Lecture Notes in Computer Science 1118

Eugene C. Freuder (Ed.)

Principles and Practice of Constraint Programming – CP96

Second International Conference, CP96
Cambridge, MA, USA, August 1996
Proceedings

Springer

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	1 Jun 97	final 15 Apr 96 - 30 Sep 96

4. TITLE AND SUBTITLE

Support of Conference on Constraint Programming

5. FUNDING NUMBERS

N00014-96-1-0752

6. AUTHOR(S)

Pascal Van Hentenryck

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Brown University
Providence, RI 02912

8. PERFORMING ORGANIZATION REPORT NUMBER

Report 1
(Proceedings)

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

ONR
800 North Quincy
Arlington, VA 22217-5660

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The proceedings of the conference "Principles and Practice of Constraint Programming -- CP 96" are attached

14. SUBJECT TERMS

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.

Standard Form 298 Back (Rev. 2-89)

Lecture Notes in Computer Science 1118
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Advisory Board: W. Brauer D. Gries J. Stoer

Springer
Berlin
Heidelberg
New York
Barcelona
Budapest
Hong Kong
London
Milan
Paris
Santa Clara
Singapore
Tokyo

Eugene C. Freuder (Ed.)

Principles and Practice of Constraint Programming — CP96

Second International Conference, CP96
Cambridge, MA, USA, August 19-22, 1996
Proceedings

Springer

Volume Editor

Eugene C. Freuder
University of New Hampshire, Department of Computer Science
Kingsbury Hall M208, College Road, Durham, NH 03824, USA

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Principles and practice of constraint Programming : second international conference ; proceedings / CP '96, Cambridge, MA, USA, August 19 - 22, 1996 / Eugene C. Freuder (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Budapest ; Hong Kong ; London ; Milan ; Paris ; Santa Clara ; Singapore ; Tokyo : Springer, 1996
(Lecture notes in computer science ; Vol. 1118)
ISBN 3-540-61551-2
NE: Freuder, Eugene C. [Hrsg.]; CP <2, 1996 Cambridge, Mass.>; GT

CR Subject Classification (1991): D.1, D.3.2-3, I.2.3-4,F.3.2, F.4.1, I.2.8, H.3.3

ISSN 0302-9743
ISBN 3-540-61551-2 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN 10513429 06/3142 – 5 4 3 2 1 0 Printed on acid-free paper

Preface

Many problems are naturally expressed in terms of constraints, for example, scheduling, design, configuration, and diagnosis problems. The constraint programming community is providing the infrastructure for understanding and computing with constraints. The Second International Conference on Principles and Practice of Constraint Programming brings together members of this community to share recent progress.

The conference is dedicated to Paris Kanellakis, who died in a tragic airplane crash in December of 1995. He was one of the founders of this conference and a pillar of this community. A Kanellakis Prize was awarded to the paper that best exemplifies the interdisciplinary spirit of the conference.

Thirty-six papers will be presented at the conference, selected from over one hundred submissions. Twenty-two selections will be presented at a conference poster session. Full text of the papers and abstracts of the posters are included in the proceedings.

The conference has received support from Applied Logic Systems, Inc. (ALS) and the Office of Naval Research (ONR). The Kanellakis Prize has received sponsorship from MIT Press, Peter Revesz, and Springer-Verlag. The Conference is being held in cooperation with the American Association for Artificial Intelligence (AAAI), the Canadian Society for the Computational Studies of Intelligence (CSCSI), and Tufts University. The Program Chair received invaluable assistance from Daniel and Mihaela Sabin.

Updated information about the conference will be posted at the conference web site: http://www.cs.ualberta.ca/~ai/cp96/. The conference will be held in Cambridge, Massachusetts, USA, August 19–22, 1996.

July 1996

Eugene C. Freuder, Program Chair
Durham, New Hampshire, USA

In Memoriam: Paris C. Kanellakis

On December 20, 1995, Paris C. Kanellakis died unexpectedly and tragically, together with his wife, Maria-Teresa Otoya, and their children, Alexandra and Stephanos. They were heading to Cali, Columbia, for an annual holiday reunion when their airplane crashed in the Andes.

Paris was born in Greece in 1953. He graduated in electrical engineering from the National Technical University of Athens in 1976; his undergraduate thesis was entitled *Easy-to-test Criteria for Weak Stochastic Stability of Dynamical Systems* and was supervised by Prof. E.N. Protonotarios. In 1978, Paris received his M.Sc. thesis in electrical engineering and computer science from the Massachusetts Institute of Technology. His M.Sc. thesis, *Algorithms for a Scheduling Application of the Asymmetric Travelling Salesman Problem*, was supervised by Profs. R. Rivest and M. Athans. In 1982, he was awarded his Ph.D. degree from the same institution; his thesis was supervised by Prof. C.H. Papadimitriou and was entitled *On the Complexity of Concurrency Control for Distributed Databases.*

Paris joined the department of computer science at Brown University as assistant professor in 1981. He was promoted to associate professor with tenure in 1986, and to full professor in 1990. He was awarded an IBM Faculty Development Award in 1985 and an Alfred Sloan Foundation Fellowship in 1987. He served as an associate editor for the *Journal of Logic Programming* and for the new journal *Constraints*, as well as for *Information and Computation*, *ACM Transactions on Database Systems*, *SIAM Journal of Computing*, and *Theoretical Computer Science*. He served as invited speaker, program chair, and program committee

member at many prominent conferences. In the constraint programming area, he was program chair (together with J.-L. Lassez and V. Saraswat) of the First International Workshop on Principles and Practice of Constraint Programming, held in Newport in April 1993. This workshop, and the subsequent one held in Orcas Island by Alan Borning, were instrumental in starting the series of constraint programming conferences and Paris played a critical role in organizing the community and the conferences. In the related area of logic programming, he was an invited speaker at the 6th International Conference on Logic Programming in Lisbon, where his talk was entitled *A Logical Query Language with Object Identity and Strong Typing.* He was on the program committees of logic programming conferences in 1989, 1990, 1992, 1993, and 1996.

As a scientist, Paris was a careful thinker, investigating fundamental issues in computer science, opening new technical areas, and challenging conventional belief whenever appropriate. He made numerous contributions to computer science in areas as diverse as databases (relational, object-oriented, and constraint databases, concurrency control), programming languages (lambda calculus, logic programming, rewriting systems, type inference), distributed computing (concurrency and fault-tolerance), complexity theory, and combinatorial optimization. Underlying those contributions was a unifying theme: the use of logic, complexity theory, and algorithmics to understand the foundations of practical systems, to analyse their efficiency, and to improve their functionality. This theme was nicely exemplified in his work on object-oriented databases featured at the logic programming conference in Lisbon. Here his desire to understand the object-oriented database O_2 led him to invent, in collaborative work, an object-based data model, a new formalization of object identity, new programming tools, and new indexing algorithms.

A beautiful account of Paris' recent research accomplishments by S. Abiteboul, G. Kuper, H. Mairson, A. Shvartsman, and M. Vardi appeared in the March issue of *ACM Computing Surveys.* It was a major source of inspiration for this short article, in which only some of Paris' contributions to constraint programming (taken broadly) can be outlined.

The first issue of the *Journal of Logic Programming* featured an article by C. Dwork, P. Kanellakis, and J. Mitchell entitled *On the Sequential Nature of Unification.* The paper shows that the decision problem *"Do two terms unify?"* is complete for PTIME which, informally speaking, means that unification cannot be sped up with a polynomially bounded number of processors. This paper was published during a period of intense activity on the parallelization of Prolog. Together with J. Mitchell, Paris subsequently used the essential idea behind the proof to show that type inference in ML was PSPACE-hard, i.e., as hard as any problem that can be solved in polynomial space. This result contradicted the popular belief at the time that ML typing was efficient. His subsequent joint paper, in collaboration with H. Mairson and J. Mitchell, showed the problem to be complete for EXPTIME. Paris' most recent work on the lambda calculus (in collaboration with G. Hillebrand and H. Mairson) led to a new syntactic characterization of the complexity classes, which emerged from their research on

a functional programming foundation for a logic-based database query language.

Together with G. Kuper and P. Revesz, Paris was the founder of the area of constraint databases, whose essential idea is to replace, in the relational model, the concept of tuples by a conjunction of constraints. They investigated the query complexity of this scheme (which parallels in the database world the area of constraint logic programming) for various classes of constraints. Together with his colleagues and his students, he was also engaged in a long-term research to build the implementation technology (in particular, the indexing structures) necessary to make this technology practical. In particular, together with D. Goldin, he investigated constraint query algebras, a class of monotone constraints that allows an efficient projection algorithm, and similarity queries with scaling and shifting. Part of this work was featured in CP'95 and in a journal article to appear in *Constraints*.

Those of us who collaborated closely with Paris have lost not only an outstanding scientist but also an esteemed colleague and a dear friend. As a colleague, Paris had the poise, the personality, and the energy to rally communities behind him and he used these skills to improve our academic and professional environment. We also mourn a friend with a charming and engaging personality and a mediterranean passion —and a family whose warmth and hospitality will be sorely missed.

In writing these few pages, I came to understand one more time how fortunate I was to collaborate with Paris, to observe him in his daily scientific and family life, and to benefit from his insights, vision, and broad expertise; and of course to realize how my life has changed since I met him. He was a very special person.

Pascal Van Hentenryck
Brown University

Program Chair

Eugene C. Freuder
University of New Hampshire

Local Arrangements Chair

Isabel F. Cruz
Tufts University

Publicity Chair

Peter van Beek
University of Alberta

Invited Lecture

George L. Nemhauser
Georgia Institute of Technology

Lectures in Honor of Paris Kanellakis

Dina Q. Goldin
Brown University

Harry G. Mairson
Brandeis University

Kanellakis Prize Paper

A Test for Tractability
Peter Jeavons, David Cohen, and Marc Gyssens

Organizing Committee

Alan Borning (University of Washington)
Jacques Çohen (Brandeis University)
Alain Colmerauer (University of Marseille)
Eugene Freuder (University of New Hampshire)
Herve Gallaire (Xerox Corporation, Grenoble)
Jean-Pierre Jouannaud (University of Paris Sud)
Paris Kanellakis (Brown University)
Jean-Louis Lassez, chair (New Mexico Tech)
Ugo Montanari (University of Pisa)
Anil Nerode (Cornell University)
Vijay Saraswat (Xerox Corporation, PARC)
Pascal Van Hentenryck (Brown University)
Ralph Wachter (Office of Naval Research)

Program Committee

Endre Boros (Rutgers University)
Philippe Codognet (INRIA-Rocquencourt)
Rina Dechter (University of California, Irvine)
Boi Faltings (Swiss Federal Institute of Technology)
Bjorn Freeman-Benson (Object Technology International)
Eugene Freuder (University of New Hampshire)
Thom Frühwirth (Ludwig-Maximilians-Universität München)
Martin Golumbic (Bar-Ilan University)
Hoon Hong (RISC)
John Hooker (Carnegie Mellon University)
Joxan Jaffar (National University of Singapore)
Deepak Kapur (State University of New York, Albany)
Alan Mackworth (University of British Columbia)
Kim Marriott (Monash University)
William Older (Bell-Northern Research)
Leszek Pacholski (University of Wroclaw)
Catuscia Palamidessi (University of Genova)
Jean-François Puget (ILOG)
Raghu Ramakrishnan (University of Wisconsin, Madison)
Peter Revesz (University of Nebraska, Lincoln)
Thomas Schiex (INRA)
Bart Selman (AT&T Bell Laboratories)
Helmut Simonis (COSYTEC)
Gert Smolka (DFKI)
Wayne Snyder (Boston University)
Edward Tsang (University of Essex)
H. Paul Williams (University of Southampton)
Makoto Yokoo (NTT Communication Science Laboratories)

Referees

Abdennadher, S.
Aiba, A.
Bacchus, F.
Beldiceanu, N.
Beringer, H.
Bing, L.
Boros, E.
Borrett, J.
Brisset, P.
Brodsky, A.
Chomicki, J.
Choueiry, B.
Codognet, C.
Codognet, P.
Crawford, J.
Davenport, A.
David, P.
Dechter, R.
Diaz, D.
Domenjoud, E.
Engelson, S.
Fabris, M.
Falaschi, M.
Faltings, B.
Feldman, R.
Freeman-Benson, B.
Frühwirth, T.
Gabbrielli, M.
Gelle, E.
Gervet, C.
Golumbic, M.
Heintze, N.
Hong, H.
Hooker, J.
Jaffar, J.
Kapur, D.
Kogan, A.
Koubarakis, M.
Leung, H.
Lopez, G.
Lynch, C.
Mackworth, A.
Marriott, K.
McAllester, D.
McAloon, K.
Miyashita, K.
Mumick, I.
Neubacher, A.
Niehren, J.
Nieuwenhuis, R.
Nuijten, W.
Older, W.
Pacholski, L.
Pai, D.
Palamidessi, C.
Puget, J.-F.
Ramakrishnan, R.
Revesz, P.
Saxena, T.
Schiex, T.
Selman, B.
Sidebottom, G.
Simonis, H.
Smolka, G.
Snyder, W.
Sondergaard, H.
Srivastava, D.
Stolzenburg, F.
Stuckey, P.
Subramaniam, M.
Swaminathan, R.
Tison, S.
Trick, M.
Tsang, E.
Van Emden, M.
Verfaillie, G.
Würtz, J.
Wadge, B.
Walsh, T.
Wang, C.
Weigel, R.
Yap, R.
Yokoo, M.
Zhang, Y.

Posters

On Confluence of Constraint Handling Rules

Slim Abdennadher, Thom Frühwirth, Holger Meuss

Computer Science Department, University of Munich
Oettingenstr. 67, 80538 Munich, Germany
{Slim.Abdennadher,Thom.Fruehwirth,Holger.Meuss}@informatik.uni-muenchen.de

Abstract. We introduce the notion of confluence for Constraint Handling Rules (CHR), a powerful language for writing constraint solvers. With CHR one simplifies and solves constraints by applying rules. Confluence guarantees that a CHR program will always compute the same result for a given set of constraints independent of which rules are applied. We give a decidable, sufficient and necessary syntactic condition for confluence.
Confluence turns out to be an essential syntactical property of CHR programs for two reasons. First, confluence implies correctness (as will be shown in this paper). In a correct CHR program, application of CHR rules preserves logical equivalence of the simplified constraints. Secondly, even when the program is already correct, confluence is highly desirable. Otherwise, given some constraints, one computation may detect their inconsistency while another one may just simplify them into a still complex constraint.
As a side-effect, the paper also gives soundness and completeness results for CHR programs. Due to their special nature, and in particular correctness, these theorems are stronger than what holds for the related families of (concurrent) constraint programming languages.
Keywords: constraint reasoning, semantics of programming languages, committed-choice languages, confluence and determinacy.

1 Introduction

Constraint Handling Rules (CHR) [Frü95] have been designed as a special-purpose language for writing constraint solvers. A constraint solver stores and simplifies incoming constraints. CHR is essentially a committed-choice language consisting of guarded rules that rewrite constraints into simpler ones until they are solved.

In contrast to the family of the general-purpose concurrent constraint languages (CC) [Sar93] and the ALPS[1] [Mah87] framework, CHR allow "multiple heads", i.e. conjunctions of atoms in the head of a rule. Multiple heads are a feature that is essential in solving conjunctions of constraints. With single-headed CHR rules alone, unsatisfiability of constraints could not always be detected (e.g `X<Y, Y<X`) and global constraint satisfaction could not be achieved.

Nondeterminacy in CHR arises when two or more rules can fire. It is obviously desirable that the result of a computation in a solver will always be the same, semantically and syntactically, no matter in which CHR rules are applied. This property of constraint solvers will be called confluence and investigated in this paper.

[1] Saraswat showed in [Sar93], that ALPS can be recognized as a subset of $cc(\downarrow, \rightarrow)$

We will introduce a decidable, sufficient and necessary syntactic condition for confluence. This condition adopts the notion of critical pairs as known from term rewrite systems [DOS88, KK91, Pla93]. Monotonicity of constraint store updates, an inherent property of constraint logic programming languages, plays a central role in proving that joinability of critical pairs is sufficient for local confluence.

Confluence turns out to be important with regard to both theoretical and practical aspects: We show that confluence implies correctness of a program. By correctness we mean that the declarative semantic of a CHR program is a consistent theory. Unlike CC programs, CHR programs can be given a declarative semantics since they are only concerned with defining constraints (i.e. first order predicates), not procedures in their generality. Furthermore we show how to strengthen the declarative reading of a CHR program if it is confluent. A practical application of our definition of confluence lies in program analysis, where we can identify non–confluent parts of CHR programs by examining the critical pairs. Programs with non–confluent parts essentially represent an ill-defined constraint solving algorithm.

Our work extends previous approaches to the notion of determinacy in the field of CC languages: Maher investigates in [Mah87] a class of flat committed choice logic languages (ALPS). He defines the class of deterministic ALPS programs as those programs whose guards are mutually exclusive. The class of deterministic ALPS programs is less expressive than confluent CHR programs. Saraswat defines for the CC framework a similar notion of determinacy [Sar93], which is also more restrictive than confluence. We also give two reasons, why CHR cannot be made deterministic in general.

Our approach is orthogonal to the work in program analysis in [MO95] and [FGMP95], where a different, less rigid notion of confluence is defined: A CC program is confluent, if different process schedulings (i.e. different orderings of decisions at nondeterministic choice points) give rise to the same set of possible outcomes. The idea of [MO95] is to introduce a non-standard semantics, which is confluent for all CC programs.

The paper is organized as follows. The next section introduces the syntax of constraint handling rules, their declarative and operational semantics. Then this section contributes to the relationship between the declarative and operational semantics of CHR programs by giving soundness and completeness results. Section 3 presents the notion of confluence for CHR. In section 4 we show that confluence implies logical correctness of a program. This leads to a stronger completeness and soundness result for finite failed computation. Finally, we conclude with a summary and directions for future work.

2 Syntax and Semantics of CHR

We assume some familiarity with (concurrent) constraint programming (CCP) [JL87, JM94, SRP91, Sar93, Sha89]. There is a distinguished class of predicates, the *constraints*. We assume, that there is a built-in constraint solver that solves, checks and simplifies built-in (predefined) constraints. On the other hand, the user-defined constraints are those defined by a CHR program. This implies, that we have two disjoint sets of constraint symbols for the built–in and the user–defined constraints.

As a special purpose language, CHR usually extend a host language such as Prolog or Lisp with (more) constraint solving capabilities. This also means, that auxiliary computations in CHR programs can be performed in the host language. Without loss of generality, to keep this paper self-contained, we will not address host language issues here. We also restrict ourselves to the main kind of CHR rule.

Definition 1. A CHR *program* is a finite set of simplification rules[2]. A *simplification rule* is of the form

$$H_1, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \mid B_1, \ldots, B_k.$$

where the multi–head $H_1, \ldots, H_i$ is a conjunction[3] of user-defined constraints and the guard $G_1, \ldots, G_j$ is a conjunction of built-in constraints and the body $B_1, \ldots, B_k$ is a conjunction of built-in and user-defined constraints called goals.

2.1 Declarative Semantics

Unlike CC programs, CHR programs can be a given a declarative semantics since they are only concerned with defining constraints (i.e. first order predicates), not procedures in their generality.

Declaratively, a simplification rule

$$H_1, \ldots, H_i \Leftrightarrow G_1, \ldots, G_j \mid B_1, \ldots, B_k.$$

is a logical equivalence provided the guard is true in the current context

$$\forall \bar{x}\ (\exists \bar{y}\ (G_1 \wedge \ldots \wedge G_j)) \rightarrow (H_1 \wedge \ldots \wedge H_n \leftrightarrow \exists \bar{z}\ (B_1 \wedge \ldots \wedge B_k)),$$

where $\bar{x}$[4] are the variables occuring in $H_1, \ldots, H_n$ and $\bar{y}, \bar{z}$ are the other variables occuring in $G_1, \ldots, G_j$ and $B_1, \ldots B_k$ respectively.

The declarative interpretation of a CHR program P is given by the set $\mathcal{P}$ of logical equivalences and a consistent built–in theory CT which determines the meaning of the built–in constraints appearing in the program. The constraint theory CT specifies among other things the ACI properties of the logical conjunction $\wedge$ in the built-in constraint store, the properties of the equality constraints $\doteq$ (Clarks axiomatization) and the properties of the basic constraints *true* and *false*.

Definition 2. A CHR program P is *correct* iff $\mathcal{P} \cup CT$ is consistent.

2.2 Operational Semantics of CHR

We define the operational semantics as a transition system.

[2] There are two other kinds of rules [BFL+94], which are not treated here.
[3] For conjunctions in rules we use "," instead of "$\wedge$".
[4] we use $\bar{x}$ as an abbreviation for a sequence of variables

States

Definition 3. A *state* is a triple

$$<C_U, C_B, \mathcal{V}>.$$

C_U is a conjunction of both user-defined and built-in constraints that remains to be solved. C_B is a conjunction of built-in constraints accumulated up to this point of execution. $\mathcal{V}$ is an ordered set of variables.

