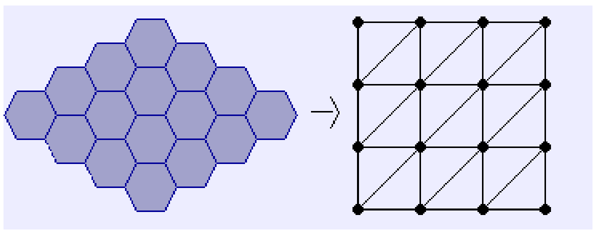


Fig. 3. Representing Hex as a graph

We take the graph as being a model for a security system, where one participant is the administrator of the system, while the other is an attacker. We refer to this security model as sHex. If the attacker makes a path of his color from one side to the other it means he has successfully infiltrated the system, while if the defender has made a path of his own color, then he has successfully defended the system from attack. Due to limited resources and inter-connectivity of system components, the system has certain limitations on what colors can be assigned and so we overlay a graph coloring problem on the board. The reasoning behind this was that blocking off a certain aspect of the system to the attacker may result in creating an opening elsewhere within the system. If ever the system detects a violation of these limitations it automatically restores to its last valid back-up and removes the values that caused the violation from the relevant node(s). These new limitations on the combinations of colors between certain nodes add constraints and backtracking to the basic hex game.

Additionally, in sHex, we now allow n different colors for a node to represent different "strengths" of the attacker's offense. A path of the 1st color being best for the attacker and of the nth color being a complete block by the defender. Agents are also allowed to select any color, which is not known to cause a conflict, for a node, not merely their own color. Thus the attacker's objective now becomes to create the strongest path he can, with the strength of a path being calculated based upon how many of each color node are in the path; a path of all 1st color nodes being the best possible path, and all nth color nodes the worst possible path.

4 Playing sHex

The game plays the same as the system described in [1]. On their turn, each agent must decide: how deep to look ahead, how to order branches for searching, how to evaluate partial solutions and how to decide upon the best move. The following is how our agents in our experiments are configured to play sHex.

Propagation The same propagation (MAC)[3] is used by all agents in sHex.

Depth of search The agents are limited to looking ahead 2 moves in our experiments for complexity reasons.

Evaluation Functions Each agent calls a heuristic evaluation function to estimate how good a position is for that agent. The function first calculates what it believes to be the best strength path from one side to the other that the current agent can complete. Then it calculates the same for the opponent (between his respective sides of the board). The ratio of these two values is what the heuristic evaluation function returns as its estimation of the quality of the given position for the current agent.

The means we use to calculate the best path it is possible for an agent to form is as follows. Each node is assigned a weight based on how good a color it is assigned or what the best color in its domain is, if it is unassigned. Each edge is then assigned a length equal to the sum of the weights of the two nodes it connects. We calculate virtual connections between nodes according to the H-search algorithm described in [2] and each virtual connection is added as a new weighted edge in the graph, though the weighting for a virtual connection edge is slightly increased compared to that of the original edges to indicate that a virtual connection is not quite as strong as a true edge. We then calculate the shortest distance path possible between the two sides of the board that this agent is attempting to connect using Dikjstra's algorithm [4].

Branch Ordering All agents in sHex simply select variables and values lexicographically during lookahead search.

Game Strategies The strategies agents can use are Minimax, Maximax, Maximin, MaxWS as described previously.

5 Experiments and Results

We have generated 40 random problems with 4x4 nodes and 4 colors, 20 of which had 5% constraint density and 20 of which had 10% constraint density. The constraint densities are quite low so as not to overly limit the lines of play open to the participants to the point that their choice of strategy would no longer affect their choices of moves. The strategies Minimax, Maximax and Maximin are equivalent on these problems as a result of the nature of the evaluation function, because one player's evaluation is the inverse of his opponent's evaluation. Thus we ran tests with the players using Minimax and MaxWS. The constraints we used are a uniformly random selection from: *equals*, *not-equals*, *greater than*, *less than*, *greater than or equal* and *less than or equal*. The problems have on average a 50% tightness but tightness for each individual problem varies.

For any pair of strategies, *strat1* and *strat2*, we perform four tests on each test problem. We take the cases where the first player is using *strat1* and has an attacker objective, where the first player is using *strat1* and has a defender objective, where the first player is using *strat2* and is an attacker and where the first player is using *strat2* and is a defender. By its nature, the first player has a large advantage in the game of Hex and similarly so in sHex, thus we expect strategies to perform better when they are the first player than when they are the second.

Table 1 shows the performance for the different strategies in our tests with 5% constraint density. Table 2 shows performance on the 10% constraint density tests. The optimal score possible is a score of 0. Table 3 shows how many cases were won for each test class. In terms of total wins, the Minimax algorithm performed better than MaxWS in most conditions, only performing poorer when playing as the second player in the lower density problems. When you are the first player and are unconcerned about win quality, Minimax is always the best choice. When you are the second player and unconcerned about win quality, if the first player is using Minimax then your best strategy is MaxWS, while if the first player is using MaxWS it is better to use Minimax as the second player. However in terms of quality of win, MaxWS generally out performs Minimax in the cases where it does win.

Table 1. Average scores with 5% constraint density

	Minimax	MaxWS
Winning Player1 Attacker	-0.23042	-0.24365
Winning Player1 Defender	-0.26888	-0.14411
Winning Player2 Attacker	-0.47247	-0.29050
Winning Player2 Defender	-0.39853	-0.37840
Losing Player1 Attacker	-25.40837	-41.21123
Losing Player1 Defender	-31.49409	-26.82651
Losing Player2 Attacker	-56.56792	-89.48099
Losing Player2 Defender	-47.34488	-50.52685

Table 2. Average scores with 10% constraint density

	Minimax	MaxWS
Winning Player1 Attacker	-0.41256	-0.40406
Winning Player1 Defender	-0.41621	-0.36553
Winning Player2 Attacker	-0.71606	-0.68821
Winning Player2 Defender	-0.53862	-0.54663
Losing Player1 Attacker	-1.84773	-2.26522
Losing Player1 Defender	-1.47981	-1.70215
Losing Player2 Attacker	-19.30188	-24.30217
Losing Player2 Defender	-22.47613	-23.46171

Table 3. Total Wins

	Minimax	MaxWS
5% density, Player1	65	58
5% density, Player2	18	19
10% density, Player1	60	57
10% density, Player2	22	21

The initial results indicate that playing order is important, different strategies should be used depending on which player you are, and what strategy your opponent is using. Also important is whether you want to win as often as possible or to have the best quality win (but at the cost of not necessarily winning as often).

6 Conclusions and Future Work

We have developed a broader version of adversarial constraint satisfaction, one which supports a larger variety of constraints and has more complicated objectives, by applying it to a new domain. We have shown that the difficulty of evaluating complex objectives may be offset by utilizing existing knowledge from other fields. We have shown that different agent configurations can be advisable depending on the adversary's configuration. In the future, we wish to develop an improved intermediate position evaluation function, that takes account of all potential paths rather than just the best potential path and in which one player's evaluation is not the inverse of his opponent's, to allow more differences between strategies. We also would like to extend the system to a more realistic security model.

7 Acknowledgements

This work has received funding from the Irish Research Council for Science Engineering and Technology(IRCSET) Embark Initiative and from Microsoft Research Cambridge (MSRC). We also wish to acknowledge the support of Youssef Hamadi from MSRC.

References

1. Brown, K.N., Little, J., Creed, P.J., Freuder, E.C.: Adversarial constraint satisfaction by game-tree search. Proceedings of the 16th European Conference on Artificial Intelligence 2004 (2004) 151–155
2. Anshelevich, V.V.: A hierarchical approach to computer hex. Artificial Intelligence **134**(1-2) (2002) 101–120
3. Sabin, D., Freuder, E.C.: Contradicting conventional wisdom in constraint satisfaction. Proceedings of the Second International Workshop on Principles and Practice of Constraint Programming (PPCP-94) **874** (1994) 10–20
4. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik **1**(1) (1959) 269–271

The importance of Relaxations and Benders Cuts in Decomposition Techniques: Two Case Studies

Alessio Guerri (student) and Michela Milano (supervisor)

DEIS, University of Bologna
V.le Risorgimento 2, 40136, Bologna, Italy
{aguerri, mmilano}@deis.unibo.it

When solving combinatorial optimization problems it can happen that using a single technique is not efficient enough. In this case, simplifying assumptions can transform a huge and hard to solve problem in a manageable one, but they can widen the gap between the real world and the model. Heuristic approaches can quickly lead to solutions that can be far from optimality. For some problems, that show a particular structure, it is possible to use decomposition techniques that produce manageable subproblems and solve them with different approaches. Benders Decomposition [1] is one of such approaches applicable to Integer Linear Programming. The subproblem should be a Linear Problem. This restriction has been relaxed in [4] where the technique has been extended to solvers of any kind and called Logic-Based Benders Decomposition (LBBD). The general technique is to find a solution to the first problem (called Master Problem (MP)) and than search for a solution to the second problem (Sub-problem (SP)) constraining it to comply with the solution found by the MP. The two solvers are interleaved and they converge to the optimal solution (if any) for the problem overall. When solving problems with a Benders Decomposition based technique, a number of project choices arises:

- At design level, the objective function (OF) depends either on MP or SP variables, or both. This choice affects the way the two solvers interact.
- Generation of Benders Cuts; Benders Cuts are constraints, added to the MP model once a SP as been solved, that remove some solutions.
- Relaxation of the SP; to avoid the generation of trivially infeasible MP solutions, some relaxations of the SP should be added to the MP model.

We focus on two particular problems, (1) the allocation and scheduling problem on Multi-Processor System-on-Chip (MPSoC) platforms (ASP), we investigated in [2], and (2) the dynamic voltage scaling problem on energy-aware MPSoC (DVSP), we investigated in [3]. These are very hard problems and they have never been solved to optimality by the system design community. We used LBBD to solve the problems.

The aim of this paper is to show the importance of the relaxations and Benders Cuts in terms of their impact on the search time and on the number of times the two solvers iterate. In addition, we claim that, in general, a tradeoff between the complexity of the cuts and relaxations introduced and their impact on the number of iterations must be found.

1 Benders Decomposition

The Benders Decomposition (BD) technique works on problems where two loosely constrained sub-problems can be recognized. Let us consider a problem modelled using two sets of variables x and y. The Benders Decomposition technique solves to optimality the master problem (MP) involving only variables x, producing the optimal solution $\bar{x}$, then it solves the original problem where the variables x values are fixed to $\bar{x}$, namely the sub-problem (SP). Depending on the objective function (OF), two cases can appear: (i) if the OF depends only on variables x, the SP is simply a feasibility problem; if $\bar{x}$ is a feasible solution for the SP, it is the optimal solution for the original problem, otherwise the SP must communicate a no-good saying that $\bar{x}$ is not feasible and another one must be found; (ii) if the OF depends on both x and y variables (or only on y variables), the MP finds an optimal solution w.r.t. its OF (or feasible if the OF depends only on y), $\bar{x}$, then passes the solution to the SP and, when the SP finds an optimal solution w.r.t. its OF, the SP must tell the MP that the solution found is the optimal one unless a better one can be found with a different assignment to variables x. In both cases, the two solvers are interleaved and they converge to the optimal solution (if any) for the problem overall.

To avoid the inefficient *generate and test* behaviour of the MP and the SP interaction it is useful to add to the MP a relaxation of the SP. The relaxation provides a lower bound (or an upper bound if it is a maximization problem) on the SP optimal solution.

The original BD technique models both the MP and the SP using Integer Linear Programming (IP), while LBBD [4] extends BD to cope with any solver. In [5] LBBD is applied to planning and scheduling problems. A set of activities must be assigned and scheduled on a given set of homogeneous facilities. The allocation master problem is modelled using an IP approach, while the scheduling sub-problem is modelled using Constraint Programming (CP). Once the allocation problem is solved, the scheduling part becomes easier since the scheduling problem does not contain alternative resources. Precedence constraints are posted only among activities allocated to the same facility, so the scheduling SP can be decomposed in a number of simpler one machine scheduling problems, one for each facility. In both our problems the scheduling does not decompose since precedence constraints link tasks that possibly run on different processors.

2 Problem Description

We describe here the two problems we faced in [2] and [3].

Problem 1: Allocation and scheduling problem on a MPSoC (ASP)

- Given a set of tasks $t_1 \dots t_n$, with duration $d_1 \dots d_n$ and memory requirements $s_1 \dots s_n$ for the internal state, $pd_1 \dots pd_n$ for the program data and $c_1 \dots c_n$ for communication,
- given precedences and communications among tasks, and realtime constraints imposing deadlines on tasks and processors,

- given an MPSoC platform [7], where a set of homogeneous processors $p_1 \dots p_m$ each with a local memory slot, a system bus and a remote memory are integrated on the same chip,
- ◇ find an allocation of tasks to processors and of memory requirements to storage devices such that the total communication on the system bus is minimized. We have a contribution to the OF each time a memory requirement is allocated on the remote memory and each time two communicating tasks execute on different processors.

Problem 2: Allocation, scheduling and voltage selection problem on an energy-aware MPSoC (DVSP)

- given an energy-aware MPSoC platform [6], where a set of homogeneous processors able to change their frequency and a system bus are integrated on the same chip,
- Given a set of tasks $t_1 \dots t_n$, each annotated with a tuple of durations $\{d_{11} \dots d_{1f}\} \dots \{d_{n1} \dots d_{nf}\}$ (one for each processor speed) and a communication requirement $c_1 \dots c_n$,
- given precedences and communications among tasks, and realtime constraints imposing deadlines on tasks and processors,
- given time and energy overhead for a processor to switch from a frequency to another,
- ◇ find an allocation of tasks to processors and of frequency to task executions such that the total power consumption is minimized. We have a contribution to the objective function each time an activity (task or communication) is performed and each time two activities running at different speeds are scheduled one just after the other on the same processor.

In both problems, we model and solve the allocation using an Integer Programming approach, while we use Constraint Programming to solve the scheduling SP. We can immediately see that the main difference between the ASP and the DVSP concerns the objective function. In the ASP, when the allocation is done, we know all the contributions to the objective function and thus the SP is simply a feasibility problem. In the DVSP instead the OF depends on both the MP and the SP.

3 Improving the models

3.1 Generation of Logic-based Benders cut

In the following we describe the Benders Cuts used.
ASP: A no-good is generated when the optimal solution of the MP is not feasible for the SP. We investigated two no-goods.

– We have variables X_{ij} that assume the value 1 if task i is allocated to processor j, 0 otherwise. The no-goods impose that for each set of tasks S_p allocated to a processor p, they should not be all reassigned to the same processor in the next iteration. The resulting no-good is $\sum_{p=1}^{m} \sum_{i \in S_p} X_{ip} < n$.

– The cuts described above remove only complete solutions. It is possible to refine the analysis and to find tighter cuts that remove only the allocation of tasks to bottleneck resources. So, when a SP failure occurs, we solve a one machine scheduling for each processor p considering constraints involving only tasks running on p. For each processor p where the problem is infeasible, we generate the cut $\sum_{i \in S_p} X_{ip} < |S_p|$. Finding this cut is a NP-hard problem, but we will show experimentally when it pays off.

DVSP: Here the OF depends on both MP and SP. If there is no feasible schedule given an allocation, the cuts are the same computed for the ASP. If the schedule exists we have to produce a cut stating that the one just computed is the optimal solution unless a better one exists with a different allocation. These cuts produce a lower bound on the setup of single processors. The cuts can therefore be of two types:

– We have variables X_{tpf}, taking value 1 if task t executes on processor p at frequency f. Let us consider J_p the set of couples (Task, Frequency) allocated to processor p. No-goods are the following: $\sum_{(t,f) \in J_p} X_{tpf} < |J_p|, \forall p$.
– Suppose a SP solution has an optimal setup cost $Setup^*$. It is formed by independent setups, one for each processor $Setup^* = \sum_{p=1}^{m} Setup_p^*$. We have a bound on the setup LB_{Setup_p} on each processor and therefore a bound on the overall setup $LB_{Setup} = \sum_{p=1}^{m} LB_{Setup_p}$. The constraints introduced in the master problem are: $Setup_p \geq LB_{Setup_p}$, and $LB_{Setup_p} = Setup_p^* - Setup_p^* \sum_{(t,f) \in J_p} (1 - X_{tpf})$, where J_p has the same meaning introduced above.

The cuts described remove only one allocation. Indeed, we have also produced cuts that remove some symmetric solutions.

3.2 Relaxation of the subproblem

In the MP models, deadlines are not taken into account, so the simplest kind of relaxation is based on the tasks execution times. The sum of the execution times of all the activities (tasks and communications) allocated to the same processor must not exceed the deadline. The deadline constraint can still be violated during the scheduling, but a huge number of infeasible solutions is surely cut.

In the **DVSP** this procedure can be improved by adding other relaxations expressing bounds on the setup cost and setup time in the master problem based only on information derived from the allocation. Let us consider, for each processor, the set of frequencies appearing at least once. A bound on the sum of the energy spent during the frequency switches can be computed as follows: let us introduce in the model variables Z_{pf} taking value 1 if the frequency f is allocated at least once on the processor p, 0 otherwise. Let us call E_f the minimum energy for switching to frequency f, i.e. $E_f = min_{i,i \neq f}\{E_{if}\}$. $Setup_p \geq \sum_{f=1}^{M} (Z_{pf} E_f - max_f\{E_f | Z_{pf} = 1\})$. This bound helps in reducing the number of iterations between the master and the subproblem. Similarly, we can compute a bound on the setup time to tighten the constraints involving deadlines described above.

4 Experimental Results

We have generated 500 DVSP and 400 ASP realistic instances, with the number of tasks varying from 7 to 19 and the number of processors from 3 to 10. We consider applications with a pipeline workload. We assume for the DVSP that each processor can run at three different frequencies. All the considered instances are solvable and we found the proved optimal solution for each of them. Experiments were performed on a 2.4GHz Pentium 4 with 512 Mb RAM. We used ILOG CPLEX 8.1, ILOG Solver 5.3 and ILOG Scheduler 5.3 as solving tools.

4.1 Algorithm performances

In [2] and [3] we compared the hybrid approaches with pure approaches modelling the problem as a whole using only IP or CP. For the ASP we found that the pure approaches search times are order of magnitude higher w.r.t. the hybrid, while for the DVSP the pure approaches are not able to find even a feasible solution within the time limit. In this section we will show the effectiveness of the cuts used. We consider ASP and DVSP instances with task graphs representing a pipeline workflow. Note that here, since we are considering applications with pipeline workload, if n is the number of tasks to be allocated, the number of scheduled tasks is n^2, corresponding to n iterations of the pipeline. Results are summarized in Table 1 for the ASP and in Table 2 for the DVSP. The first three rows contain respectively the number of tasks allocated and scheduled and the number of processors considered in the instances. The last two rows represent respectively the search time and the number of iterations. Each value is the mean over all the instances with the same number of tasks and processors. We can see that for all the DVSP instances the optimal solution can be found within four minutes and the number of iterations is typically low. For the ASP instances the optimal solution can be found within one minute and the mean number of iterations is very close to 1.

To show the effectiveness of the relaxations used for the DVSP we solved the instances considering either both or only one of the two relaxations described in 3.2. Table 3 shows the percentage of occurrence of a given number of iterations when solving the DSVP with different relaxations. Using both of them (row All) we can see that the optimal solution can be found at the first step in one half of the cases and the number of iterations is at most 5 in almost the 90% of cases. We tried to solve the problems using only one relaxation; rows Time and Bound show the results when considering only the relaxation on the deadlines and on the SP OF lower bound respectively. We can see that, for most of the cases, the number of iterations is higher than 10. In addiction, the search time on average rises up to 1 order of magnitude and, in the worst cases, the solution cannot be found within two hours.

To show the effectiveness of the cuts used for the ASP, we selected a hard ASP instance with 34 activities and we solved it with different deadline values, starting from a very weak one to the tightest one. Table 4 shows the number of iterations when solving these instances respectively without (row Base) and with (row Advanced) the second kind of cuts described in 3.1 for descending deadline values (row Deadline). We can see that, when the number of iterations is high, the cuts reduce them notably. These cuts are extremely tight, but the time to generate them is one order of magnitude greater w.r.t. the time to generate the Base cuts, therefore they are helpful only on hard instances.

We tried to introduce tighter cuts and relaxations, but we experimentally see that the computation time increases. This is because the cuts and the relaxations complicate the model too much. In general, a tradeoff between the complexity of the cuts and the reduction in terms of iterations must be found.

Alloc	7	7	9	9	11	11	11	13	13	15	15	15	17	17	19	19	19
Sched	49	49	81	81	121	121	121	169	169	225	225	225	289	289	361	361	361
Procs	3	4	4	5	4	5	6	5	6	5	6	7	6	7	4	7	9
Time(s)	0,42	0,41	0,50	0,57	0,60	0,85	1,26	2,84	6,14	0,98	9,53	14,37	7,71	9,25	3,85	27,85	46,69
Iters	1,01	1,05	1,01	1,07	1,06	1,09	1,10	1,08	1,09	1,03	1,07	1,12	1,11	1,02	1,03	1,06	1,11

Table 1. Search time and number of iterations for ASP instances

Alloc	7	7	9	9	11	11	11	13	13	15	15	15	17	17	19	19	19
Sched	49	49	81	81	121	121	121	169	169	225	225	225	289	289	361	361	361
Procs	3	4	4	5	4	5	6	3	7	4	5	7	5	6	3	6	10
Time(s)	1,43	2,24	5,65	6,69	15,25	2,17	2,14	5,90	34,53	12,34	22,65	51,07	60,07	70,40	3,07	120,1	209,4
Iters	2,91	3,47	4,80	3,41	4,66	4,50	3,66	1,90	6,34	4,45	10,53	6,98	7,15	9,20	1,96	6,23	10,65

Table 2. Search time and number of iterations for DVSP instances

Iter	1	2	3	4	5	6	7	8	9	10	11+
All	50,27	18,51	7,11	4,52	4,81	2,88	2,46	2,05	1,64	1,64	4,11
Time	35,23	10,32	3,47	4,76	3,12	2,84	2,13	2,06	1,04	1,11	33,92
Bound	28,6	10,12	5,64	3,78	4,35	2,91	1,29	1,48	1,12	0,84	39,87

Table 3. Number of iterations distribution ratio with different relaxations

Deadline	1000000	647824	602457	487524	459334	405725	357491	345882	340218	315840	307465
Base	3	1	1	18	185	192	79	6	4	2	2
Advanced	3	1	1	6	16	23	17	4	3	3	2

Table 4. Number of iterations varying the deadline and with different Benders Cuts

References

1. J. F. Benders. Partitioning procedures for solving mixed-variables programming problems. *Numerische Mathematik*, 4:238–252, 1962.
2. L. Benini, D. Bertozzi, A. Guerri, and M. Milano. Allocation and scheduling for mpsocs via decomposition and no-good generation. In *Proceedings of CP 2005*, pages 107–121, 2005.
3. L. Benini, D. Bertozzi, A. Guerri, and M. Milano. Allocation, scheduling and voltage scaling on energy aware mpsocs. In *Proceedings of CPAIOR2006*, 2006.
4. J. N. Hooker. A hybrid method for planning and scheduling. In *Procs. of the 10th Intern. Conference on Principles and Practice of Constraint Programming - CP 2004*, pages 305–316, Toronto, Canada, Sept. 2004. Springer.
5. J. N. Hooker. Planning and scheduling to minimize tardiness. In *Procs. of the 11th Intern. Conference on Principles and Practice of Constraint Programming - CP 2005*, pages 314–327, Sites, Spain, Sept. 2005. Springer.
6. M. Ruggiero, A. Acquaviva, D. Bertozzi, and L. Benini. Application-specific power-aware workload allocation for voltage scalable mpsoc platforms. In *2005 International Conference on Computer Design*, pages 87–93, 2005.
7. W. Wolf. The future of multiprocessor systems-on-chips. In *In Procs. of the 41st Design and Automation Conference - DAC 2004*, pages 681–685, San Diego, CA, USA, June 2004. ACM.