Definition 4. A variable X in a state $<C_U, C_B, \mathcal{V}>$ is called *global*, if it appears in $\mathcal{V}$. It is called *local* otherwise.

Definition 5. The pair (C_1, C_2) (C_1 and C_2 are conjunctions of constraints) is called *enclosed* by the ordered set $\mathcal{V}$ iff all variables shared by C_1 and C_2 are contained in $\mathcal{V}$.

We can attribute to each state $<C_U, C_B, \mathcal{V}>$ the formula

$$\exists Y_1, \ldots, Y_m \; C_U \wedge C_B$$

as a logical meaning, where $Y_1, \ldots, Y_m$ are the local variables in C_U and C_B. Note that the global variables remain unbound in the formula.

Update We define now the basic operation of the built–in constraint solver: The main task of *update* is transforming a state into a logically equivalent state with a normalized built–in constraint store. update performs the following tasks:

- normalize the built–in constraint store according to CT
- propagate equality constraints through the state
- remove redundant equality constraints where one side is a local variable.

Definition 6. update normalizes a state by performing the following operations in sequence:

1. update produces a unique representation of the built-in constraint store according to the theory CT.
2. Equality constraints of the form $X \doteq t$ receive a special treatment: occurrences of X in all constraints (except the equality itself) in the built-in constraint store and goal store are replaced by t.
3. All equality constraints of the form $X \doteq t$ or $Y \doteq X$ are removed, if X is local. These equality constraints will be called *local*. This reflects the validity of formulas $(\exists X \; X \doteq a)$, which follows from the axioms in CT (see example 2.1).

Example 2.1

$$\text{update}(<p(Y) \wedge q(Z), Y \doteq f(X) \wedge Z \doteq a, [Y]>) = <p(f(X)) \wedge q(a), Y \doteq f(X), [Y]>$$

Under an enclosement condition update is compatible with addition of constraints. This result is given by the following lemma, which is proven by contradiction.

Lemma 7. If C is a conjunction of built–in constraints and (C, C_B) is enclosed by $\mathcal{V}$ and update($<C_U, C_B, \mathcal{V}>$) = $<C'_U, C'_B, \mathcal{V}>$ then

$$\text{update}(<C_U, C_B \wedge C, \mathcal{V}>) = \text{update}(<C'_U, C'_B \wedge C, \mathcal{V}>).$$

The enclosement condition in the lemma above reflects the sensitivity of update with respect to local variables. It guarantees that equality constraints involving variables appearing in the added constraint C are not removed due to locality. If the condition is violated, the claim is false:

Example 2.2

$$\text{update}(<\text{true}, X \doteq 2, []>) = <\text{true}, \text{true}, []>,$$

adding the built-in constraint $X \doteq 1$ on both sides results for the left side in:

$$\text{update}(<\text{true}, X \doteq 2 \wedge X \doteq 1, []>) = <\text{true}, \text{false}, []>$$

but for the right side in:

$$\text{update}(<\text{true}, \text{true} \wedge X \doteq 1, []>) = <\text{true}, \text{true}, []>$$

Definition 8. *Entailment* ($\rightarrow_o$) tests whether a given conjunction of built–in constraints is implied by another conjunction of built–in constraints in the context of a state and is defined as follows:

$$<C_{U1}, C_{B1}, \mathcal{V}> \rightarrow_o <C_{U2}, C_{B2}, \mathcal{V}> \quad \text{iff}$$
$$<C'_{U1}, C'_{B1}, \mathcal{V}> \; = \; \text{update}(<C'_{U2}, C'_{B1} \wedge C'_{B2}, \mathcal{V}>).$$

where update($<C_{U1}, C_{B1}, \mathcal{V}>$) = $<C'_{U1}, C'_{B1}, \mathcal{V}>$ and update($<C_{U2}, C_{B2}, \mathcal{V}>$) = $<C'_{U2}, C'_{B2}, \mathcal{V}>$.

Computation Steps Given a CHR program P we define the transition relation $\mapsto_P$ by introducing two kinds of *computation steps*:

Solve $<C \wedge C_U, C_B, \mathcal{V}> \; \mapsto_P \; \text{update}(<C_U, C \wedge C_B, \mathcal{V}>)$
if C is a built-in constraint.

The built–in constraint solver updates the state after adding the built–in constraint C to the built–in store C_B.

Simplify $<H' \wedge C_U, C_B, \mathcal{V}> \; \mapsto_P \; \text{update}(<C_U \wedge B, H \doteq H' \wedge C_B, \mathcal{V}>)$
if $(H \Leftrightarrow G \mid B)$ is a variant with fresh variables of a rule in P and $<H', C_B, \mathcal{V}> \rightarrow_o <H', H' \doteq H \wedge G, \mathcal{V}>$.

To simplify user-defined atoms means to apply a simplification rule on these atoms. This can be done if the atoms match with the head atoms of the rule and the guard is entailed by the built-in constraint store. The atoms occuring in the body of the rule are added to the goal constraint store.

Notation. By $c(t_1, \ldots, t_n) \doteq c(s_1, \ldots, s_n)$ we mean $t_1 \doteq s_1 \wedge \ldots \wedge t_n \doteq s_n$, if c is a user-defined constraint. By $p_1 \wedge \ldots \wedge p_n \doteq q_1 \wedge \ldots \wedge q_n$ we mean $p_1 \doteq q_1 \wedge \ldots \wedge p_n \doteq q_n$.

Definition 9. $S \mapsto_P^* S'$ holds iff

$$S = S' \text{ or } S = \text{update}(S') \text{ or } S \mapsto_P S_1 \mapsto_P \ldots \mapsto_P S_n \mapsto_P S' \quad (n \geq 0).$$

We will write $\mapsto$ instead of $\mapsto_P$ and $\mapsto^*$ instead of $\mapsto_P^*$, if the program P is fixed.

Lemma 10. Update has no influence on application of rules, i.e.

$$S \mapsto S' \text{ implies } \text{update}(S) \mapsto S'.$$

The *initial state* consists of a goal G, an empty built-in constraint store and the list $\mathcal{V}$ of the variables occuring in G,

$<G, \text{true}, \mathcal{V}>$.

A computation state is a *final state* if

- its built-in constraint store is false, then it is called *failed*;
- no computation step can be applied and its built-in constraint store is not false. Then it is called *successful*.

Definition 11. A *computation* of a goal G is a sequence $S_0, S_1, \ldots$ of states with $S_i \mapsto S_{i+1}$ beginning with the the initial state $S_0 = <G, \text{true}, \mathcal{V}>$ and ending in a final state or diverging. A finite computation is *successful* if the final state is successful. It is *failed* otherwise.

Definition 12. A *computable constraint* C of G is the conjunction $\exists \bar{x}\ C_U \wedge C_B$, where C_U and C_B occur in a state $<C_U, C_B, \mathcal{V}>$, which appears in a computation of G. $\bar{x}$ are the local variables.
A *final constraint* C is the conjunction $\exists \bar{x}\ C_U \wedge C_B$, where C_U and C_B occur in a final state $<C_U, C_B, \mathcal{V}>$.

Equivalence and Monotonicity The following definition reflects the AC1 properties of the goal store and the fact that all states with an inconsistent built–in constraint store are identified.

Definition 13. We identify states according to the equivalence relation $\cong$:
$<C_U, C_B, \mathcal{V}> \cong <C'_U, C_B, \mathcal{V}>$ iff C_U can be transformed to C'_U using the AC1 properties of the conjunction $\wedge$, or C_B is *false*.

We have to ensure that the equivalence $\cong$ is well-defined, i.e. that it is compatible with the operations we perform on states. We have six different operations working on states, 1-3 are explicitly used for computation steps, whereas 4-6 occur only in the proof for the theorem on local confluence:

1. **Solve**
2. **Simplify**
3. update
4. add a constraint to the goal store or built-in constraint store
5. form a variant
6. replace the global variable store by another ordered set of variables

It is easy to see that all these operations are congruent with the relation $\cong$, i.e. the following holds for each instance o of an operation:

$$S_1 \cong S_2 \text{ implies } o(S_1) \cong o(S_2)$$

Therefore we can reason about states modulo $\cong$.

The next definition defines the notion of monotonicity, which guarantees that addition of new built-in constraints does not inhibit entailment (and hence the application of **Simplify**):

Definition 14. A built–in constraint solver is said to be monotonic iff the following holds:

$$<C_{U1}, C_B, \mathcal{V}> \rightarrow_o <C_{U2}, G, \mathcal{V}> \text{ implies } <C_{U1}, C_B \wedge C, \mathcal{V}> \rightarrow_o <C_{U2}, G, \mathcal{V}>.$$

Lemma 15. Every built-in constraint solver (where update fulfills the stated requirements) is monotonic.

2.3 Relation between the declarative and the operational semantics

We present results relating the operational and declarative semantics of CHR. These results are based on work of Jaffar and Lassez [JL87], Maher [Mah87] and van Hentenryck [vH91].

Lemma 16. Let P be a CHR program, G be a goal. If C is a computable constraint of G, then

$$\mathcal{P}, CT \models \forall\, (C \leftrightarrow G).^5$$

Proof. By induction over the number of computation steps.

Theorem 17 Soundness of successful computations. Let P be a CHR program and G be a goal. If G has a successful computation with final constraint C then

$$\mathcal{P}, CT \models \forall\, (C \leftrightarrow G).$$

Proof. Immediately from lemma 16.

The following theorem is stronger than the completeness result presented in [Mah87], in the way that we can reduce the disjunction in the strong completeness theorem to a single disjunct. This is possible, since the computation steps preserve logical equivalence (lemma 16).

Theorem 18 Completeness of successful computations. Let P be a CHR program and G be a goal. If $\mathcal{P}, CT \models \forall\, (C \leftrightarrow G)$ and C is satisfiable, then G has a successful computation with final constraint C' such that

$$\mathcal{P}, CT \models \forall\, (C \leftrightarrow C').$$

[5] $\forall F$ is the universal closure of a formula F.

The next theorem gives a soundness and completeness result for correct CHR programs.

Theorem 19 Soundness and Completeness of failed computations.
Let P be a correct CHR program and G be a Goal. The following are equivalent:

a) $\mathcal{P}, CT \models \neg\exists G$
b) G has a finitely failed computation.

3 Confluence of CHR programs

We extend the notion of determinacy as used by Maher in [Mah87] and Saraswat in [Sar93] to CHR by introducing the notion of confluence. The notion of deterministic programs is less expressive and too strict for the CHR formalism, because it is not always possible to transform a CHR program into a deterministic one. This has two reasons, of which the first also holds for the CC formalism:

The constraint system must be closed under negation so that a single-headed CHR program can be transformed into one with non-overlapping guards.

Example 1. We want to extend the built-in solver, which contains the built-in constraints $\leq$ and $\doteq$, with a user-defined constraint `maximum(X,Y,Z)` which holds if `Z` is the maximum of `X` and `Y`. The following could be part of a definition for the constraint `maximum`:

`maximum(X,Y,Z)` $\Leftrightarrow$ `X`$\leq$`Y` | `Z`$\doteq$`Y`.
`maximum(X1,Y1,Z1)` $\Leftrightarrow$ `Y1`$\leq$`X1` | `Z1`$\doteq$`X1`.

This program cannot be transformed into an equivalent one without overlapping guards.

The second reason is that CHR rules have multiple heads. We can get into a situation, where two rules can be applied to different but overlapping conjunctions of constraints. In general it is not possible to avoid commitment of one of the rules (and thus making the program deterministic[6]) by adding constraints to the guards.

Example 2. Consider the following part of a CHR program defining interactions between the boolean operations `not`, `imp` and `or`.

`not(X,Y), imp(X,Y)` $\Leftrightarrow$ `true` | `X`$\doteq$`0, Y`$\doteq$`1`.
`not(X1,Y1), or(X1,Z1,Y1)` $\Leftrightarrow$ `true` | `X1`$\doteq$`0, Y1`$\doteq$`1, Z1`$\doteq$`1`.

Note that both rules can be applied to the goal `not(A,B)`$\wedge$`imp(A,B)`$\wedge$`or(A,C,B)`. When we want that only the fist rule can be applied, we have to add a constraint to the guard of the first rule, that `or(A,C,B)` doesn't exist. Such a condition is meta–logical and syntactically not allowed.

[6] We extend the notion of deterministic programs to our formalism in the natural way that only one rule can commit by any given goal.

In the following we will adopt and extend the terminology and techniques of conditional term rewriting systems (CTRS) [DOS88]. A straightforward translation of results in the field of CTRS was not possible, because the CHR formalism gives rise to phenomena not appearing in CTRS. These include the existence of global knowledge (the built–in constraint store) and local variables.

Definition 20. A CHR program is called *terminating*, if there are no infinite computation sequences.

Definition 21. Two states S_1 and S_2 are called *joinable* if there exist states S_1', S_2' such that $S_1 \mapsto^* S_1'$ and $S_2 \mapsto^* S_2'$ and S_1' is a variant of S_2' ($S_1' \sim S_2'$).

Definition 22. A CHR program is called *confluent* if the following holds for all states S, S_1, S_2:

If $S \mapsto^* S_1, S \mapsto^* S_2$ then S_1 and S_2 are joinable.

Definition 23. A CHR program is called *locally confluent* if the following holds for all states S, S_1, S_2:

If $S \mapsto S_1, S \mapsto S_2$ then S_1 and S_2 are joinable.

For the following reasoning we require, that rules of a CHR program contain disjoint sets of variables. This requirement means no loss of generality, because every CHR program can be easily transformed into one with disjoint sets of variables.

In order to give a characterization for local confluence we have to introduce the notion of critical pairs:

Definition 24. If one or more atoms $H_{i_1}, \ldots, H_{i_k}$ of the head of a CHR rule $H_1, \ldots, H_n \Leftrightarrow G \mid B$ unify with one or more atoms atom $H'_{j_1}, \ldots, H'_{j_k}$ of the head of another or the same CHR rule $H'_1, \ldots, H'_m \Leftrightarrow G' \mid B'$ then the triple

$$(G \wedge G' \wedge H_{i_1} \doteq H'_{j_1} \wedge \ldots \wedge H_{i_k} \doteq H'_{j_k} \mid B \wedge H'_{j_{k+1}} \wedge \ldots \wedge H'_{j_m} =\downarrow= B' \wedge H_{i_{k+1}} \wedge \ldots \wedge H_{i_n} \mid \mathcal{V})$$

is called a *critical pair* of the two CHR rules. $\{i_1, \ldots, i_n\}$ and $\{j_1, \ldots, j_m\}$ are permutations of $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$ respectively, $\mathcal{V}$ is the set of variables appearing in $H_1, \ldots, H_n, H'_1, \ldots, H'_m$.

Example 3. Consider example 1. There are two trivial[7] and the following nontrivial critical pair:

```
(X≤Y ∧ Y1≤X1 ∧ X≐X1 ∧ Y≐Y1 ∧ Z≐Z1 |
 Z≐Y =↓= Z1≐X1 | [X,Y,Z,X1,Y1,Z1])
```

The rules of example 2 have the nontrivial critical pair (We omit the global variable store for reasons of clarity):

```
(X≐X1 ∧ Y≐Y1 |
 imp(X,Y) ∧ X1≐0 ∧ Y1≐1 ∧ Z1≐1 =↓= or(X1,Z1,Y1) ∧ X≐0 ∧ Y≐1 | [..])
```

[7] We call critical pairs of the form $(G \mid B =\downarrow= B \mid \mathcal{V})$ trivial.

Trivial critical pairs in example 1 are stemming from unifying the heads of either the first or second rule with themselves. Note that not every critical pair stemming from one rule only is trivial. If the head of a rule contains a constraint symbol more than once, the resulting critical pair may be nontrivial.

Definition 25. A critical pair $(G \mid B_1 =\!\downarrow\!= B_2 \mid \mathcal{V})$ is called *joinable* if $<B_1, G, \mathcal{V}>$ and $<B_2, G, \mathcal{V}>$ are joinable.

Example 4. The first critical pair in example 3 is joinable, if the built–in constraint solver simplifies $\mathtt{X} \leq \mathtt{Y} \wedge \mathtt{Y} \leq \mathtt{X}$ to the constraint $\mathtt{X} \doteq \mathtt{Y}$.

The following lemmas are necessary to prove theorem 33. The proofs for these lemmas can be found in [AFM96]. The first lemma states that the global variables are not touched when testing the variance of two states. Crucial for this lemma is the fact that $\mathcal{V}$ is an ordered set.

Lemma 26. If

$$<C_{U1}, C_{B1}, \mathcal{V}> \sim <C_{U2}, C_{B2}, \mathcal{V}>$$

then the variables in $\mathcal{V}$ are not modified by variable renaming.

The following lemma shows that enclosement guarantees that addition of built–in constraints is compatible with update:

Lemma 27. If $(C, C_U \wedge C_B)$ is enclosed by $\mathcal{V}$ and

$$\begin{aligned} <C_U, C_B, \mathcal{V}> &\mapsto^* <C'_U, C'_B, \mathcal{V}> \quad \text{then} \\ <C_U, C_B \wedge C, \mathcal{V}> &\mapsto^* \text{update}(<C'_U, C'_B \wedge C, \mathcal{V}>). \end{aligned}$$

We apply lemma 27 to prove lemma 28, stating the enclosement conditions under which joinability of states is compatible with addition of built–in constraints.

Lemma 28. If

$$\begin{aligned} <C_{U1}, C_{B1}, \mathcal{V}> &\mapsto^* <C'_{U1}, C'_{B1}, \mathcal{V}>, \\ <C_{U2}, C_{B2}, \mathcal{V}> &\mapsto^* <C'_{U2}, C'_{B2}, \mathcal{V}>, \\ <C'_{U1}, C'_{B1}, \mathcal{V}> &\sim <C'_{U2}, C'_{B2}, \mathcal{V}>, \end{aligned}$$

and $(C, C_{U1} \wedge C_{B1})$ and $(C, C_{U2} \wedge C_{B2})$ are enclosed by $\mathcal{V}$, then

a)

$$\begin{aligned} <C_{U1}, C_{B1} \wedge C, \mathcal{V}> &\mapsto^* \text{update}(<C'_{U1}, C'_{B1} \wedge C, \mathcal{V}>), \\ <C_{U2}, C_{B2} \wedge C, \mathcal{V}> &\mapsto^* \text{update}(<C'_{U2}, C'_{B2} \wedge C, \mathcal{V}>), \\ \text{update}(<C'_{U1}, C'_{B1} \wedge C, \mathcal{V}>) &\sim \text{update}(<C'_{U2}, C'_{B2} \wedge C, \mathcal{V}>), \end{aligned}$$

b)

$$\begin{aligned} <C_{U1} \wedge C, C_{B1}, \mathcal{V}> &\mapsto^* \text{update}(<C'_{U1} \wedge C, C'_{B1}, \mathcal{V}>), \\ <C_{U2} \wedge C, C_{B2}, \mathcal{V}> &\mapsto^* \text{update}(<C'_{U2} \wedge C, C'_{B2}, \mathcal{V}>), \\ \text{update}(<C'_{U1} \wedge C, C'_{B1}, \mathcal{V}>) &\sim \text{update}(<C'_{U2} \wedge C, C'_{B2}, \mathcal{V}>). \end{aligned}$$

Definition 29. We call two states $<C_{U1}, C_{B1}, \mathcal{V}>$ and $<C_{U2}, C_{B2}, \mathcal{V}>$ *update equivalent* iff

$$\text{update}(<C_{U1}, C_{B1}, \mathcal{V}>) = \text{update}(<C_{U2}, C_{B2}, \mathcal{V}>)$$

Lemma 30. If $<C_U, C_B, \mathcal{V}>$ and $<C'_U, C'_B, \mathcal{V}>$ are update equivalent and $<C_U, C_B, \mathcal{V}> \mapsto^* S'$, then $<C'_U, C'_B, \mathcal{V}> \mapsto^* S'$.

Proof. The lemma follows directly from lemma 10.

The next lemma gives a condition when joinability is compatible with changing the global variable store:

Lemma 31. Let $<C_{U1}, C_{B1}, \mathcal{V}>$ and $<C_{U2}, C_{B2}, \mathcal{V}>$ be joinable. Then the following holds:

a) $<C_{U1}, C_{B1}, \mathcal{V}'>$ and $<C_{U2}, C_{B2}, \mathcal{V}'>$ are joinable,
   if $\mathcal{V}'$ consists only of variables contained in $\mathcal{V}$.
b) $<C_{U1}, C_{B1}, \mathcal{V} \circ \mathcal{V}'>$ and $<C_{U2}, C_{B2}, \mathcal{V} \circ \mathcal{V}'>$ are joinable,
   if $\mathcal{V}'$ contains only fresh variables ($\circ$ denotes concatenation).

The following theorem is an analogy to Newman's Lemma for term rewriting systems [Pla93] and is proven analogously:

Theorem 32 confluence of CHR programs. If a CHR program is locally confluent and terminating, it is confluent.

Theorem 33 gives a characterization for locally confluent CHR programs. The proof is given in [AFM96] and relies on lemmas 26 to 31.

Theorem 33 local confluence of CHR programs. A terminating CHR program is locally confluent if and only if all its critical pairs are joinable.