Backdoors, Backbones and Clause Learning: Towards Direct Backdoor Search

Student: Peter Gregory
Supervisors: Derek Long
Maria Fox
University of Strathclyde
Glasgow, UK
firstname.lastname@cis.strath.ac.uk

Abstract

Is it possible to exploit backdoors directly? If it is then a good characterisation of backdoors is required. One way to find this characterisation is through empirical analysis of different features that could predict backdoor membership. This work describes some preliminary steps required in this analysis. Backdoor distributions, the relationship with backbones and the effect of clause learning are discussed.

1 Introduction

A backdoor of a CSP problem is a subset of the variables that make the rest of the problem solvable in polynomial time. It is typical that the backdoor sizes of CSPs are small. If there were a way of characterising backdoors then searching directly for them could be an effective new form of search. This work describes the preliminary steps required to achieve this.

The backbone of a CSP is the set of variables that take the same assignments in every solution. Previous work has shown that backdoor membership is negatively correlated with backbone membership [2]. Further insight into why this is the case is discussed here.

This work is motivated by the desire to exploit backdoor variables directly in order to improve search. This work describes the preliminary steps of this investigation: finding an effective characterisation of backdoor variables. The relationship with backbones is discussed as work done in the past regarding backbone membership could possibly be reused to find backdoor variables.

2 Preliminaries

A constraint satisfaction problem (CSP) P is defined as a triple, $(X, \mathcal{D}, \mathcal{C})$. X is a finite set of n variables, $X = \{x_1, x_2, ..., x_n\}$. $\mathcal{D}$ is a finite set of domains, $\mathcal{D} = \{D(x_1), D(x_2), ..., D(x_n)\}$, such that $D(x_i) = \{v_{i_1}, v_{i_2}, ..., v_{i_m}\}$ is the finite set of possible values for variable x_i and $\mathcal{C}$ is the set of constraints $\mathcal{C} = \{C_1, C_2, ..., C_m\}$. A constraint C_i is a relation over a subset of the variables $S_i \subseteq X$ that represents the assignments to the variables in S_i that are legal simultaneously. If $S_i = \{x_{i_1}, ..., x_{i_l}\}$, then $C_i \subseteq D_{i_1} \times ... \times D_{i_l}$.

An assignment to a variable is a pair $\langle x_i, v\rangle$ such that $(v \in D(x_i))$, meaning variable x_i is assigned the value v. A solution S to a CSP P is a set of assignments $S = \{\langle x_1, v_1\rangle, \langle x_2, v_2\rangle, ..., \langle x_n, v_n\rangle\}$, such that all constraints in $\mathcal{C}$ are satisfied. A partial assignment V_S is a set of assignments to variables in $S = \{x_{i_1}, ..., x_{i_l}\}$ such that $S \subset X$. We also use the notation $P[v/x]$ to represent the simplified CSP under the assignment $\langle x, v\rangle$ and $P[V_S]$ to represent the simplified CSP under the partial assignment V_S.

2.1 Backdoors

A sub-solver is an algorithm that solves a tractable subproblem of a general problem class. Paraphrasing Garey and Johnson [3], a subproblem of the general CSP is obtained whenever we place additional restrictions on the allowed instances of the general CSP problem class. A sub-solver A_Π is an polynomial time algorithm that determines only problem instances of subproblem $\Pi \subset CSP$.

Given a sub-solver A, a weak backdoor B_w is a set of variables $B_w = \{x_{i_1}, ..., x_{i_l}\}$ such that there is at least one assignment b_w to B_w such that A determines $S[b_w]$ satisfiable. A strong backdoor B_s is a set of variables $B_s = \{x_{i_1}, ..., x_{i_l}\}$ such that *for every* assignment b_s to B_s, A determines $P[b_s]$ correctly. Note, this definition of a strong backdoor includes strong backdoors for both satisfiable and unsatisfiable CSPs. For unsatisfiable CSPs, every assignment to B_s is determined unsatisfiable by A. For satisfiable instances, at least one assignment to B_s is determined satisfiable by A, and all of the assignments not determined satisfiable are correctly determined as unsatisfiable (and never rejected).

2.2 SAT and Unit Propagation

Boolean Satisfiability (SAT herein) is a special case of CSP. SAT restricts the domains of every variable to two values, $true$ and $false$. It also restricts the constraints to a set of clauses. A clause is a disjunction of literals. The variables correspond to logical variables, and the clauses disjunctions of logical literals, rather than writing $\langle x_i, true\rangle$ and $\langle x_i, false\rangle$, we will use the shorthand x_i and $\neg x_i$ instead. We will also occasionally refer to x_i and $\neg x_i$ as being in positive and negative *phase* respectively.

A clause with only one literal x_i is called a unit clause. To satisfy the formula, that literal must be true in any solution. This means that the complementary literal $\neg x_i$ can be pruned from any clause (as $\neg x_i$ cannot be made true). This procedure can run iteratively, if some pruned clause(s) have now been reduced to unit clauses. This procedure is called Unit Propagation and runs in linear time. In DPLL [4, 5], it is used in conjunction with backtracking search to solve SAT problems. DPLL forms the basis of many modern SAT solving algorithms.

3 Distribution of Backdoors

This section of the work concentrates on the way backdoors are spread across the variable space in different types of SAT problems. The problem instances we will study are encoded from several different problem domains. We look at planning problems,

Algorithm 1 MINIMAL BACKDOOR

```
1: $candidate \leftarrow X$
2: $member \leftarrow \emptyset$
3: while $candidate \neq \emptyset$ do
4:     $c \leftarrow x_i, x_i \in candidate$
5:     $candidate \leftarrow candidate \setminus c$
6:     if $b\_dpll(candidate \cup member)$ did not determine $P$ then
7:         $member \leftarrow member \cup \{c\}$
8:     end if
9: end while
10: return $member$
```

graph colouring problems, quasigroup completion problems and random 3SAT problems (from the phase trasnsition region). These were all selected from the satlib web resource [7] for their variety in structure.

The algorithm MINIMAL BACKDOOR reduces a candidate minimal backdoor until there are no variables that can be removed from the candidate that yield a backdoor. The entire set of variables is trivially a backdoor, and this is what the candidate is initialised to. Minimisation is achieved by simply removing each variable, in random order, and testing if the remaining structure is a backdoor using a depth-bounded DPLL search (written *b_dpll* in the algorithm). If it is, then the variable is not part of the minimal backdoor, and is discarded. If not, then the variable is reintroduced into the candidate. This is similar to the MINWEAKBACKDOOR algorithm in [2]. The difference being, that algorithm used *literals* and not variables as the constituents of the backdoor. This means that different instantiation of the variables in their backdoors could give a smaller *weak* backdoor. It also means that MINWEAKBACKDOOR cannot detect (or minimise) *strong* backdoors.

Algorithm 1 is simple in approach, and is only useful for reasonably small problems. However, for our study Algorithm 1 has advantages over methods previously employed to find backdoors [1, 2]. Previous work has relied on the assignments made by a SAT solver when finding solutions. A modified version of the *satz-rand* solver that outputs the chosen decision variables is the method used to find backdoors. This approach will find backdoors admitted by the randomised heuristic of *satz-rand*. But we are not interested in just those backdoors, we are interested in general backdoors. Algorithm 1 provides a method of finding backdoors that is independent of any solver and any variable ordering heuristic.

The results of finding 100 different backdoors using MINIMAL BACKDOOR for our test instances are given in Table 1. As has previously been noted, backdoors are typically only a small fraction of the total variables. It is interesting to note the sizes of backdoor as a proportion of variables differs greatly between problem domain. The backdoors in the unstructured random 3SAT instances have particularly large backdoors compared to the planning and quasigroup completion problems.

An important question, indeed a critical one, is whether some variables occur in many backdoors whilst others occur in few, if any. We can look at the frequencies that variables occur in the backdoors we found in Table 1. In the blocksworld problem bw-medium for instance, there were 75 variables out of 116 that occurred in any backdoor. The most frequent variable found occurred in 11% of all backdoors. The 25 most frequent variables occurred in an average of 6.1% of all backdoors found. There are also 41 variables in this problem that never occur in a backdoor. These facts together show

Problem	$\lvert X\rvert$	Med$\mathcal{B}$	$\lvert I\rvert$	Med$\lvert B \cap I\rvert$	Med$\mathcal{B}_C$
bw-medium	116	2	97	0	1
bw-huge	459	3	459	2	1
qg1-07	343	5	189	0	2
qg2-07	343	5	169	0	3
qg7-09	729	2	505	0	1
flat30-50	90	5	0	0	4
flat75-5	225	12	0	0	11
uf75-05	75	8	68	2	5
uf100-05	100	11	58	1	7

Table 1: Table of statistics for the studied instances. $|X|$ is the number of variables in the instance. Med$\mathcal{B}$ is the median number of variables in the found backdoors. $|I|$ is the size of the backbone. Med$|B \cap I|$ is the median number of variables in the found backdoors that are also in the backbone. Med$\mathcal{B}_C$ is the median number of backdoor variables when clause learning is turned on.

that indeed, for this instance, there are particular variables that tend to be "backdoor variables".

All of the variables in flat-30-50 occurred in at least one backdoor. The most frequent variable occurred in 10% of the backdoors. The least frequent occurred in just 2% of the backdoors. The frequencies between the two extreme vary much more uniformly than the blocksworld example. In spite of this fact, it is still the case that certain variables occur more often in backdoors five times more often than others.

4 The Backbone and Backdoors

The backbone I_P is the set of variables in X that have a fixed value in every solution. That is, $I_P \subseteq X$ is the backbone of a CSP P if there is a partial assignment X_{I_P} such that $P[X_{I_P}]$ is satisfiable and I_P is the maximum set with this property. Note, we assign the letter I to denote the backbone as it is a solution invariant. Recall that we have defined $P[X_S]$ as the CSP P simplified under partial assignment $X_S \subset X$. The backbone of a CSP simplified by a partial assignment is given by $I_{P[X_S]}$ and will be referred to as the *augmented backbone*. This is because as variables are assigned, the backbone of the CSP grows monotonically. If an augmented backbone $I_{P[X_S]} = X$ then the partial assignment $P[X_S]$ is a *unique solution identifier*. That is, there is only one value that each variable in X can take once the partial assignment $P[X_S]$ is made.

It has been previously observed that backdoor variables are not often backbone variables [2]. There is occasionally an intersection between the two structures, but it appears accidental. So a better question is: what is the reason that backbones and backdoors appear to be (typically) disjoint? Let us start by making some observations.

If all of the backbone variables are set correctly, could this be a backdoor? No. The backbone variables are those whose assignments are implied by the problem. Thus, if setting the backdoor correctly implied another variable/value assignment, this other variable *must be* in the backbone also. Once we have this piece of information, we can see that partial/ full assignments to backbone variables only have the capacity to imply *other backbone variables*. Since a backdoor implies *every* variable's value for a given solution, the backbone variables cannot be a backdoor.

As variables are assigned in search, the sub-spaces that we move into have monotonically growing backbones. Indeed, when a problem is solved a problem using assignment and propagation, all of the variables are trivially in the augmented backbone (as in the final state all variables are set). Since we have shown backbone variables can only imply themselves, it is true that in any sub-space of the search tree, the next choice should not be in the augmented backbone, as this can't imply any variables other than those already in the augmented backbone.

4.1 Identifying Unique Solutions

When search is in a state where all variables are in the augmented backbone, then there is a single solution (in that sub-space). This doesn't mean that search is necessarily complete, some problems with single solutions are hard to solve. But it does mean search is at the stage where unit propagation *may* be able to solve the problem, because there is now a single solution in our sub-space.

So, we have a necessary, but not sufficient, property of any backdoor – *assignment of part of the backdoor must identify a unique solution.* The next enquiry naturally concerns the question: how is the remainder of the backdoor composed? In this situation, several variables have been assigned such that, in the current sub-space, there is a unique solution to the studied instance (but the problem is not solved). Our problem now must be that we don't have enough information in the current clauses to cause propagation of the remaining variables. We can infer information using clause learning.

5 Clause Learning and Backdoors

When searching for backdoors, we disabled the clause learning features of the SAT solver. This is because clause learning "interferes" with backdoors. The effect it has on backdoors is to make them smaller. In terms of search, this is obviously a desirable effect. We performed the same experiments as before (finding 100 backdoors for our test cases) whilst leaving clause learning turned on. In every case, the median backdoor size found was reduced by at least one, and as many as four in the case of the *uf100-05* instance. Although it may seem a small reduction, a linear reduction in the size of the average backdoor means there is an exponential increase in the number of backdoors to the problem.

6 Conclusions and Future Work

If backdoors are to be exploited directly, then we must be able to identify good candidate members. This work shows that if a good approximation to the backbone is available, then we can reduce the choice for backdoor variables. It also shows that clause learning can have a positive effect on backdoors, it can reduce their size. This is because it adds useful information to the problem such that more of the augmented backbone can be propagated at each point in search.

Future work will entail rigourous analysis of more properties that may indicate backdoor membership. These properties will be taken from different variable ordering heuristics, structural properties such as ratio between positive and negative phase literals, and other measures that could prove indicative of backbone membership. Once that analysis is complete, a combination of the features will be used to predict backdoors and will form the basis of a search algorithm.

References

[1] Williams, R., Gomes, C., Selman, B.: Backdoors to typical case complexity. In Gottlob, G., Walsh, T., eds.: Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Morgan Kauffman (2003) 1173–1178

[2] Kilby, P., Slaney, J., Thiebaux, S., Walsh, T.: Backbones and backdoors in satisfiability. In: Proceedings of AAAI-2005. (2005)

[3] Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)

[4] Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM **7**(3) (1960) 201–215

[5] Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. Communications of the ACM **5**(7) (1962) 394–397

[6] Beame, P., Kautz, H., Sabharwal, A.: Understanding the power of clause learning. In Gottlob, G., Walsh, T., eds.: Proceedings of the 18th International Joint Conference on Artificial Intelligence, Morgan Kauffman (2003) 1194–1201

[7] Hoos, H.H., Sttzle, T.: SATLIB: An Online Resource for Research on SAT. In I.P.Gent, H.v.Maaren, T.Walsh, eds.: Proceedings of the Third International Conference on the Theory and Applications of Satisfiability Testing, IOS Press (2000) 283–292

[8] Dechter, R.: Constraint Processing. Morgan Kauffman (2003)

New Propagators for the SPREAD Constraint

Students: Pierre Schaus, Jean-Noël Monette.
Supervisors: Yves Deville, Pierre Dupont.

Computer Sciences and Engineering Department (INGI)
Université catholique de Louvain
Place Sainte-Barbe, 2
1348 Louvain-la-Neuve
{pschaus, jmonette, yde, pdupont}@info.ucl.ac.be

1 Introduction

In assignation problems, it is often desirable to have a fair or balanced solution. One example of such a problem is BACP. The goal is to assign periods to courses such that the academic load of each period is balanced, i.e., as similar as possible [1]. A perfectly balanced solution is generally not possible. A standard approach is to include the balance property in the objective function. Alternatively the constraint *SPREAD* introduced by Pesant and Régin [2] could be used to reduce the search tree while simplifying the model. Constraining the variance of assignments to fall below an upper bound is a proper way to enforce the balance property.

Given a set of variables X and two variables μ and σ, $SPREAD(X, \mu, \sigma)$ states that the collection of values taken by the variables of X exhibits an arithmetic mean μ and a standard deviation σ. While the *SPREAD* constraint in [2] also involves the median, this will not be considered here. We present a simplified version of the propagator from σ and μ to X. We also introduce a propator to narrow the upper bound of σ which is missing in [2].

Section 2 introduces some statistical background, definitions of constraint programming and explains the results from [2] that will be used. Then, section 3 describes our propagator from μ and σ on the domains of the set of variables X and finally section 4 describes the propagator to lower the upper bound of the standard deviation interval from the set of variables X.

2 Background [2]

In this section we introduce some background and definitions necessary to understand the rest of the article. We also present the results from [2] we use in our propagator. We assume the reader familiar with common statistical notions such as *mean*, *standard deviation* and *variance*. Note simply that a convenient way to compute the variance of a set of values $X = \{x_1, x_2, ..., x_n\}$ is the following: $\sigma^2 = \left(\frac{1}{n}\sum_{i=1}^{n} x_i^2\right) - \mu^2$.

We use the following notations for the variables and domains considered in this paper:

– A finite-domain (discrete) variable x takes a value in $D(x)$, a finite set called its domain. We denote the smallest (resp. largest) value x may take as $x^{\min}$ (resp. $x^{\max}$).
– A bounded-domain (continuous) variable y takes a value in $I_D(y) = [y^{\min}, y^{\max}]$, an interval on $\mathbb{R}$ called its domain as well.
– Given a finite-domain variable x, $I_D(x)$ denotes its domain relaxed to the continuous interval $[x^{\min}, x^{\max}]$. By extension for a union of domains $\mathcal{D} = \bigcup_{i=1}^{n} D(x_i)$, $I_\mathcal{D}$ represents the interval $[\min_{i=1}^{n} x_i^{\min}, \max_{i=1}^{n} x_i^{\max}]$.

To narrow the variables in X, we need a way to find the minimum possible variance with a fixed mean. [2] explains how to solve this optimization problem.

Definition 1 (Minimization of the variance on X). *Let $X = \{x_1, x_2, ..., x_n\}$ be a set of finite-domain (discrete) variables. For some fixed number q we denote by $\mathit{\Pi}_1(X, q)$ the problem:* $\min \sum_{i=1}^{n}(x_i - q/n)^2$ *such that* $\sum_{i=1}^{n} x_i = q$, $x_i \in I_D(x_i)$, $1 \leq i \leq n$ *and we denote by* $opt(\mathit{\Pi}_1(X, q))$, *or simply* $opt(\mathit{\Pi}_1)$, *the optimal value to this problem.*

In the above definition, $opt(\mathit{\Pi}_1)$ corresponds to n times the minimal variance and q to n times μ.

Definition 2. *An assignment* $A : x \rightarrow I_D(x)$ *over* X *is said to be a* v*-centered assignment when* $A(x) = x^{\max}$ *if* $x^{\max} \leq v$, $A(x) = x^{\min}$ *if* $x^{\min} \geq v$ *and* $A(x) = v$ *otherwise.*

Lemma 1 ([2]). *Any optimal solution to* $\mathit{\Pi}_1(X, q)$ *is a* v*-centered assignment.*

Lemma 1 gives a necessary condition for an assignment to be optimal for $\mathit{\Pi}_1(X, q)$ but the v value can be anywhere in $I_\mathcal{D}$. [2] introduces a splitting of $I_\mathcal{D}$ into intervals. Any such interval is either included in a domain or has an empty intersection with it but no partial overlap occurs. This splitting simplifies the problem of finding where the optimal v lies within $I_\mathcal{D}$.

Definition 3. *Let* $B(X)$ *be the sorted sequence of bounds of the relaxed domains of the variables of* X*, in non-decreasing order and with duplicates removed. Define* $\mathcal{I}(X)$ *as the set of intervals defined by a pair of two consecutive elements of* $B(X)$*. The* k^{th} *interval of* $\mathcal{I}(X)$ *is denoted by* I_k*. For an interval* $I = I_k$ *we define the operator* $prev(I) = I_{k-1}, (k > 1)$.

They are at most $2.n - 1$ intervals in $\mathcal{I}(X)$. Let assume that the value v of the optimal solution to $\mathit{\Pi}_1(X, q)$ lies in the interval $I \in \mathcal{I}(X)$. We denote by $R(I) = \{x | x^{\min} \geq \max(I)\}$ the variables lying to the right of I and by $L(I) = \{x | x^{\max} \leq \min(I)\}$ the variables lying to the left of I. By lemma 1, all variables $x \in L(I)$ take their value $x^{\min}$ and all variables in $R(I)$ take their value $x^{\max}$. It remains to assign the variables overlapping I. We denote these variables by $M(I) = \{x | I \subseteq I_D(x)\}$ and the cardinality of this set by $m = |M(I)|$. By lemma 1, the variables of $M(I)$ must take a common value v. The sum constraint of $\mathit{\Pi}_1(X, q)$ can be rewritten as $\sum_{x \in R(I)} x^{\min} + \sum_{x \in L(I)} x^{\max} + \sum_{x \in M(I)} v = q$.

Let denote the sum of extrema by $ES(I) = \sum_{x \in R(I)} x^{\min} + \sum_{x \in L(I)} x^{\max}$. The sum constraint from Π_1 implies that v must be equal to $v^* = (q - ES(I))/m$. This results in a valid assignment only if $v^* \in I$. This condition is satisfied if $q \in V(I) = [ES(I) + \min(I).m, ES(I) + \max(I).m]$.

By defining $\underline{S}(X) = \sum_{x \in X} x^{\min}$ and $\overline{S}(X) = \sum_{x \in X} x^{\max}$, an important property concerning $\mathcal{I}(X)$ and the definition of $V(I)$ is $\min(V(I_1)) = \underline{S}(X)$, $\max(V(I_{|\mathcal{I}(X)|})) = \overline{S}(X)$ and for two consecutive intervals I_k, I_{k+1} from $\mathcal{I}(X)$, we have $\min(V(I_{k+1})) = \max(V(I_k))$, thus leaving no gap.

Given a value q such that $q \in [\underline{S}(X), \overline{S}(X)]$ and $I^q \in \mathcal{I}(X)$ such that $q \in V(I^q)$, the following assignment gives the optimal value to $\Pi_1(X, q)$:

Definition 4. *$A_{I^q}(x) = x^{\max}$ for $x \in L(I^q)$, $A_{I^q}(x) = x^{\min}$ for $x \in R(I^q)$ and $v = A_{I^q}(x) = (q - ES(I^q))/m$ for $x \in M(I^q)$.*

Example 1. Let $X = \{x_1, x_2, x_3\}$ with $I_D(x_1) = [1, 3]$, $I_D(x_2) = [2, 6]$ and $I_D(x_3) = [3, 9]$ then $\mathcal{I}(X) = \{I_1, I_2, I_3, I_4\}$ with $I_1 = [1, 2]$, $I_2 = [2, 3]$, $I_3 = [3, 6]$, $I_4 = [6, 9]$. $M(I_2) = \{x_1, x_2\}$, $L(I_2) = \phi$, $R(I_2) = \{x_3\}$, $ES(I_2) = 3$ and $V(I_2) = [7, 9]$. Similarly, $V(I_1) = [6, 7]$, $V(I_3) = [9, 15]$ and $V(I_4) = [15, 18]$. For $q = 10$ we have $q \in V(I_3)$ thus $I^{10} = I_3$. $L(I_3) = \{x_1\}$, $M(I_3) = \{x_2, x_3\}$ and $R(I_3) = \phi$. $A_{I_3}(x_1) = 3$, $A_{I_3}(x_2) = A_{I_3}(x_3) = 3.5$. For $q = 9$, we have $q \in V(I_2)$ and $q \in V(I_3)$. Whichever interval we choose between I_2 and I_3, we find the same optimal assignment $A_{I_2}(x_1) = A_{I_3}(x_1) = 3$, $A_{I_2}(x_2) = A_{I_3}(x_2) = 3$ and $A_{I_2}(x_3) = A_{I_3}(x_3) = 3$.