The theorem also means that we can decide whether a program (which we do not know is terminating or not) will be confluent in case it is terminating.

Example 5. This example illustrates the case that an unjoinable critical pair is detected. The following CHR program is an implementation of `merge/3`, i.e. merging two lists into one list as the elements of the input lists arrive. Thus the order of elements in the final list can differ from computation to computation.

```
merge([],L2,L3) ⇔ true | L2≐L3.
merge(M1,[],M3) ⇔ true | M1≐M3.
merge([X|N1],N2,N3) ⇔ true | N3≐[X|N], merge(N1,N2,N).
merge(O1,[Y|O2],O3) ⇔ true | O3≐[Y|O], merge(O1,O2,O).
```

There are 8 critical pairs, 4 of them stemming from different rules.
If `merge/3` meets the specification, there is space for nondeterminism that causes non-confluence. Indeed, a look at the critical pairs reveals one critical pair stemming from the third and fourth rule that is not joinable:

```
([X|N1]≐O1 ∧ N2≐[Y|O2] ∧ N3≐O3 |
 N3≐[X|N] ∧ merge(N1,N2,N) =↓= O3≐[Y|O] ∧ merge(O1,O2,O) | [..])
```

It can be seen from the unjoinable critical pair above that a state like `<merge([a],[b],L),true,[L]>` can either result in putting `a` before `b` in the output list `L` or vice versa, since a **Simplify**–step can result in differing unjoinable states, depending on which rule is applied. Hence - not surprisingly - `merge/3` is not confluent.

4 Correctness and Confluence of CHR Programs

Definition 34. Given a CHR program P, we define the computation equivalence $\leftrightarrow_P^*$: $S_1 \leftrightarrow_P S_2$ iff $S_1 \mapsto S_2$ or $S_1 \mapsfrom S_2$. $S \leftrightarrow_P^* S'$ iff there is a sequence $S_1, \ldots S_n$ such that S_1 is S, S_n is S' and $S_i \leftrightarrow_P S_{i+1}$ for all i. We will write $\leftrightarrow$ instead of $\leftrightarrow_P$ and $\leftrightarrow^*$ instead of $\leftrightarrow_P^*$, if the program P is fixed.

For the sake of simplicity and clarity we prove the following two lemmas only for the special case that all rules are ground–instantiated, without guards and that *true* and *false* are the only built–in constraints used. One can extend the proof to full CHR by transforming each rule of a CHR program into (possibly infinitely many) ground–instantiated rules. This includes evaluating the built–in constraints in the guards and bodies.

Lemma 35. If P is confluent, then $<\text{true}, \text{true}, \mathcal{V}> \leftrightarrow_P^* <\text{true}, \text{false}, \mathcal{V}>$ does not hold.

Proof. We show by induction on n that there are no states $S_1, T_1, S_2, \ldots, T_{n-1}, S_n$ such that

$$<\text{true}, \text{true}, \mathcal{V}> {}^*\!\!\mapsfrom S_1 \mapsto^* T_1 {}^*\!\!\mapsfrom S_2 \mapsto^* \ldots \mapsto^* T_{n-1} {}^*\!\!\mapsfrom S_n \mapsto^* <\text{true}, \text{false}, \mathcal{V}>$$

Base case: $<\text{true}, \text{true}, \mathcal{V}> {}^*\!\!\mapsfrom S_1 \mapsto^* <\text{true}, \text{false}, \mathcal{V}>$ cannot exist, because $<\text{true}, \text{true}, \mathcal{V}>$ and $<\text{true}, \text{false}, \mathcal{V}>$ are different (no variants) final states and P is confluent.

Induction step: We assume that the induction hypothesis holds for n, i.e. $<\text{true}, \text{true}, \mathcal{V}> {}^*\!\!\mapsfrom S_1 \mapsto^* T_1 {}^*\!\!\mapsfrom S_2 \mapsto^* \ldots \mapsto^* T_{n-1} {}^*\!\!\mapsfrom S_n \mapsto^* <\text{true}, \text{false}, \mathcal{V}>$ doesn't exist. We prove the assertion for $n+1$ by contradiction:
We assume that a sequence of the form $<\text{true}, \text{true}, \mathcal{V}> {}^*\!\!\mapsfrom S_1 \mapsto^* T_1 {}^*\!\!\mapsfrom S_2 \mapsto^* T_2 {}^*\!\!\mapsfrom \ldots {}^*\!\!\mapsfrom S_n \mapsto^* T_n {}^*\!\!\mapsfrom S_{n+1} \mapsto^* <\text{true}, \text{false}, \mathcal{V}>$ exists. We will lead this assumption to a contradiction.

Since P is confluent, $<\text{true}, \text{false}, \mathcal{V}>$ and T_n are joinable. Since $<\text{true}, \text{false}, \mathcal{V}>$ is a final state, there is a computation of T_n that results in $<\text{true}, \text{false}, \mathcal{V}>$ ($T_n \mapsto^* <\text{true}, \text{false}, \mathcal{V}>$), and hence $S_n \mapsto^* <\text{true}, \text{false}, \mathcal{V}>$. Therefore there is a sequence of the form

$<\text{true}, \text{true}, \mathcal{V}> {}^*\!\!\mapsfrom S_1 \mapsto^* T_1 {}^*\!\!\mapsfrom S_2 \mapsto^* T_2 {}^*\!\!\mapsfrom \ldots {}^*\!\!\mapsfrom S_{n-1} \mapsto^* T_{n-1} {}^*\!\!\mapsfrom S_n \mapsto^* <\text{true}, \text{false}, \mathcal{V}>$,

which is a contradiction to the induction hypothesis.

Lemma 36. If $<\text{true}, \text{true}, \mathcal{V}> \leftrightarrow^* <\text{true}, \text{false}, \mathcal{V}>$ does not hold, then $\mathcal{P} \cup CT$ is consistent.

Proof. We show consistency by defining an interpretation which is a model of $\mathcal{P}$, and therefore of $\mathcal{P} \cup CT$.

We define $I_0 := \{\{C_1, \ldots, C_n\} | <C_1 \wedge \ldots \wedge C_n, \text{true}, \mathcal{V}> \leftrightarrow^* <\text{true}, \text{true}, \mathcal{V}>\}$. Let be $I := (\bigcup I_0) \backslash \{\text{true}\}$ ($\bigcup M$ is the union of all members of M). false $\notin I$, because $<\text{false}, \text{true}, \mathcal{V}> \leftrightarrow^* <\text{true}, \text{true}, \mathcal{V}>$ does not hold. Therefore I is a Herbrand interpretation.

We show that $I \models \mathcal{P}$:
For all formulas $H_1 \wedge \ldots \wedge H_n \leftrightarrow B_1 \wedge \ldots \wedge B_m \in \mathcal{P}$ the following equivalences hold:

$$
\begin{aligned}
& I \models H_1 \wedge \ldots \wedge H_n \\
\text{iff } & \{H_1, \ldots, H_n\} \subseteq I \\
\text{iff } & <H_1 \wedge \ldots \wedge H_n, \text{true}, \mathcal{V}> \leftrightarrow^* <\text{true}, \text{true}, \mathcal{V}> \\
\text{iff } & <B_1 \wedge \ldots \wedge B_m, \text{true}, \mathcal{V}> \leftrightarrow^* <\text{true}, \text{true}, \mathcal{V}> \\
\text{iff } & \{B_1, \ldots, B_m\} \subseteq I \\
\text{iff } & I \models B_1 \wedge \ldots \wedge B_m.
\end{aligned}
$$

Therefore $I \models H_1 \wedge \ldots \wedge H_n \leftrightarrow B_1 \wedge \ldots \wedge B_m$ for all formulas $H_1 \wedge \ldots \wedge H_n \leftrightarrow B_1 \wedge \ldots \wedge B_m$ in $\mathcal{P}$.

Theorem 37. If P is confluent, then $\mathcal{P} \cup CT$ is consistent.

Proof. The theorem follows directly from the lemmas 35 and 36.

Maher proves the following result for deterministic programs: if any computation sequence terminates in failure, then every (fair) computation sequence terminates in failure. We extend this result on confluent programs and give, compared to theorem 19, a closer relation between the operational and declarative semantics.

Definition 38. A computation is *fair* iff the following holds:
If a rule can be applied infinitely often to a goal, then it is applied at least once.

Lemma 39. Let P be a confluent CHR program and G be a goal which has a finitely failed derivation. Then every fair derivation of G is finitely failed.

The following theorem is a consequence of the above lemma and theorem 19.

Theorem 40. Let P be a confluent program and G be a Goal.
The following are equivalent:

a) $\mathcal{P}, CT \models \neg \exists G$
b) G has a finitely failed computation.
c) every fair computation of G is finitely failed.

5 Conclusion and Future Work

We introduced the notion of confluence for Constraint Handling Rules (CHR). Confluence guarantees that a CHR program will always compute the same result for a given set of user-defined constraints independent of which rules are applied.

We have given a characterization of confluent CHR programs through joinability of critical pairs, yielding a decidable, syntactically based test for confluence. We have shown that confluence is a sufficient condition for logical correctness of CHR programs. Correctness is an essential property of constraint solvers.

We also gave various soundness and completeness results for CHR programs. Some of these theorems are stronger than what holds for the related families of (concurrent) constraint programming languages due to correctness.

Our approach complements recent work [MO95] that gives confluent, non-standard semantics for CC languages to make them amenable to abstract interpretation and analysis in general, since our confluence test can find out parts of CC programs which are confluent already under the standard semantics.

Current work integrates the two other kinds of CHR rules, the propagation and the simpagation rules, into our condition for confluence. We are also developing a tool in ECL^{i}PS^e(ECRC Constraint Logic Programming System [Ecl94]) which tests confluence of CHR programs. Preliminary tests show that most existing constraint solvers written in CHR are indeed confluent, but that there are inherently non-confluent solvers (e.g. performing Gaussian elimination), too. We also plan to investigate completion methods to make a non-confluent CHR program confluent.

Acknowledgements

We would like to thank Heribert Schütz and Norbert Eisinger for useful comments on a preliminary version of this paper and Michael Marte for his implementation of a confluence tester.

References

[AFM96] S. Abdennadher, T. Frühwirth, and H. Meuss. Confluent simplification rules. Technical report, LMU Munich, January 1996.

[BFL+94] P. Brisset, T. Frühwirth, P. Lim, M. Meier, T. Le Provost, J. Schimpf, and M. Wallace. *ECL^{i}PS^e 3.4 Extensions User Manual.* ECRC Munich Germany, July 1994.

[DOS88] N. Dershowitz, N. Okada, and G. Sivakumar. Confluence of conditional rewrite systems. In *1st CTRS*, pages 31–44. LNCS 308, 1988.

[Ecl94] *ECL^{i}PS^e 3.4 User Manual*, July 1994.

[FGMP95] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence in concurrent constraint programming. In Alagar and Nivat, editors, *Proceedings of AMAST '95, LNCS 936.* Springer, 1995.

[Frü95] T. Frühwirth. Constraint handling rules. In A. Podelski, editor, *Constraint Programming: Basics and Trends.* LNCS 910, March 1995.

[JL87] J. Jaffar and J. L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages POPL-87, Munich, Germany*, pages 111–119, 1987.

[JM94] J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 20:503–581, 1994.

[KK91] Claude Kirchner and Hélène Kirchner. *Rewriting: Theory and Applications.* North-Holland, 1991.

[Mah87] M. J. Maher. Logic Semantics for a Class of Committed-Choice Programs. In *Fourth International Conference on Logic Programming*, pages 858–876, Melbourne, Australia, May 1987.

[MO95] K. Marriott and M. Odersky. A confluent calculus for concurrent constraint programming with guarded choice. In Ugo Montanari Francesca Rossi, editor, *Principles and Practice of Constraint Programming, Proceedings First International Conference, CP'95, Cassis, France*, pages 310–327, Berlin, September 1995. Springer.

[Pla93] David A. Plaisted. Equational reasoning and term rewriting systems. In D. Gabbay, C. Hogger, J. A. Robinson, and J. Siekmann, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 1, chapter 5, pages 273–364. Oxford University Press, Oxford, 1993.

[Sar93] V. A. Saraswat. *Concurrent Constraint Programming.* MIT Press, Cambridge, 1993.

[Sha89] E. Shapiro. The family of concurrent logic programming languages. In *ACM Computing Surveys*, volume 21:3, pages 413–510, September 1989.

[SRP91] V. A. Saraswat, M. Rinard, and P. Panangaden. The semantics foundations of concurrent constraint programming. In *Conference Record of the Eighteenth Annual ACM Symposium on principles of Programming Languages*, pages 333–352, Orlando, Florida, January 1991. ACM Press.

[vH91] P. van Hentenryck. Constraint logic programming. In *The Knowledge Engineering Review*, volume 6, pages 151–194, 1991.

A Labelling Arc Consistency Method for Functional Constraints

M.S. Affane and H. Bennaceur
Laboratoire d'Informatique de Paris-Nord, CNRS URA 1507
Université Paris-Nord, Institut Galilée
avenue J.B.Clément - 93430 Villetaneuse - France
phone: 1/49403573 fax:1/48260712
email: {msa,bennaceu}@ura1507.univ-paris13.fr

Abstract. Numerous arc consistency algorithms have been developed for filtering constraint satisfaction problems (CSP). But, few of them considered the semantic of the constraints. Arc consistency algorithms work with a queue containing element to reconsider. Then, some constraints may be checked many times. Recently, Liu has proposed an improved specific version AC5+ of the AC5 algorithm. AC5+ deals with a subclass of functional constraints, called "Increasing Functional Constraints (IFC)". It allows some IFC constraints of a CSP to be checked only once, when achieving arc consistency. In this paper, we propose a labelling arc consistency method (LAC) for filtering CSPs containing functional constraints. LAC uses two concepts:arc consistency and label-arc consistency. It allows all functional constraints to be checked only once, and some general constraints to be checked at most twice. Although, the complexity of LAC is still in O(ed) for functional constraints, where e is the number of constraints and d the size of the largest domain, the technique used in LAC leads to improve the performances and the effectiveness of classical arc consistency algorithms for CSPs containing functional constraints. The empirical results presented show the substantial gain brought by the LAC method.

I Introduction

A constraint satisfaction problem consists of assigning values to variables which are subject to a set of constraints.
Numerous arc consistency algorithms have been developed for filtering constraint satisfaction problems (CSP) [1, 2, 7] before or during the search for a solution. But, few of them considered the semantic of the constraints. Important classes of constraints (functional, anti-functional, monotonic,...) have been studied in the last years. These types of constraints arise in several concrete applications, such as job scheduling [6], and constraint logic programming languages [4, 10]. The basic constraints used in these languages are special cases of functional and monotonic constraints. David [3] has proposed a filtering algorithm "Pivot consistency" for the functional constraints. Van Hentenryck and al. [11] have developed a generic arc consistency algorithm AC5. This algorithm achieves arc consistency in O(ed) on functional, anti-functional and monotonic constraints, where e is the number of constraints and d is the size of the largest domain. Arc consistency algorithms work with a queue containing the elements to reconsider. Then, some constraints may be rechecked many times during the constraint propagation process. More recently, Liu [6] has proposed an improved version AC5+ of the AC5 algorithm. AC5+ deals with a subclass of functional constraints, called "Increasing Functional Constraints (IFC)". It allows some IFC constraints of a CSP to be checked only once, when achieving arc consistency.

In this paper, we propose a labelling arc consistency method (LAC) for filtering CSPs containing functional constraints. This method is based on the following two properties of functional constraints. The first is that, if for any two variables X_i and X_j of a CSP, there exists a sequence of functional constraints between X_i and X_j, then each value of a domain participates in at most one solution of this CSP. The second is that the composition of a set of functional constraints is a functional constraint. These properties are exploited for developing an algorithm allowing all functional constraints to be checked only once, and some general constraints to be checked at most twice. This is achieved by affecting a label to each value and each domain of a CSP, and defining a local consistency concept, called label-arc consistency, by modifying slightly the arc consistency concept. The labelling of domains and values allows the keeping trace of solutions to which participate some values. LAC applies the label-arc concept only on the constraints C_{ij} such that the domains D_i and D_j have the same label. This leads to improve the performances and the effectiveness of classical arc consistency algorithms for filtering CSPs containing functional constraints. Naturally, this technique can be embedded in any general arc consistency algorithm.
LAC deals not only with the class of functional constraints containing the class of IFC constraints considered by AC5+, but also with some general constraints. AC5+ does not guarantee that any IFC constraint of a CSP will be checked only once, while LAC checks all functional constraints only once, and in addition some general constraints are checked at most twice. Then, the proportion of constraints checked uselessly many times by AC5 and AC5+ is diminished using the LAC method. The performances and the effectiveness of LAC on CSPs containing functional constraints are more important than the classical arc consistency algorithms, since, in one hand, this method avoids to check many times certain constraints, and in other hand, LAC allows in some cases the solving of the CSP or the improving of the effectiveness of the filtering.
Another advantage of the LAC method is its incremental behaviour. If we want to add a constraint to a label-arc consistent CSP (P), the functional constraints and some general constraints of (P) will not be checked for making the new CSP label-arc consistent. The empirical results presented show the substantial gain brought by the LAC method.
The rest of this paper is organized as follows. Section 2 reviews some needed definitions on the constraint satisfaction framework. Section 3 describes the related works. Section 4 presents the LAC method. Section 5 presents empirical results and section 6 concludes.

2. Preliminaries

In this section, we present the formalism of CSPs introduced by Montanari [9].
A binary CSP P is defined by (X, D, C, R), where:
-X is a set of n variables $\{X_1,..., X_n\}$;
-D is a set of n domains $\{D_1,..., D_n\}$ where D_i is the set of all possible values for X_i;
-C is a set of m constraints where C_{ij} (i<j) is the constraint between the variables X_i and X_j is defined by its relation R_{ij};
-R is a set of m relations R_{ij}, where R_{ij} is a subset of the Cartesian product $D_i \times D_j$ specifying the compatible values between X_i and X_j.
The constraint graph represents variables and constraints of the CSP in the form of a network, where each variable is represented by a vertex and each constraint by an edge.

For all constraint C_{ij}, the predicate $R_{ij}(v_r,v_s)$ holds if and only if (v_r,v_s) belongs to the relation R_{ij}.
We recall now the definitions of arc consistency, functional constraint and increasing functional constraint.
Definition 1 A domain D_i of D is arc consistent iff, for each $v_r \in D_i$, $\forall X_j \in X$ such that $C_{ij} \in C$, there exists $v_s \in D_j$ such that $R_{ij}(v_r,v_s)$. A CSP is arc consistent iff $\forall D_i \in D$, D_i is arc consistent and D_i not empty.
Definition 2 A constraint C_{ij} is functional iff for all $v_r \in D_i$ (resp. $v_s \in D_j$) there exists at most one $v_s \in D_j$ (resp. $v_r \in D_i$) such that $R_{ij}(v_r, v_s)$. We note $v_s=f_{ij}(v_r)$ and $v_r=f_{ji}(v_s)$.
Definition 3 A constraint C_{ij} is an Increasing Functional Constraint (IFC) iff C_{ij} is functional and for all $v_r, v_s \in D_i$ such that $f_{ij}(v_r)$ and $f_{ij}(v_s)$ exist in D_j then $v_r < v_s$ implies $f_{ij}(v_r) < f_{ij}(v_s)$.
Observe that if C_{ij} is an IFC then the constraint is functional, the reciprocal is false.

3. Related Work

Our work is motivated by the AC5 and AC5+ algorithms developed by Van Hentenryck and al. [11], and Liu [6], respectively. AC5 is a generic consistency algorithm which can be specialized for functional constraints, anti-functional constraints, monotonic constraints, and their piecewise constraints. As classical arc consistency algorithms, AC5 manages a queue containing elements to be reconsidered, then some constraints can be checked uselessly many times.
AC5+ proposed an improved version of AC5. It allows some IFC constraints of a CSP to be checked only once, when achieving arc consistency. The drawbacks of AC5+ is that its use is limited to a restrictive class of functional constraints. It even does not guarantee that any IFC constraint of a CSP will be checked only once, since this depends on the order in which the constraints are checked. There are also many general constraints checked uselessly many times.
The proposed method LAC checks all functional constraints only once, and some general constraints at most twice. Then, the proportion of constraints checked uselessly many times is diminished. The effectiveness of LAC on CSPs containing functional constraints outperforms the classical arc consistency algorithms since this method uses a filtering concept more general than the arc consistency concept. Although our method is still in O(ed) for functional constraints, the computational experiments show that it is more efficient than AC5 and AC5+.

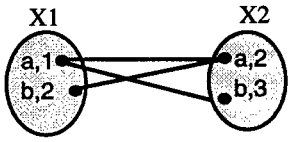
4. The LAC algorithm

We begin this section with some needed definitions. Then we present the core of the technique used in LAC, and the corresponding algorithm.

Definition 4 Assume that the domains of a CSP and the values of each domain are labelled. A value v_r of D_i is label-arc consistent iff for each domain D_j having the same label as D_i, there exists a value v_s such that we have $R_{ij}(v_r, v_s)$, and the values v_r and v_s have the same label.