3 Propagation from μ and σ to X

To simplify the presentation, we first assume that σ is an interval $[\sigma^{\min}, \sigma^{\max}]$ and μ is a given value. We will consider afterwards the general case where μ is an interval.

Let us denote $q = n\mu$, $\pi_1^{\max} = n(\sigma^{\max})^2$ and $I^q \in \mathcal{I}(X)$ is such that $q \in V(I^q)$. If $opt(\Pi_1) > \pi_1^{\max}$ the constraint fails because there exists no consistent assignment. Otherwise, for a variable $x \in R(I^q)$ (resp. $\in L(I^q)$) we compute its maximal value (resp. minimal value) and for a variable $x \in M(I^q)$ we compute both. As the problem is symmetrical we only consider the maximal value computation for $x \in R(I^q) \cup M(I^q)$. For these variables we will see that shifting its domain to $x + d$ increases $opt(\Pi_1)$ quadratically. The bound $\pi_1^{\max}$ is reached for $d = d^{\max}$. The propagator considers each variable x in turn, computes its $d^{\max}$ and prunes $D(x) \leftarrow D(x) \cap [x^{\min}, x^{\min} + d^{\max}]$. All the domains can be updated once after consideration of all variables in X. Alternatively, each pruned domain can directly be used for the propagation on the other variables. In either case, the process must be iterated until a fix-point is reached. The simplest approach is the first one and as can be seen in the exemple of Figure 1, the propagation is already effective in this way. The analysis of the $FindDMax$ algorithm described below shows that $d^{\max}$ is computed in $O(n)$ making our propagator running in $O(n^2)$.

Searching $d^{\max}$ for $x \in R(I^q)$ X' denotes X after the shift $x' = x + d$. Let $\Pi_1(X', q)$, $ES'(I^q)$ and $V'(I^q)$ be the corresponding quantities for X'. We have $ES'(I^q) = ES(I^q) + d$ and $V'(I^q) = V(I^q) + d$.

Let assume that $d \leq d_1 = q - \min(V(I^q))$ such that v' remains in I^q. Only the v value will change in the optimal assignment: $v' = v - d/m$. We have $opt(\Pi_1(X', q)) = \left(\sum_{x_i \in L(I^q)} (x_i^{\max})^2\right) + \left(\sum_{x_i \in R(I^q)} (x_i^{\min})^2\right) + d^2 + 2dx^{\min} + \left(\sum_{x_i \in M(I^q)} (v - \frac{d}{m})^2\right) - \frac{q^2}{n} = opt(\Pi_1(X, q)) + d^2 + 2dx^{\min} + m\left(\frac{d^2}{m^2} - 2\frac{d}{m}v\right)$. The value $d^{\max}$ is the positive solution of a second degree equation $ad^2 + 2bd + c$, where $a = (1 + \frac{1}{m})$, $b = x^{\min} - v$ and $c = opt(\Pi_1(X, q)) - \pi_1^{\max}$.

Until now, we made the assumption that $d \leq d_1$. If $d^{\max} > d_1$ this value is not valid since v does not lie within I^q anymore. In this case x is shift by d_1 and the interval $I^{q\prime} = prev(I^q)$ is considered. The resulting Algorithm 1 searching for $d^{\max}$ runs in $O(n)$ since they are at most $|\mathcal{I}(\mathcal{X})| < n$ recursive calls and that the body runs in $O(1)$.

```
Algorithm: FindDMax($x, I^q$)
Data: $x \in R(I^q)$; $I^q \in \mathcal{I}$; $q \in V(I^q)$;
Result: $d^{\max}$ s.t. $opt(\Pi_1(X', q)) = \pi_1^{\max}$ with $x' = x + d^{\max}$
$d_1 = q - \min(V(I^q))$;
$d^{\max} = \frac{-b+\sqrt{b^2-ac}}{a}$;
if $d^{\max} < d_1$ then
    return $d^{\max}$;
else
    if $I^q = I_1$ then
        return $d_1$;
    else
        return $d_1$+FindDMax($x + d_1, prev(I^q)$);
    end
end
```
Algorithm 1: FindDMax

Searching $d^{\max}$ with $x \in M(I^q)$ can be reduced to searching for $d^{\max}$ with a new variable x' with $x'^{\min} = v$. When x is increased ($x' = x + d$), the optimal assignment does not change if $d \leq v - x^{\min}$. For $d = v - x^{\min}$ two new intervals are created replacing the old I^q: $I_j = [\min(I^q), v]$ and $I_k = [v, \max(I^q)]$ with $q = \max(V'(I_j)) = \min(V'(I_k))$. The optimal assignment is the same but a new problem $\Pi_1(X', q)$ is created with $q \in V'(I_j)$ and $x' \in R(I_j)$. This case reduced to searching for $d^{\max}$ with $x' \in R(I_j)$ is exposed above. The final $d^{\max}$ relative to the variable x is given by: $d^{\max} = v - x^{\min} + FindDMax(x', I_j)$ where $x' = x + v - x^{\min}$.

Figure 1 shows an example of the effect of the propagator.

Extension to $\mu = [\mu^{\min}, \mu^{\max}]$ The generalization $\mu = [\mu^{\min}, \mu^{\max}]$ is equivalent to $q \in [q^{\min} = n\mu^{\min}, q^{\max} = n\mu^{\max}]$. This extension does not affect our

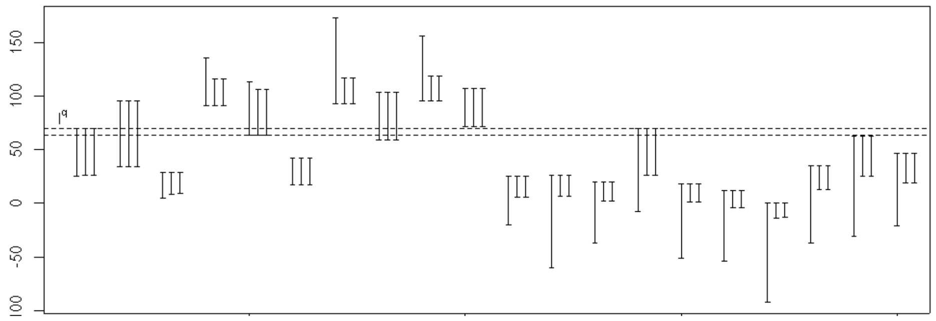


Fig. 1. The propagation on a typical run. The I^q interval lies between the two horizontal lines. The posted constraint is $SPREAD(X, 50, [0, 23])$. There are 20 variables and the domains after one and two propagations are represented on the left of each original domain. We can see that the second propagation does not prune a lot anymore.

propagator but only requires an additional step before the call to $FindDMax$ for each variable: the computation of a suitable $q \in [q^{\min}, q^{\max}]$. The computation of $d^{\max}$ in the algorithm depends on the value of q. To express this explicitly we denote $d^{\max}$ as a function of q: $d^{\max}(q)$. Since it can be shown to be concave and derivable, one can search a q^0 such that $d^{\max}(q)$ is maximum: $\frac{\partial d^{\max}}{\partial q}\big|_{q=q^0} = 0$. It can be shown that q^0 is the only valid solution of a second degree equation . As $q \in [q^{\min}, q^{\max}]$, if $q^0 > q^{\max}$ (resp. $< q^{\min}$) then $FindDmax$ is called with $q = q^{\max}$ (resp. $q = q^{\min}$). If $q^0 \in [q^{\min}, q^{\max}]$, $FindDmax$ is called with $q = q^0$.

4 Propagation from X to $\sigma^{\max}$

To narrow the upper bound of σ we need a way to compute the maximal variance on X such that $\sum_{i=1}^{n} x_i = q$. This can be shown to be a convex maximization problem (NP-hard in general [3]). Even the relaxed problem without the sum constraint remains a convex maximization problem but it is easier to design an upper bound on it because of a known characterization of the optimal solution with respect to the extrema of the domains.

Definition 5 (Maximization of the variance on X). *Let $X = \{x_1, x_2, ..., x_n\}$ be a set of finite-domain (Discrete) variables. We denote by $\Pi_2(X)$ the problem: $\max \sum_{i=1}^{n}(x_i - \sum_{j=1}^{n} x_j/n)^2$. We denote by opt($\Pi_2(X)$) the optimal value for the problem.*

Lemma 2 (Optimal solution to $\Pi_2(X)$). *Any optimal solution to $\Pi_2(X)$ must be an assignment on the extrema of the domains i.e. on $x^{\max}$ or $x^{\min}$.*

They are 2^n possible extrema assignments for X. We denote $\underline{\mu} = \underline{S}(X)/n$ and $\overline{\mu} = \overline{S}(X)/n$. For some variables the optimal assignment can be deduced immediatly. Indeed if $x^{\min} > \overline{\mu}$, an optimal solution to $\Pi_2(X)$ is such that $x = x^{\max}$. The case $x^{\max} < \underline{\mu}$ is symmetrical. There are additional cases where extrema assignment can be deduced. Note that if x would be assigned to $x^{\min}$, the upper bound for μ would become $\overline{\mu}^* = \overline{\mu} - \frac{x^{\max} - x^{\min}}{n}$.

In the example on the left of Figure 2, an optimal solution would assign $x = x^{\max}$ because the lower bound on the distance of $x^{\max}$ to μ is greater than the upper bound on the distance of $x^{\min}$ to $\overline{\mu}^*$. More generally, in each case where the lower-bound using an extremum is larger than the upper-bound using the other extremum, the optimal assignment corresponds to the first extremum.

Assigning a variable x to $x^{\min}$ will decrease $\overline{\mu}$ and assigning a variable x to $x^{\max}$ will increase $\underline{\mu}$ resulting possibly in a larger set of variables for which an optimal assignment can be deduced. All such extrema can be found in $O(n^2)$.

Since $opt(\Pi_2(X)) = \sum_i x_i^2 - \left(\sum_i x_i\right)^2 / n$, an upper bound $\overline{opt}(\Pi_2(X))$ can be computed using the assigned values and $x_i^{\max}$ (resp. $x_i^{\min}$) in the first (resp. second) sum otherwise. This upper bound can be used to narrow the interval σ by posting the constraint $n.\sigma^2 \leq \overline{opt}(\Pi_2(X))$. For the example on the right of Figure 2 with 50 variables, the algorithm find the optimal solution i.e. $\overline{opt}(\Pi_2(X)) = opt(\Pi_2(X))$. The deduced extrema's are indicated with a $\oplus$. The worst case for propagating on σ would correspond to all variables with an identical domain.

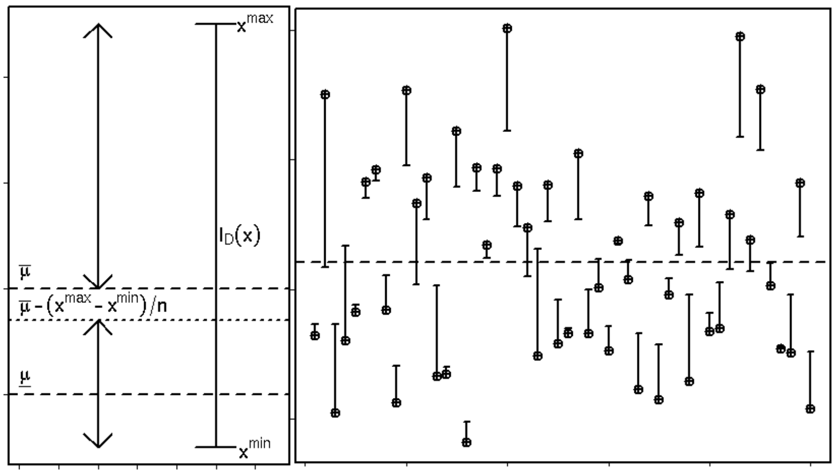


Fig. 2. Left figure: $x = x^{\max}$ because the lower bound on the distance from $x^{\max}$ to $\underline{\mu}$ is smaller than the upper bound on the distance from $x^{\min}$ to μ. Right figure: $\overline{opt}(\Pi_2(X)) = opt(\Pi_2(X))$. The deduced extrema's are indicated with a $\oplus$

References

1. *Problem 30 of CSPLIB (www.csplib.org).*
2. Jean-Charles Regin Gilles Pesant. Spread: A balancing constraint based on statistics. *Lecture Notes in Computer Science*, 3709:460–474, 2005.
3. Lieven Vandenberghe Stephen Boyd. *Convex Optimization*. Cambridge University Press, 2004.

Supervision of robot tasks planning through constraint networks acquisition.

Student: Mathias PAULIN
Supervisor: Jean SALLANTIN

LIRMM (CNRS/University of Montpellier), France,
{paulin, js}@lirmm.fr

Abstract. In recent years, robotic, and more precisely humanoïd robots, have made great improvements, especially in some close links with Artificial Intelligence. However, each time you have to design a new robot, it still exists a specific difficulty. You have, with some specific elementary actions, to combine them in a high-level vision to describe some complex tasks. In this article, we propose a theoretical framework using constraint networks to supervise planning of robot tasks. Each elementary action is modelled with a CSP automatically acquired by Machine Learning. To describe high-level task we only have to sequence some of the multiple acquired CSPs using planning tools. Some experimental results on a one-leg jumping robot have validated the automatically constraint networks modelling process.

1 Introduction

In recent years, robotic have made great improvements as the Sony's AIBO dog [1] or humanoïd robots such ASIMO [2] testify some. Current robots can perform a great number of elementary actions which are automatically executed. However, each time they have to design a new robot, the main problem that roboticians have, consists in linking and combining these elementary actions in order to perform a complex task. An elementary action is modelled by sets of mathematical equations which involve really complex physical laws, such as mechanical energy conservation law, kinetic momentum, partial derivative equations... Combine and link these elementary actions in order to perform a complex task is consequently difficult and requires a lot of computation hours, where the result is often an unique pre-computed sequence of elementary actions.

In this article, we propose to interact with roboticians in the planning process. Our approach is based on elementary actions automatically modelled with constraint networks by Machine Learning (using the CONACQ platform [4, 6]). The aim of the CSP acquisition is to **automatically** obtain a vocabulary $\mathcal{V}$ such that each word $w \in \mathcal{V}$ expresses an elementary action of the robot. The CSP modeling has the double advantage of abstracting a declaratory model from the elementary actions and to have good computational properties which is a capital aspect for planning. Thus, determining a valid sequence of elementary actions consists in building a valid sentence of words from $\mathcal{V}$. To perform high-level task, we sequence these multiple networks using planning tools. With this approach, we wish to relieve the modeling task of roboticians and contribute to the automatic constitution of laws of order for robots.

This paper is organized as follows. Section 2 presents some formal definitions of the basic concepts used in the article. In Section 3 we describe the theoretical framework we propose to tackle tasks planning of robots. Before concluding (Section 5), we present a first experiment which is a first empirical validation of our approach.

2 Preliminaries

2.1 Constraint Programming

A constraint network $\mathcal{P}$ is defined as a triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ where $\mathcal{X} = \{x_1, \ldots, x_n\}$ is a finite set of n variables, $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ is the set of their domains, and where $\mathcal{C} = \{c_1, \ldots, c_m\}$ is a sequence of constraints on $\mathcal{X}$. A constraint c_i is defined by the sequence $var(c_i)$ of variables it involves, and by the relation $rel(c_i)$ specifying the allowed tuples on $var(c_i)$.

The instanciation of values on the variables of $var(c_i)$ *satisfies* c_i if it belongs to $sol(c_i)$. An complete instance e on $\mathcal{X}$ is a tuple $(v_1, \ldots, v_n) \in D(x_1) \times \ldots D(x_n)$. An instance e is a solution of a constraint network if it satisfies all the constraints of the network. Otherwise, it is a non solution.

2.2 Constraint network acquisition

The goal of the constraint network acquisition consists in automatically acquiring a model from solutions and non solutions of the problem we want to model with a constraint network [4, 6, 7]. For the constraint acquisition process, the set $\mathcal{X}$ of the variables and their values domains $\mathcal{D}$ are known, and we dispose of E^+, a subset of solutions of the studied problem, and of E^- a set of non solutions. The goal of the acquisition process is to model the problem in a constraint solver. Our learning bias $\mathcal{B}$ is then a constraint library from this solver.

A constraints sequence belongs to the learning bias if and only if this sequence involves only constraints of the bias library. Given a set of variables $\mathcal{X}$, their domains $\mathcal{D}$, two sets E^+ and E^- of instances on $\mathcal{X}$, and a learning bias $\mathcal{B}$, the constraint acquisition problem consists in finding a set of constraints $\mathcal{C}$ such that $\mathcal{C} \in \mathcal{B}$, $\forall e^- \in E^-$, e^- is a non solution of $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, and, $\forall e^+ \in E^+$, e^+ is a solution of $(\mathcal{X}, \mathcal{D}, \mathcal{C})$.

2.3 Planning problem

Planning[1] consists in finding, among a set of possible actions, which actions have to be applied, and how apply its, in order to attain a goal defined in a environment called World. The STRIPS formalism is commonly used to express planning problems. STRIPS uses the concepts of State, Goal and Action which are defined as follows. A ***State*** is a conjunction of positive literals describing the World at a specific time. A ***Goal*** is a particular state which describes partially the World at a final time. A state s ***satisfies*** a goal g if s contains all the literals in g. An ***Action*** A is defined by its name, its parameters, its *precondition* and its *effect*. The ***precondition*** of A is a conjunction of

[1] In this section, we present the planning problem in its ordinary definition [9].

literals which expresses the set of conditions which must be verified in order to execute A. The ***effect*** of A is a conjunction describing the state changes when A is executed.

Given an initial state S_i, a final state S_f corresponding to a goal, and $\mathcal{A}$ a set of actions, the planning problem consists in finding a sequence of actions from $\mathcal{A}$ such that if all these actions are performed from S_i, they allow to reach a state satisfying S_f.

3 Theoretical framework

The aim of approach we propose in this article consists in supervising quickly and easily the planning of tasks for robots. In this section, we describe the theoretical framework we propose and explain how and why using CSPs is a good alternative to the common roboticians approach.

3.1 General principle

Given a robot able to perform a set $\mathcal{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_m\}$ of elementary actions. An action $\mathcal{A}_i$ is considered as an ***elementary*** action when it is defined in a non decomposable way (in time, in moves, in robotic motor activation, etc.). Given a task which can't be performed with a single elementary action, the roboticians problem consists in combining different elementary actions in order to perform the given task.

For several years, Constraint Programming has been a success growing and is largely used in many industrial applications. The main reason of this success lies in its declaratory aspect and inherent dissociation between a model describing the problem (”What”) and the techniques of resolution used to solve it (”How”). As presented previously, elementary actions are modelled by complex mathematical equations which are very difficult to solve. We propose consequently to model elementary actions by constraint networks using simple constraints which have good computational properties. To guarantee the requirements of speed and simplicity, we will use a set of very simplified constraints which will answer the planning problems and not complex mathematical equations. To determine a plan for performing a specific task, we will then use planning tools and constraints solver to determine a sequence on elementary actions which must be executed in order to perform the task.

3.2 Proposed approach

In the first part of this sub-section, we explain how we model elementary actions with CSPs. The second part presents how we use constraint acquisition techniques to automatically acquire CSPs. Finally, we describe how planning tools allow us to combine and sequence some of the multiple acquired CSPs in order to carry out a high level task.

Modeling elementary actions with CSPs

The approach we propose in this article is a generic one and could be deployed for any poly-articulated robot. In this section, we consider then a robot R which can perform the set $\mathcal{A} = \{\mathcal{A}_1, ..., \mathcal{A}_m\}$ of elementary actions in a specific environment.

The robot R is consisted by a set $\delta = \{d_1, ..., d_n\}$ of descriptors which describe various dimensions of R and its environment, and by the set $\alpha = \{a_1, ..., a_k\}$ of its motors. Let be $\mathcal{A}_i \in \mathcal{A}$ an elementary action of R and $\mathcal{L}$ a constraints library. We propose to model $\mathcal{A}_i$ with CSPs in the STRIPS formalism by modelling three constraint networks $\mathcal{P}_p$, $\mathcal{P}_a$ and $\mathcal{P}_e$ defined as follows:

$$\begin{cases} \mathcal{P}_p = (X_p, \mathcal{D}_p, \mathcal{C}_p) \text{ models the conditions in which } \mathcal{A}_i \text{ can be performed,} \\ \mathcal{P}_a = (X_a, \mathcal{D}_a, \mathcal{C}_a) \text{ models how the motors must react to perform } \mathcal{A}_i, \\ \mathcal{P}_e = (X_e, \mathcal{D}_e, \mathcal{C}_e) \text{ models the state of } R \text{ after the execution of } \mathcal{A}_i. \end{cases}$$

In the previous definition, X_p, X_a and X_e are subsets of α, $\mathcal{C}_p$, $\mathcal{C}_a$ and $\mathcal{C}_e$ are subsets of $\mathcal{L}$, and $\mathcal{D}_p$ (*resp.* $\mathcal{D}_a$, $\mathcal{D}_e$) is the set of domains of the variables from X_p (*resp.* X_a, X_e). It should be noted that this CSP modeling allows to acquire, for each elementary action $\mathcal{A}_i$, a model which abstracts a set of possible instances of $\mathcal{A}_i$. Our approach models a *space* of possible solutions (and not a single one). Combine several elementary actions between them is consequently more easily.

Automatic CSPs building by constraint acquisition

In order to *automatically* model an elementary action $\mathcal{A}_i$, we have choose to use the constraint network acquisition platform CONACQ [4, 6].

For the acquisition process, we have to dispose of a constraints library $\mathcal{L}$, called learning bias, and of two disjoint sets E^+ and E^- which respectively contain positive instances of $\mathcal{A}_i$ and negative ones. This training data, simply given by roboticians, will be analysed by CONACQ which will automatically build (*c.f.* Section 4) a constraint networks $\mathcal{P}$ matching with the constraint acquisition problem. The constraint networks $\mathcal{P}_p$, $\mathcal{P}_a$ and $\mathcal{P}_e$ are then immediatly deduced from $\mathcal{P}$ as included sub-networks of $\mathcal{P}$.

The constraints library $\mathcal{L}$ must answer two partially opposite requirements: *simplicity*, in order to garantee good computationnal properties, and *modeling capability*, in order to efficiently describe each elementary action. The choice of $\mathcal{L}$ will be consequently a capital aspect of the modeling process and will be realized in interaction with roboticians in order to obtain the *best* ratio between simplicity and modeling capability.

Tasks planning

For each elementary action $\mathcal{A}_i \in \mathcal{A}$, we restrict then the acquired model to $(\mathcal{P}_p^i, \mathcal{P}_e^i)$, and express a task T to execute in the STRIPS formalism with the initial state E_i and a goal E_f. Finding a sequence of elementary actions allowing to perform T consists in these conditions in solving the planning problem $(E_i, E_f, \mathcal{A})$.