Definition 5 A domain D_i is label-arc consistent iff each value of D_i is label-arc consistent. A CSP (P) is label-arc consistent iff each domain of (P) is label-arc consistent and not empty.

Example Assume that D_1 and D_2 have the same label, the values a and b of D_1 are labelled by 1 and 2 respectively, and the values a and b of D_2 are labelled by 2 and 3 respectively. Although the domains D_1 and D_2 are arc consistent, applying label-arc consistency leads to a removing of values a from D_1 and b from D_2, since these values have different labels.



4.1 Functional constraints

The LAC method is based on two important properties of functional constraints. The first property exploited by the LAC method leads to check all functional constraints only once, while the second property allows the checking of some general constraints at most twice.

Property 1 Let (P) be a CSP, and assume that for each pair of variables X_i and X_j of (P), there exists a sequence of functional constraints between X_i and X_j. Then for each value v_r of a domain D_i, there exists at most one solution such that X_i takes the value v_r.

Proof Assume that, for a value v_r of D_i there exists two different solutions S1 and S2 of (P) such that X_i takes the value v_r. Then, there exists a variable X_j of (P) such that X_j takes two different values in S1 and S2. Either there is a constraint between X_i and X_j which is not functional, or for each sequence of constraints between X_i and X_j there is at least one of them which is not functional. Otherwise, implicitly X_i and X_j are linked with a functional constraint, since the composition of a sequence functional constraints is a functional constraint. ♦
Before presenting the second property, we first describe the labelling process which will be used for proving this property.

The principle of LAC

As classical arc consistency algorithms, the LAC method performs the filtering in two phases. The first phase deals with the checking of constraints and the labelling process. While, the second phase deals with the constraint propagation process.
Assume that initially each domain D_i is labelled by i and each value v_r of D_i is labelled by r. The labelling process is performed during the first phase as follows. Progressively, the constraints of a CSP are checked. When a functional constraint C_{ij} is encountered, we perform one of the following treatments.
(i) - Either the domains D_i and D_j are labelled by i and j respectively, in which case arc consistency between these domains is applied. For each tuple (v_r, v_s) of R_{ij}, the value v_s will be labelled by r. The domain D_j will be labelled by i.

(ii) - Or the domains D_i and D_j are labelled by k ($k \neq i$ and $k \neq j$) and j respectively, arc consistency between these domains is applied. For each tuple (v_r, v_s) of R_{ij}, the value v_s will be labelled by the same label as the one of v_r. The domain D_j will be labelled by k. The case when D_i and D_j are labelled by i and k ($k \neq j$ and $k \neq i$) respectively is similar to (ii).
(iii) - Or, the labels of the domains D_i and D_j are different, let k_1 and k_2 be the labels of D_i and D_j respectively. Arc consistency between the domains D_i and D_j is applied. For each tuple (v_r, v_s) of R_{ij}, the value v_s will be labelled by the same label as the one of v_r. The domain D_{k2} will be labelled by k_1. Notice that all the domains which were labelled with k_2 will be implicitly[1] labelled with k_1, and the values of domains having the same label as v_s will be implicitly[2] labelled with the label of v_r.
(iv) - Or the domains D_i and D_j have the same label, in which case label-arc consistency between the domains D_i and D_j is applied.
In all cases the constraint C_{ij} will not be reconsidered.
When a general constraint C_{ij} is encountered : either the domains D_i and D_j have the same labels, label-arc consistency between these domains is applied, and this constraint will not be reconsidered, or arc consistency between these domains is applied.
Note that in (i), (ii) and (iii), the labelling process is propagated over the domains and the values.
During the second phase, the constraint propagation process leads to recheck only some general constraints. When a label-arc consistency is performed on a general constraint, this constraint will not be reconsidered.
Intuitively, the LAC algorithm consists in partitioning the set of domains of a CSP into a subsets of domains. Each subset D_k groups the domains having a same label k, and is represented by the domain D_k.

Definition 7 D_k is a representative domain if it is labelled by k. A representative value v_r of a representative domain is a value labelled by r.

We now present the second property showing that some general constraints will be considered as a functional constraint after applying label-arc consistency. Then these constraints will not be reconsidered during the constraint propagation process.

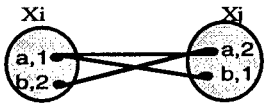
Property 2 Let $C_{i_1i_2}$, $C_{i_2i_3}$,..., $C_{i_{k-1}i_k}$ be a sequence of functional constraints, and $C_{i_pi_q}$ $1 \leq p \leq q \leq k$ a general constraint. After applying label-arc consistency between the domains D_{i_p} and D_{i_q}, for each value $v_r \in D_{i_p}$ (resp. $v_s \in D_{i_q}$) labelled by l_r (resp. l_s) there exists exactly one value v_s in D_{i_q} (resp. v_r in D_{i_p}) labelled by l_r (resp. l_s) such that $R_{ij}(v_r, v_s)$ holds.

Proof Since the constraints $C_{i_1i_2}$, $C_{i_2i_3}$,..., $C_{i_{k-1}i_k}$ are functional, the labelling process leads to label the domains D_{i_p} and D_{i_q} by a same label (the index of their representative domain). And, in D_{i_p} (resp. D_{i_q}) there is no two values having a same

[1]The domains will not be effectively labelled again, the new labelling of a domain can easily be deduced when this domain is manipulated (see the ComputeLabelDomain procedure).
[2]It means that the labels of these values can be deduced (see the ComputeLabelValue procedure).

label. Since the constraint C_{ij} is implicitly functional (this constraint is the composition of functional constraints), then after applying label-arc consistency between D_{i_p} and D_{i_q}, for each value $v_r \in D_{i_p}$ (resp. $v_s \in D_{i_q}$) labelled by l_r (resp. l_s) there exists exactly one value v_s in D_{i_q} (resp. v_r in D_{i_p}) labelled by l_r (resp. l_s) such that $R_{ij}(v_r, v_s)$ holds. ♦

We can associate to each subset of domains D_k a subproblem CSP P_k satisfying the hypothesis of Property 1 and for each general constraint C_{ij} of P_k, there is a sequence of functional constraints between X_i and X_j. These general constraints are in reality functional constraints since the composition of a set of functional constraints is a functional constraint. Then, by Property 2, after applying label-arc consistency, the labelling allows the identification of these general constraints[3] which are in reality functional. For instance, thanks to the labelling, the general constraint C_{ij} , where D_i and Dj have a same label,



is treated as a functional constraint by our method. In the rest of this paper, we call these constraints implicit functional constraints.
If a modification must be made on any domain of D_k, this modification will be effectively made on the representative domain D_k. Namely, if a value v_s must be removed from a domain D_j labelled by k, we remove effectively its representative value from D_k, since by Property 1, the value v_s participates to a solution of a CSP if and only if its representative value in D_k participates also to that solution. This leads not to reconsider all functional constraints.

The algorithm The first phase of LAC consists in checking the constraints and labelling the domains. The second phase deals with the constraint propagation process. Two arrays are used for stocking the labels. An array label_domain, initialized as label_domain[i] = i for i=1,..., n, indicates that each domain is represented by itself. And an array label_value, initialized as label_value[i][j]=j for i=1,..., n, j=1,...,d_i, where d_i is the size of the domain D_i.

```
LAC(CSP)
begin
(phase 1)
for each constraint Cij of C;
    If Cij is functional
    then FunctionalTreatment(i, j);
    else NonFunctionalTreatment(i, j);
(phase 2)
while L is not empty
    consider an element (k1, label1) of L;
    while Sk1,label1 is not empty
            consider an element (i, r, j, s) of Sk1,label1 ;
            k2 = ComputeLabelDomain(i);
            If (k1=k2)
                then CheckLabelNonFunctional(i,j,k1);
```

[3]Note that our method does not modify these constraints, the fact that these constraints are functional is indicated by the labels.

```
        else perform arc consistency between Di and Dj[4];
            L← L U {the set of representatives inconsistent values of Di and Dj};
end.
```

Fig. 1.

Figure 1 shows the LAC algorithm. During the first phase, LAC. checks each constraint exactly once, and uses two procedures which provide a specific treatment for functional constraints and general constraints. During the second phase, LAC uses the procedure CheckLabelNonFunctional(i, j) which is called when the domains are labelled by a same label. Note that L contains the current representative values to remove.

The procedures in Figures 2 and 3 present the specific treatment devoted to functional and non-functional constraints respectively. Two possible cases are examined by these procedures. Either the labels of the domains D_i and D_j are equal (this means that there exists a sequence of functional constraints between X_i and X_j), in which case the CheckLabelFunctional (resp. CheckLabelNonFunctional) procedure is called for performing label-arc consistency between the domains D_i and D_j. Or, D_i and D_j have two different representative domains D_{k1} and D_{k2} respectively. In the case, the FunctionalTreatment procedure propagates the labelling in order to make all domains, which are labelled with k_1 and k_2 have the same representative domain D_{k1} or D_{k2}. While the NonFunctionalTreatment procedure performs arc consistency between the domains D_i and D_j.

```
FunctionalTreatment(i, j)
begin
k1=ComputeLabelDomain(i)
k2=ComputeLabelDomain(j)
If (k1=k2) then
        CheckLabelFunctional(i, j, k1);
    else
        PropagateLabel(i, j, k1, k2);
end.
```

Fig. 2.

```
NonFunctionalTreatment(i, j)
begin
k1=ComputeLabelDomain(i)
k2=ComputeLabelDomain(j)
If (k1=k2) then
    CheckLabelNonFunctional(i, j, k1);
else
    perform arc consistency between Di and Dj;
    L← L U {the set of representatives
    inconsistent values of Di and Dj};
end.
```

Fig. 3.

The PropagateLabel procedure consists in propagating the labels from D_{k1} to D_{k2}. Let D_i and D_j be two domains labelled by k_1 and k_2 respectively, and C_{ij} a functional constraint. This procedure labels the representative domain D_{k2} by k_1, then the domains of the two sets will have a same label k1. When a domain D_{k2} is labelled by k_1, we transfer all necessary information[5] concerning D_{k2} to the representative domain D_{k1}. In order to do this, we use the notion of first support introduced in AC6 [1], by modifying slightly the data structure of this algorithm. We recall that this algorithm manages, for each value (X_k, v_l), a list $S_{k,l}$ representing the set of values for which $X_k, v_l)$ is the first support. In our case, we manage a list $S_{k,l}$ which contains elements

[4] We use the specific treatment of arc consistency algorithm AC5 devoted to different types of constraints (anti-functional, monotonic,...).

[5] The information necessary for making the constraint propagation process.

of the form (i,r,j,s) for which the value (X_j, v_s) is the first support of the value (X_i, v_r) relatively to the constraint C_{ij}, and $v_l \in D_k$ is the representative value of $v_s \in D_j$. This allows us to keep a trace for searching a new first support for the value (X_i, v_r) when the current representative value (X_k, v_l) is removed. This indicates that the new first support will be searched in D_j. Thus, the transfer of the information concerning D_{k2} to the representative domain D_{k1} is made by updating the lists $S_{k1,l1}$ for $l1=1,...,d_{k1}$ where d_{k1} is the size of D_{k1}. This is done with at most d operations.

```
PropagateLabel(i, j, k1, k2)
begin
domain ← empty;
for each value v_r ∈ D_i
label1=ComputeLabelValue(i, r);
if (label1)
then if (v_s=f_ij(v_r)) ∈ D_j
  then
     label2=ComputeLabelValue(j, s);
     If (not(label2))
       then L← L U {(k1, label1)};
            remove v_label1 from D_k1;
       else
         S_k1,v_label1 ← Concatenate(S_k1,v_label1, S_k2,v_label2)
         label_value[k2][label2]=label1;
         domain ← domain U {v_s};
else
    L← L U {(k1, label1)};
    emove v_label1 from D_k1;
for each value v_s ∈ D_j and v_s ∉ domain
do label=ComputeLabelValue(j, s);
        if (label) then L← L U {(k2, label)};
                    remove v_label from D_k2;
label_domain[k2]=k1;
end.
```

Fig. 4.

It suffices to update some pointers. Namely, to perform this we concatenate the lists $S_{k2,l2}$ to $S_{k1,l1}$ for each pair of values v_{l1} and v_{l2} of D_{k1} and D_{k2} where v_{l1} and v_{l2} are the representative values of $v_r \in D_i$ and $f_{ij}(v_r) \in D_j$.

The procedures in figures 5 and 6 perform label-arc consistency between the domains D_i and D_j. The CheckLabelFunctional procedure deals with the functional constraints while the CheckLabelNonFunctional deals with general constraints.

```
CheckLabelFunctional(i, j, k)
begin
domain ← empty;
for each value v_r ∈ D_i
label1=ComputeLabelValue(i, r);
if (label1)
then if (v_s=f_ij(v_r)) ∈ D_j
    then
        label2=ComputeLabelValue(j, s);
        if (not(label2)) then L← L U {(k, label1)};
                             remove v_label1 from D_k;
        else if label1≠ label2) then
                        L← L U {(k, label1)};
                        remove v_label1 from D_k;
                        L← L U {(k, label2)};
                        remove v_label2 from D_k;
             else domain ← domain U {v_s};
    else
        L← L U {(k, label1)};
for each value v_s ∈ D_j and v_s ∉ domain
do label=ComputeLabelValue(j, s);
     if (label) then L← L U {(k, label)};
                             remove v_label from D_k;
end.
```

Fig. 5.

```
CheckLabelNonFunctional(i, j, k)
begin
domain ← empty
for each value v_r ∈ D_i
do label1=ComputeLabelValue(i, r);
if (label1)
then  support=false;
      for each value v_s ∈ D_j
      do if R_ij(v_r,v_s)
        then
            label2=ComputeLabelValue(j, s);
            if (label1=label2) then
                        support=true;
                        domain ← domain U {v_s};
                        break;
    if (not(support)) then L← L U {(k, label1)};
                             remove v_label1 from D_k;
for each value v_s ∈ D_j and v_s ∉ domain
do label=ComputeLabelValue(j, s);
  if (label) then L← L U {(k, label)};
                        remove v_label from D_k;
end.
```

Fig. 6.

The procedures in figures 7 and 8 supply the label of a domain and the label of a value respectively. The function ComputeLabelDomain returns the label k of a domain D_i. D_k is the representative domain of D_i. The function ComputeLabelValue returns the label of a value v_r or false when the representative value of v_r is already removed (this means that v_r is inconsistent).

```
ComputeLabelDomain(i)
begin
rep=i
while (label_domain[rep] ≠ rep)
        do rep = label_domain[rep];
return (rep);
end.
```

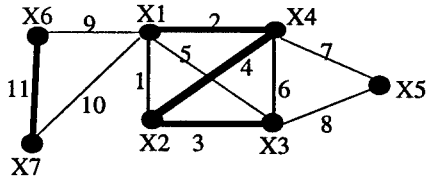
Fig. 7.

```
ComputeLabelValue(i, r)
begin
label= r;
rep=i;
while (label_domain[rep] ≠ rep)
 do label = label_value[rep][label];
    rep = label_domain[rep];
    if (vlabel ∉ Drep) then return(false);
return(label);
end.
```

Fig. 8.

An illustrating example

Let us consider the following CSP whose constraint graph is :



The functional constraints are represented by bold edges.
The following 8 points show the treatment performed by LAC during the first phase on the CSP. The order of constraint checking is chosen in the aim to have a good illustration of the LAC method.

1) The constraint (1) is general, then arc consistency is performed between D_1 and D_2.
2) The constraint (2) is functional, then arc consistency is performed between D_1 and D_4, and the domain D_4 is labelled by 1.
3) The constraint (3) is functional, then arc consistency is performed between D_2 and D_3, and the domain D_3 is labelled by 2.
4) The constraint (4) is functional, then arc consistency is performed between D_4 and D_2, and the domain D_2 is labelled by 1, since D_4 is already labelled by 1.
5) The constraint (5) is general, D_1 and D_3 have the same label 1 since D_2 is labelled by 1, then label-arc consistency is performed between D_1 and D_3.
6) The constraint (6) is general, D_3 and D_4 have the same label 1, then label-arc consistency is performed between D_3 and D_4.
7) Once the constraints 7, 8, 9 and 10 are checked, we perform arc consistency between the domains D_4 and D_5, D_3 and D_5, D_1 and D_6, D_1 and D_7.
8) The constraint (11) is functional, then arc consistency is performed between D_6 and D_7, and the domain D_7 is labelled by 6.

Remark that the set of domains of this problem is partitioned into 3 subsets of domains. $D_1 = \{D_1, D_2, D_3, D_4\}$ the subset of domains labelled by 1 (their representative domain is D1), $D_6 = \{D_6, D_7\}$ the subset of domains labelled by 6, and $D_5 = \{D_5\}$, the domain D_5 is represented by itself.

During the phase 2, all functional constraints and the general constraints 1 and 5 will not be reconsidered. The constraint 6 is reconsidered at most once.

The following theorem shows the completeness of the LAC algorithm.

Theorem 1 Let (P) be a CSP and (P') the CSP obtained after applying the LAC algorithm. The CSP (P') is label-arc consistent.

Proof As classical arc consistency algorithms, LAC proceeds in two phases. In the first, each constraint is checked once, and a queue L of elements to reconsider in the second phase is built. At the end of the first phase, the set of domains of a CSP are partitioned into subsets $D_1,..., D_p$: $p \leq n$ of domains. Each of these subsets groups together the domains having a same label k, and is represented by the domain D_k. Note that we can associate to each subset D_k of domains a subproblem CSP P_k.
The second phase performs the constraint propagation process. We will show, in this phase, that it is not necessary to reconsider the functional constraints, nor to reconsider any general constraint of a subproblem P_k more than once. Let $D_i \in D_k$, and v_j be a value of D_i, and k the label of D_i. Assume that v_j is detected inconsistent, then the representative value v_r ($v_r \in D_k$) of v_j is enqueued in L. The propagation process leads to check the constraints of P_k and the general constraints linking the variables of P_k to other subproblems. The functional constraints of each P_k will not be checked since by property 1 there is at most one solution containing the value v_j satisfying the constraints of P_k, and this solution is represented by the value v_r of D_k. The general constraints of P_k will be checked at most once, since when a general constraint of P_k is considered in this phase, the label-arc consistency concept is applied to this constraint, and from property 2 this constraint will be an implicit functional constraint and then not reconsidered. The other non functional constraints are treated as in AC5 algorithm using the labelling and the notion of representative domain and value. ♦

4.2 Analysis

We present here some results showing that the technique used in LAC makes this algorithm more powerful than classical arc consistency algorithms. Namely, we show, on the one hand, that LAC allows all functional constraints to be checked only once, and some general constraints at most twice. And, on the other hand, some cases of CSPs can be solved by only achieving LAC, without backtracking.

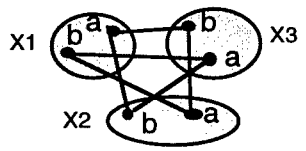
Note As shown in 4.1, LAC uses sometimes the label-arc consistency concept for filtering the domains of a CSP, then it is clear that the effectiveness of LAC outperforms the one of classical arc consistency algorithms.

Theorem 2 Let (P) be a CSP. If all the constraints of (P) are functional, then (P) can be solved by applying LAC. Furthermore, each constraint of (P) is checked only once.

Proof The LAC algorithm checks the constraints of (P) progressively. Then, when a constraint C_{ij} is checked, either the domains D_i and D_j have a same label, in which case the label-arc consistency concept is applied to this constraint. If for every tuple (v_r, v_s) of R_{ij}, the labels of v_r and v_s are different, we conclude that (P) has no solution (since their representative domain becomes empty). Or, the domain D_i is

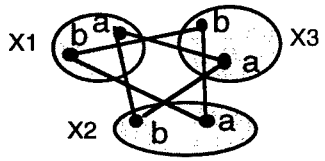
labelled by k1 and D_j by k2 (k1≠k2), in which case the classical arc consistency concept is applied, and for each tuple (v_r, v_s) of R_{ij}, the values v_r and v_s will have a same label. Assume that after checking each constraint of (P) once, all the domains are not empty; we conclude that (P) has a solution. Let X_i be a variable of (P) and v_r a value of D_i. Since the constraints of (P) are all functional, then by property 1 there exists at most one solution of (P) such that $X_i=v_r$. The values of other variables forming this solution will have the same label as v_r. All, solutions of (P) can be found by taking for each of them the values having the same label. Then, (P) has a solution if and only if all the domains of (P) are not empty after applying the LAC algorithm. ♦

Example Let us consider the following sample CSP with 3 variables and 3 functional constraints whose constraint graph is



We can verify easily that this CSP is arc consistent, but we cannot say whether the CSP has no solution by using classical arc consistency algorithms.

Applying LAC to this problem, the checking of C_{12} leads to label the domains D_1 and D_2 by 1, the values a and b of D_1 and D_2 respectively by 1, and the values b and a of D_1 and D_2 respectively by 2. In the same way, the checking of the constraint C_{13} leads to label D_3 by 1, the values a and b of D_3 by respectively 2 and 1. When, the constraint C_{23} is checked, since D_2 and D_3 have the same label, we apply the label-arc consistency concept between these domains, then their representative domain D_1 will be empty since the values a and b of D_2 and D_3 are not label-arc consistent. Thus, we deduce that this problem has no solution.
In the same way for the following CSP with 3 variables and 3 functional constraints,



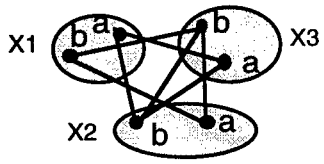
We cannot verify with the classical arc consistency algorithms whether this problem has a solution. Since after applying LAC, all the domains will not be empty, all solutions of this problem can be computed by taking for each of them the values having the same label.

The class of problems CSPs which can be solved using LAC, can be extended to problems containing general constraints.

Definition 8 Let (P) be a CSP and G=(V, U) its constraint graph, where V and U are respectively the set of vertices and edges representing the variables and the constraints of (P). And, let E be the set of edges representing the general constraints of (P). The partial graph G_b of G is defined by (V, U-E), (G_b is obtained by removing from G the set of edges E).

Theorem 3 If G_b is connected, then LAC achieves label-arc consistency on (P) by checking each functional constraint once and each general constraint at most twice. And, if each general constraint C_{ij} of (P) is checked when D_i and D_j have a same label, then LAC guarantees the solving of (P).

Proof LAC checks the constraints of (P) progressively. Then, when a functional constraint is encountered, we make the same reasoning as in the proof of theorem 2. When, a general constraint C_{ij} is checked, since G_b is connected, then either D_i and D_j have a same label in which case label-arc consistency concept is applied to this constraint, and for each tuple (v_r, v_s) of R_{ij}, the values v_r and v_s will have the same label. By property 2, C_{ij} will be an implicit functional constraint, and this constraint will not be reconsidered. Or, the domain D_i is labelled by k1 and D_j by k2 ($k1 \neq k2$), then arc consistency concept is applied to this constraint. In this case, the constraint C_{ij} may be reconsidered at most once, since when it is reconsidered we are sure that the domains D_i and D_j will have a same label and the label-arc consistency will be applied to it, then by property 2 this constraint will be an implicit functional constraint and then not be checked again.
If each general constraint C_{ij} of (P) is checked when D_i and D_j have a same label, then for each tuple (v_r, v_s) of R_{ij}, the values v_r and v_s will have the same label. And, if after applying LAC, the domains of (P) are not empty, we conclude that (P) has a solution, and all solutions of (P) can be found by taking for each of them the values having the same label. ♦
Note that if each general constraint of (P) is checked twice, then LAC guarantees the solving of (P).
Example Let us consider the following CSP with 3 variables and 3 constraints, C_{12} and C_{13} are functional, while C23 is a general constraint.



The checking of C_{12} leads to label the domains D_1 and D_2 by 1, the values a and b of D_1 and D_2 respectively by 1, and the values b and a of D_1 and D_2 respectively by 2. In the same way, the checking of the constraint C_{13} leads to label D_3 by 1, the values a and b of D_3 by respectively 1 and 2.
When, the constraint C_{23} is checked, since D_2 and D_3 have the same label, we apply the label-arc consistency concept between these domains, this constraint will be an implicit functional constraint. We can not verify with the classical arc consistency algorithms whether this problem has a solution. Since after applying LAC, all the domains will not be empty, all solutions of this problem can be computed by taking for each of them the values having the same label.

Assume that G_b is not connected, and let GP be the partial graph of G obtained by adding to G_b each edge representing a general constraint of (P) having its extremities in a same connected component of G_b. The following corollary points out the general constraints of a CSP checked at most twice.

Corollary The general constraints corresponding to the edges added to G_b are all checked at most twice.

Proof By theorem 3, the general constraints of each connected component of G_b are checked at most twice. ♦

Note The LAC method can be used as a decomposition strategy for solving CSPs containing functional constraints. Namely, if we want to solve completely the

problem, we can use the LAC method as a preprocessing for filtering and partitionning the set of its domains into subsets $D_1,..., D_p : p \leq n$ of domains. Each partition D_k has a representative domain D_k and may be associated to a subproblem (P_k) of (P). We instantiate only the variables of the representatives domains for covering all the search space of solutions, since each value of each representative domain D_k participates at most in one solution of (P_k). Thus, the theoretical complexity of solving such problems is in $O(d^p)$.

5. Computational experiments

We have compared the performances of LAC, AC5 and AC5+ over CSPs containing functional and IFC constraints. We have used for all these algorithms the notion of first support introduced in AC6 [1].
The experiments were performed over randomly generated problems using the random model proposed in [5].
The generator considers four parameters

- n, the number of variables
- d, the domain size of each variable
- pc, the probability that a constraint C_{ij} between two variables exists
- pu, the probability that a given tuple (a,b) belongs to R_{ij} The comparisons are concentrated on the three parameters pc $\in \{0.3, 0.6, 0.9\}$, pu $\in \{0.4, 0.7\}$ and nf the number of functional constraints varying between 4 and 40. Each algorithm was applied to 20 randomly generated problems, for each tuple of values of pc, pu and nf. All test problems have n=32 and d=16. For each instance (pu, pc, and nf fixed), and for each method M we counted the ratio r_M = (the running time of M / number of removed values by M) expressing the efficiency of the method M (while the ratio decreases the efficiency of the method M increases).

LAC versus AC5
Figures 1a to 1c show the comparison between LAC and AC5. The x-axis represents the number of functional constraints and the y-axis the ratio r_{LAC} / r_{AC5}. This ratio express the relative efficiency between LAC and AC5 (when the ratio is little than 1, LAC outperforms AC5). We have remarked that these methods have almost the same running time for problems having few functional constraints ($4 \leq nf < 8$). The LAC method becomes faster than AC5 when the number of functional constraints increases. The number of removed values by LAC is always greater than the one removed by AC5.

LAC versus AC5+
As mentioned above the AC5+ algorithm performs a specific treatment only for IFC constraints, while our method deals with functional constraints including the IFC constraints. For this reason, we have considered only problems containing IFC and general constraints for evaluating the performances of these methods (All functional constraints of the test problems are IFC constraints). Figures 2a to 2c show the comparison between LAC and AC5+. The x-axis represents the number of functional constraints and the y-axis the ratio r_{LAC} / r_{AC5+}. This ratio express the relative efficiency between LAC and AC5+. We remark that the LAC method outperforms AC5+ on all tested problems. We can see that the ratio r_{LAC} / r_{AC5+} decreases with the number of IFC constraints, this means that the efficiency of the LAC method

increases with the number of functional constraints. This may be explained by the fact that AC5+ performs the specific treatment on the most part of the IFC constraints when the problems contain few IFC constraints ($4 \leq nf < 12$), since in this case the order of the checking of the constraints does not affect the number of IFC constraints checked only once. When the number of IFC constraints exceeds 12 the LAC method outperforms largely AC5+, since in this case the number of IFC constraints checked many times by AC5+ increases.

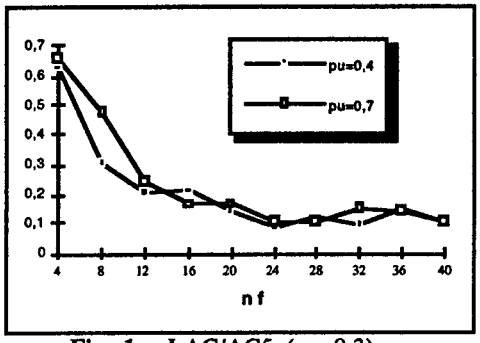


Fig. 1a. LAC/AC5 (pc=0,3)

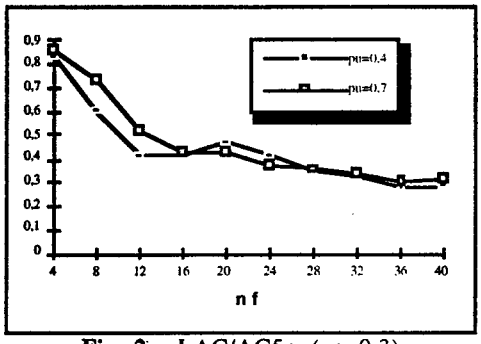


Fig. 2a. LAC/AC5+ (pc=0,3)

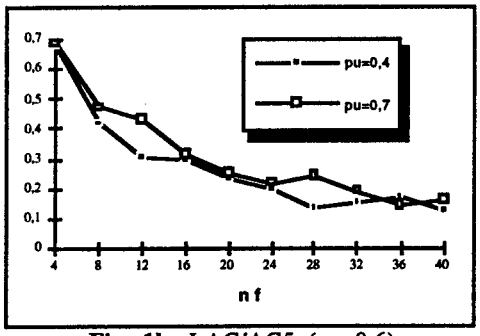


Fig. 1b. LAC/AC5 (pc=0,6)

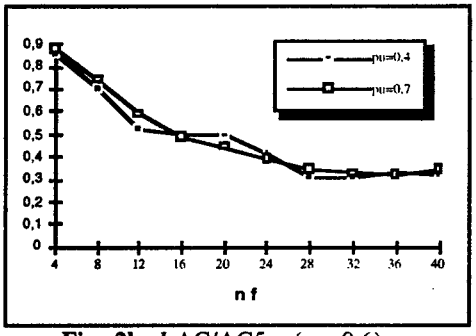


Fig. 2b. LAC/AC5+ (pc=0,6)

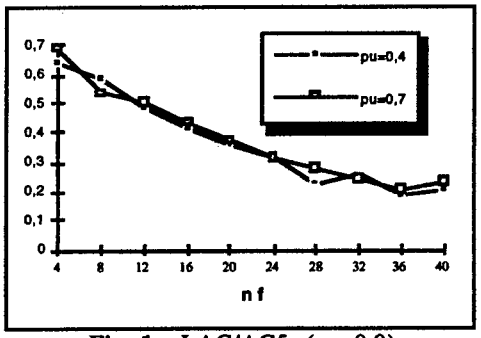


Fig. 1c. LAC/AC5 (pc=0,9)

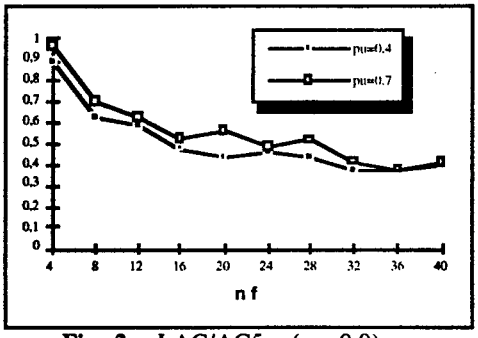


Fig. 2c. LAC/AC5+ (pc=0,9)

The computational experiments show the gain brought by the LAC method. The efficiency of the LAC method is justified by its performances (the number of checks of constraints is lower than the one checked by AC5 and AC5+) and its effectiveness (the number of removed values is greater than the one of removed by AC5 and AC5+).

6. Conclusion

We have proposed a labelling arc consistency method LAC for filtering CSPs containing functional constraints. This method uses two consistency concepts, the classical arc consistency and the label-arc consistency. The performances and the effectiveness of LAC on CSPs containing functional constraints are more important than the classical arc consistency algorithms, since on the one hand, this method avoids to check many times some constraints, and on the other hand, LAC allows in some cases the solving of the CSP or the improving of the effectiveness of the filtering. The technique used in LAC makes its incremental aspect efficient since as seen above this method does not recheck a part of the set of constraints. The main application of LAC is the constraint programming languages. The empirical results presented show the substantial gain brought by the LAC method.

7. References

[1] Bessière C., "Arc-consistency and arc-consistency again", Research Note in Artificial Intelligence, Vol. 65, 1 pp. 179-190.
[2] Bessière C., Freuder E.C., and Régin J-C, "Using Inference to Reduce Arc Consistency Computation", IJCAI95, Montreal, pp592-598.
[3] David P., "When functional and bijective constraints make a CSP polynomial", IJCAI93, Chambery, France, pp. 224-229.
[4] Dincbas M. and al., "Solving large combinatorial problems in logic programming", Journal of Logic Programming, 8, pp. 75-93, 1990.
[5] Hubbe P. and Freuder H., "An efficient Cross-Product Representation of the Constraint Satisfaction Problem Search Space", In proc. of AAAI, 1992, P. 421-427.
[6] Liu B., "Increasing Functional Constraints Need to Be Checked Only Once", IJCAI95, Montreal, pp 586-591.
[7] Mohr R. and Henderson T.C., "Arc and path consistency revisited", Artificial Intelligence, 28-2, 1986, pp. 225-233.
[8] Mohr R. and Masini G., " Running efficiently arc consistency", Springer, Berlin, 1988, pp. 217-231.
[9] Montanari U., "Networks of constraints: Fundamental properties and applications to picture processing", Inform. Sci., vol. 7 n°2, 1974, p. 95-132.
[10] Van Hentenryck P., "Constraint satisfaction in Logic Programming", MIT press, Cambridge, MA, 1989.
[11] Van Hentenryck P., Devilles Y. and Teng C-M. A generic arc consistency algorithm and its specifications. Artificial Intelligence, 27, pp. 291-322, 1992.

Constraint Satisfaction in Optical Routing for Passive Wavelength-Routed Networks

Dhritiman Banerjee and Jeremy Frank

Department of Computer Science
University of California, Davis
Davis, CA 95616
{*banerjed,frank*}*@cs.ucdavis.edu*

Abstract. A wavelength-routed, optical network employs all-optical channels *(lightpaths)* on multiple wavelengths to establish a rearrangeable interconnection pattern *(virtual topology)* for transport of data. A lightpath may span multiple fiber-links, and may be routed optically at an intermediate node without undergoing electronic conversion. We examine the problem of establishing a set of lightpaths in an optical network, which employs a passive wavelength routing device called a Latin Router (LR). Latin Routers are attractive for optical network design because of their fault-tolerance and low cost, but make traditional routing algorithms difficult to implement due to the complexity of the constraints they impose on legitimate routes and colors. We employ a local search algorithm to search the space of virtual topologies in order to satisfy a maximum number of given lightpath requests. We use the same algorithm to maximize the number of single-hop connections for a given network. We show that the algorithm can satisfy a high percentage of lightpaths under low to moderate network loads. Experiments reveal that we can establish $O(N)$ lightpaths in an optical network with N nodes. We believe that our work is the first known attempt at designing optical wide-area networks using Latin Routers.

Keywords: wavelength routing, latin routers, local search, optical networks

1 Introduction

A lightpath is used in a wavelength-routed, optical network to establish high-speed, all-optical, channels which can span multiple fiber links without undergoing electronic processing at the intermediate nodes of the network. For example, in Fig. 1, optical lightpath LP4 is established directly connecting nodes 4 and 2 through an all-optical channel. In the absence of wavelength conversion devices at the intermediate nodes of the network, a lightpath will be on the same wavelength on all the fiber-links through which it traverses; the lightpath will be switched optically at the intermediate nodes, e.g., lightpath LP4 is optically switched at node 5 and node 1 before it finally terminates at node 2. A lightpath is typically a unidirectional channel of communication, i.e., a lightpath from

node 4 to node 2 does not necessarily mean that there will be a lightpath from node 2 to node 4.

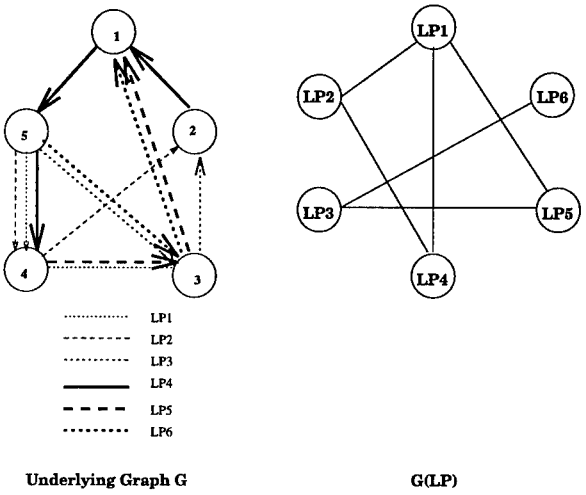


Fig. 1. From G to $G(VC)$.

Using all-optical lightpaths in the network architecture considerably reduces the processing time at intermediate switching nodes, by optically switching forwarded traffic. If two lightpaths traverse one or more common fiber links, the lightpaths must necessarily be operated on different wavelengths. For example, in figure 1 lightpaths LP1 and LP4 traverse a common fiber 4-5, and hence should be on different wavelengths. Typically the number of wavelengths available in the network is fixed at some maximum number, and is limited by the technology used to build the network.

A routing node in this network employs an optical component for wavelength routing of all-optical lightpaths. We use a Latin Router as a passive wavelength router in the optical component of the router, because of its low cost and robustness. A $K \times K$ Latin Router (LR) (shown in Fig. 2) provides complete connectivity between every input and output port, by passively routing K^2 optical connections on K wavelengths. A certain router, called the *Shift Latin Router* (SLR), has a fixed cyclical-permutation-based interconnection pattern between its input and output ports, e.g., wavelength k on input port i is always routed to the same wavelength on output port $(i + k)$ *modulo* K, $\forall i, k \in [0, 1, \ldots, N-1]$. A particular feature of the Latin Router is that the number of wavelengths supported in the router is equal to the size of the Latin Router.

A common problem in optical network design is: given a physical network topology and a numbers of available wavelengths (hereafter called colors), can we establish a given set of lightpaths? Requested lightpaths are given as source-destination pairs of nodes in the underlying physical topology. Previously pro-

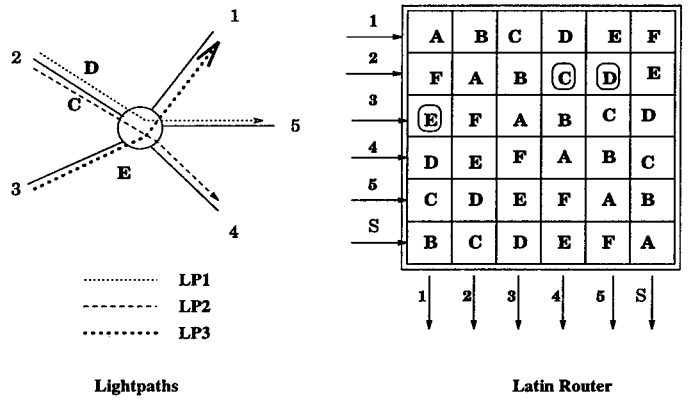


Fig. 2. Latin Square Router.

posed algorithms typically solve this problem in two distinct phases: *routing* and *coloring*. The routing algorithm uses traditional shortest-path-based algorithms to generate a set of routes for each lightpath. Each lightpath is then assigned a color, such that no two lightpaths passing through a common link are assigned the same color. Most existing network designs are based on the Wavelength Routing Switch (WRS), which is a reconfigurable router allowing any wavelength to be switched from any input port to any output port, and hence does not impose any restrictions on the routing algorithm [BM95]. The advantage of this scheme is that routing and coloring are both well-studied problems, and therefore a large number of existing techniques can be employed to solve the problem.

While the above techniques work well in WRS-based networks, we must find different solutions for LR-based networks. In particular, Shift Latin Routers make it difficult to separate the algorithmic process into routing and coloring stages. The constraints that are imposed by a Shift Latin Router on the wavelength assigned to a lightpath needs to be accommodated by the routing algorithm, otherwise unacceptably small numbers of lightpaths will have colorings which obey those constraints. Modifying the routing algorithms to minimize the impact of those constraints is difficult and substantially complicates the routing process (this is explained later in the Appendix). Is is important to develop schemes to handle the constraints that these routers impose on optical routing.

An alternative approach to constraint satisfaction is *local search* which has proven successful at rapidly solving a variety of constraint problems [SLM92, MJPL92]. Local search makes small changes to a complete assignment of variables in a constraint problem in order to improve the quality of the solution. We have devised an algorithm called the Local-search Optical Network Configuration Algorithm (LONCA[1]) which routes static requests for an optical network built using Latin Routers. LONCA addresses two problems, which are as follows.

– Establish the maximum number of a set of requested lightpaths in a network.

[1] A *lonka* is an Indian chili pepper.

– Establish the maximum number of single-hop[2] connections in a network.

LONCA operates by performing local search on the space of *virtual topologies* (i.e. possible interconnections of routers) to find such topologies which satisfy the maximum number of lightpath requests or which establishes the maximum number of single-hop in an optical network.

It is not possible to set up a complete graph as a virtual topology by establishing lightpaths to provide single-hop connectivity between every pair of nodes. In this case, traffic might need to go through multiple lightpaths, while undergoing electronic switching at the endpoints of adjacent lightpaths. Minimizing the average number of optical hops that traffic has to traverse in the network to reach from a source node to a destination node is a related optimization problem. LONCA does not handle this problem directly, but we investigate the virtual topologies maximizing the number of single-hop connections to see how many optical hops are necessary to establish all connections in the network.

In §2 we provide the problem formulation, and in the Appendix we describe the constraints imposed by the physical network topology and the routers used on the lightpath establishment problem. We discuss traditional constraint solving techniques, which separate routing and coloring, in §3. In §4, we discuss LONCA, which is based on the well-known local search paradigm. We present simulation results for this algorithm on randomly generated problem instances in §5. Our results suggest that LONCA can establish $O(N)$ lightpaths in a network on N nodes. This can help in drastically reducing the average hop distance in the network. In §6 we conclude and discuss future work.