To solve this planning tasks, we use then planning tools used as black boxes. At the present time, we have not definitively choose the planning algorithm we will use.

According to criteria the roboticians will fix, we will prefer algorithms such as GRAPHPLAN [5], CPLAN [3] or other planning tools. However, we can note that the modelling of preconditions and effects by CSPs garantees strong propagation properties and allows to expect limited computation times for planning tools.

If there exists a plan for performing T, the planner will output a sequence $\mathcal{S} = [\mathcal{A}_i | \mathcal{A}_i \in \mathcal{A}]$ of the elementary actions the robot must apply to perform T.

4 TWIG: A preliminary empirical validation

In collaboration with the LIRMM robotic department, we experimented the automatic elementary actions modeling by CSP acquisition on a one-leg jumping robot named TWIG. Represented on Figure 1, TWIG has been designed by Sebastien Krut from [8]. Figure 1 is the result of the modeling of TWIG with the 3D Computer Aided Design software SOLIDWORKS [10]. We ran experiments on COSMOSMOTION, the physical simulator of SOLIDWORKS.

TWIG is constituted by a flywheel which is used to stay in vertical position, by a jack which TWIG uses to perform a longitudinal thrust, and by helicoid spring used to absorb a landing.
To describe TWIG, we have used the following set of descriptors: U_R and U_θ are the electrics powers respectively applied to the jack motor and the flywheel. R is the distance between the movable part of the jack and the fix one. $\ddot{x}_G$ and $\ddot{y}_G$ are the accelerations of the gravity point G, and x_G, y_G its coordinates. s is the foot/ground contact.

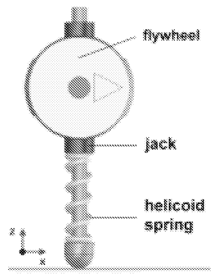


Fig.1 The TWIG robot.

For this preliminary experiment, we choose to model four actions of TWIG: Vertical jump, vertical landing, horizontal jump, and stable staying process. These actions are not elementary actions if we refer to definitions given above. However, this aspect does not have importance insofar as the purpose of this experiment were to validate that CONACQ was able to automatically model the actions of a robot with CSPs by constraint networks acquisition.

Each training instance given to CONACQ was a labelled (positive or negative) tuple $(t^i, s^i, x^i_G, z^i_G, U_R, U_\theta, t^f, s^f, x^f_G, z^f_G)$, where the t variable refers the time, and where the exponents i and f indicate respectively the variables state at the beginning and at the end of each action. CONACQ has automatically modelled the four actions by the following constraint networks:

Vertical Jump

$$\begin{cases} \mathcal{P}_p = \{(s^i = 1)\} \\ \mathcal{P}_a = \{(U_R = 500,\ U_\theta = 0)\} \\ \mathcal{P}_e = \{(s^f = 0,\ x^f_G = x^i_G,\ z^f_G = z^i_G + 3,\ t^f = t^i + 2)\} \end{cases}$$

Vertical landing

$$\begin{cases} \mathcal{P}_p = \{(s^i = 0)\} \\ \mathcal{P}_a = \{(U_R = 0,\ U_\theta = 0)\} \\ \mathcal{P}_e = \{(s^f = 1,\ x^f_G = x^i_G,\ z^f_G = z^i_G - 3,\ t^f = t^i + 20)\} \end{cases}$$

Horizontal Jump

$$\begin{cases} \mathcal{P}_p = \{(s^i = 1)\} \\ \mathcal{P}_a = \{(U_R = 300,\ U_\theta = 450)\} \\ \mathcal{P}_e = \{(s^f = 0,\ x_G^f = x_G^i + 3,\ z_G^f = z_G^i + 2,\ t^f = t^i + 6)\} \end{cases}$$

Stable staying process

$$\begin{cases} \mathcal{P}_p = \{(s^i = 1)\} \\ \mathcal{P}_a = \{(U_R = 0,\ U_\theta = 0)\} \\ \mathcal{P}_e = \{(s^f = 1,\ x_G^f = x_G^i,\ z_G^f = z_G^i,\ t^f = t^i + 1)\} \end{cases}$$

5 Conclusion

In this article, we have proposed a theoretical framework using constraint networks acquisition to acquire CSPs which model the elementary actions of a robot. We have shown in a second time how the acquired CSPs could be manipulated by planning tools in order to define quickly and automaticaly actions plans for poly-articulated robots.

Some aspects, such as the choice of the planning tools, are not yet defined totaly at present. However, the first experiment on the one-leg jumping robot TWIG has validated the automatic modelling process using CONACQ, which is the first step of our approach. So we are reasonnably optimistic for future works we have to carry out in order to make our theoretical framework as an effective platform for supervising tasks planning for robots.

Acknowledgments

We summarize in this paper the work we carried out in close co-operation with Jean Sallantin, Eric Bourreau and Rémi Coletta, for the theoretical and formal aspects, and Sébastien Krut and Christopher Dartnell for the TWIG experiments. That they all are here to thank for their contributions...

References

1. AIBO *http://www.eu.aibo.com/* Sony Aibo Europe - Official Website.
2. ASIMO *http://asimo.honda.com/* ASIMO Humanod Robot - Honda Robotic Technology.
3. P. van Beek, X. Cheng *CPlan: A constraint Programming Approach to Planning.* Proceedings of AAAI'99, 1999.
4. C. Bessiere, R. Coletta, F. Koriche, B. O'Sullivan *A SAT-Based Version Space Algorithm for Acquiring Constraint Satisfaction Problems.* Proceedings of ECML'05, Porto, Portugal, pages 747-751, October 2005.
5. A. L. Blum, M. L. Furst *Fast Planning Through Planning Graph Analysis.* Proceedings of Artificial Inteligence, pages 281-300, 1997.
6. R. Coletta, C. Bessire, B. O'Sullivan, E.C. Freuder, S. O'Connell, J. Quinqueton *Semi-automatic modeling by constraint acquisition.* Proceedings of CP-2003, Short paper, LNCS 2833, Springer Kinsale, Cork, Ireland., pages 812-816, September 2003.
7. A. Lallouet, A. Legtchenko *Two Contributions of Constraint Programming to Machine Learning.* Proceedings of ECML'05, Porto, Portugal, pages 617-624, October 2005.
8. Marc H. Raibert *Legged robots that balance.* MIT Press Series In Artificial Intelligence, Cambrige, Massachussetts, ISBN:0-262-18117-7, 1986.
9. S. Russell, P. Norvig *Artificial Intelligence - A Modern Approach.* Second Edition, Prentice Hall, ISBN:0-13-080302-2, 2003.
10. SolidWorks *http://www.solidworks.com* 3D CAD softwares.

A Specialised Binary Constraint for the Stable Marriage Problem with Ties and Incomplete Preference Lists*

Student name: Chris Unsworth
Supervisor name: Patrick Prosser

Department of Computing Science, University of Glasgow, Scotland.
{chrisu,pat}@dcs.gla.ac.uk

Abstract

Gent and Prosser proposed the first constraint model for the Stable Marriage problem with Ties and Incomplete preference lists (SMTI). Their model was based upon the simple Stable Marriage (SM) model proposed by Gent et al, in which for each man woman pair a constraint is posted consisting of a set of *no good* pairs of values. Prosser and Unsworth proposed a specialised binary constraint for SM which significantly outperforms the simple SM model proposed by Gent et al. We now propose a new specialised binary constraint for SMTI. This constraint should provide a complete solution for the problem which significantly outperforms the simple SM model proposed by Gent and Prosser. This may then allow us to solve larger more realistic sized problem instances.

1 Introduction

In the Stable Marriage problem (SM) [4] we have a set of n men $\{m_1 \ldots m_n\}$ and a set of n women $\{w_1 \ldots w_n\}$. Each man ranks the n women into a strictly ordered preference list, and the women rank the men similarly. The problem is then to produce a *stable* matching of men to women. By a matching we mean that there is a bijection from men to women, and by stable we mean that there is no incentive for partners to divorce and elope. A matching M is said to be unstable if it contains a blocking pair. A man woman pair (m_i, w_j) form a blocking pair in a matching M if m_i and w_j are not matched in M and would both prefer to be matched to each other than to be matched to their respective assigned partners in M.

The Stable Marriage problem with Ties and Incomplete lists (SMTI) is a generalisation of SM. In SMTI men and women are allowed to submit incomplete preference lists which may also contain ties. By allowing ties in the preference list we are dropping the requirement that the preference lists must be strictly ordered, thus allowing someone to express indifference between two or more potential parters. By allowing incomplete preference lists we allow someone to state that they would rather be unmatched than be matched to someone not in their preference list. An instance of SMTI can be seen in Figure 1, in which entries in brackets indicate indifference.

By allowing ties in the preference list we introduce the possibility of different definitions of a blocking pair, and thus different definitions of stability. Here when talking about stability in SMTI we refer to the most commonly used definition known as weak stability [4]. In this definition a pair (m_i, w_j) will only form a blocking pair if both m_i and w_j

*The author is supported by EPSRC. Software support was given by an ILOG SA's academic grant.

Men's lists	Women's lists
Alf : Zoe (Ann Liz) Joe	Ann : Tom Alf Bob Ian
Bob : Liz Jes (Ann Zoe)	Joe : Ian (Alf Bob Jim)
Tom : (Ann Jes Liz Zoe)	Liz : (Alf Ian) Tom Bob
Ian : Ann Jes Liz Zoe Joe	Zoe : Tom (Jim Ian Bob) Alf
Jim : Joe Zoe Jes	Jes : Ian Jim (Tom Bob)

Figure 1: An SMTI instance with 5 men and 5 women

stand to improve their position by eloping. For example if Alf was matched to Joe and Liz was matched to Ian (from the instance in Figure 1), then (Alf,Liz) would not form a blocking pair, even though Alf would rather be matched to Liz than Joe. Liz is happy enough with Ian and would not want to swap. However if Liz was matched to Tom then (Alf,Liz) would form a blocking pair.

It has been proven that a stable matching can always be found for an instance of SMTI in $O(n^2)$ time [5]. It has also been proven to be NP-hard to find a stable matching in which the maximum possible number of people are matched [6]

We now present a specialised binary constraint for SMTI. We will first show how an SMTI instance can be represented within a constraint model. We present the constraint and the methods that act upon it. We then discuss the complexity of the constraint.

2 Representing SMTI in a Constraint Model

In this constraint model we represent the men with a set of n integer variables $\{x_1 \dots x_n\}$, the women are also represented by a set of n integer variables $\{y_1 \dots y_n\}$. The man variable x_i has an initial domain of $\{1 \dots l_i^m, n+1\}$, where l_i^m is the length of m_i's preference list (the women variables have equivalent initial domains). The domain values represent preferences, meaning that if x_i were assigned the value j then this would correspond to m_i being matched to the woman in the j^{th} position in m_i's preference list. For example if x_4 was assigned the value 2 then in the instance shown in Figure 2, that would mean m_4 was matched to w_4. A variable x_i being assigned the value $n+1$ indicates that m_i is unmatched.

Men's lists	Women's lists
1: 1(3 6 2)4	1: 1 5 6(3 2 4)
2: 4 6(1 2)5	2:(2 4 6)(1 3 5)
3:(1 4)5(3 6 2)	3:(3 6)(5 1)
4: 6 4 2 1 5	4: 1 (3 5 4)2 6
5: 2 3(1 4 5)6	5: 3(2 6)4 5
6: 3(1 2 6 5 4)	6: 5(1 3 6 4)2

Figure 2: An SMTI instance with 6 men and 6 women

In [9] the preference lists are represented in two ways. The first is as a pair of two dimensional integer arrays *mpl* and *wpl*. These arrays contain the male and female preference lists respectively. For example from the instance in Figure 2 m_4's entry would be $mpl[4] = [6, 4, 2, 1, 5]$ and his second choice woman would be $mpl[4][2] = 4$ or w_4. There is also a second pair of two dimensional integer arrays mPw (man's preference for woman) and wPm (woman's preference for man). These arrays contain the inverse preference lists and can be used to find where a specific person appears in someone's preference list. For example m_4's inverse preference list would be $mPw[4] = [4, 3, -1, 2, 5, 1]$ (where the entry -1 indicates that person is unacceptable) and m_4's preference for w_2 would be $mPw[4][2] = 3$. Note that $mpl[i][k] = j \Leftrightarrow mpw[i][j] = k$.

To extend this to include ties in the preference list we first arbitrarily break the ties to flatten the preference lists. The preference lists are then held as before in a pair of two dimensional arrays. For example, m_1's preference list could look like $mpl[1] = [1, 3, 6, 2, 4]$ (depending on how the ties were broken). We then extend the inverse preference lists to include information about ties. Instead of storing a single integer to describe the position of w_j in the preference list of m_i we store a triple (α, β, γ). Here α represents the position in m_i's preference list of the first person tied with w_j, β is the position of w_j in the preference list, and γ is the position of the last person in the tie with w_j. For example the triple representing m_1's preference for w_6 would be $mPw[1][6] = (2, 3, 4)$. The full inverse preference list for m_1 would be $mPw[1] = [(1, 1, 1)(2, 4, 4)(2, 2, 4)(5, 5, 5)(-1, -1, -1)(2, 3, 4)]$. A triple in which $\alpha = \beta = \gamma$ indicates that this person in not involved in a tie. Note that a triple $(-1, -1, -1)$ is used to indicate an unacceptable partner.

3 Specialised Binary Constraint for SMTI (SMTI2)

The specialised binary constraint for the stable marriage problem with ties and incomplete lists (SMTI2) is designed to work within an AC5 type environment. SMTI2 is a binary constraint and acts over a man woman pair (m_i, w_j). SMTI2 will ensure that (m_i, w_j) do not become a blocking pair or get inconsistent values assigned to them (meaning that if m_i is matched to w_j then w_j will be matched to m_i). A constraint model using this constraint to solve an instance of SMTI will require one of these constraints for each man woman pair. To model an SMTI instance of size n would require $O(n^2)$ of these constraints.

An SMTI2 constraint acting over the man woman pair (m_i, w_j) would have the following attributes:

- x is a constrained integer variable representing m_i
- y is a constrained integer variable representing w_j
- xPy is a single triple representing m_i's preference for w_j, meaning $xPy = mPw[i][j]$.
- yPx is a single triple representing w_j's preference for m_i, meaning $yPx = wPm[j][i]$.

We also require the following procedures that act upon a constrained integer variable. We assume all these methods run in $O(1)$ time:

- $getMin(v)$ returns the smallest value in the domain of variable v.
- $getNextHigher(v, a)$ returns the smallest value in the domain of variable v that is strictly greater than the value a, if no such value exists then this procedure returns $n + 1$.
- $setMax(v, a)$ removes all values from the domain of variable v that are strictly greater than the value a.
- $removeValue(v, a)$ removes the value a from the domain of the variable v.

We assume that we require two methods to enforce AC over this constraint, $init(c)$ which gets called at the head of search when the constraint c is initialised, and $remVal(c, a)$ which is called when the value a is removed from the domain of a variable constrained by the constraint c. The methods will be presented using a Java like pseudo-code such that the . (dot) operator is an attribute selector, such that $c.b$ delivers the b attribute of c. Parentheses will be used to access individual elements of a list. For example, $c.xPy[2]$ will deliver the second item from the triple $c.xPy$.

We give these methods from a male only perspective. When a call is made to $remVal(c, a)$ we assume that the value has been removed from the domain of variable x. When a call is made to $init(c)$ we only consider the domain of x. To implement this constraint we will also require methods that consider the female perspective. The female versions of these methods can be obtained be simply swapping the gender specific terms. (i.e. c.x becomes c.y and c.xPy becomes c.yPx)

```
1.   remVal(c,a)
2.    IF a = c.xPy[2]
3.    THEN removeValue(c.y,c.yPx[2])
4.    IF getMin(c.x) == c.xPy[2]
5.    THEN IF getNextHigher(c.x,c.xPy[2]) > c.xPy[3]
6.         THEN setMax(c.y,c.yPx[3])
7.    IF getMin(c.x) > c.xPy[2]
8.    THEN setMax(c.y,c.yPx[3])
```

The $remVal(c, a)$ method is called when the value a has been removed from the domain of variable x constrained by the constraint c. If the value corresponding to y is no longer in the domain of x (line 2) then x is removed from the domain of y (line 3). If the minimum value in the domain of x corresponds to y (line 4) and the next highest value x's domain is not tied with y (line 5), then x strictly prefers y to all the remaining women in his domain, therefore y must not consider any man she likes less than x (line 6). If x is to be matched to a woman he likes less than y (line 7) then y must be matched to someone she likes no less than x (line 8).

```
1.   init(c)
2.    IF getMin(c.x) == c.xPy[2]
3.    THEN IF getNextHigher(c.x,c.xPy[2]) > c.xPy[3]
4.         THEN setMax(c.y,c.yPx[3])
```

The $init(c)$ method is called when the constraint c is initialised. This method checks if x strictly prefers y to all the remaining women in his domain (lines 2-3), if so then y must be matched to a man no worse than x (line 4).

This constraint has been implemented in JSolver [1] (the Java version of Ilog Solver). The results of initial testing are encouraging, and show that this constraint significantly outperforms the simple constraint model proposed in [3].

4 Complexity of the Constraint

Each SMTI2 constraint has size $O(1)$. A constraint model using this constraint would require $O(n^2)$ of these constraints to model an instance of SMTI, therefore a constraint model using this constraint would require $O(n^2)$ space.

Assuming that all the methods used within the $init(c)$ and $remVal(c, a)$ methods run in $O(1)$ time, then $init(c)$ and $remVal(c, a)$ will also run in $O(1)$ time. The $init(c)$ method will be called once for each of the $O(n^2)$ constraints. Each variable will be associated with $O(n)$ SMTI constraints, and each variable has $O(n)$ domain values, therefore for each variable the $remVal(c, a)$ method could be called at most $O(n^2)$ times. There are $O(n)$ variables in an instance of SMTI, therefore in the worse case enforcing AC on a constraint model using this constraint will take at most $O(n^3)$ time.

Note that with SMTI enforcing AC is not sufficient to ensure a solution is found, unlike the case with SM [2].

5 Conclusions and Future Work

We have proposed a new specialised constraint solution for SMTI. This constraint can be used to find all possible stable matching for a given instance of SMTI. Initial test have shown that this constraint offers a significant performance increase compared with the simple constraint model proposed in [3]. It will be interesting to extend these tests to a full empirical study.

In [3] Gent and Prosser conducted an empirical study of SMTI. In the study the authors generated random SMTI instances varying the number of ties and the size of the preference lists and measure the search cost to solve them. We intend to follow on this work by repeating the empirical study using SMTI2, and extending it to investigate larger instances, to see how this effects the problem.

In the same way as the specialised binary constraint for SM has been modified to solve SMTI, the specialised n-ary constraint proposed in [8] can be extended to solve SMTI. It should also be possible to extend the specialised binary and n-ary constraints for the hospital/residents problem proposed in [7] in the same way. This will then allow us to produce a complete solution for the real life hospital/residents problem.

References

[1] ILOG JSolver. http://www.ilog.com/products/jsolver/.

[2] I. P. Gent, R. W. Irving, D. F. Manlove, P. Prosser, and B. M. Smith. A constraint programming approach to the stable marriage problem. In *CP'01*, pages 225–239, 2001.

[3] I. P. Gent and P. Prosser. An empirical study of the stable marriage problem with ties and incomplete lists. In *ECAI'02*, 2002.

[4] D. Gusfield and R. W. Irving. *The Stable Marriage Problem: Structure and Algorithms.* The MIT Press, 1989.

[5] D. F. Manlove. Stable marriage with ties and unacceptable partners. Technical report, Dep. Computing Science, Univ. Glasgow, 1999.

[6] D. F. Manlove, R. W. Irving, K. Iwama, S. Miyazaki, and Y. Morita. Hard variants of stable marriage. *Theoretical Computer Science*, 276:261–279, 2002.

[7] D. F. Manlove, G. O'Malley, P. Prosser, and C. Unsworth. A constraint programming approach to the hospitals / residents problem. In *Workshop on Modelling and Reformulating Constraint Satisfaction Problems at CP'05*, 2005.

[8] C. Unsworth and P. Prosser. An n-ary constraint for the stable marriage problem. In *The Fifth Workshop on Modelling and Solving Problems with Constraints*, 2005.

[9] C. Unsworth and P. Prosser. A specialised binary constraint for the stable marriage problem. In *SARA 2005*, pages 218–233, 2005.

Automatic Generation of Alternative Representations and their Channelling Constraints

Student: Bernadette Martínez-Hernández
Supervisor: Alan M. Frisch

Department of Computer Science, University of York, United Kingdom
{berna, frisch}@cs.york.ac.uk

Abstract. Automatic modelling systems aim to reduce the number of decisions human modellers must take. To do so, these systems implement common modelling guidelines and techniques. One of the widespread methods to improve a model is the introduction of channelling constraints when two (or more) alternative representations are simultaneously used to solve a problem. In this paper we discuss the automatic generation of the alternatives and their channelling constraints.

1 Introduction

Constraint modelling is the process of encoding a problem into finite-domain decision variables and a set of constraints posed over them. Efficient constraint modelling is a hard task, often learnt by novice constraint users from modelling examples. In order to reduce the modelling time many automatic modelling systems have arisen. The construction of an automatic modelling system requires a good understanding of the modelling process as well as each commonly used modelling technique. In this paper we study the modelling technique of combining (redundant) alternative models through the addition of *channelling constraints*. Channelling constraints (or channels) were defined by Cheng et al [1] and used to increase propagation by combining two alternative models of a permutation: primal and dual. Channels are now used in many efficient models for all sorts of problems (see [2]), though the formalisation their automatic inclusion in a model has been somehow overlooked. This paper introduces an automatic method to generate channelling constraints. We first define representations in Section 2 to then discuss alternative representations and channels in the next section. The automatic generation of representations is described in Section 4, and finally, we detail the method of automatic generation of channels and channelled models in Section 5. Conclusions and future work are presented in the last section.

2 Representations

Throughout this paper we use the common definition of CSP (Constraint Satisfaction Problem) instance: a triple (X, D_X, C) of variables, domains and constraints. For example, the Sonet problem requires to configure a group of nodes

$$\begin{aligned}
&X && = \{\ \mathit{rings}\ \}\\
&D_X(\mathit{rings}) && = \textit{multisets (of size nrings) of sets (of maxsize capacity) of 1..nnodes}\\
&C && = \emptyset
\end{aligned}$$

Fig. 1. CSP-instance 1 of the rings of the Sonet problem.

$$\begin{aligned}
&X_1 && = \{\ \mathit{rings}_1\ \}\\
&D_{X_1}(\mathit{rings}_1) && = \textit{2-D matrix of Boolean indexed by 1..nrings and 1..nnodes}\\
&C_1 && = \forall i \in 1..\mathit{nrings}\ .\ (\textstyle\sum j \in 1..\mathit{nnodes}\ .\ \mathit{rings}[i,j]) = \mathit{capacity}
\end{aligned}$$

Fig. 2. CSP-instance 2 of the rings of the Sonet problem.

(1..*nnodes*) in a network of certain number *nrings* of rings, where each of ring has a maximum *capacity* of nodes[1]. In CSP instance 1, shown in Figure 1, the network of rings is specified as a multiset of sets. Since no constraint is imposed on the *rings* variable, the solutions of this instance are all the possible assignments to the multiset. Figure 2 shows a second CSP instance modelling the Sonet network with a Boolean 2-dimensions matrix $rings_1$, where $rings_1[i,j] = True$ if node j is in ring i. Notice that each solution of the CSP instance 2 can be mapped into a solution of the CSP instance 1. In general, we seek to find a mapping from solutions to solutions to ensure an instance represents another instance; as it is shown in the following definition.

Definition 1 *R' **represents** R via ψ, if $R' = (X', D_{X'}, C')$ and $R = (X, D_X, C)$ are CSP instances and ψ is a partial function from the total assignments of X' into the total assignments of X such that:*

- *For each total assignment w' of the variables in X', w' is a solution of C' **if and only if** $\psi(w')$ is defined and it is a solution of C,*
- *For each solution w of C, there is at least one solution w' of C' such that $\psi(w') = w$.*

*We say that R' **represents** or **is a representation** of R if for some ψ R' represents R via ψ*

For our Sonet example we can define the function ψ_1 that transforms the rows of $rings_1$ into the sets of *rings*, and then say CSP instance 2 represents CSP instance 1 via ψ_1.

[1] For simplicity, in this example we do not impose the communication and minimisation constraints.

$$\begin{aligned}
&X_2 && = \{\ \mathit{rings}_2,\ \mathit{switch}\ \}\\
&D_{X_2}(\mathit{rings}_2) && = \textit{2-D matrix of 1..nnodes indexed by 1..nrings and 1..capacity}\\
&D_{X_2}(\mathit{switch}) && = \textit{2-D matrix of Boolean indexed by 1..nrings and 1..capacity}\\
&C_2 && = \emptyset
\end{aligned}$$

Fig. 3. CSP instance 3 of the rings of the Sonet problem.

3 Alternative Representations and Channels

Let us now introduce a third CSP instance that models the network of rings. This instance is shown in Figure 3 and contains the matrices, $rings_2$ and $switch$, of integer and Boolean variables respectively; where node j is in ring i if for some k, $rings_2[i,k] = j$ and $switch[i,k] = True$. The CSP instance 3 also represents CSP instance 1. Since CSP instance 2 and CSP instance 3 represent CSP instance 1 we call them *redundant* with respect to CSP instance 1.

Two redundant CSP instances with respect to a third CSP instance are *alternative* if their sets of variables are disjoint. The *union* of a group of instances is defined as the union of their variables, domains and constraints. In fact, the union of alternative CSP instances, redundant with respect to R, also represent the CSP instance R. For example, the union of CSP instances 2 and 3 represents CSP instance 1 too.

The definition of representation connects two instances. We introduce now the concepts of *variable* and *constraint* representations that relate variables and constraints to CSP instances. These extensions of the concept of representation are used to define a *constraint-wise representation*, that is a group of instances that some how represent the constraints and variables of a CSP instance.

Definition 2 *Let x_R be a variable with domain τ_{x_R}. The CSP instance R* ***represents the decision variable*** *x_R if R represents the CSP instace $(\{x_R\}, \{\tau_{x_R}\}, \{\})$. Let C_R be a constraint over the variables $x_1, \ldots, x_n$ with domains $\tau_{x_1}, \ldots, \tau_{x_n}$. The CSP instance R* ***represents the constraint*** *C_R if R represents the CSP instance $(\{x_1, \ldots, x_n\}, \{\tau_{x_1}, \ldots, \tau_{x_n}\}, \{C_R\})$.*
Let R be the CSP instance $(X, D_X, C = \{C_1, \ldots, C_n\})$; the CSP instances $R_1, \ldots, R_n$ be representations of the constraints $C_1, \ldots, C_n$; and $R_{x_1}, \ldots, R_{x_n}$ be representations of the variables $\{x_1, \ldots, x_n\}$ in X such that no constraint in C is imposed over them. The set of instances $\{R_1, \ldots, R_n, R_{x_1}, \ldots, R_{x_n}\}$ is a ***constraint-wise representation*** *of CSP instance R if:*

- *The variables of each instance in the set $\{R_1, \ldots, R_n, R_{x_1}, \ldots, R_{x_n}\}$ are pairwise disjoint.*
- *For every constraint C_i in C, there are (pairwise variable-disjoint) CSP sub-instances R_{y_j} of R_i such that for each variable y_j the constraint C_i is imposed on, x_j is represented by R_{x_j}.*

To clarify these definitions consider the following example. Let us extend CPS instance 1 by adding to it constraints $C_m(rings)$ and $C_p(rings)$; to instance 2 $C'_m(rings_1)$; and to instance 3 $C'_p(rings_2, switch)$. We call these extensions CSP instances 1a, 2a and 3a respectively[2]. Suppose CSP instance 2a represents constraint $C_m(rings)$ and CSP instance 3a represents $C_p(rings)$. This, CSP instances 2a and 3a compose the constraint-wise representation of CSP instance 1a. In general, the union of a constraint-wise representation of an instance does not represent the instance because the solutions of both representations are not synchronised, and when propagating the constraints they do not prune consistently assignments in the representation mapping to the same assignment in the

[2] To simplify the examples constraints C_m, C'_m, C_p, C'_p remain undefined.

instance. To synchronise the different alternative representations of a variable in a constraint-wise representation of an instance we add channelling constraints. For the union of instances 2a and 3a we introduce the following channel:

$$\begin{aligned}&\forall i \in 1..nrings\ .\ ((\sum j \in 1..nrings\ .\ \forall k \in 1..nnodes\ .\ \exists l \in 1..nnodes\ rings_1[i,k] = rings_1[j,l])\\ &= (\sum j \in 1..nrings\ .\ \forall k \in 1..nnodes\ .\ \exists l \in 1..capacity\ .\ rings_1[i, rings_2[j,l]] \wedge switch[j,l]))\end{aligned} \tag{1}$$

We generalise the synchronisation intuition in the following definition of channelling constraints.

Definition 3 *Let R_1 represent R via ψ_1 and R_2 represent R via ψ_2. Let $vars(R_1)$ and $vars(R_2)$ be disjoint sets of variables. The set of constraints C_h is considered a set of* ***channelling constraints between*** R_1 ***and*** R_2 *if:*

- *For each solution x_1 of R_1 there is at least one total assignment x_2 (of the variables in R_2) such that the composed assignment $x_1 \cup x_2$ satisfies the constraints in C_h. Similarly for each solution x_2 there must be an assignment x_1 such that the composed assignment $x_1 \cup x_2$ satisfies the constraints in C_h.*
- *For all total assignments x_1 and x_2 where the composed assignment $x_1 \cup x_2$ satisfies the constraints in C_h, $\psi_1(x_1)$ and $\psi_2(x_2)$ are either both undefined or take the same value.*

By adding correct channels to the union of a constraint-wise representation of an instance we can now ensure it represents the instance.

Theorem 1 *Let R_{cstr} be the constraint-wise representation of R, where variable x of R has two variable representations R_{x_1} and R_{x_2} in R_{cstr}. Let C_h be the correct channelling constraint between R_{x_1} and R_{x_2}. Then, the CSP instance $\bigcup R_{cstr} \cup C_h$ represents R.*

4 Refinement

Refinement is an automatic modelling method used by CONJURE, a system introduced by Frisch et al [3]. The refinement of CONJURE is a generalisation of the refinement restricted to instances, that is, the refinement explained in this paper.

Given an input CSP instance R, the refinement process generates constraint-wise representations of R, where each one composes a single instance by joining its elements. Every constraint-wise representation is generated by independently producing the representations of the constraints and variables of R by means of the recursive application of refinement rules. For example, the *rings* variable of CSP instance 1 is transformed into CSP instance 1′ of Figure 4 by a rule that transforms multisets into arrays (explicit representation). Then the variable *rings*′ is fed to the refinement process to obtain CSP instance 2 of Figure 2 after applying the rule that transforms sets into Boolean arrays (occurrence representation). Regardless of the rule used to refine a variable or constraint, the transformations applied to variables are recorded using tags called *representation annotations.* These tags contain information specifying representation constructed by the rule. From the set of annotations of a produced instance

$$\begin{array}{ll} X' & = \{\ \mathit{rings}'\ \} \\ D'_X(\mathit{rings}') & = \textit{1-D matrix of sets (of maxsize capacity) of 1..nnodes} \\ & \quad \textit{indexed by 1..nrings} \\ C' & = \emptyset \end{array}$$

Fig. 4. CSP instance 1′ of the rings of the Sonet problem.

we can generate sequences of annotations connecting the variables of refined instance R with their produced representations. For example, for the refinements of CSP instance 2 and 3 of CSP instance 1 we have the sequences

$$[\texttt{represent}(exp, rings, rings'), \forall i \in 1..nrings.\ \texttt{represent}(occ, rings'[i], rings_1[i])] \quad (2)$$

$$[\texttt{represent}(exp, rings, rings'), \forall i \in 1..nrings.\ \texttt{represent}(varexp, rings'[i], (rings_2[i], switch[i]))]$$

Note these sequences are also produced by refining CSP instance 1a into the union of instances 2a and 3a. Any CSP instance R', produced by the refinement process of an input CSP instance R, represents R as long as it does not contains several alternative representations of any of the variables in R. For example, one of the refinements of CSP instance 1a is the union of CSP instances 2a and 3a which is clearly not a representation of instance 1a. As we showed in the previous section, channels need to be introduced to transform this union into a proper representation.

5 Systematic generation of channelling constraints

Using the annotations, we can track the final representation of each variable of a refined CSP instance. More importantly, two representations of a variable are identical if we can *unify* their sequences of annotations obtained from the refinement each one was produced. Whenever one of the instances returned after the refinement contains two (or more) alternative redundant representations of the same variable, we use the sequence of annotations in a second refinement of the variable for whom alternative representations were produced. We follow the next steps:

1. A dummy variable with the same domain is created. For our example of the union of CSP instance 2a and 3a, we create *ringsdummy*; a variable whose domains is, as well as the *rings* variable, composed of multisets of sets.
2. We modify the sequences of annotations to assign a path conducting each variable to an alternative representation. In our example we change only the last sequence of (4) into:

$$[\texttt{represent}(exp, ringsdummy, ringsdummy'),$$
$$\forall i \in 1..nrings.\ \texttt{represent}(varexp, ringsdummy'[i], (rings_2[i], switch[i]))] \quad (3)$$

3. Equality is a channelling constraint between two variables with the same domain. The refinement of a channelling constraint between two variables produces the representation of the two variables plus the channelling constraints between them. For that reason the equality constraint between two variables is refined. For the example, we refine *rings = ringsdummy.*

From the set of representations we get after this process is finished we select only those whose sequences of annotations unify with the previously modified ones. In our example the channel (1) is produced. Theorem 1 ensures the algorithm of generation produces correct channelling constraints, converting constraint-wise representations into proper representations of instances. In many cases a direct implementation of a channel propagate over an extensive set of constraints does not give the most efficient pruning. We can obtain specialised global constraints by add to the system refinement rules that produce them instead of the usual constraints.

6 Conclusions and future work

We have identified that channels in a refined-based automatic may only be needed when a variable is used in several constraints. We introduce a method of automatic generation of the channels that depends upon keeping track of the series of transformation used to construct alternative redundant representations of the same variable. We use the tags that annotate the consecutive transformations to restrict the refinement of an equality constraint. The purpose of this second refinement is the automatic generation of the needed channelling constraints using the same refinement technique.

Future work includes the addition of global constraints for the channels between representations of variables of compound domains and/or uncommon representations, for example the channel (1) between the occurrence and the variable-sized-explicit representation (introduced in [4]). Also, the literature provides examples of combinations of alternative representations where some redundant constraints or variables have been deleted to improve the propagation (e.g. [5]). We need to implement them and possibly find similar cases.

References

1. Cheng, B.M.W., Lee, J.H.M., Wu, J.C.K.: Speeding up constraint propagation by redundant modeling. In: CP 1996. (1996) 91–103
2. Walsh, T., Hnich, B.: Why channel? multiple viewpoints for branching heuristics. In: Proceedings of the CP'03 Second International Workshop on Modelling and Reformulating Constraint Satisfaction Problems: Towards Systematisation and Automation. (2003)
3. Frisch, A.M., Jefferson, C., Martínez-Hernández, B., Miguel, I.: The rules of constraint modelling. In: Nineteenth Int. Joint Conf. On Artificial Intelligence (IJCAI). (2005) 109–116
4. Jefferson, C., Frisch, A.M.: Representations of sets and multisets in constraint programming. In: Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems. (2005) 102–116 Held at the 11th International Conference on Principles and Practice of Constraint Programming.
5. Walsh, T.: Permutation problems and channelling constraints. In Nieuwenhuis, R., Voronkov, A., eds.: Proceedings of LPAR-2001. LNAI, Springer (2001) 377–391

The Effect of Constraint Representation on Structural Tractability

Student: Chris Houghton
Supervisor: David Cohen

Department Of Computer Science,
Royal Holloway, University Of London, UK

Abstract. Tractability results for structural subproblems have generally been considered for explicit relations listing the allowed assignments. In this paper we define a representation which allows us to express constraint relations as either an explicit set of allowed labelings, or an explicit set of disallowed labelings, whichever is smaller. We demonstrate a new structural width parameter, which we call the interaction width, that when bounded allows us to carry over well known structural decompositions to this more concise representation. Our results naturally derive new structurally tractable classes for SAT.

1 Introduction

An instance of the constraint satisfaction problem is a collection of variables to be assigned, a universe of possible values and a collection of constraints. Each constraint has a relation which restricts the allowed simultaneous assignments to a set of these variables. This set of variables is called the scope of the constraint.

The constraint satisfaction problem is, in general, NP-hard. As such, it is an important area of research to identify subproblems which are tractable.

The structure of a constraint satisfaction problem instance (CSP) is defined to be the hypergraph whose vertices are the variables of the instance and whose hyperedges are the constraint scopes.

When we are given a CSP to solve, the constraint relations will be expressed in some encoding, that is, by some sequence of symbols. We classify these encodings based on the expression method used. The set of encodings that we allow a class of CSPs to be expressed by is called a representation. Practitioners generally use the most concise representation they can to express a problem.

Most research on tractability has been concerned with explicitly allowed encodings [2], which we shall refer to as the positive representation. In this paper we wish to investigate how well our current theory translates into a more natural representation.

We define a notion of structural width for a hypergraph which we call the interaction width. We are able to show that certain structural classes with bounded interaction width are tractable.

We describe SAT in terms of our complement representation and show why this is a natural approach for discussing the structural tractability of classes of SAT instances.

2 CSPs and Representations

Definition 1. *A constraint satisfaction problem instance (CSP) is a triple $\langle V, D, C\rangle$ where; V is a set of variables, D is a finite set which we call the universe of the problem, and C is a set of constraints.*

Each constraint *$c \in C$ is a pair $\langle \sigma, \rho\rangle$, where σ (called the constraint scope) is a subset of V and ρ (called the constraint relation) is a set of labelings of σ. Each* labeling *is a function from σ into the universe D.*

A solution *to a CSP, $P = \langle V, D, C\rangle$ is a mapping $s : V \to D$ such that for every $\langle \sigma, \rho\rangle \in C$ we have that s restricted to σ is an element of ρ.*

Definition 2. *A* hypergraph*, H, is a pair $\langle V, E\rangle$, where V is a set, called the* vertices *of H, and E is a set of subsets of V, called the* hyperedges *of H.*

For any CSP $P = \langle V, D, C\rangle$, the structure *of P, denoted $\sigma(P)$, is the hypergraph $\langle V, \{\sigma \mid \langle \sigma, \rho\rangle \in C\}\rangle$.*

The class of CSPs whose structure is an *acyclic* hypergraph is tractable [1]. In particular an acyclic hypergraph has a *join tree* which can be used to solve the instance.

Example 1. Let $\mathcal{A}$ be the class of CSPs generated by taking an instance of graph 3-coloring and adding a universal constraint (over all variables) which allows all labeling. This does not alter solution but the universal constraint forces instances of $\mathcal{A}$ to have acyclic structure.

This anomaly relies on the universal constraint being expressed by listing every possible assignment to all variables in the CSP.

An encoding is the way in which a constraint relation is expressed. It is usual for the labelings in a constraint relation to be encoded directly as the allowed assignments to the variables in the scope. We shall refer to this encoding as the *positive encoding.* Alternatively, it is also acceptable for the labelings to be encoded as the disallowed assignments to the variables in the scope. We shall refer to this encoding as the *complement encoding.*

Definition 3. *A* representation*, R, is a set of possible encodings.*

A CSP, $P = \langle V, D, C\rangle$, is said to be expressed in a representation, R, if for every constraint, $\langle \sigma, \rho\rangle \in C$, the relation, ρ, is expressed by the encoding in R which has the smallest size for ρ. (This is similar to the concept of Minimum Description Length [3].)

The representation which allows only the positive encoding is called the positive representation *(*Pos*) and the representation which allows only the compliment encoding is called the* compliment representation *(*Comp*).*

The representation which allows both the positive encoding and the compliment encoding is called the mixed representation *(*Mixed*).*

3 Tractability with respect to representation

We show that the tractable classes of each of these representations is distinct by demonstrating classes which distinguish them.

Definition 4. *A class of CSPs is called* tractable *if there is a polynomial time algorithm to decide membership and to solve the instances of the class.*

Define by $\mathrm{T}(\mathcal{R})$ *the tractable classes of representation* $\mathcal{R}$.

Proposition 1. *Consider two representations,* $\mathcal{R}$ *and* $\mathcal{Q}$*, such that* $\mathcal{Q} \subseteq \mathcal{R}$. *Assuming that there is a polynomial conversion from relations expressed with respect to* $\mathcal{Q}$ *to relations expressed with respect to* $\mathcal{R}$*, then for a set,* S*, of CSPs, we have that if* $S \in \mathrm{T}(\mathcal{R})$ *then* $S \in \mathrm{T}(\mathcal{Q})$.

Proof. Let P be any CSP in S expressed with respect to $\mathcal{Q}$. We use the polynomial time conversion to change the expression of P from $\mathcal{Q}$ to $\mathcal{R}$ and then solve using the algorithm for S with respect to $\mathcal{R}$.

Corollary 1. *Let* S *be a set of CSPs such that* $S \in \mathrm{T}(\mathrm{Mixed})$. *We have that* $S \in \mathrm{T}(\mathrm{Pos})$ *and* $S \in \mathrm{T}(\mathrm{Comp})$.

Proof. It is straightforward to see that any relation expressed in the larger of the two encodings must list more than half the possible number of assignments. The universal constraint on this scope is at most twice as big so we can generate the other encoding for this relation.

The class $\mathcal{A}$ from Ex. 1 is in $\mathrm{T}(\mathrm{Pos})$, but not in $\mathrm{T}(\mathrm{Comp})$. This is because the universal constraint has no size when expressed in Comp and so is directly equivalent to graph 3-coloring.

Example 2. Let $\mathcal{B}$ be the class of CSPs with $2n$ variables generated by taking an instance of graph 3-coloring over n of the variables and adding a single constraint over the remaining n variables which allows only a single assignment.

The class of instances, $\mathcal{B}$, in example 2 is not tractable when expressed in Pos. The added constraint over the n variables which are not part of the graph coloring instance is small when expressed in the positive encoding, and so the problem is directly equivalent to graph coloring. When expressed in Comp, the size of this constraint dominates the size of the graph coloring component and so a simple polynomial time algorithm is to test all possible assignments to the graph coloring component.

We have now shown that $\mathrm{T}(\mathrm{Pos})$ is incomparable to $\mathrm{T}(\mathrm{Comp})$ as neither is contained in the other. However, as we have also shown that anything in $\mathrm{T}(\mathrm{Mixed})$ must also be in both $\mathrm{T}(\mathrm{Pos})$ and $\mathrm{T}(\mathrm{Comp})$, $\mathrm{T}(\mathrm{Mixed})$ must be a proper subset of both $\mathrm{T}(\mathrm{Pos})$ and $\mathrm{T}(\mathrm{Comp})$. This poses the question; does $\mathrm{T}(\mathrm{Mixed})$ contain any interesting (non-trivial) classes?

4 Converting Mixed to Pos

We shall show that by bounding some new notion of structural size, called the *interaction width*, we can convert certain CSPs from Mixed to Pos. If there are subclasses of CSP classes in T (Pos) for which there is a polynomial conversion from Mixed to Pos, then such subclasses are tractable with respect to Mixed.

If we were to represent a hypergraph as a Venn Diagram where the hyperedges are the sets, then an interaction region of the hypergraph is a region of the Venn Diagram. Interaction width is the maximal number of regions over any of the hyperedges.

Definition 5. *Let $H = \langle V, E\rangle$ be a hypergraph. We define the* interaction *on vertex $x \in V$, denoted $\tau(x)$, to be the set of edges containing x so that $\tau(x) = \{e \in E \mid x \in e\}$.*
We define I to be the set of interactions for all vertices so that $I = \{\tau(x) \mid x \in V\}$. We define $I(e)$ to be the set of interactions for the vertices which are in the edge e so that $I(e) = \{X \in I \mid e \in X\}$.

The interaction region*, $V(X)$, associated with the interaction $X \in I$ is the set of vertices which are in the same interaction as X that is, $V(X) = \{x \in V \mid \tau(x) = X\}$.*

The interaction width*, denoted $I_w(H)$, of H is the largest number of non-singleton interactions associated with any of its edges;*

$$I_w(H) = \max\{|I(e) - \{\{e\}\}| \mid e \in E\}.$$

There are two types of interaction region in which we may not have enough information to do the conversion in polynomial time. We call these ‘isolated regions’ and ‘trivial compliment regions’. We shall also show that the reduction from one structure to the new structure preserves other structural notions of width.

For a hypergraph, $H = \langle V, E\rangle$, we define the removal of a set of vertices, $V' \subseteq V$, to be the hypergraph $H' = \langle V', E'\rangle$ where $E' = \{e \cap V' \mid e \in E\}$.

It is straightforward to show that removing a set of vertices does not increase the structural decomposition width of a hypergraph. Structural decomposition of the original structure may therefore be used to solve the converted instance.

Our conversion requires that we project out certain interaction regions.

Definition 6. *The projection of a constraint, $\langle\sigma, \rho\rangle$ onto a subset, X, of its scope is the constraint $\langle X, \{f_{|X} \mid f \in \rho\}\rangle$.*

The method for performing projection on constraints whose relations are sets of allowed assignments is well defined, but it is not clear how to perform projection in polynomial time for constraint whose relation is expressed as sets of disallowed assignments. Method 1 does this for any constraint expressed using Comp and gives the resulting constraint expressed with respect to Comp.

Method 1. *Given an encoding of a constraint, $\langle\sigma, \rho\rangle$, where ρ is a set of disallowed assignments, and a subset of the variables in the scope, $\sigma' \subseteq \sigma$ we can project ρ onto σ' in the following way;*

Restrict the assignments of ρ so that they are only over the variables of σ' to give ρ'. For every l in ρ', if every possible extension to l exists in ρ then keep l in ρ', else discard l.

After performing projection on a constraint represented with respect to Pos, it may then allow more than half the possible assignments and need to be converted to Comp (which can be done in polynomial time for the same reason).

Definition 7. *Given a hypergraph, $H = \langle V, E\rangle$, for each $e \in E$ the region associated with the interaction $\{e\}$ is called an* isolated region*.*

If we could project out the isolated regions from a CSP, then we could solve the problem over the remaining variables and then extend any solution on the reduced structure to the isolated regions of the original.