2 Routing in Optical Networks

We discuss the problem formulation, and briefly mention the traditional routing algorithms used to satisfy a set of lightpath requests. In the Appendix, we discuss how different parameters in the underlying graph and the chosen router can affect the constraint-satisfaction problem.

2.1 Problem Formulation

The input to the problem is a graph $G =< V, E >$, $|V| = N$, a number of available wavelengths k and a set of i requested lightpaths $C = \{s_i, d_i\}$. The number of wavelengths available restricts the number of lightpaths which can traverse a single link. Lightpaths are directed; in other words, we may desire to have a directed lightpath from s_i to d_j, without a lightpath from d_j to s_i. Traditionally we employ a *routing* algorithm which takes the list $C = \{s_i, d_i\}$ and produces a set of uncolored routes $LP = \{lp_{i1}, lp_{i2}...lp_{ij}\}$ such that $\forall i$, $lp_{i1} = s_i \bigwedge lp_{ij} = d_i$. Two lightpaths lp_a, lp_b can't have the same *color* if they share an edge $(i, j) \in G$. This motivates a graph coloring problem where each connection

[2] A single-hop connection is a lightpath connecting two nodes of the network, such that they can communicate with each other in one optical hop

is isomorphic to a node of an auxiliary graph; two nodes have an edge between them in the auxiliary graph if their corresponding lightpaths share an edge in the underlying graph G. We shall refer to this auxiliary graph as the graph *induced* by LP and denote it as $G(LP)$. Fig. 1 shows an example of constructing $G(LP)$ from G. We observe, for example, that lp_1, lp_2 and lp_4 all use edge $4-5$ in the graph, and so they form a 3-clique in $G(LP)$.

Given N nodes in G, $G(LP)$ has $|LP| < N(N-1)$ nodes. The number of edges in $G(LP)$ is dependent on the routing of the lightpaths. Intuitively, shorter routes for lightpaths result in the fewer edges in $G(LP)$, because fewer lightpaths share the fiber links. The network topology, and the size of the router used, have also been shown to impact lightpath routing [BH95].

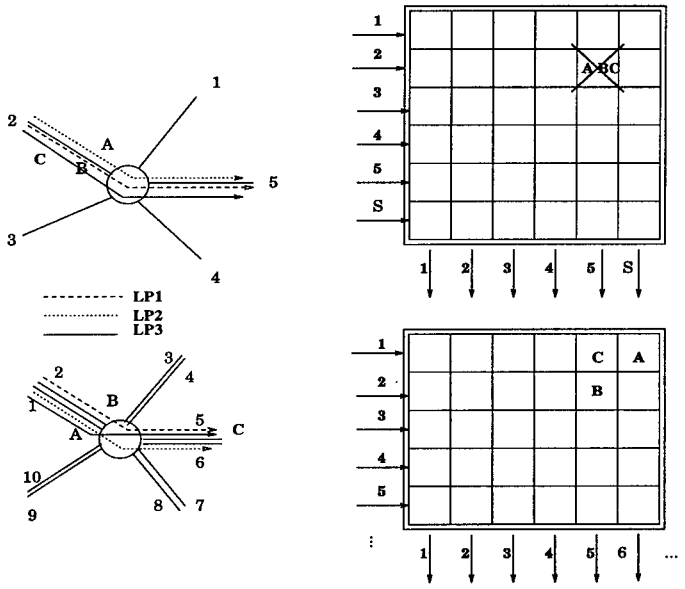


Fig. 3. Lightpaths in a Multigraph.

2.2 Physical Network

The characteristics of the network topology represented by G, and the router used at the network nodes, influences the solution to the problem. It is clear that the more edges G has, the less constrained the graph coloring problem is likely to be, because there will be fewer lightpaths sharing a fiber, thus decreasing the number of edges in $G(LP)$ If the underlying graph is a *multigraph*, i.e., there are multiple fibers connecting adjacent nodes in the physical topology, then more connections can pass between two heavily congested nodes without an increase in the number of colors the fiber must support. In Fig. 3 we see 3 lightpaths

passing from edge 2 to edge 5. In a simple graph this would result in a 3-clique in $G(LP)$; however, clever selection of the edges the lightpath uses can result in a less restrictive constraint graph. We discuss the impact of fiber multiplicity on lightpath routing in §3.

2.3 Optical Wavelength Routers

Optical networks can be built using many types of optical routers. The different capabilities in these routers impose different constraints on legal routes and have an impact on the constraint problem instance. A WRS [BM95] can perform arbitrary routing of wavelengths; it allows multiple wavelengths to be optically switched from any input fiber to any output fiber, as long as multiple lightpaths on the same wavelength don't need to be switched onto the same output fiber; however WRSs are costly to build.

The routing pattern in an *Arbitrary Latin Router* (ALR) is based on the Latin Square; for a router of size K the routing pattern consists of a $K \times K$ table. Each fiber attached to a node in G is attached to a single row (input port) and a single column (output port) of the table, and table entry (i, j) corresponds to the wavelength switched from input port i to output port j as shown in figure 2. The routing table has the property that no color appears twice in the same row or in the same column of the table, and only one color occupies each entry of the table. These two properties ensure that there is only a single wavelength which is switched from any input port to any output port, and that lightpaths switched onto an output port are distinct colors. Rows and columns which have no edge attached to them may be used as *access ports*, and are used to terminate lightpaths originating or ending at the node. Access ports are labeled with an S in figure 2. We do not know of any existing prototypes of Arbitrary Latin Routers.

A more restricted form of Latin Routers is called a *Shift Latin Router* (SLR). This router has the additional property that the wavelengths appear in increasing order in the top row of the table, and each subsequent row is the previous row rotated by one table cell; formally, wavelength k on input port i is always routed to the same wavelength on output port $(i+k)$ *modulo* K, $\forall i, k \in [0, 1, \ldots, N-1]$. Such a router appears in Fig. 2. We have used letters to denote wavelengths in order to avoid confusion with the labels for the incoming edges. In this figure, we observe that lp_3 is routed from input edge three to output edge one, and is assigned wavelength E.

Latin Routers impose a variety of different constraints on both routing and coloring. Traditional routing techniques can handle some of the constraints for ALRs [CB95], but SLRs add immense complexity to the problem, requiring intricate routing algorithms and the generation of polynomially many constraints even before coloring begins. This is because there are $O(NK^4)$ coloring constraints (N being the number of nodes, and $K \times K$ being the size of each router) which are imposed by the static routing property of the Latin Router, in addition to the coloring constraints imposed by two lightpaths sharing the same fiber. The reader interested in more details is referred to the Appendix.

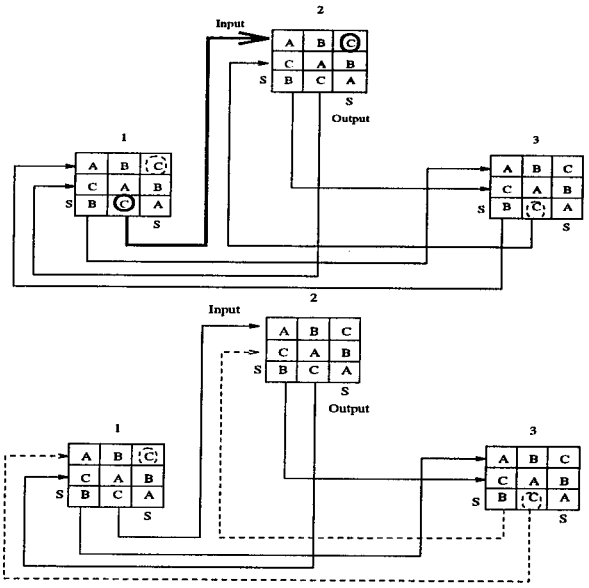


Fig. 4. Interconnection using Latin Squares.

In figure 4 we show an example of a 3 node network composed of 3x3 Latin Routers. Each undirected edge of the graph is represented by 2 directed edges between the routers. As before, an S by the input port (row) denotes the port at which connections originate at the node, while an S at the output port (column) denotes connections terminating at the node. Let us suppose we want a lightpath established between nodes 1 and 2. Input port 3 of node 1 is the designated port where connections originate, and the edge from output port 2 runs from node 1 to node 2. This edge enters port 1 of node 2, and output port 3 of node 2 terminates this lightpath. In order for this to be a valid lightpath, table entry (3,2) of node 1 and table entry (1,3) of node 2 must have the same wavelength; if we look at the circled table entries we see that both entries have wavelength C. So this edge denotes a lightpath. If we want to establish a lightpath from 3 to 1, however, we can't take the edge from 3 to 1 since entry (3,1) of node 3 is B and entry (1,3) of node 1 is C. In fact, we can't establish a lightpath from node 3 to node 1 in this configuration at all.

3 Previous Work

We briefly discuss previous work on algorithms designed to route and color lightpaths on optical networks. In most proposed algorithms, routing and coloring are treated as separate phases of the algorithm [BM95, RS94, ZA94]. We first discuss algorithms for Wavelength Routing Switches, and then for Arbitrary Latin Routers.

3.1 Lightpath establishment for WRS-based Networks

[BM95] present an analysis of routing and coloring techniques for establishing lightpaths using a WRS. Since the WRS imposes no additional constraints on coloring, the routing algorithm should provide a graph coloring problem instance which is as easy to solve as possible. One approach is to find a set of p short routes for each lightpath and pick the route which adds the fewest edges to $G(LP)$. A modification of Dijkstra's algorithm can be used to compute the p-shortest paths for each connection.

Once $G(LP)$ is created, any coloring algorithm can be used to color the resulting graph. On-line coloring algorithms have been used [BM95], although other fast algorithms such as min-conflicts can also be used [MJPL92]. An interesting special case of the problem occurs when the underlying network topology is a single cycle. Clearly the shortest path in such a network is an arc of length less than or equal to N/2. We see that the resulting constraint graph is a *circular arc graph*. While coloring such graphs is still known to be $\mathcal{NP}$-Hard, [Tuc75] gives an upper bound on the number of colors required by such graphs and a multi-commodity flow algorithm which solves the problem; [MIR93] gives on-line algorithms which approximate the number of required colors to within a constant factor.

3.2 Lightpath Establishment for ALR-based Networks

[CB95] present a discussion of routing in networks using Arbitrary Latin Routers. Their approach is to use p-shortest path as described above to find available routes for each connection. The algorithm selects the route which minimally constrains routers, and then assigns a wavelength which minimally constraints those routers. This algorithm was developed for ALRs and does not address the question of routing in networks composed of SLRs. To our knowledge no previous work has devised an optical routing algorithm for networks using SLRs.

4 Satisfying Connections Using Local Search

Traditional constraint satisfaction techniques use routing to generate an easy coloring problem, then satisfy the coloring constraints. There are polynomially many coloring constraints related to the configuration of the SLRs, and it is difficult to write routing algorithms which result in easy coloring problems due to the intricate nature of these constraints.

Local search has been successful in finding satisfying assignments for solvable K-SAT problems [SLM92] and graph coloring problems [MJPL92]. While it does not guarantee a solution, in practice local search algorithms tend to solve constraint problems with solutions very quickly. Local search algorithms examine small changes to a complete assignment of variables in a constraint problem and select one of the changes which improves the number of satisfied constraints the most. This procedure is sometimes referred to as *gradient ascent* or *hill-climbing*.

Since complete search of all virtual topologies appears to be costly, we felt that local search was an attractive alternative.

We decided to use local search to find a virtual topology which would satisfy the most lightpath requests; in effect, we merge the routing and coloring stages into the same algorithm. To do so, we make small changes in the virtual topology by changing the connections between edges in the graph and ports on the SLRs, thereby changing the mapping of colors onto edges. For instance, in figure 4, if we wanted a lightpath between nodes 3 and 1 and the nodes were connected in the fashion indicated, we would be unable to satisfy this request. However, if we were to change the edges assigned to the output ports of node 3 so that the edge from node 3 to node 1 was connected to output port 2, then the wavelength at entry (3,2) of node 3 and the wavelength at entry (1,3) of table 1 (corresponding to the termination of a connection) are both C, indicating that this is now a valid lightpath. To make this change, we would switch the edge running from node 3 to node 2 to output port 1 on node 3. Swapping pairs of input port or output port locations defines a natural neighborhood for local search.

We present the Local-Search Optical Network Configuration Algorithm (LONCA) in figure 5. Making a change to a single router is simply a matter of swapping the position of 2 of the edges attached to either input or output ports of a Latin Router. If the router size is K then there are at most $K^2 - K$ row swaps and column swaps possible for each router in G. These local changes to the network configuration allow us to move through the space of all virtual topologies. At each iteration of the local search algorithm, we select the edge swap which increases the number of satisfied lightpath requests the most; if there are several such changes we choose any one among them at random. For a single iteration, we examine swaps in only one router due to the cost of analyzing the updates and the large number of swaps which must be examined.

```
procedure LONCA(G, VC, RouterSize, MaxSwaps)
   connect all routers in G at random
   for i=1 to MaxSwaps
     pick a router at random
     for each row and column swap
       evaluate the number of connections satisfied
       update set of best swaps
     end for
     pick one of the best swaps and do it
     if all connections satisfied exit
   end for
end
```

Fig. 5. LONCA Algorithm Sketch.

A related problem to that of satisfying requested connections is that of max-

imizing the number of connections established in the network. In some cases network designers may not have a clear set of requests in mind, and may try to maximize the number of lightpath that can be established in the network, in order to minimize the average hop distance in the network. We use LONCA with a complete graph as the requested virtual topology to solve this problem.

5 Empirical Results

We tested LONCA on physical networks of different sizes and with differing numbers of requested connections. In each case we generated physical networks in the following way: we required each network to be a biconnected graph with average degree varying uniformly between two and seven, hence the number of edges in the networks were $4.5N$. These assumptions are based on characteristics of present-day fiber networks. Lightpath requests were generated by selecting source-destination pairs of nodes chosen uniformly at random without replacement. For these experiments we assume that all routers are of the same size K and that the multiplicity m of each edge in the graph is the same.

5.1 Satisfying Requested Lightpaths

Our first set of experiments was designed to analyze LONCAs ability to satisfying a set of requested connections. We generated 100 sets of connections for each of 5 different physical topologies, each consisting of 50 nodes. We analyzed networks with two different configurations: $m = 1, K = 8$ and $m = 2, K = 15$. We chose the router size such that a node in the graph with maximum degree would be guaranteed at least one access port for termination of lightpaths; hence $K = 7m + 1$. Moreover, Latin Router prototypes of sizes 8 and 15 have been reported in the literature [DEK91]. We ran LONCA once per set of connections with MaxSwaps set to 500 since our initial tests indicated that after 500 swaps LONCA was unable to improve the number of requests satisfied. We generated numbers of connections ranging from $2N - 10N$ and computed the proportion of connections satisfied. We present the results in Fig. 6. Each line of the plot indicates the performance of differing numbers of connections for the same physical network.

We observe that the proportion of established connections is higher for a fiber multiplicity of two than for a multiplicity of one. We observe, as expected, that in both cases the percentage of lightpaths satisfied decreases as the number of requests increases. However, the degradation in number of requests satisfied is relatively slow, with over 70% of 500 requests satisfied for a router size of 15 and a fiber multiplicity of two.

There are likely to be connections which we cannot be established due to violations of the Arbitrary Latin Router constraint. For example, if 10 requested lightpaths originate at a node with a physical degree of seven and a single source port, we can only set up seven of these connections. We were able to analyze those lightpath requests which were not satisfied with respect to the number

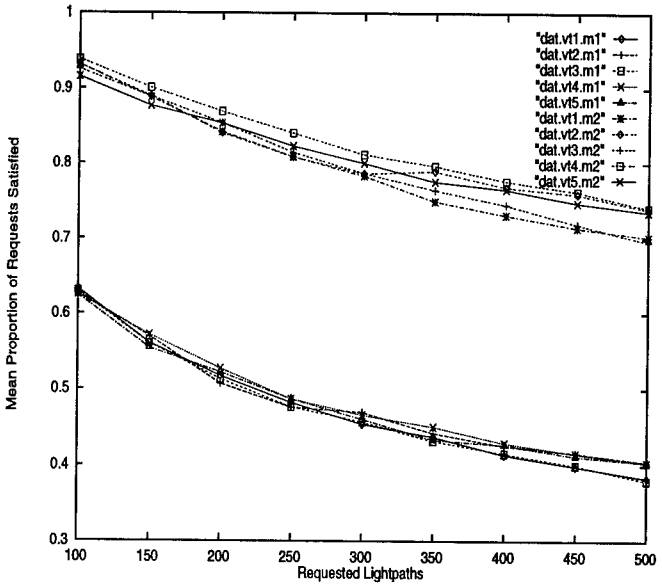


Fig. 6. Mean number of requested connections established.

of lightpath requests for the source and destination nodes. We found that for $m = 1$, only a small fraction of lightpath request failures were due to source or destination port overload, and that for $m = 2$ no lightpath request failures were caused by this problem.

5.2 Maximizing Single-Hop Connections

In our next experiment we ran LONCA on 100 different networks of sizes 50-100 incremented by 10. Our objective was to maximize the number of single-hop connections established in the given networks; in these experiments we asked LONCA to attempt to establish all $N(N-1)$ directed connections possible in the network. We ran LONCA 10 times per network configuration (to average out the randomizing effect of the local search algorithm) again using two network configurations: $m = 1, K = 8$ and $m = 2, K = 15$ with MaxSwaps set at 500. Fig. 7 shows the scaling in the number of connections we were able to establish. We see that LONCA is able to set up about $11.4N$ connections on N nodes when $m = 1$ and $22.8N$ connections when $m = 2$; we conjecture that LONCA can establish about $11.4mN$ connections in a network with multiplicity m, but we need to run more experiments to verify this claim.

We examined the resulting networks to determine how many nodes could be reached in one or two optical hops. We report on the results obtained for 100 node networks; again we tested the case for 100 different topologies with $m = 1, K = 8$ and $m = 2, K = 15$. When $m = 1$ LONCA was able to establish an average of 74% of all $N^2 - N$ connections in one or two optical hops. If m

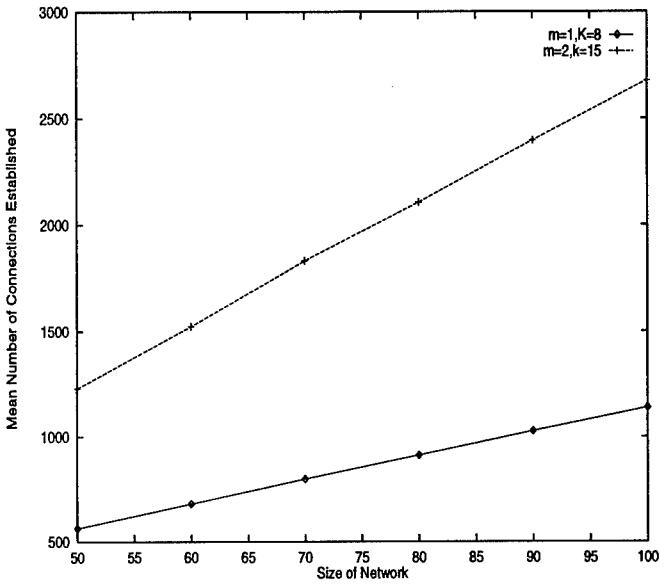


Fig. 7. Mean number of single-hop connections established.

increases to two, LONCA can establish an average of 99% of the connections (Fig. 8).

6 Conclusions and Future Work

We have presented and analyzed LONCA, a local search algorithm designed to perform routing of static lightpaths in all-optical networks. We show that LONCA is able to satisfy a high proportion of lightpath requests effectively in networks with high edge multiplicity, for a large number of requested connections. We also showed that LONCA can establish $O(N)$ lightpaths in a network of size N, and can effectively connect every node in a 100 node network with nearly every other node simultaneously in one or two optical hops, using a router size of 15×15, and fiber multiplicity of two.

m	Mean	Sdev
1	74.07%	1.43%
2	99.28%	0.23%

Fig. 8. Percentage of Connections Established in 2 Hops.

We were disappointed that we could not effectively analyze the reason for LONCA's inability to establish lightpaths. Clearly, understanding the problem

in establishing lightpaths will help us create more effective algorithms for establishing lightpaths in the future. We plan to study theoretical upper and lower bounds on the number of lightpaths that can be established in networks of Shift Latin Routers.

There are many variants of local search algorithms which improve performance. Most of these variants force more vigorous exploration of the assignment space by promoting changes to the assignment involving frequently unsatisfied constraints or frequently ignored variables. We hope to investigate the impact of these improvements on LONCA's performance.

We have only tested physical networks of one type in our experiments. We hope to continue experimenting with both more sparse and more dense networks to analyze LONCA's ability to satisfy constraints. We have suggested that when a network is a Hamiltonian Cycle that we may be able to devise better routing algorithms and coloring algorithms. Other special case networks such as regular graphs are also worth examining.

A major drawback to our experiments is the selection of sets of requests to examine. In the analysis section we mention that, even before routing, we can guarantee some connections will not be established due to excessive load at either the source or the destination. We therefore wish to examine a somewhat different problem; given a model of generating requests for lightpaths, how do we build inexpensive networks using SLRs which perform within some specified tolerance?