Definition 8. *Any interaction for which all constraints (over at least two scopes) are encoded with respect to* Comp*, is called a* compliment interaction*.*

Given a CSP, $P = \langle V, D, C\rangle$ with a compliment interaction, X, let $C' \subseteq C$ be the set of constraints whose scopes are contained in X. Let ρ' be the set of all labellings from the constraints in C' restricted to the vertices of the complement region X, i.e. $\rho' = \left\{ l \mid \forall\langle\sigma, \rho\rangle \in C', l \in \rho_{|V(X)} \right\}$. If l does not contain all possible assignments over the variables in X, $|l| < |D|^{|X|}$, then we call X a trivial complement region*.*

We can remove trivial compliment regions as not all disallowed assignments exist in ρ' so any missing assignment must be allowed by all extensions for every constraint in C'. We can see that ρ' can be generated in polynomial time. By assuming an order on assignments we can easily check if one is missing from ρ'. We can stop after finding a single missing assignment and remember it for the purpose of extending solutions later.

Let $\mathcal{H}$ be a set of hypergraphs with interaction width i. We can now provide an algorithm for converting any CSP represented with respect to Mixed and whose structure is in $\mathcal{H}$ to a solution preserved CSP represented with respect to Pos.

Method 2. *INPUT: A CSP $P = \langle V, D, C\rangle$ and the hypergraph of P, $H = \langle V, E\rangle$.*

1. *Find and project out all isolated regions.*
2. *Find and project out trivial complement regions, remembering extensions. (Let H' be the reduced structure and $P' = \langle V', D, C'\rangle$ be the reduced instance after projecting out isolated regions and trivial complement regions.)*
3. *Convert the reduced instance to the positive representation.*
   - *Create a mapping, L, which maps from interactions I of the hypergraph H' to sets of assignments on the respective interaction regions such that*

  - *For each interaction, $X \in I$, if there exists a constraint, $\langle \sigma', \rho' \rangle \in C'$ whose relation is expressed with respect to* Pos *and whose scope contains $V(X)$, the region of the interaction X, then set $L(X)$ to be $\rho|_{V(X)}$. Otherwise, set $L(X)$ to be the set of all possible assignments to $V(X)$ over D.*
- *Create a new CSP, $\bar{P} = \langle V', D, \bar{C} \rangle$ with structure H' such that*
  - *For each hyperedge σ' of H', create the new constraint in $\langle \sigma', \bar{\rho} \rangle$ in $\bar{P}$ such that $\bar{\rho}$ is the product over $X \in I(\sigma')$ of $(L(X))$.*
  - *For each constraint $\langle \sigma', \bar{\rho} \rangle \in \bar{C}$, then for every constraint $\langle \sigma', \rho' \rangle \in C'$, if ρ' is represented with respect to* Pos*, then $\bar{\rho} := \bar{\rho} \cap \rho'$. Otherwise, $\bar{\rho} := \bar{\rho} - \rho'$.*

It is straightforward to show that this algorithm runs in polynomial time for bounded interaction width. The bound is required for generating the new constraints on the products of the assignments over the regions.

Once we have solved the reduced CSP we can then extend any solutions to the original CSP by extending solutions to the trivial complement regions and the isolated regions. We have already shown that this is easy to do.

For any tractably identifiable structural decomposition, such as bounded width hypertrees [2], we generate a new tractable class with respect to the representation Mixed.

5 Structurally Tractable classes of SAT

Each clause in a SAT instance only disallows a single assignment. There can be no polynomial time conversion from SAT clauses to Pos as this would lead to a possible exponential blow-up in the size of an instance unless the arity is bound.

However, there is a natural representation of SAT in Mixed where each claude is a constraint with a single disallowed assignment. Structural tractability results (with bounded interaction width) naturally extend to SAT.

Szeider [4] has also developed a structural tractability result for SAT which is based on the *treewidth* of the so called *incidence graph*. He has shown that any class of instances with bounded treewidth of this graph is a fixed parameter tractable class for SAT. We can show that even just for SAT, these two structurally tractable classes are incomparable, so there are two distinct structural tractability results for SAT. However, ours has a natural extension to domains of larger size, so we hope may be applicable to other practical problems.

References

1. C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30:479–513, 1983.
2. Georg Gottlob, Nicola Leone, and Francesco Scarcello. A comparison of structural csp decomposition methods. *Artif. Intell.*, 124(2):243–282, 2000.
3. J. Rissanen. Modeling by shortest data description. In *Automatica, vol. 14*, 1978.
4. Stefan Szeider. On fixed-parameter tractable parameterizations of sat. In *SAT*, pages 188–202, 2003.

Combining BDDs with Cost-Bounding Constraints for Interactive Configuration

Student: Tarik Hadžić
Supervisor: Henrik Reif Andersen

Computational Logic and Algorithms Group, IT University of Copenhagen,
Rued Langgaards Vej 7, DK-2300 Copenhagen S, Denmark
{tarik,hra}@itu.dk

Abstract. Binary Decision Diagram (BDD) [1] is a data structure representing a conjunction of propositional formulas in a format that supports efficient answers to many important queries (satisfiability, equivalence, model counting etc). Although exponentially large in worst case, it has a compact representation for many important propositional formulas occurring in practice [2]. In particular, BDDs are successfully applied in the area of *interactive configuration* [3] where the key user functionality of *calculating valid domains* (CVD) [4] is implemented by an algorithm enforcing *generalized arc-consistency* [5].

In order to extend the range of current BDD-based interactive configuration, we are investigating which constraints can be implicitly conjoined with a BDD and still allow for efficient calculation of valid domains. In [6] we described first such bounding-cost constraint. Here we show that BDD extensions with several cost-bounding constraints necessarily require an NP-hard algorithm for implementing CVD functionality. We also discuss further algorithmic approaches to tackle this problem.

1 Introduction

Interactive configuration problems are special applications of Constraint Satisfaction Problems (CSP) where a user is assisted in interactively assigning values to variables by a software tool. The key user functionality delivered by this tool is the *calculation of valid domains* (*CVD*) for each unassigned variable. Application areas include customizing physical products (such as PC's and cars) and services (such as airplane tickets and insurances).

The domains calculated by the *CVD* functionality should be *complete* (all valid configurations should be reachable through user interaction), *backtrack-free* (a user is never forced to change an earlier choice due to incompleteness in the logical deductions), and the *CVD* feedback should be in *real-time*. If we understand the underlying CSP model as defining a single constraint C, the requirement for backtrack-freeness corresponds to making the constraint C generalized arc-consistent. In general this is an NP-hard task. Therefore, in order to provide real-time guarantees for user interaction, current approaches use off-line compilation of the CSP model into a tractable datastructure representing the solution space of all valid configurations [3, 7, 8].

In [3, 10, 9] the interactive configuration over CSP models involving propositional constraints is delivered by first compiling constraints into a BDD [1]. Then the CVD functionality is delivered online by algorithms that are provably efficient in the size of the compiled representation [4]. Although having exponential worst-case size, in the real industrial applications of product configuration, the compiled BDDs can often be kept small [9]. This is essential for guaranteeing a real-time response time to a user interaction.

Adding global constraints to a CSP model can make the size of a compiled BDD to explode. Therefore we are investigating whether we can avoid such explosion by keeping global constraints outside a BDD, but still develop efficient algorithms for CVD. In [6] we demonstrated one such case where a *cost bounding constraint* was combined efficiently with a BDD. Here, we show that for a conjunction of a BDD with several cost-bounding constraints there is no polynomial time algorithm calculating valid domains.

The rest of the paper is organized as follows. In section 2 we describe a BDD approach to interactive configuration. In section 3 we show the infeasibility result. Finally we conclude in the fourth section.

2 BDD-Based Interactive Configuration

The input *model* describing the knowledge about valid variable assignments is a special kind of a *Constraint Satisfaction Problem* (CSP) [5]:

A *configuration model* C is a triple (X, D, F) where X is a set of variables $\{x_0, \ldots, x_{n-1}\}$, $D = D_0 \times \ldots \times D_{n-1}$ is the Cartesian product of their finite domains $D_0, \ldots, D_{n-1}$ and $F = \{f_0, ..., f_{m-1}\}$ is a set of propositional formulae over atomic propositions $x_i = v$, where $v \in D_i$, specifying conditions on the values of the variables.

Concretely, every domain can be viewed as $D_i = \{0, \ldots, |D_i| - 1\}$. An *assignment* of values $v_{i_0}, \ldots, v_{i_k}$ to variables $x_{i_0}, \ldots, x_{i_k}$ is denoted as a set of pairs $\rho = \{(x_{i_0}, v_{i_0}), \ldots, (x_{i_k}, v_{i_k})\}$. The domain of the assignment $dom(\rho)$ is the set of variables which are assigned: $dom(\rho) = \{x_i \mid \exists v \in D_i.(x_i, v) \in \rho\}$ and if $dom(\rho) = X$ we refer to ρ as a *total assignment*. We say that a total assignment ρ is *valid* if it satisfies all the rules, which is denoted as $\rho \models F$. A partial assignment $\rho, dom(\rho) \subseteq X$ is *valid* if there is at least one total assignment $\rho' \supseteq \rho$ that is valid, $\rho' \models F$, i.e. if there is at least one way to successfully complete the existing configuration process. Given the starting domains D_i and a partial user assignment ρ valid domains are: $D_i^\rho = \{v \in D_i \mid \exists \rho'.(\rho' \models F \; and \; \rho \cup \{(x_i, v)\} \subseteq \rho'\}$.

In product configuration, the knowledge about product components and product rules is usually modelled by representing all the choices for a component as values in a variable domain. Then, a valid total assignment ρ completely specifies a configurable product.

The compilation of a configuration model to a BDD and the implementation of CVD functionality is described in [4]. The description is based on the Clab [11] configuration framework.

2.1 Cost-Bounded Configuration

Consider an extension of the configuration model where the selection of each choice $v \in D_i$ is associated with a cost.

A *cost bounded configuration model* C_c is a quadruple (X, D, F, c) where $C(X, D, F)$ is a standard configuration model and c is a cost function such that $c^i_v \in \mathbf{Z}$ denotes the integer cost of a choice $x_i = v$, $x_i \in X$, $v \in D_i$.

The cost of assignment ρ is defined as $c(\rho) = \sum_i c^i_{\rho(x_i)}$. Given the starting domains D_i, a partial user assignment ρ and a user-designated maximum cost C_{max}, the configurator should calculate and display the valid domains involving only those choices that can be extended to a configuration of maximum cost C_{max}. The cost-bounded valid domains are: $D_i^{\rho,C_{max}} = \{v \in D_i \mid \exists\rho'.(\rho' \models F \text{ and } \rho \cup \{(x_i, v)\} \subseteq \rho' \text{ and } c(\rho') \leq C_{max})\}$.

In [6] we described a polynomial-time algorithm for implementing this max-bounded CVD algorithm, where the maximum cost C_{max} can be interactively changed during user interaction. We did not explicitly encode a cost information as a propositional theory, thus avoiding the exponential increase in the size of existing BDD B. In practice, the implementation facilitates an interactive product configurator where a user can interactively limit the price of any configurable product, and get the same user functionality as in the standard configuration.

3 Infeasibility Result

An important user functionality is to prune choices based on more than one cost function. For example, in a product configuration scenario, a user might want to interactively limit both the maximum price and the weight of a product. We propose a direct extension of the cost bounded configuration model:

Definition 1 (2-cost bounded configuration model). *A* 2-cost bounded configuration model *$C_{a,b}$ is a 5-tuple (X, D, F, a, b) where $C(X, D, F)$ is a standard configuration model and a, b are cost functions such that $a^i_v, b^i_v \in \mathbf{Z}$ denote the integer costs of a choice $x_i = v$, $x_i \in X$, $v \in D_i$.*

The costs of assignment ρ are defined as $a(\rho) = \sum_i a^i_{\rho(x_i)}, b(\rho) = \sum_i b^i_{\rho(x_i)}$. Given the starting domains D_i, a partial user assignment ρ and a user-designated maximum costs cA, cB, the configurator should calculate and display the valid domains involving only those choices that can be extended to a configuration not exceeding cA and cB. The 2-cost bounded valid domains are: $D_i^{\rho,cA,cB} = \{v \in D_i \mid \exists\rho'.(\rho' \models F \text{ and } \rho \cup \{(x_i, v)\} \subseteq \rho' \text{ and } a(\rho') \leq cA \text{ and } b(\rho') \leq cB)\}$.

In this section we show that it is not possible to construct an algorithm for calculating valid domains $D_i^{\rho,cA,cB}$, in time polynomial in the size of the given BDD representing $C(X, D, F)$. To do so, we will first define an auxiliary problem used in the proofs to follow.

Definition 2 (Set-Sum-Partition). *Given an input instance defined by a finite set S of positive integers and by integer constants C_A and C_B such that $\sum_{c \in S} c \leq C_A + C_B$, the* set-sum-partition *problem asks whether S can be partitioned into subsets $S_A \subseteq S$, $S_B \subseteq S$ such that $\sum_{a \in S_A} a \leq C_A$, $\sum_{b \in S_B} b \leq C_B$.*

We will also use a *subset-sum* problem [12], defined by a finite set of positive integers S, and an integer C. The problem asks whether there exist a subset $S' \subseteq S$ such that $\sum_{c \in S'} c = C$.

Theorem 1. *The set-sum-partition problem is NP-hard.*

Proof. [1] We will prove our claim by reducing the NP-hard *subset-sum problem* [12] to the *set-sum-partition* (SSP). Given a subset-sum instance $S = \{c_1, \ldots, c_n\}, C$; take $D = \sum_{c \in S} c$ to be the sum of all elements in the set. Now, we can create an instance of the SSP problem by taking: $C_A = C, C_B = D - C$.

If SSP returns *true* then the subset-sum returns true. Namely, there exists a partition S_A, S_B, $\sum_{c \in S_A} c \leq C$, $\sum_{c \in S_B} c \leq D - C$, hence

$$D = \sum_{c \in S} c = \sum_{c \in S_A} c + \sum_{c \in S_B} c \leq C + (D - C) = D.$$

Therefore, $\sum_{c \in S_A} = C$ and the subset-sum returns true.

If SSP returns *false* then the subset-sum returns false. Namely, if there was a solution $S' \subseteq S$ to the subset-sum, it would suffice to take $S_A = S'$, $S_B = S \setminus S'$ to get a solution to the SSP. □

Now we will show that an algorithm calculating 2-cost valid domains (*2-cost CVD*) cannot have a polynomial time complexity w.r.t. the input BDD B representing the standard configuration model. The size of input B is $|B| = |V| + |E| + |X_b|$ where V is the set of BDD nodes, E is the set of edges and X_b refers to the set of Boolean variables encoding the finite-domain variables X (the notation is adopted from [6]).

Theorem 2 (Infeasibility result). *Given the configuration model $C(X, D, F)$ and its compiled solution space B, and the partial user assignment ρ, the problem of calculating valid domains $D_i^{\rho, cA, cB} = \{v \in D_i \mid \exists \rho' . \rho' \models F$ and $\rho \cup \{(x_i, v)\} \subseteq \rho'$ and $a(\rho) \leq cA$ and $b(\rho) \leq cB\}$ is NP-hard in the size of input $|B|$.*

Proof. We will prove our claim by reducing the NP-hard *set-sum-partition* (SSP) (Theorem 1) to the problem of calculating valid domains $D_i^{\rho, cA, cB}$ over a linear size BDD B.

Given the input instance to the SSP defined with a set of positive integers $S = \{c_1, \ldots, c_n\}$ and constants C_A, C_B, we will construct a configuration model $C(X, D, F)$ as follows: Define $2n$ Boolean variables $X = \{x_1, \ldots, x_{2n}\}$. Take the constraint set F to contain n constraints $\{x_1 \neq x_2, \ldots, x_{2n-1} \neq x_{2n}\}$. The compiled BDD representation B with respect to natural ordering $x_1 < \ldots < x_{2n}$ has $3n$ internal nodes.

For each $k = 1, \ldots, n$ define costs as follows:

$$a_1^{2k-1} = c_k, a_0^{2k-1} = 0, a_1^{2k} = 0, a_0^{2k} = 0$$

[1] The proof of Theorem 1 is to be credited to Andrzej Wasowski, who first brought it to our attention.

$$b_1^{2k-1} = 0, b_0^{2k-1} = 0, b_1^{2k} = c_k, b_0^{2k} = 0$$

Take $cA = C_A, cB = C_B$. Now, if calculated domains are non-empty, then there is at least one satisfying solution $\rho \models F$. To construct a solution to SSP it suffices to take:

$$S_A = \{c_k \mid \rho(x_{2k-1}) = 1, k = 1, \dots, n\},$$

$$S_B = \{c_k \mid \rho(x_{2k}) = 1, k = 1, \dots, n\}.$$

Since $x_{2k-1} \neq x_{2k}$ for any c_i, $i = 1, \dots, n$ exactly one of the variables x_{2k-1}, x_{2k} must be assigned 1. This implies that the sets S_A, S_B form a partition.

It remains to show that $\sum_{c \in S_A} c \leq C_A$ and $\sum_{c \in S_B} c \leq C_B$. Observe that it holds: $\sum_{c \in S_A} c = \sum_{k=1}^{n} c_k \cdot \rho(x_{2k-1})$. With respect to our choice of the cost function a it also holds:

$$c_k \cdot \rho(x_{2k-1}) = a^{2k-1}_{\rho(x_{2k-1})} = a^{2k-1}_{\rho(x_{2k-1})} + a^{2k}_{\rho(x_{2k})}. \tag{1}$$

Now, substituting equation (1) in $\sum_{k=1}^{n} c_k \cdot \rho(x_{2k-1})$ we get

$$\sum_{c \in S_A} c = \sum_{k=1}^{n} (a^{2k-1}_{\rho(x_{2k-1})} + a^{2k}_{\rho(x_{2k})}) = \sum_{k=1}^{2n} a^{k}_{\rho(x_k)} = a(\rho) \leq cA = C_A.$$

Hence, $\sum_{c \in S_A} c \leq C_A$. Analogously we show $\sum_{c \in S_B} c \leq C_B$.

Therefore, if calculating valid domains returns all non-empty domains then there is a solution to SSP.

The opposite also holds. If one of domains was empty, the SSP has no required partitioning. Namely, if there was partitioning S_A, S_B for the SSP problem, a solution $\rho \models F$ could be constructed as follows: $\rho(x_{2k-1}) = 1$ iff $c_k \in S_A$, $\rho(x_{2k}) = 1 - \rho(x_{2k-1})$, $k = 1, \dots, n$. In this case, every domain D_i would have at least one element, $\rho(x_i)$. This would contradict to initial assumption.

We have shown that the problem of calculating valid domains restricted by two cost functions is NP-hard w.r.t to a BDD representing the compiled configuration model. $\square$

The infeasibility result indicates that unless P=NP, there is no efficient way to deliver multi-cost bounding on top of compiled BDD.

4 Conclusions and Future Work

In this work we considered the tractability of calculating valid domains (CVD) over BDDs combined with the cost-bounding constraints. We extended our results from [6] by showing that it is NP-hard to deliver CVD functionality when a BDD is combined with multiple cost-bounding constraints.

In the future we plan to investigate approximation schemes for implementing multi-cost bounding. We also plan to further characterize constraints that can be efficiently combined with BDDs for calculating valid domains.

Acknowledgments

We would like to thank Andrzej Wasowski for bringing to our attention the Set-Sum-Partition problem. We also thank our reviewers for their comments.

References

1. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers **8** (1986) 677–691
2. Meinel, C., Theobald, T.: Algorithms and Data Structures in VLSI Design. Springer (1998)
3. Møller, J., Andersen, H.R., Hulgaard, H.: Product configuration over the internet. In: Proceedings of the 6th INFORMS. (2001)
4. Hadzic, T., Jensen, R., Andersen, H.R.: Notes on Calculating Valid Domains. `http://www.itu.dk/~tarik/cvd/cvd.pdf` (2006, online)
5. Dechter, R.: Constraint Processing. Morgan Kaufmann (2003)
6. Hadzic, T., Andersen, H.R.: A BDD-Based Polytime Algorithm for Cost-Bounded Interactive Configuration. In: Proceedings of The Twenty-First National Conference on Artificial Intelligence (AAAI-06). (2006) To appear.
7. Amilhastre, J., Fargier, H., Marquis, P.: Consistency restoration and explanations in dynamic CSPs-application to configuration. Artificial Intelligence **1-2** (2002)
8. Madsen, J.N.: Methods for interactive constraint satisfaction. Master's thesis, Department of Computer Science, University of Copenhagen (2003)
9. Subbarayan, S., Jensen, R.M., Hadzic, T., Andersen, H.R., Hulgaard, H., Møller, J.: Comparing two implementations of a complete and backtrack-free interactive configurator. In: CP'04 CSPIA Workshop. (2004) 97–111
10. Hadzic, T., Subbarayan, S., Jensen, R.M., Andersen, H.R., Møller, J., Hulgaard, H.: Fast backtrack-free product configuration using a precompiled solution space representation. In: PETO Conference, DTU-tryk (2004)
11. Jensen, R.M.: CLab: A C++ library for fast backtrack-free interactive product configuration. `http://www.itu.dk/people/rmj/clab/` (online)
12. Garey, M.R., Johnson, D.S.: Computers and Intractability-A Guide to the Theory of NP-Completeness. W H Freeman & Co (1979)

High-Level Nondeterministic Abstractions in C++

Student: Andrew See[1]
Supervisors: Laurent Michel[1] and Pascal Van Hentenryck[2]

[1] University of Connecticut, Storrs, CT 06269-2155
[2] Brown University, Box 1910, Providence, RI 02912

Abstract. This paper presents high-level abstractions for nondeterministic search in C++ which provide the counterpart to advanced features found in recent constraint languages. The abstractions have several benefits: they explicitly reflect the nondeterministic nature of the code, avoid the need for goal interpreters, simplify debugging, and are efficiently implementable using macros and continuations. Their efficiency is demonstrated by comparing their performance with the C++ library GECODE. A more detailed discussion of this work is presented in the full length paper [15].

1 Introduction

The ability to specify search procedures has been a fundamental asset of constraint programming languages since their inception (e.g., [1, 2, 12]) and a differentiator compared to earlier tools such as Alice [6] and MIP systems where search was hard-coded in the solver. Indeed, by programming the search, users may define problem-specific branching procedures and heuristics, exploit unconventional search strategies, break symmetries dynamically, and specify termination criteria for the problem at hand. The last two decades have also witnessed significant progress in this area (e.g., [5, 7, 8, 11, 13, 14]).

The embedding of constraint programming in mainstream languages such as C++ has also been a fundamental step in its acceptance, especially in industry. With constraint programming libraries, practitioners may use familiar languages and environments, which also simplifies the integration of a constraint programming solution within a larger application. ILOG SOLVER [9] is the pioneering system in this respect: it showed how the nondeterministic abstractions of constraint logic programming (e.g., goals, disjunction, and conjunction) can be naturally mapped into C++ objects. To specify a search procedure, users thus define C++ objects called goals, and combine them with logical connectives such as `or` and `and`. In recent years, constraint programming libraries have been enhanced to accommodate search strategies [8, 3] (originally proposed in OZ [11]) and high-level nondeterministic abstractions [7] (originally from OPL [13]).