References

[BH95] R. A. Barry and P. A. Humblet. Models of blocking probability in all optical networks with and without wavelength changers. *Proceedings of IEEE Infocom*, pages 402–411, June 1995.

[BM95] D. Banerjee and B. Mukherjee. Practical approaches for routing and wavelength assignments in large all optical wavelength routed networks. *IEEE Special Issue JSAC/JLT on Optical Networks*, 1995.

[CB95] C. Chien and S. Banerjee. Optical switch configuration and lightpath assignment in wavelength routing multihop lightwave networks. *Proceedings of the 14th IEEE Infocom*, 1995.

[CPS90] O. Chien, H. Plugfelder, and J. Smith, editors. *Quasigroups and Loops: Theory and Applications*. Helderman Verlag, 1990.

[DEK91] C. Dragone, C. A. Edwards, and R. C. Kistler. Integrated optics nxn multiplexer on silicon. *IEEE Photonics Technology Letters*, 3(10):896–898, 1991.

[MIR93] M. Marathe, H. B. Hunt III, and S. S. Ravi. Efficient approximation algorithms for domatic partition and on-line coloring of circular arc graphs. *5th International Symposium on Algorithms and Information*, 1993.

[MJPL92] S. Minton, M. Johnston, A. Phillips, and P. Laird. Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 1992.

[RS94] R. Ramaswami and K. Sivarajan. Optical routing and wavelength assignment in all optical networks. *Proceedings of IEEE Infocom*, pages 970–983, 1994.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. *Proceedings of the 10th National Conference on Artificial Intelligence*, 1992.

[Tuc75] A. Tucker. Coloring a family of circular arcs. *SIAM Journal of Applied Mathematics*, 1975.

[ZA94] Z. Zhang and A. Acampora. A heuristic wavelength assignment algorithm for multihop wdm networks with wavelength routing and wavelength reuse. *Proceedings of IEEE Infocom*, pages 534–543, 1994.

A Appendix: The Constrainedness of Latin Routers

In this section we discuss the constraints Latin Routers impose upon constraint satisfaction in more detail. One of the features of a Latin Router is that the number of wavelengths supported by Latin Router is equal to the size of the Latin Router. It is not known how to interconnect a network of nodes employing Latin Routers of different sizes (as this would bring a mismatch in the number of wavelengths supported in different parts of the network). Therefore, we assume that the sizes of the Latin Router used is the same for every node in the network. The router size K must be larger than the maximum physical degree in the physical topology, i.e., $K > \Delta$. In addition, we notice that if the degree of a node is much less than K, there may be a large number of free ports in the node. Also, since the number of colors supported is equal to the dimension of the router, we must increase the router dimension if we need more colors to color the constraint graph, subject to limitations imposed by the maximum router size. Finally, since one table entry switches an incoming connection on one edge out to another edge and each table entry only switches one color, we know that a demand to switch two colors from one incoming edge to the same outgoing edge can't be satisfied. This imposes restrictions on the constraint graph which are known at routing time; namely two or more lightpaths can't share two adjacent edges in G. Fig. 3 shows this situation: we cannot route two virtual circuits from edge 2 to edge 5 using a Latin Router. This restriction also has an impact on the number of virtual circuits originating and terminating at a node; if there are s source ports and the degree of the node in the physical network is d then only sd connections can originate or terminate at the node. We refer to this as the bypass restriction in an LR-based network.

A.1 Routing for Arbitrary Latin Routers

We notice that the only additional constraint that an Arbitrary Latin Router imposes on establishing connections is due to bypass restrictions; there are no additional coloring constraints. This implies that if we can manage to route without violating the bypass restrictions, we can utilize existing coloring algorithms to assign wavelengths to the lightpaths.

This problem is alleviated by increasing the size of the router and by increasing the fiber multiplicity of edges in the network; two routes can now be established on different fibers but routing connections to the same node of the

network. Fig. 3 shows how increasing the multiplicity alleviates the problem in Latin Routers. If the multiplicity is μ we see that we can route μ^2 connections sharing two adjacent edges; the ease in restrictions extends to routes to and from the source ports as well. With reference to the special case when the underlying network is a single Hamiltonian Cycle we observe that if Latin Routers are to be used, we must increase the multiplicity of edges in the physical network. We observe if the maximum clique in $G(LP)$ has size L then the multiplicity of edges must be $> \sqrt{L}$, resulting in router sizes of at least $\Delta\sqrt{L}$.

A.2 Routing for Shift Latin Routers

The Shift Latin Router is a more restricted form of Latin Routers. We can characterize the form this router must take if we label the available wavelengths as integers from 1 to K. If we examine any four table entries (i,a),(i,b), (j,a) and (j,b) and refer to the color of entry (x,y) as c_{xy} then $c_{ia} - c_{ib} + c_{jb} - c_{ja} \bmod K = 0$. The number of Latin Squares with this property is many times fewer than the number of arbitrary Latin Squares [CPS90]. This restriction turns out to be quite difficult to accommodate in lightpath establishment. Routing and Wavelength Assignment algorithms determine the sequence of edges used by lightpaths and then assign these paths non-conflicting colors. However, in this case we need to make certain that all the color entries obey the above restrictions; in addition to encoding the normal constraints we must also generate and accommodate $O(NK^4)$ additional constraints imposed by the special structure of the Shift Latin Router. To see that this is the correct number of constraints, notice we select 2 entries from each of 2 rows: the total number is $K(K-1)^2$. Since there are N routers we have a total of $O(NK^4)$. The heuristics we designed to account for all of these intricacies were highly complicated and expensive to run. Further, these constraints are highly restrictive on the space of solutions: of the k^4 colorings on 4 variables, a Latin Router constraint on these 4 variables leaves only $k(k-1)^2$ colorings remaining, which is incredibly restrictive compared to the edge constraints on 2 variables, each of which leave $\frac{k^2-k}{k^2}$ colorings.

Using CSP Look-Back Techniques to Solve Exceptionally Hard SAT Instances

Roberto J. Bayardo Jr.
bayardo@cs.utexas.edu
http://www.cs.utexas.edu/users/bayardo/

Robert Schrag
schrag@cs.utexas.edu
http://www.cs.utexas.edu/users/schrag/

Dept. of Computer Sciences and Applied Research Laboratories
University of Texas at Austin
Taylor Hall, Rm. 141 (C0500)
Austin, TX 78712

Abstract. While CNF propositional satisfiability (SAT) is a sub-class of the more general constraint satisfaction problem (CSP), conventional wisdom has it that some well-known CSP look-back techniques -- including backjumping and learning -- are of little use for SAT. We enhance the Tableau SAT algorithm of Crawford and Auton with look-back techniques and evaluate its performance on problems specifically designed to challenge it.

The Random 3-SAT problem space has commonly been used to benchmark SAT algorithms because consistently difficult instances can be found near a region known as the phase transition. We modify Random 3-SAT in two ways which make instances even harder. First, we evaluate problems with structural regularities and find that CSP look-back techniques offer little advantage. Second, we evaluate problems in which a hard unsatisfiable instance of medium size is embedded in a larger instance, and we find the look-back enhancements to be indispensable. Without them, most instances are "exceptionally hard" -- orders of magnitude harder than typical Random 3-SAT instances with the same surface characteristics.

1 Introduction

Given the usual framework of backtrack search for systematic solution of the finite-domained constraint satisfaction problem (CSP), techniques intended to improve efficiency can be divided into two classes: *look-ahead* techniques, which exploit information about the remaining search space, and *look-back* techniques, which exploit information about search which has already taken place. The former class includes variable ordering heuristics, value ordering heuristics, and dynamic consistency enforcement schemes such as forward checking. The latter class includes schemes for backjumping (also known as intelligent backtracking) and learning (also known as nogood or constraint recording). In CSP algorithms, techniques from both classes are popular; for instance, one common combination of techniques (e.g. [1, 12, 28]) is forward checking, conflict-directed backjumping [23], and an ordering heuristic preferring variables with the smallest domains.

CNF propositional satisfiability (SAT) is a specific kind of CSP in which every variable ranges over the values {true, false}. For SAT, the most popular systematic

algorithms are variants of the Davis-Logemann-Loveland modification [8] to the procedure originally defined by Davis and Putnam [7]; hereafter we refer to this procedure as "DP". In CSP terms, the procedure is equivalent to backtrack search with forward checking and an ordering heuristic favoring unit-domained variables. Two effective modern variants of this algorithm are Tableau [5] and POSIT [11], both amounting to DP with highly optimized variable ordering heuristics. Are these SAT algorithms missing anything by not incorporating conflict-directed backjumping or another look-back technique? The standard Random 3-SAT problem space commonly used to benchmark SAT algorithms may not be a good place to look for the answer: Tableau is able to solve millions of instances from Random 3-SAT without any apparent trouble.

Here we challenge Tableau with modifications to Random 3-SAT to make instances more difficult. Random 3-SAT is already a source of consistently hard problem instances -- those in the region of the *phase transition* occurring when the ratio of constraints to variables increases through a critical value [27]. The phase transition separates an *under-constrained* region, where almost all instances are satisfiable and easy, from an *over-constrained* region, where almost all instances are unsatisfiable and relatively easy. We modify Random 3-SAT in two ways: first, we force problems to have structural regularities intended to confuse variable selection heuristics; second, we embed hard unsatisfiable instances into larger instances, making the unsatisfiability of the resulting instances difficult to identify. We also modify Tableau to incorporate some popular look-back techniques, and we evaluate the enhanced algorithm with the new problem spaces. In the case of highly regular problems, we find that look-back techniques offer little or no advantage; for solving our embedded problems, we find them indispensable.

Researchers working with random spaces for other CSPs [1, 17], with other SAT problem spaces [14, 15], or with Random 3-SAT but an algorithm other than Tableau [14, 15], have found rare instances in the under-constrained region so difficult as to render the mean difficulty higher there than in the transition region. Crawford and Auton [5] using Tableau and Random 3-SAT find no such "exceptionally hard" instances (EHIs). Compared to a reference problem space harder than Random 3-SAT ("Variable Regular 3-SAT" -- see Section 4), our embedding procedure generates instances that are "exceptionally hard" for Tableau -- orders of magnitude harder than other instances with the same surface characteristics. Our EHIs could as well result from Random 3-SAT, albeit with low probability. These instances have a clause to variable ratio that places them in the under-constrained region of the reference problem space. Given the difficulty and consistency with which they are generated, we believe they are useful as benchmarks for SAT algorithms.[1]

We find that look-back techniques greatly reduce the incidence of EHIs produced by our embedding procedure. The result is similar to that of Baker [1] who, using a random graph-coloring CSP space, finds no EHIs with respect to a conflict-directed backjumping and learning algorithm. In contrast, some instances produced by our embedding procedure remain difficult even for Tableau with conflict-directed back-

1 Implementations of the problem generators and algorithms defined in this paper are available through the Web page of the first author: http://www.cs.utexas.edu/users/bayardo/.

jumping and learning enhancements. Related work has identified other 3-SAT instances that are difficult for Tableau or other DP variants [16, 22]. The instances among these which we have tested are trivial for Tableau enhanced with CSP look-back techniques (see Section 6).

2 Definitions

SAT involves determining whether a given Boolean expression has a satisfying truth assignment. Any Boolean expression can be transformed to conjunctive-normal form (CNF) which allows a conjunction of clauses $C_1 \wedge C_2 \wedge \ldots \wedge C_m$ where each *clause* C_i is a disjunction of *literals* $l_1 \vee l_2 \vee \ldots \vee l_i$. A literal is either a *variable* x_i or its *negation* $\neg x_i$, $1 \leq i \leq n$. Expressions in conjunctive normal form are easily seen to be instances of the CSP: each variable in the Boolean expression corresponds to a variable in the CSP with a Boolean domain, and each clause of i literals is a constraint disallowing exactly one truth assignment to the i variables mentioned. SAT restricted to conjunctive normal form with exactly k literals per clause is known as k-SAT. A common restriction of SAT that retains its NP-completeness is 3-SAT.

By *problem*, we mean an abstract description such as the definition for CSP, SAT, or 3-SAT which can be instantiated in different concrete ways -- e.g., by enumerating specific variables and constraints. By *instance*, we mean one of these particular instantiations. By *problem space*, we mean a parameterized set of problems, where each parameter represents a dimension of the space. Thus, one point in the larger space of 3-SAT is 3-SAT with exactly 75 variables and 325 clauses. In a *random problem space*, the probability distribution for the occurrence of a particular instance at any point depends on the operation of a non-deterministic procedure given the parameter values for that point as inputs. The procedure for the Random 3-SAT problem space is given below.

RANDOM 3-SAT: Inputs are the number of variables n and the number of clauses m. Three distinct variables are randomly selected out of the pool of n possible variables. Each variable is negated with probability $1/2$. These literals are combined to form a clause. m clauses are created in this manner and conjoined to form the 3-CNF Boolean expression.

For scaling across different problem sizes (different values of n), we use the *constraint ratio* m/n which is expressed in units of clauses per variable. Instances with high median difficulty can be found at the *crossover point*, occurring where half the generated instances are satisfiable. The crossover point may be thought of as the mid-point of the phase transition region. It turns out that the location of the crossover point is fairly stable in constraint ratio terms -- around 4.26 for Random 3-SAT [5].

3 Tableau and Look-Back Enhancements

We use Tableau [5] as our baseline SAT algorithm. Crawford and Auton show Tableau to be very effective at solving instances from Random 3-SAT and at overcoming the incidence of EHIs in the under-constrained region. For a full discussion of the heu-

ristics which lead to its success, please see [5]. We create three look-back-enhanced versions of Tableau: one applying conflict-directed backjumping (CBJ) [23], another CBJ with third-order learning [12], and the last CBJ with unrestricted learning (sometimes referred to as "dependency-directed backtracking" [30]).

As look-back techniques, backjumping and learning are invoked when the algorithm reaches a failure point where at least one variable assignment must be undone before search can progress. Both exploit a set of "culprit" variables whose assignments are determined to be responsible for the failure. The method used to identify the set of culprits is critical in the effectiveness of the techniques. The culprit identification scheme used by Prosser's conflict directed backjumping is widely used [1, 12, 28], requires little overhead [28], and is provably more effective than some of its predecessors [20]. Given a culprit identification scheme, the next issue to be decided is how to exploit the culprits. Pure CBJ simply backs up to the most recent culprit to have been assigned a value without recording the culprit assignments. At the other extreme is unrestricted learning which records every assignment of culprit variables (called a *nogood*). A useful middle-ground is to apply CBJ and to record nogoods only if they are below a certain size. For instance, third-order learning [12] records only those nogoods mentioning three or fewer variables. In the SAT context, this corresponds to recording derived clauses of three or fewer literals.

In the experiments that follow, we concentrate on CBJ and bounded learning enhancements of Tableau. We experiment briefly with unrestricted learning, but find it too expensive on the more difficult instances.

4 Regularity-Inducing 3-SAT Generators

Tableau and other modern SAT algorithms exploit irregularities within the search space to realize inevitable dead-ends as quickly as possible. We were interested in identifying the effects of highly regular instances on Tableau and its enhancements. Various other studies [6, 13, 29, 31] have investigated the effects of increased regularity on SAT and CSP solving, finding that higher regularity increases mean difficulty. While look-back techniques do not significantly improve Tableau's mean performance on the instances below, we discover interesting phase transition properties which we exploit to develop a harder problem generator in the following section.

We first define two new generation procedures based on Random 3-SAT that progressively eliminate certain sources of irregularity. An obvious potential irregularity within a Random 3-SAT instance is that some variables may appear more often than others. The following generation procedure removes this irregularity nearly completely:

VARIABLE-REGULAR 3-SAT: Inputs are the number of variables (n) and the number of clauses (m). The instance is constructed by putting $\lfloor 3(m/n) \rfloor$ occurrences of each variable in a "bag". A random set of unique variables is then added to the bag so that there are exactly $3m$ variables in it. To construct each clause, three distinct variables are removed from the bag. Each variable is negated with probability $1/2$ to form a clause, and clauses are conjoined to form the 3-CNF

expression. If there are only one or two distinct variables remaining within the bag, additional distinct variables are selected randomly from the set of all variables.

Since variables are negated at random in Variable-Regular 3-SAT, a given variable may appear negated more often than not (or vice versa). The next generation procedure removes this source of irregularity nearly completely. This problem space is equivalent to the "doubly balanced" SAT space investigated independently by Dubois and Boufkhad [10], and similar to those defined by Genisson and Sais [13].

LITERAL-REGULAR 3-SAT: Inputs are the number of variables (n) and the number of clauses (m). There are $2n$ possible literals given n variables, so $\lfloor 3m/2n \rfloor$ occurrences of each literal are placed in a bag. A random set of unique literals is then added to the bag so that there are exactly $3m$ literals in it. To construct each clause, three literals on distinct variables are removed from the bag. If there are only 1 or 2 distinct variables mentioned in literals remaining the bag, additional distinct variables are randomly selected from the set of all variables and negated with probability $1/2$.

Data on the location of the phase transition and mean problem difficulty with respect to Tableau for instances generated by the regularity-inducing generators appear in Figure 1. Each point plotted in both graphs results from 500 instances solved by our implementation of Tableau.

While both generators increase regularity, one exhibits a phase transition to the right of Random 3-SAT's, and the other to the left. The first graph in the figure displays the phase transition properties for each procedure when n is fixed at 140. Smith and Dyer [29] find a similar rightward shift with CSPs when decreasing constraint-graph regularity. They point out that it is more difficult to assign a value to a highly constrained variable than to a less constrained one; thus, greater variability in constraint graph degree should lead on average to greater frequency of unsatisfiability for given n and m. This helps to explain why variable-regularity shifts the phase transition to the right from Random 3-SAT. Genisson and Sais [13] reported a similar leftward shift with their literal-regular 3-SAT generator. We believe that literal-regular instances typically require fewer clauses for unsatisfiability because the balance of positive and negative variable occurrences provides more opportunities for resolution, leading to a greater probability of ultimately deriving the empty clause.

Also displayed in Figure 1 are mean problem difficulty at various values of input parameters. Difficulty is represented by the number of branch points encountered by Tableau. *Branch points* are defined as search tree nodes where Tableau does a significant amount of work (i.e. more than just unit propagation) in branching on both possible truth values [5]. As expected, mean difficulty at crossover increased with regularity. Literal-Regular 3-SAT exhibits the highest mean difficulty; at this relatively low value of n, its peak is almost an order of magnitude higher than that of Random 3-SAT.

We repeated the experiments at larger and smaller values of n in order to see how the phase transition and mean difficulty changed, this time averaging over 1000

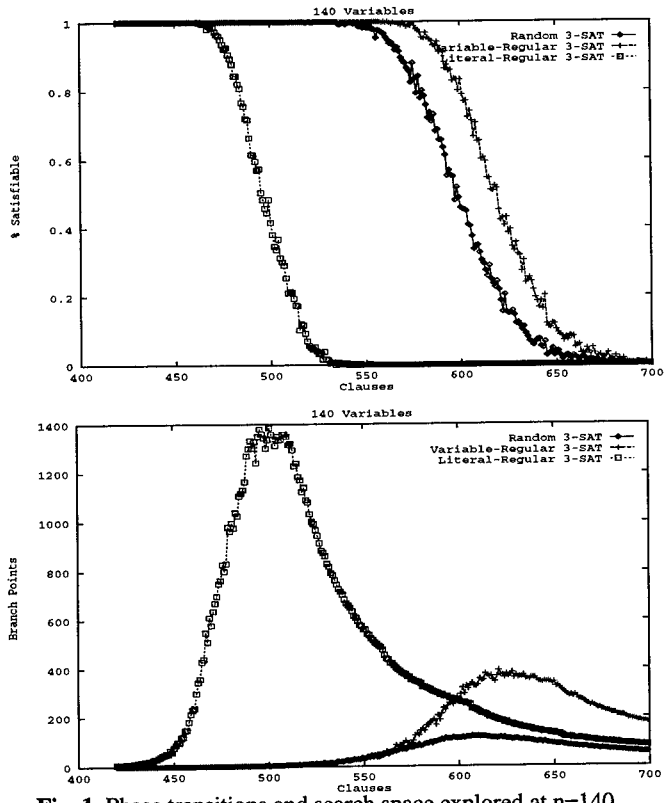


Fig. 1. Phase transitions and search space explored at n=140.

instances per data point. Recall that for Random 3-SAT the location of the crossover point is fairly stable at a constraint ratio near 4.26. The graphs in Figure 2 illustrate the crossover points for the other problem spaces, and the respective difficulty at the crossover point. Crossover point locations for these new generators are also stable in constraint ratio terms. We derived a crossover point constraint ratio of 4.41 for Variable-Regular 3-SAT and 3.54 for Literal-Regular 3-SAT.

Measuring difficulty in branch points explored by Tableau, it is known that difficulty of crossover point problems from Random 3-SAT approximately doubles every time the number of variables is increased by 20 (at least up to $n = 300$) [5]. For Variable-Regular 3-SAT, we see that difficulty increases by a factor of 2.4 with an increment of 20 variables (within the range explored). For Literal-Regular 3-SAT, the difficulty increases with a factor of approximately 2.9.