However these and other similar libraries, while widely successful, still have two inconveniences as far as specifying search procedures. On the one hand, constraint programming libraries require users to create objects (e.g., goals) to specify search procedures. This may obscure the natural nondeterministic structure of the code and may produce some non-trivial interleaving of C++ code and library functions. On the other hand, their implementations typically rely on an interpreter, complicating the debugging process which alternates between library and user code, while not showing users the inherent nondeterministic structure of their applications.

This paper is an attempt to mirror, in constraint programming libraries, the high-level nondeterministic abstractions of modern constraint programming languages. The paper shows that it is indeed possible and practical to design a search component in C++ that

- reflects the nondeterministic structure of the application directly;
- avoids the need for a goal interpreter;
- simplifies the debugging process;
- is as efficient as existing libraries.

The technical idea underlying the paper is to map the nondeterministic abstractions of COMET [14] into C++ using macros and continuations. Since continuations are not primitive in C++, it is necessary to show how they can be implemented directly in the language itself.

The rest of the paper is organized as follows. Section 2 presents the nondeterministic abstractions and their benefits. Section 3 briefly discusses how to implement continuations in C++. Section 4 presents the experimental results which shows that the nondeterministic abstractions can be implemented efficiently and compare well with the search implementation of GECODE.

2 The Search Abstractions

This section describes the search abstractions in C++. Section 2.1 starts by describing the nondeterministic abstractions used to define the search tree to explore. These abstractions are parameterized by a search controller that specifies how to explore the search tree. Search controllers are discussed in the full length paper and are presented in depth in [14].

2.1 Nondeterministic Abstractions

The nondeterministic abstractions are mostly modelled after OPL [13].

Static Choices The `try` construct creates a binary search node representing the choice between two alternatives. The snippet

```
TRY(sc)
  cout << "yes" <<endl;
OR(sc)
  cout << "no" <<endl;
ENDTRY(sc)
```

nondeterministically produces two lines of output: the first choice displays `yes`, while the second one displays `no`. When the search controller `sc` implements a depth-first strategy, the instruction first executes the first choice, while the second choice is executed upon backtracking.

```
TRYALL(<sc>, <param>, <low>, <high>, <condition>, <ordering>)
  [<Statement>]*
ENDTRYALL(<sc>)
```

Fig. 1. The Syntax of the TRYALL Construct.

```
0.  int x[3] = -1, -1, -1;                                    0,0,0
1.  EXPLOREALL(sc)                                            0,0,1
2.    for(int i=0; i<3 ; i++) {                               0,1,0
3.      TRY(sc)                                               0,1,1
4.        x[i] = 0;                                           1,0,0
5.      OR(sc)                                                1,0,1
6.        x[i] = 1;                                           1,1,0
7.      ENDTRY(sc)                                            1,1,1
8.    }
9.    cout << x[0]<<',''<<x[1]<<','<<x[2]<<endl;
10. ENDEXPLOREALL(sc)
```

Fig. 2. An Example of Encapsulated Search.

Dynamic Choices The TRYALL construct iterates over a range of values, filtering and ordering the candidate values dynamically. Figure 1 depicts the general syntax of the construct. The first parameter `<sc>` is the search controller. The `<param>` argument is the local variable used to store the selected value. Parameters `<low>` and `<high>` define the range of values, while `<condition>` holds for those values to consider in the range. Finally, the expression `<ordering>` specifies the order in which to try values. For instance, the snippet

```
TRYALL(sc, p, 0, 5, (p%2)==0, -p)
  cout << "p = "<< p << endl;
ENDTRYALL(sc)
```

nondeterministically produces three lines of output: `p=4`, `p=2`, and `p=0`. The instruction binds the parameter `p` to values 0 through 5 in increasing order of `-p` and skips those violating the condition `(p%2)==0`.

Encapsulated Search The EXPLOREALL construct implements an encapsulated search that initializes the search controller and produces all solutions to its body. Figure 2 illustrates an encapsulated search for implementing a simple labeling procedure. The body of the encapsulated search (lines 2–9) iterates over the values 0..2 (line 2) and nondeterministically assigns `x[i]` to 0 or 1 (lines 3–7). Once all the elements in array `x` are labeled, the array is displayed in line 9. The right part of Figure 2 depicts the output of the encapsulated search for a depth-first search controller. Other similar constructs implement encapsulated search to find one solution or to find a solution optimizing an objective function.

It is important to emphasize some benefits of the nondeterministic abstractions. First, the code freely interleaves nondeterministic abstractions and arbitrary C++ code: it does not require the definition of classes, objects, or goals. Second, the nondeterministic structure of the program is clearly apparent, simplifying debugging with traditional support from software environments. In particular, C++ debuggers can be used on these nondeterministic programs, enabling users to follow the control flow of their programs at a high level of abstraction. For example, by line stepping, users can step over the implementation of the `TRYALL` statement into the body as they would expect from the syntax, since the macros expand to a single line of `C++` code. Breakpoints also function normally with the nondeterministic abstractions. The details of the macros implementing the nondeterministic abstractions are presented in the full length version.

3 Search Nodes as `C++` Continuations

As in Comet [14], the nondeterministic abstractions are implemented using continuations. Since `C++` does not support continuations natively, this section describes briefly how to implement continuations in the language itself. Recall that a continuation captures the current state of computation, i.e., the program counter, the stack, and the registers (but not the heap). Once captured, the continuation can be executed at a later time, causing the computation to resume from the previous state. This ability to return to search nodes is the fundamental building block of the search abstractions described in the previous section.

The implementation of continuations uses the standard `C` functions `setjmp` and `longjmp` and is thus portable to any architecture with correct implementations of these functions. In absence of `setjmp` and `longjmp`, continuations can be implemented using `getContext` and `setContext`, or in assembly. Our implementation based on `setjmp/longjmp` has been successfully tested on three different hardware platforms (Intel x86, PowerPC, and UltraSparc) and four operating systems (Linux, Windows XP, Solaris, and OSX). The only platform where it fails is Itanium because its implementation of `setjmp` and `longjmp` is non-conformant. Further details regarding the implementation are presented in the full length paper.

It is important to note that the implementation of Ilog Solver [4] also uses `setjmp` and `longjmp` [10]. The novelty here is to save the stack before calling `setjmp` and restoring the stack after calling `longjmp`. The benefits are twofold. On the one hand, it enables the implementation of high-level nondeterministic abstractions such as `tryall` in `C++`. On the other hand, it enables continuations to be called at any time during the execution even if the stack has fundamentally changed. As a result, continuations provide a sound basis for complex search procedures jumping from node to node arbitrarily in the search tree.

4 Experimental Results

This section presents the experimental results demonstrating the efficiency of the implementation. It first shows that the cost of using continuations is not prohibitive. Then, it demonstrates that the abstractions are comparable in efficiency to the search procedures of existing constraint libraries. The CPU Times are given on a Pentium IV 2.0 GHz running Linux 2.6.11.

n	16	18	20	22	24	26	28	30	32
R	.007	.028	.152	1.486	.405	.443	3.748	78.900	133.86
N	.014	.063	.308	2.878	.731	.762	6.175	125.64	205.82
$(N-R)/R$	1.00	1.25	1.026	0.937	0.805	0.720	0.648	0.592	0.538

Table 1. The Pure Cost of the Nondeterministic Abstractions

n	16	18	20	22	24	26	28	30	32
Gecode	.06	.20	.98	7.83	2.08	2.08	16.34	301.27	507.08
$ND + Gecode$	0.5	.19	.97	7.55	2.02	1.96	16.28	292.80	495.69

Table 2. Performance Comparison in Seconds on GECODE Only.

On the Efficiency of Continuations One possible source of inefficiency for the nondeterministic abstractions is the overhead of capturing and restoring continuations. To quantify this cost, we use a simple backtrack search for the queens problem and we compare a search procedure written in C++ (and thus with a recursive style) with a search procedure using the nondeterministic abstractions (and thus with an iterative style). Table 1 shows the runtime of the recursive (R) and nondeterministic (N) search procedures and the percentage increase in CPU time. The results show that the percentage increase in CPU time decreases as the problem size grows and goes down to 54% for the 32-queens problem. These results are quite interesting, since they use a mainstream, non-garbage collected, language which is supposed to be highly efficient. Moreover, these programs do not involve any constraint propagation and do not need to save and restore the states of domain variables and constraints. As such, these tests represents the pure cost of the abstraction compared to the hand-coded implementation.

Programming Search Engines We now compare the nondeterministic abstractions with the search engine of an existing library: GECODE [3]. In order to focus the comparison on the search components, this experiment only uses GECODE as the underlying solver. It compares the built-in implementation of depth-first search in GECODE with an implementation using our nondeterministic abstractions. Recall that GECODE manipulates computation spaces representing the search tree. The following `C++` code

```
0.  EXPLORE(gecode)
1.    while (gecode->needBranching()) {
2.      int alt = gecode->getNbAlternatives();
3.      TRYALL4(gecode, a, 0, alt-1)
4.        gecode->tryCommit(a);
5.      ENDTRYALL4(gecode);
6.  }
7.  ENDEXPLORE(gecode);
```

is an implementation of depth-first search for GECODE that uses our nondeterministic abstraction. The code iterates branching until the tree is fully explored (line 1). To branch, the code retrieves the number of alternatives (line 2) and performs a `TRYALL` to try each alternative (line 4). The code uses a `gecode` controller to clone and restore the spaces appropriately (which is not shown for space reasons). Note also the combination of a `C++ while` instruction with `TRYALL`.

Table 2 depicts the computational results. This evaluation has the merit of comparing the search procedures with exactly the same constraint solver and the search procedure coded by the designer of the library. The nondeterministic abstractions are slightly more efficient than the builtin implementation of Gecode, although they perform exactly the same number of clones, failures, and propagation calls. The results thus indicate that the nondeterministic abstractions are not only expressive and natural; they are also very efficient.

Acknowledgments

This work was supported by the National Science Foundation grant DMI-0423607.

References

1. A. Colmerauer. Opening the Prolog-III Universe. *BYTE Magazine*, 12(9), 1987.
2. N.C. Heintze, S. Michaylov, and P.J. Stuckey. CLP($\Re$) and some Electrical Engineering Problems. In *ICLP-87*, May 1987.
3. http://www.gecode.org/. Generic Constraint Development Environment, 2005.
4. Ilog Solver 4.4. Reference Manual. Ilog SA, Gentilly, France, 1998.
5. F. Laburthe and Y. Caseau. SALSA: A Language for Search Algorithms. In *CP'98*, Pisa, October 1998.
6. J-L. Lauriere. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1):29–127, 1978.
7. L. Michel and P. Van Hentenryck. Modeler++: A Modeling Layer for Constraint Programming Libraries. In *CP-AI-OR'2001*, April 2001.
8. L. Perron. Search Procedures and Parallelism in Constraint Programming. In *CP'99*, pages 346–360, Alexandria, Virginia, October 1999.
9. J-F. Puget. A C++ Implementation of CLP. In *Proceedings of SPICIS'94*, Singapore, November 1994.
10. J.-F.. Puget. Personal Communication, March 2006.
11. C. Schulte. Programming Constraint Inference Engines. In *CP'97*, 519–533, Linz, Austria, October 1997.
12. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.
13. P. Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, Cambridge, Mass., 1999.
14. P. Van Hentenryck and L. Michel. Nondeterministic Control for Hybrid Search. In *CP-AI-OR'05*, Prague, May 2005.
15. L. Michel and A. See and P. Van Hentenryck High-Level Nondeterministic Abstractions in C++. In *CP'06*, Nantes, France, September 2006.

A Comparison of Time-Space Schemes

Student: Robert Mateescu
Supervisor: Rina Dechter

School of Information and Computer Science
University of California, Irvine, CA 92697-3425
{dechter,mateescu}@ics.uci.edu

Abstract. We investigate two parameterized algorithmic schemes for graphical models that can accommodate trade-offs between time and space: 1) AND/OR Cutset Conditioning (**AOC(i)**) and 2) Variable Elimination with Conditioning (**VEC(i)**). We show that **AOC(i)** is better than the vanilla versions of **VEC(i)**, and use the guiding principles of **AOC(i)** to improve **VEC(i)**. Finally, we show that the improved versions of **VEC(i)**can be simulated by **AOC(i)**, which emphasizes the unifying power of the AND/OR framework.

1 Introduction

In this paper we compare AND/OR search [1] and alternating elimination and conditioning controlled by induced-width w (**VEC**) [2, 3]. By analyzing them using the context minimal AND/OR graph data structure [4], we show that **VEC(i)** can be improved via the AND/OR search principle and by careful caching, to the point that both schemes become identically good. We show that the recently proposed AND/OR cutset conditioning [5] (improving cutset, and w-cutset schemes) can simulate any execution of **VEC**, if the latter is augmented with AND/OR search over the conditioning variables.

The analysis is done in the general context of graphical models, assuming no determinism. This is still useful in the context of constraint networks, providing a comparison of the total space that the algorithm might need to traverse.

2 Preliminaries

Definition 1 (graphical model). *A* graphical model *is a 3-tuple* $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$*, where:* $\mathbf{X} = \{X_1, \ldots, X_n\}$ *is a set of variables;* $\mathbf{D} = \{D_1, \ldots, D_n\}$ *is the set of their finite domains of values;* $\mathbf{F} = \{f_1, \ldots, f_r\}$ *is a set of real-valued functions.*

Definition 2 (pseudo tree). *A* pseudo tree *of a graph* $G = (\mathbf{X}, E)$ *is a rooted tree* $\mathcal{T}$ *having the same set of nodes* $\mathbf{X}$*, such that every arc in* E *is a backarc in* $\mathcal{T}$ *(i.e., it connects nodes on the same path from root).*

Definition 3 (induced graph and induced width). *An* ordered graph *is a pair* (G, d)*, where* G *is an undirected graph, and* $d = (X_1, ..., X_n)$ *is an ordering of the nodes. The* width of a node *in an ordered graph is the number of neighbors that precede it in the ordering. The* width of an ordering d*, denoted* $w(d)$*, is the maximum width over* *all nodes. The* induced width of an ordered graph, $w^*(d)$, *is the width of the induced ordered graph obtained as follows: for each node, from last to first in* d, *its preceding neighbors are connected in a clique. The* induced width of a graph, w^*, *is the minimal induced width over all orderings. The induced width is equal to the* treewidth *of a graph.*

3 Description of Algorithms

AOC and **VEC**are both parameterized memory intensive algorithms that need to use space in order to achieve the worst case time complexity of $O(n\ k^{w^*})$, where k bounds domain size. The task that we consider is one that is equivalent to solutions counting.

3.1 AND/OR Cutset Conditioning - AOC

The AND/OR search space is a recently introduced [1, 4, 5] unifying framework for advanced algorithmic schemes for graphical models. Its main virtue consists in exploiting independencies between variables during search, which can provide exponential speedups over traditional search methods oblivious to problem structure.

Given a graphical model $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F} \rangle$, its primal graph G and a pseudo tree $\mathcal{T}$ of G, the associated AND/OR search tree has alternating levels of OR and AND nodes. The OR nodes are labeled correspond to branching according to values of variables, while the AND nodes correspond to problem decomposition The structure of the AND/OR search tree is based on the underlying pseudo tree $\mathcal{T}$. The AND/OR search tree can be traversed by a depth first search algorithm, thus using linear space.

Theorem 1 ([6–8, 5]). *Given a graphical model* $\mathcal{M}$ *and a pseudo tree* $\mathcal{T}$ *of depth* m, *the size of the AND/OR search tree based on* $\mathcal{T}$ *is* $O(n\ k^m)$, *where* k *bounds the domains of variables. A graphical model of treewidth* w^* *has a pseudo tree of depth at most* $w^* \log n$, *therefore it has an AND/OR search tree of size* $O(n\ k^{w^* \log n})$.

The AND/OR search tree may contain *unifiable* nodes, that root identical conditioned subproblems. When unifiable nodes are merged, the search space becomes a graph. The depth first search algorithm can therefore be modified to cache previously computed results, and retrieve them when the same nodes are encountered again. Some unifiable nodes can be identified based on their *contexts* [8]. We only use caching based on *OR context*, denoted as $context(X) = [X_1 \dots X_k]$, which is the set of ancestors of X in $\mathcal{T}$ ordered descendingly, that are connected in the primal graph to X or to descendants of X. The *context minimal* AND/OR graph is obtained by merging all the context unifiable OR nodes. An example will appear later in Figure 4.

Theorem 2 ([7, 1]). *Given a graphical model* $\mathcal{M}$, *its primal graph* G *and a pseudo tree* $\mathcal{T}$, *the size of the context minimal AND/OR search graph based on* $\mathcal{T}$ *is* $O(n\ k^{w^*_{\mathcal{T}}(G)})$, *where* $w^*_{\mathcal{T}}(G)$ *is the induced width of* G *over the depth first traversal of* $\mathcal{T}$, *and* k *bounds the domain size.*

AND/OR Cutset Conditioning (**AOC**) [5] is a search algorithm that combines AND/OR search spaces with cutset conditioning. The conditioning (cutset) variables form a *start* pseudo tree. The remaining variables (not belonging to the cutset), have bounded conditioned context size that can fit in memory.

Given a primal graph G, of a graphical model and a pseudo tree $\mathcal{T}$ of G, a *start pseudo tree* $\mathcal{T}_{start}$ is a connected subgraph of $\mathcal{T}$ that contains the root of $\mathcal{T}$.

We can now define algorithm **AOC(i)**, that depends on a parameter i that bounds the maximum size of a context that can fit in memory. Given a pseudo tree $\mathcal{T}$, we first find a start pseudo tree $\mathcal{T}_{start}$ such that the context of any node not in $\mathcal{T}_{start}$ contains at most i variables that are not in $\mathcal{T}_{start}$. This can be done by starting with the root of $\mathcal{T}$ and then including as many descendants as necessary in the start pseudo tree until the previous condition is met. $\mathcal{T}_{start}$ now forms the cutset, and when its variables are instantiated, the remaining conditioned subproblem has induced width bounded by i. The cutset variables can be explored by linear space (no caching) AND/OR search, and the remaining variables by using full caching, of size bounded by i. The cache tables need to be deleted and reallocated for each new conditioned subproblem.

Adaptive Caching for AND/OR Search The cutset principle inspires a refined caching scheme for AND/OR search, which we will call *adaptive caching* (in the sense that it adapts to the available memory), that caches some values even at nodes with contexts greater than the bound i that defines the memory limit. Lets assume that $context(X) = [X_1 \dots X_k]$ and $k > i$. During search, when variables $X_1, \dots, X_{k-i}$ are instantiated, they can be regarded as part of a cutset. The problem rooted by X_{k-i+1} can be solved in isolation, like a subproblem in the cutset scheme, after the variables $X_1, \dots, X_{k-i}$ are assigned their current values in all the functions. In this subproblem, $context(X) = [X_{k-i+1} \dots X_k]$, so it can be cached within space bounded by i. However, when the search retracts to X_{k-i} or above, the cache table for X needs to be deleted and reallocated when a new subproblem rooted at X_{k-i+1} is solved.

Algorithm **AOC(i)** is essentially an AND/OR search with adaptive caching bounded by i for all variables. The AND/OR search algorithm that caches only the full contexts bounded by i is **AO(i)**.

Proposition 1. *AOC(i) increases space requirements linearly compared to* ***AO(i)****, but the time savings can be exponential.*

3.2 Variable Elimination with Conditioning - VEC

Variable Elimination with Conditioning (**VEC**) [2, 3] is an algorithm that combines the virtues of both inference and search. **VEC** works by interleaving elimination and conditioning of variables. Typically, given an ordering, it prefers the elimination of a variable whenever possible, and switches to conditioning whenever space limitations require it, and continues in the same manner until all variables have been processed. We say that the conditioning variables form a *conditioning set*, or *cutset* (this can be regarded as a w*-cutset*, where w defines the induced width of the problems that can be handled by elimination). The vanilla version of **VEC** will also be called **VEC-OR** because the cutset is explored by OR search rather than AND/OR. When there are no conditioning variables, **VEC** becomes the well known Variable Elimination (**VE**) algorithm. In this case **AOC** also becomes the usual AND/OR graph search (**AO**).

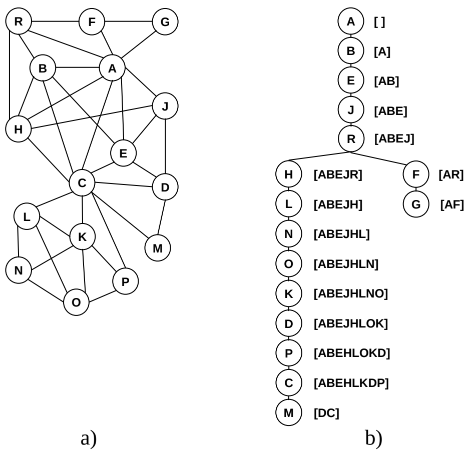


Fig. 1. Primal graph and pseudo tree

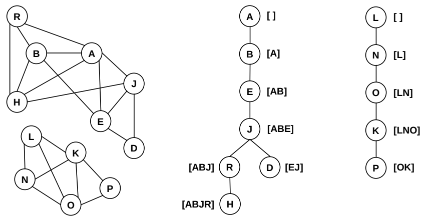


Fig. 2. Components after conditioning on C

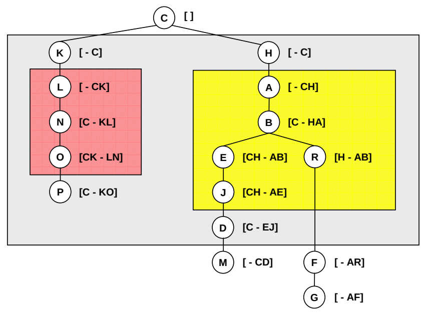


Fig. 3. Pseudo tree for **AOC(2)**

Theorem 3 (VE and AO are identical [4]). *Given a graphical model with no determinism and a pseudo tree, **VE** traverses the full context minimal graph bottom-up by layers (breadth first), while **AO** is a top-down depth-first search that explores (and records) the full context minimal graph as well.*

4 AOC(i) Compared to VEC(i)

We will begin by following an example. Consider the graphical model given in Figure 1a having binary variables, the ordering $d_1 = (A, B, E, J, R, H, L, N, O, K, D, P, C, M, F, G)$, and the space limitation $i = 2$. The pseudo tree corresponding to this ordering is given in Figure 1b. The context of each node is shown in square brackets.

If we apply **VEC** along d_1 (processing from last to first), variables G, F and M can be eliminated. However, C cannot be eliminated, because it would produce a function with scope equal to its context, $[ABEHLKDP]$, violating the bound $i = 2$. **VEC** switches to conditioning on C and all the functions that remain to be processed are modified accordingly, by instantiating C. The primal graph has two connected components now, shown in Figure 2. Notice that the pseudo trees are based on this new graph, and their shape change from the original pseudo tree.

Continuing with the ordering, P and D can be eliminated (one variable from each component), but then K cannot be eliminated. After conditioning on K, variables O, N and L can be eliminated (all from the same component), then H is conditioned (from the other component) and the rest of the variables are eliminated. To highlight the conditioning set, we will box its variables when writing the ordering, $d_1 = (A, B, E, J, R, \boxed{H}, L, N, O, \boxed{K}, D, P, \boxed{C}, M, F, G)$.

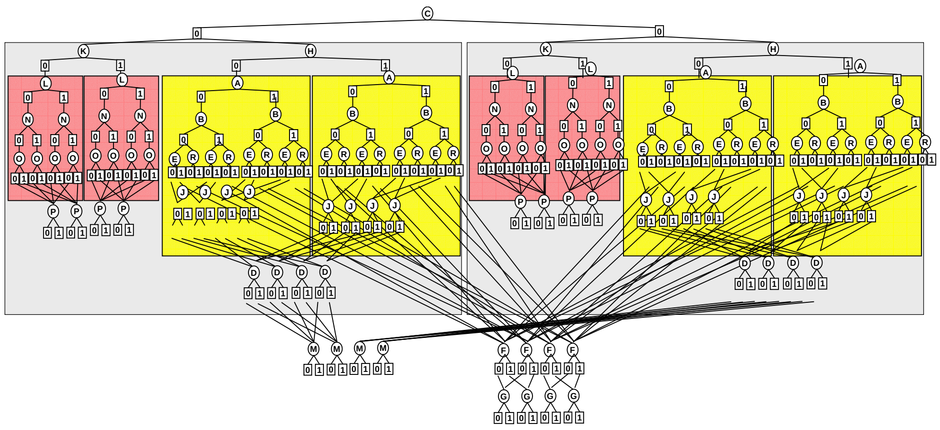


Fig. 4. Context minimal graph

If we take the conditioning set $[HKC]$ in the order imposed on it by d_1, reverse it and put it at the beginning of the ordering d_1, then we obtain:

$$d_2 = \left(\boxed{C}, \underline{\left[\boxed{K}, \underline{\left[\boxed{H}, \underline{[A,B,E,J,R]}_H, L,N,O\right]}_K, D,P\right]}_C, M,F,G\right)$$

where the indexed squared brackets together with the underlines represent subproblems that need to be solved multiple times, for each instantiation of the index variable.

So we started with d_1 and bound $i = 2$, then we identified the corresponding conditioning set $[HKC]$ for **VEC**, and from this we arrived at d_2. We are now going to use d_2 to build the pseudo tree that guides **AOC(2)**, given in Figure 3. The outer box corresponds to the conditioning of C. The inner boxes correspond to conditioning on K and H, respectively. The context of each node is given in square brackets, and the *2-context* is on the right side of the dash. For example, $context(J) = [CH\text{-}AE]$, and 2-$context(J) = [AE]$. The context minimal graph corresponding to the execution of **AOC(2)** is shown in Figure 4.

We can now follow the execution of both **AOC** and **VEC** along this context minimal graph. After conditioning on C, **VEC** solves two conditioned subproblems (one for each value of C), which are the ones shown on gray backgrounds. However, the vanilla version **VEC-OR** is less efficient than **AOC**, because it uses an OR search over the cutset variables, rather than AND/OR. In our example, the subproblem on A, B, E, J, R would be solved eight times by **VEC-OR**, once for each instantiation of C, K and H, rather than four times. It is now easy to make the first improvement to **VEC**, so that it uses an AND/OR search over the conditioning set, an algorithm we call **VEC-AO**.

Let's look at one more condition that needs to be satisfied for the two algorithms to be identical. If we change the ordering to $d_3 = (A, B, E, J, R, \boxed{H}, L, N, O, \boxed{K}, D, P, F, G, \boxed{C}, M)$, ($F$ and G are eliminated after conditioning on C), then the pseudo tree is the same as before, and therefore the context minimal graph for **AOC** is still the one shown in Figure 4. However, **VEC-AO** would require more effort, because the elimination of G and F is performed twice now (once for each instantiation of C), rather than once as was for ordering d_1. This shortcoming can be eliminated by defining a pseudo tree based version for **VEC**, rather than one based on an ordering. The final algorithm, **VEC(i)** is given below (where $N_G(X_i)$ is the set of neighbors of X_i in the graph G). Note that the guiding pseudo tree is regenerated after each conditioning.

```
Algorithm VEC(i)
input  : $\mathcal{M}=\langle \mathbf{X},\mathbf{D},\mathbf{F}\rangle$; $G=(\mathbf{X},E)$; $d=(X_1,\ldots,X_n)$; $i$
output: Solutions count.
generate the bucket tree $\mathcal{T}$ for $d$;
while $\mathcal{T}$ not empty do
    if (($\exists X_i$ leaf in $\mathcal{T}$)$\wedge$($|N_G(X_i)|\leq i$)) then eliminate $X_i$ else pick $X_i$ leaf of $\mathcal{T}$;
      for each $x_i \in D_i$ do
          assign $X_i = x_i$;
          call VEC(i) on each connected component of conditioned subproblem
```

Theorem 4 (AOC(i) can simulate VEC(i)). *Given a graphical model $\mathcal{M} = \langle \mathbf{X},\mathbf{D},\mathbf{F}\rangle$ and an execution of **VEC(i)**, there exists a pseudo tree that guides an execution of **AOC(i)** that traverses the same context minimal graph.*

5 Conclusion

We have compared two parameterized algorithmic schemes for graphical models that can accommodate time-space trade-offs. They have emerged from seemingly different principles: **AOC(i)** is search based and **VEC** combines search and inference.

We show that if the graphical models contain no determinism, **AOC(i)** can have a smaller time complexity than the vanilla versions of **VEC(i)**. This is due to a more efficient exploitation of the graphical structure of the problem through AND/OR search, and the adaptive caching scheme that benefits from the cutset principle. These ideas can be used to enhance **VEC(i)**. We show that if **VEC(i)** uses AND/OR search over the conditioning set and is guided by the pseudo tree data structure, then there exists an execution of **AOC(i)** that is identical to it. AND/OR search with adaptive caching (**AOC(i)**) emerges therefore as a unifying scheme, never worse than **VEC(i)**. All the analysis was done by using the context minimal data structure, which provides a powerful methodology for comparing the algorithms.

References

1. Dechter, R., Mateescu, R.: Mixtures of deterministic-probabilistic networks and their and/or search space. In: UAI'04. (2004) 120–129
2. Rish, I., Dechter, R.: Resolution vs. search; two strategies for sat. Journal of Automated Reasoning **24(1/2)** (2000) 225–275
3. Larrosa, J., Dechter, R.: Boosting search with variable-elimination. Constraints **7(3-4)** (2002) 407–419
4. Mateescu, R., Dechter, R.: The relationship between and/or search and variable elimination. In: UAI'05. (2005) 380–387
5. Mateescu, R., Dechter, R.: And/or cutset conditioning. In: IJCAI'05. (2005) 230–235
6. Freuder, E.C., Quinn, M.J.: Taking advantage of stable sets of variables in constraint satisfaction problems. In: IJCAI'85. (1985) 1076–1078
7. Bayardo, R., Miranker, D.: A complexity analysis of space-bound learning algorithms for the constraint satisfaction problem. In: AAAI'96. (1996) 298–304
8. Darwiche, A.: Recursive conditioning. Artificial Intelligence **125**(1-2) (2001) 5–41

Encoding Binary Quantified CSPs as Quantified Boolean Formulae

Student: Peter Nightingale, Supervisor: Ian Gent, {pn, ipg}@dcs.st-and.ac.uk

School of Computer Science, University of St Andrews, Fife KY16 9SX, Scotland

Abstract. Quantified problems such as QBF and quantified CSP have been gaining attention recently. Both of these problems can be used to model PSPACE problems arising in AI, for example planning under uncertainty, adversarial game playing and model checking. QBF has received more attention so far, so the search algorithms are better developed, and to take advantage of this Gent et al [1] developed an encoding of binary QCSP as QBF. The encoding progressed through three iterations, *global acceptability*, *local acceptability* and *adapted log*. In this paper I present the next step in that sequence, simplifying and improving adapted log. The new encoding compares favourably with the previous encodings and the direct solution algorithm QCSP-Solve [7].

1 Introduction

Quantified problems such as Quantified Boolean Formulae (QBF) and Quantified Constraint Satisfaction Problems (QCSP) have been gaining attention recently. Both of these problems can be used to model PSPACE problems arising in AI, for example planning under uncertainty, adversarial game playing and model checking. QBF has received more attention so far, so the search algorithms are arguably more advanced (for example, rules specific to QBF [5], conflict and solution directed backjumping [4] and efficient data structures [8] have been developed). To take advantage of this, Gent et al [1] developed an encoding of binary QCSP to QBF. Note that binary QCSP is not trivial; it is PSPACE-complete as is general QCSP and QBF. The encoding progressed through three iterations, *global acceptability*, *local acceptability* and *adapted log*. In this paper I present the next step in that sequence, called *enhanced log*, simplifying and improving adapted log and gaining a performance improvement.

Enhanced log includes an idea from the enhanced binary encoding presented by Frisch et al [2] for encoding non-Boolean satisfiability (SAT) to Boolean SAT. Incorporating the enhanced binary idea allows the new encoding to take advantage of the pure literal rule present in some QBF solvers to eliminate the need for indicator variables.

First I present the encoding, then empirically evaluate it on a suite of random binary QCSPs, in comparison with the adapted log encoding and a direct solver, QCSP-Solve [7]. The enhanced log encoding outperforms the adapted log by two orders of magnitude on average, and can also significantly outperform QCSP-Solve.

2 Background

In the QCSP, variables may be universally ('for all values', $\forall$) and existentially ('there exists a value', $\exists$) quantified in sequence. For example, the QCSP below reads as 'there exists a value of x, such that for all values of y, there exists a value of z such that the single constraint is satisfied'.

$$\exists x \in \{1..5\}, \forall y \in \{1..5\}, \exists z \in \{1..5\} : 2x + 5y + 3z = 30$$

In this case the QCSP is unsatisfiable, since no value of x extends to 5 satisfying assignments for each value of y. QBF is similar, however the variables have domain $\{T, F\}$ and constraints are disjuncts of positive or negative literals. QBF is therefore a subset of QCSP. For both QCSP and QBF, representing the set of constraints as $\mathcal{C}$, the semantics of the QCSP/QBF $\mathcal{QC}$ can be defined recursively as follows. If $\mathcal{Q}$ is of the form $\exists x_1 Q_2 x_2, \ldots, Q_n x_n$ then $\mathcal{QC}$ is true iff there exists a value $a \in D(x_1)$ such that $Q_2 x_2, \ldots, Q_n x_n \mathcal{C}[x_1 = a]$[1] is true. If $\mathcal{Q}$ is of the form $\forall x_1 Q_2 x_2, \ldots, Q_n x_n$ then $\mathcal{QC}$ is true iff for all values $a \in D(x_1)$, $Q_2 x_2, \ldots, Q_n x_n \mathcal{C}[x_1 = a]$ is true. If $\mathcal{C}$ is empty then the problem is true.

3 Encoding the constraints

For both existential and universal QCSP variables, there is a set of QBF variables x_i^v representing a variable v, with the intention that $x_i^v = T \iff v = i$. The other aspects of representing a QCSP variable are covered in the next section.

The direct and support encodings [3] of the constraints simulate forward checking and arc-consistency respectively (i.e. unit propagation over the constraint clauses sets x_i^v variables to the same effect as forward checking or arc-consistency in the original QCSP). Gent et al [1] did not include the support encoding because it performs poorly with all the encodings described there. There is no reason to believe it would be better with enhanced log, since it cripples the pure literal rule for x_i^v variables. I omit it here for this reason.

A constraint C between two variables a and b with domains A and B is represented by the relation $R_{a,b} \subseteq A \times B$ containing the allowed pairs. The direct encoding forbids each of the disallowed pairs.

– **Conflict clauses** For all tuples $\langle i, j \rangle \notin R$,

$$\neg x_i^a \vee \neg x_j^b$$

For comparison the adapted log encoding has conflict clauses $\neg x_i^a \vee \neg x_j^b \vee z_v$ where z_v is the variable indicating an unacceptable assignment to universal variable v or some outer universal variable. For constraints involving existentials we use z_v for the last universal v quantified before the first existential QCSP variable; for constraints involving a universal v we use z_v, or the innermost if the constraint involves two universals. All indicator variables are existential and in the innermost block.

[1] Where $[x_1 = a]$ means the assignment of a to x_1.

4 Encoding the QCSP variables

Each existential QCSP variable e is encoded by a set of Boolean variables $x^e_1 \dots x^e_d$ where d is the domain size of v, and $x^e_i = T \iff e = i$. The x^e variables are existentially quantified together. At least one of the x^e variables must be assigned T, so the following clause is added to the formula. The conflict clauses contain only negative literals, so it is sufficient that they are all satisfied and each set of x variables has at least one assigned T. Adding clauses to state that exactly one is assigned true is not necessary. Also, the additional clauses would stop the pure literal rule working effectively on the x^e variables. This is identical to the adapted log encoding.

– **At least one** clause

$$x^e_1 \vee x^e_2 \vee \dots \vee x^e_d$$

Each universal variable u is encoded, as in the adapted log encoding, by $\lceil \log_2 d \rceil$ variables b^u_i which are universally quantified together. The order of variables is preserved in the quantifier sequence. We also introduce $x^u_1 \dots x^u_d$ variables as before, which are existentially quantified at the end of the quantifier sequence. These x^u variables are used in the constraint clauses, to avoid having several b^u literals in conjunction to represent a value, then distributing conjunction over disjunction.

The b^u_i variables are channelled to the x^u variables with the following clause set. This is given for the case where $d = 5$.

– **Channelling clauses**

$$\begin{gathered} x^u_1 \vee b^u_2 \vee b^u_1 \vee b^u_0 \\ x^u_2 \vee b^u_2 \vee b^u_1 \vee \neg b^u_0 \\ x^u_3 \vee b^u_2 \vee \neg b^u_1 \\ x^u_4 \vee \neg b^u_2 \vee b^u_1 \\ x^u_5 \vee \neg b^u_2 \vee \neg b^u_1 \end{gathered}$$

There are 8 possible assignments to the b^u variables, and 5 values, so for the values 3, 4 and 5 there are two b^u assignments mapped onto each, hence all 8 assignments are valid. In contrast to adapted log, no local acceptability variable (called z_u in [1]) is present, because no assignments to previous universal variables can be invalid. However, the problem is that the QBF solver can search two equivalent subtrees in some cases, for example when $b^u_2 = T$ and $b^u_1 = T$, the solver can branch on b^u_0 which is not contained in any clause.

By contrast, in the adapted log encoding, three assignments to b^u are invalid, and cause the indicator z_u and all subsequent indicators $z_{v>u}$ to be set through a group of collector clauses.

After setting b^u_2 and b^u_1, if either is set to T then the first two clauses above are satisfied and in the reduced formula b^u_0 does not exist. Both b^u_0 and $\neg b^u_0$ are pure, so if the solver implements the pure literal rule then it will not branch on this variable. This solves the repeated subtree problem mentioned above, on the condition that b_0^u is set last. Also, in common with the adapted log encoding, the channelling works only from b^u to x^u variables, so only positive x^u literals are included in the clause set above, therefore the pure literal rule can detect cases where the x_i^u is involved in no conflicts. In some circumstances, this can also lead to the elimination of b^u variables. For example, if x_4^u and x_5^u become pure, then b_2^u becomes pure as well and the search is reduced.

I briefly argue that this encoding is correct. (1) The conflict clauses directly encode the semantics of the binary constraints. (2) Since the conflict clauses are negative, it is required that at least one of the x^v variables is set true. This is accomplished with the at least one clause for existentials, and channelling for universals: whatever the assignment to the b^v variables, at least the corresponding x_i^v variable is set true. (3) The quantification order is preserved. The combination of (1), (2) and (3) preserves the satisfiability of the original QCSP but not its set of solutions. In a solution tree for the QBF, there are duplicate branches for the superfluous universal assignments.

The space complexity for the enhanced log and adapted log encodings is $O(ed^2)$ to express the constraints and $O(d \log d)$ for the channelling (and indicator variables), so $O(ed^2)$ in total (where d: domain size, e: number of constraints).

5 Empirical evaluation

To evaluate the enhanced log encoding, it is compared to the adapted log encoding and also to the state-of-the-art direct QCSP solver QCSP-Solve [7]. This is done with a suite of randomly generated binary QCSPs. The generation method is similar to CSP model b, in that the parameters represent proportions (for uniformity) rather than probabilities. Constraints between two universal variables are not generated, because if such a constraint contained any conflicts the problem would be trivially false. Also no constraints are generated from existentials to universals (i.e. the existential is quantified before the universal) because these can be resolved to a unary constraint and removed by preprocessing.

To avoid the flaw described in [1], when generating a constraint linking a universal a and an existential variable b, for each value of a a unique value of b is chosen[2]. From these d mappings, $q_{\forall\exists} \times d$ satisfying tuples are chosen. As a result, if the domain size is greater than the number of universal variables, the flaw cannot occur. For constraints between two existentials, $q_{\exists\exists} \times d^2$ satisfying tuples are chosen from all possible tuples. For experiments of this size ($n = 24$), flaws in constraints between existentials are not an issue. For each parameter set, 100 QCSP instances are generated.

To compare the two encodings with QCSP-Solve, we used a local implementation of CSBJ (originally described by Giunchiglia et al [9]) with unit propagation and the pure literal rule. Figure 1 shows the results, extending the second experiment of Gent et al [7]. The instances have 24 variables, with 8 existentially

[2] All random choices are made with uniform distribution.

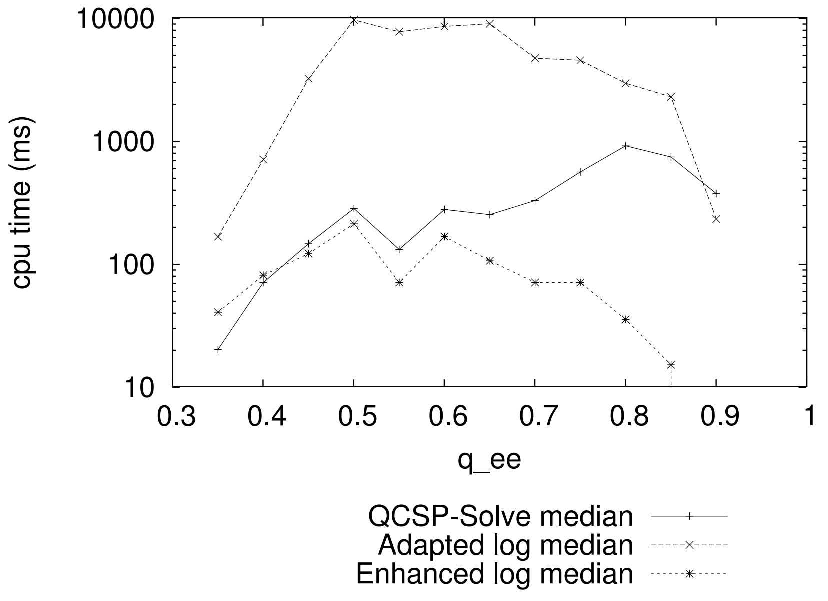


Fig. 1. CPU time comparison. $n = 24$, $n_\forall = 8$, $d = 9$, $p = 0.2$, $q_{\forall\exists} = 0.5$

quantified followed by 8 universally quantified followed by 8 more existentially quantified. The domain size is 9, ensuring that the instances are unflawed. The proportion of generated constraints from those that are allowed to be generated is 0.2. $q_{\forall\exists}$ is 0.5, and $q_{\exists\exists}$ is varied across the satisfiability phase transition. For each point, 100 instances were generated. The median average is used because of high outliers. The time taken to encode the instances is not included, but since it is a linear encoding this is negligible for difficult instances.

The closest setting of $q_{\exists\exists}$ to the phase transition is 0.55, with 43 instances satisfiable from 100. This also approximately coincides with the difficulty peaks for the encodings, but not for QCSP-Solve. For lower values of $q_{\exists\exists}$ fewer of the instances are satisfiable and the enhanced log encoding is less competitive with QCSP-Solve. For example at $q_{\exists\exists} = 0.35$, 99 of the instances are unsatisfiable, and QCSP-Solve outperforms the enhanced log encoding. Where $q_{\exists\exists} = 0.8$ all the instances are satisfiable and the enhanced log encoding outperforms QCSP-Solve. At $q_{\exists\exists} = 0.9$, the median for the enhanced log encoding fell below the resolution of the timer, so it is not shown on the graph.

This suggests that CSBJ is more effective than QCSP-Solve in pruning or backjumping over universal variables, because in a loosely-constrained instance the main cost is branching on universals. QCSP-Solve contains a method called solution directed pruning to achieve this, but it may be less effective than the solution backjumping incorporated into CSBJ. This could be tested but remains for future work.

6 Conclusions

I have introduced the enhanced log encoding, based on the adapted log and enhanced binary encodings. It exploits the pure literal rule present in some QBF solvers to replace the indicator variables of the adapted log encoding. The enhanced log encoding is effective, in some cases outperforming the state-of-the-art direct solution algorithm.

The advantage of this encoding over QCSP-Solve could be due to a rule in the QBF solver, in which case it would be interesting to develop the analogue rule for QCSP. Alternatively it could be simply greater efficiency of the QBF solver, however this seems unlikely because the CPU times peak in different places in figure 1. Investigating this further is left for future work. Other interesting directions would be to investigate the support encoding with enhanced log, and to develop encodings of common non-binary constraints.

It is not expected that, in the long term, QBF will outperform QCSP on large structured problems. The contribution here is to enable QCSP researchers to better compare their methods against QBF, and to identify methods used in QBF which would benefit QCSP. One iteration of this has already happened with the pure value rule being incorporated into QCSP-Solve.

7 Acknowledgements

The first author is funded by an EPSRC doctoral training grant. We would also like to thank Andrew Rowley for providing the implementation of CSBJ and for contributing to an earlier part of this work [1], and Kostas Stergiou for helpful comments and use of QCSP-Solve.

References

1. Ian P. Gent, Peter Nightingale and Andrew Rowley, Encoding Quantified CSPs as Quantified Boolean Formulae, in Proc. ECAI 2004, pages 176-180, 2004.
2. Alan M. Frisch and Timothy J. Peugniez, Solving Non-Boolean Satisfiability Problems with Stochastic Local Search, in Proc. IJCAI 2001, pages 282-288, 2001.
3. Ian P. Gent, Arc Consistency in SAT, in Proc. ECAI 2002, pages 121-125, 2002.
4. Enrico Giunchiglia, Massimo Narizzano and Armando Tacchella, Backjumping for quantified Boolean logic satisfiability, in Proc. IJCAI 2001, pages 275-281, 2001.
5. Marco Cadoli, Marco Schaerf, Andrea Giovanardi and Massimo Giovanardi, An Algorithm to Evaluate Quantified Boolean Formulae and its Experimental Evaluation, in Journal of Automated Reasoning, 28(2) pages 101-142, 2002.
6. Lucas Bordeaux and Eric Monfroy, Beyond NP: Arc-Consistency for Quantified Constraints, in Proc. CP-2002, pages 371-386, 2002.
7. Ian P. Gent, Peter Nightingale and Kostas Stergiou, QCSP-Solve: A Solver for Quantified Constraint Satisfaction Problems, to appear in Proc. IJCAI 2005.
8. Andrew Rowley, Watching Clauses in Quantified Boolean Formulae, Apes Research Group technical report 62, APES-62-2003, 2003.
9. Enrico Giunchiglia, Massimo Narizzano and Armando Tacchella, Backjumping for quantified Boolean logic satisfiability, Artificial Intelligence 145(1-2) pages 99-120, 2003.

Author Index