We added conflict-directed backjumping and third-order learning to Tableau, anticipating that learning (which derives new clauses during search) could create irregularities for the variable ordering heuristics to exploit. We found that neither scheme improved runtime nor reduced search space explored beyond a few percentage points. Tableau, while performing worse on these instances than typical Random 3-SAT instances, has a respectable growth rate when compared to naive DP variants. For instance, Crawford and Auton [4] find that DP using a most-constrained-first variable

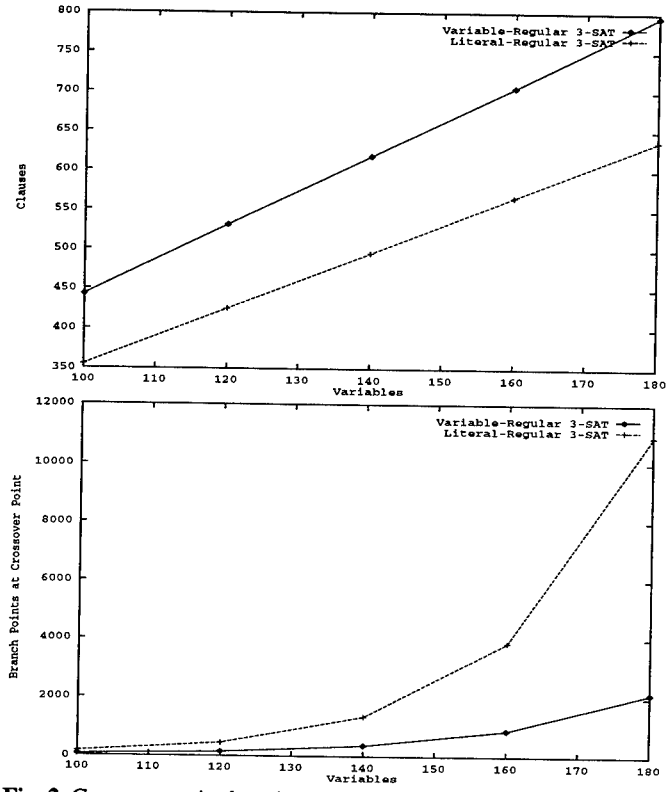


Fig. 2. Crossover point locations and difficulty at the crossover point.

selection heuristic requires over 10,000 branch points for Random 3-SAT problems with as few as 100 variables. After assigning a few variables, we suspect enough irregularities appear in the resulting sub-problems to make Tableau's look-ahead heuristics effective. The irregularities created by learning were not significant in comparison.

5 Manufacturing and Solving Exceptionally Hard Problems

Several researchers [1, 14, 15, 17] have found rare instances in the under-constrained region of various problem spaces so difficult as to render the mean difficulty higher than that of instances from the crossover point. Some of these instances have been found to contain small unsatisfiable sub-problems [15]. In this section, instead of randomly generating instances from the under-constrained region of a particular problem space in search of exceptionally hard instances, we actively generate them by creating under-constrained instances with small unsatisfiable sub-problems.

A simple approach for generating an under-constrained instance containing an unsatisfiable sub-problem is to take the union of the clauses from an under-constrained instance and an unsatisfiable one. However, most instances created using this approach have surface characteristics that render them easy. For example, if the two combined instances consist of disjoint variables, a simple connected components algorithm could

be used to identify each sub-problem for their independent solution. Without such a preprocessing phase, we have found that Tableau can perform poorly on such instances if the variable occurring most frequently appears in the under-constrained instance: Tableau will initially attempt to solve the under-constrained instance, and for each solution, it attempts (and fails) to solve the unsatisfiable one. The addition of conflict-directed backjumping completely remedies this behavior, since it allows each sub-problem to be effectively solved independently in conjunction with the most-constrained first variable ordering heuristic. If the variable names of the combined instances are overlapping, then variables shared between the instances almost always occur most frequently (unless additional steps are taken to prevent this). This causes Tableau to branch initially on those variables, allowing it to determine unsatisfiability without exceptional difficulty.

In this section, we show how the phase transition characteristics of the regular problem spaces can be exploited to conveniently generate under-constrained instances containing well-concealed small unsatisfiable sub-problems. The procedure is used to generate instances whose constraint ratios suggest they should be easily satisfiable with reference to another problem space. Instead, they frequently turn out to be exceptionally hard. We find that look-back enhancements provide a substantial degree of insurance against poor performance, though they fail to eliminate it completely.

In principle, an instance can be "exceptionally hard" only with reference to a given algorithm and a particular problem space. For example, looking at Figure 2b, crossover instances from Literal-Regular 3-SAT would be considered hard with reference to Random 3-SAT. As a convention, we take an EHI to be any instance which requires at least an order of magnitude more work than the mean difficulty of crossover instances in the reference problem space. Here, we use Variable-Regular 3-SAT as our reference space since the generator produces only variable regular instances. In our experiments with Variable-Regular 3-SAT and those with Random 3-SAT, we found no instances at crossover requiring more than 5 times the mean number of branch points, so our EHIs are much harder than any of the observed crossover instances.

The procedure below conceals a problem instance within a larger one. It uses the Variable-Regular 3-SAT procedure to embed the input instance P' with n' variables and m' clauses within a larger randomly generated instance of size n, m.

EMBEDDING PROCEDURE: Inputs are an instance P' of size n', m', and parameters n, m specifying the size of the instance to be generated. The number of occurrences of any variable in P' is required to be no more than $\lfloor 3(m/n) \rfloor$. The variables of P' are first renamed randomly to variables within the set of n variables to appear in the generated instance. Next, a bag is filled with variable occurrences exactly as is done by Variable-Regular 3-SAT with parameters n, m. Then, for every occurrence of a variable appearing in the renamed m' clauses, an occurrence of that variable is removed from the bag. Afterwards, the remaining $m - m'$ clauses are generated as defined by the Variable-Regular instance generator.

If P' is unsatisfiable, then the resulting problem will also be unsatisfiable since it contains the (renamed) clauses of P'. The procedure below uses the technique to create unsatisfiable variable-regular instances.

EXCEPTIONALLY HARD 3-SAT: Inputs are four parameters, n, m, n', m'. We begin by using the Literal-Regular 3-SAT procedure to generate an instance P' of size n', m'. The procedure is invoked until an unsatisfiable instance is produced. Then, clauses are greedily removed at random from P' until there is no single clause that can be removed without rendering the instance satisfiable. At this point, the reduced P' is checked to ensure no variable appears more than $\lfloor 3(m/n) \rfloor$ times. If a variable appears too often, then we start over. Otherwise, we embed P' into a size n, m instance using the previously-described embedding procedure.

The above procedure must find an unsatisfiable instance that can be effectively concealed by way of low variable occurrences. Its strategy for maximizing the probability of concealment is as follows. First, it selects an unsatisfiable instance from Literal-Regular 3-SAT. On average, these require fewer clauses for unsatisfiability than instances from Random 3-SAT and Variable-Regular 3-SAT due to the left-shifted phase transition. The procedure then applies the greedy reduction phase to make the instance even easier to hide (we find that reduction typically removes 20-40% of the clauses from unsatisfiable crossover instances).[1] In the embedding phase, the reduced instance is padded with clauses that suffice to make the result a possible output of Variable-Regular 3-SAT. The right-shifted phase transition of Variable-Regular 3-SAT allows padding with more clauses than Literal-Regular or Random 3-SAT for producing what superficially appear to be under-constrained instances.

The following tables report the performance of Tableau and its enhancements on instances from Exceptionally Hard 3-SAT. Tableau is denoted by "Tab", Tableau with conflict-directed backjumping "Tab+CBJ" and Tableau with conflict-directed backjumping and third-order learning "Tab+CBJ+lrn". We continue to report problem difficulty in terms of branch points because overhead of these additional enhancements was small. The overhead of conflict-directed backjumping alone was negligible (less than 3%) since Tableau expends most of its effort selecting branch variables. Learning derives new clauses which had to be tested even by the branch-variable selection procedure. Third-order learning, however, typically recorded only a few clauses, keeping overhead well within 20% even on the hardest problems. At the end of this section, we discuss preliminary experiments with unrestricted learning.

We used a constraint ratio of 3.5 or less for determining m for the various values of n since it is well within the under-constrained region of Variable-Regular 3-

1 This phase is NP-hard, though it presents no practical problem as long as we choose to embed small instances. For greater efficiency, we could embed an instance selected from a set of pre-reduced instances instead of generating and reducing a new instance for embedding with each invocation of the Exceptionally Hard 3-SAT procedure.

SAT. For n', m', we used the formula $n = 3.5m + 5$ to produce hard Literal-Regular 3-SAT instances for embedding.

To facilitate experiments with large numbers of instances (10,000 per value of n), we had the algorithm halt and report failure beyond a threshold of branch points an order of magnitude or more than the mean required for Variable-Regular crossover instances with the same number of variables (10,000 branch points for $n = 75$ and $n = 140$, 50,000 branch points for $n = 200$). We also report the mean difficulty of a smaller number (200) of problems on which the failure mechanism was turned off.

Table 1. Exceptionally hard problem statistics for n=75, m=225, n'=10, m'=40

Algorithm	Mean Difficulty of Solved Instances (branch points)	Failure Rate (%) [> 10000]	Mean Difficulty of 200 instances (branch points)
Tableau	2,672	66.5	87,993
Tab + CBJ	113	0	113
Tab + CBJ + lrn	3	0	3

Table 1 shows that unenhanced Tableau performs poorly even on very small instances from Exceptionally Hard 3-SAT. For Variable -Regular crossover instances with $n = 75$, Tableau requires a mere 20 branch points on average. Performance on 75 variable Exceptionally Hard 3-SAT instances is over 3 orders of magnitude worse. While Tableau with backjumping is effective at solving these problems, the addition of learning makes them trivial. For problems barely beyond $n = 75$, we found that Tableau's failure rate rapidly approached 100%.

Table 2. Exceptionally hard problem statistics for n=140, m=490, n'=20, m'=75

Algorithm	Mean Difficulty of Solved Instances (branch points)	Failure Rate (%) [> 10000]	Mean Difficulty of 200 instances (branch points)
Tab + CBJ	1,904	57.2	55,122
Tab + CBJ + lrn	12	0	12

The next data point (Table 2) illustrates that Tableau with conflict-directed backjumping alone begins to go awry at large enough problems. Others [15, 28] have also found that backjumping alone fails to eliminate occurrence of exceptionally hard instances, though in the context of sparse CSP or much larger SAT instances.

Table 3. Exceptionally hard problem statistics for n=200, m=700, n'=30, m'=110

Algorithm	Mean Difficulty of Solved Instances (branch points)	Failure Rate (%) [> 50000]
Tab + CBJ + lrn	78	0

Table 3 shows that the learning algorithm remains completely effective even at larger problem sizes when the embedded instance is small. The failure cutoff is increased to 50,000 branch points for these larger problems since the average Variable-Regular crossover instance of 200 variables requires approximately 5000 branch points. It appears, however, that if the embedded and actual instances are large enough, we can elicit failure even in the learning algorithm (Table 4 below). The failure rate remains relatively low at the data points we investigated. Further experimentation is required in order to determine if the failure rate approaches 100% with larger problems (as is clearly the case with the other two algorithms).

Table 4. Exceptionally hard problem statistics for n=200, m=700, n'=50, m'=180

Algorithm	Mean Difficulty of Solved Instances (branch points)	Failure Rate (%) [> 50000]
Tab + CBJ + lrn	3,702	3.4945

We have performed some experiments with unrestricted learning, but have been unable to draw any solid conclusions about its effects other than that its overhead becomes unacceptably high on sufficiently large and dense SAT instances. For example, on difficult instances from Exceptionally Hard 3-SAT, the third-order learning algorithm was 30 times faster in terms of branch points searched per second. On 50 instances from the <200, 700, 50, 180> point, the hardest instance found for the unrestricted learning algorithm required 22,405 branch points. This translates to well over 10 times the CPU time that the third-order learning algorithm required to reach failure. To account for this overhead, we feel the definition of an exceptionally hard instance for unrestricted learning algorithms should take CPU time into account. By such a definition, our implementation of unrestricted learning fails to eliminate them completely.

Baker [1] and Frost and Dechter [12] found that the overhead of their unrestricted learning algorithms was not excessive. We believe the our different findings are primarily due to the constraint density of SAT compared to binary CSP. 3-SAT instances require many constraints (clauses), since each excludes only a small fraction of potential truth assignments. Further, each constraint is defined on three variables instead of two. As a result, the set of variables responsible for each failure when solving a SAT instance is often large. Another potential cause for our different findings is that some instances produced by Exceptionally Hard 3-SAT required extensive search even of the unrestricted learning algorithm. Baker's instances were easy for his unrestricted learning algorithm, so it never had the opportunity to derive an excessive number of constraints. We believe that any SAT algorithm applying learning on instances like ours will require either limited order learning as employed here, time or relevance

limits on derived clauses [2,16], or some method for efficiently producing smaller culprit sets than conflict-directed backjumping (e.g. possibly along the lines of Dechter's deep learning schemes [9]).

6 Related and Future Work

Ginsberg & McAllester [16] evaluate a CSP algorithm they call "partial-order dynamic backtracking" on a 3-SAT problem space with restricted structure. This problem space creates an instance by arranging the variables on a two-dimensional grid and creating clauses that contain variables forming a triangle with two sides of unit Euclidean length. The algorithm incorporates look-ahead techniques not specifically geared for SAT and look-back techniques similar to CBJ and learning. They find it immune to pathologies encountered by Tableau on these instances. The hardest problems used in their evaluation were crossover instances from their new SAT problem space with 625 variables. We found these instances trivial for Tableau with CBJ and third-order learning, requiring on average 6 and at maximum 28 branch points on the 10,000 instances we attempted. We found that CBJ-enhanced Tableau has occasional difficulties on these same instances, requiring less than 22 branches on over half the instances, but over 50,000 branches on 2.51% of them. This suggests these instances may have properties similar to (though not as pronounced as) those from Exceptionally-Hard 3-SAT.

Mazure et al. [22] recently developed a look-ahead technique for DP based on GSAT [26]. The technique selects branch variables by counting the number of times a variable occurs in clauses falsified by assignments made during a GSAT search on the current sub-problem. The intent is to focus search on variables that may be part of an inconsistent kernel. They evaluate their technique on several DIMACS benchmark instances[1] which were infeasible for DP. Some of the "AIM" instances from this suite that are difficult for DP are trivial for their algorithm. We found that all of the largest (200 variable) "AIM" instances were trivial for Tableau with CBJ and learning, the most difficult requiring 27 branch points. Some of these instances were difficult for Tableau without learning but with CBJ. We have not yet explored the effect of their technique on our problem spaces.

Lee and Plaisted [21] enhance DP with a backjumping scheme similar to (but not as powerful as) CBJ which they use as a subroutine in a first-order theorem proving system. Gent and Walsh [15] experiment with an implementation of this algorithm[2] on a SAT problem space they call "Constant Probability" and find that the backjumping scheme reduces the incidence of EHIs in the under-constrained region, but fails to eliminate them at large enough problem sizes. Chvatal [3] also applies look-back to SAT through "resolution search" -- a DP variant with what appears to be a novel learning scheme.

---

1 These instances are available through anonymous FTP at ftp:dimacs.rutgers.edu within directory pub/challenge/sat/benchmarks/cnf.

2 This is the "intelligent backtracking" algorithm whose implementation they credit to Mark Stickel.

Others have developed procedures intended to generate hard random problems. Iwama et al. [19] and Rauzy [24] independently generate 3-SAT instances which are known in advance to be satisfiable or unsatisfiable. The authors do not directly compare the difficulty of these problems to problems from Random 3-SAT. Genisson and Sais [13] investigate 3-SAT generators with controlled distributions of literals -- much like our Literal-Regular 3-SAT -- and also find a left-shifted phase transition and increased difficulty. Dubois and Boufkhad also investigate literal-regular instances they call "doubly balanced" in a forthcoming paper [10]. Culberson et al. [6] and Vlasie [31] describe generators for graph coloring CSPs in which graphs are endowed with a variety of structural properties intended to make coloring difficult. Their empirical results cannot be compared to ours directly, but they also find that increased regularity increases mean problem difficulty. Whether look-back techniques are important for these problem spaces remains to be evaluated empirically.

Crawford and Auton [5] were unable to find any EHIs for Tableau in the under-constrained region of Random 3-SAT. Our results with Exceptionally Hard 3-SAT show that they do exist, albeit in Random 3-SAT with low probability. Gent and Walsh report that an improved DVO heuristic [14] and improved constraint propagation method [15] (both look-ahead techniques) fail to eliminate EHIs for their 3-SAT generators and DP implementation. Baker [1], working with graph coloring CSPs, finds no problems to be extremely hard for a conflict-directed backjumping and unrestricted learning algorithm. Selman and Kirkpatrick [25], using an earlier version of Tableau [4] and Random 3-SAT, investigate the incidence of EHIs when a given instance is subject to an equivalence-preserving random renaming of its variables. They report observing the same incidence of EHIs whether running Tableau on 5000 different permutations of the same 20 source instances, or whether sampling 5000 instances independently. This suggests these EHIs arise on account of unfortunate variable orderings. We have not yet investigated the effects of random renamings on our instances. The fact that they occur with near certainty for unenhanced Tableau when generating sufficiently large problems suggests that the source of their difficulty may be qualitatively different.

The fact that our generated EHIs are unsatisfiable means that sound but incomplete algorithms such as GSAT [26] cannot be used to address them. We are considering a related problem generator of satisfiable instances for the purpose of benchmarking sound-and-incomplete SAT algorithms. We embed hard satisfiable instances instead of unsatisfiable ones by removing one additional clause immediately following the reduction phase of Exceptionally Hard 3-SAT. Limited experimentation has shown similar (though not as pronounced) difficulties result for unenhanced Tableau, even though the resulting instances are almost always satisfiable.

7 Conclusions

We have shown that, contrary to the conventional wisdom, CSP look-back techniques are useful for SAT. We were able to significantly reduce the incidence of exceptionally hard instances (EHIs) encountered by the Tableau SAT algorithm after enhancing it with look-back techniques. We devised a new generator which usually

succeeds in producing instances which are exceptionally hard for unenhanced Tableau when compared to the difficulty of other common benchmark instances. Relatively few of these instances remained exceptionally hard for the look-back-enhanced versions of Tableau.

It may be that look-back techniques are essential to solving these instances efficiently, though we encourage experimentation with non-look-back algorithms which might refute this hypothesis. At the same time, we do not wish to minimize the significance of look-ahead techniques. Tableau's variable selection heuristics make it competitive with the best current SAT algorithms, and we would expect to encounter many more exceptionally hard instances without them. We do not believe that any generally effective SAT algorithm can totally eradicate the incidence of EHIs. We do believe that selected look-ahead and look-back techniques with modest overhead can provide some valuable insurance against them.

Acknowledgments

We wish to thank Jimi Crawford for the Tableau executable and assistance in replicating its behavior. We also thank Richard Genisson, Ian Gent, Daniel Miranker, David Mitchell, Lakhdar Sais, Mark Stickel and Toby Walsh for their comments and assistance.

References

1. A. B. Baker, *Intelligent Backtracking on Constraint Satisfaction Problems: Experimental and Theoretical Results*. Ph.D. Dissertation, Dept. of Computer and Information Science, University of Oregon, March 1995.
2. R. J. Bayardo, A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem, In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, 1996.
3. V. Chvatal, Resolution search. *Discrete Applied Math*, to appear, 1996.
4. J. M. Crawford and L. D. Auton, Experimental results on the crossover point in satisfiability problems. *AAAI-93,* 21-27, 1993.
5. J. M. Crawford and L. D. Auton, Experimental results on the crossover point in random 3SAT. *Artificial Intelligence* 81(1-2), 31-57, 1996.
6. J. Culberson, A. Beacham and D. Papp, Hiding our colors, Workshop on Studying and Solving Really Hard Problems, International Conference on Principles and Practice of Constraint Programming, 1995.
7. M. Davis and H. Putnam, A computing procedure for quantification theory, *JACM* 7, 201-215, 1960.
8. M. Davis, G. Logemann, D. Loveland, A machine program for theorem proving, *CACM* 5, 394-397, 1962.
9. R. Dechter, Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence* 41(3):273-312, 1990.
10. O. Dubois and Y. Boufkhad, From very hard doubly balanced SAT formulae to easy unbalanced SAT formulae, variations of the satisfiability threshold, *Proceedings of the DIMACS workshop on the Satisfiability Problem: Theory and Applications*, Ding-Zhu Du, Jun Gu, and Panos Pardalos, eds., March 1996.

11. J. W. Freeman, *Improvements to Propositional Satisfiability Search Algorithms*. Ph.D. Dissertation, Dept. of Computer and Information Science, University of Pennsylvania, 1995.
12. D. Frost and R. Dechter, Dead-end driven learning. In *Proceedings of AAAI-94*, 294-300, 1994.
13. R. Genisson and L. Sais, Some ideas on random generation of k-SAT instances, AAAI-94 Workshop on Experimental Evaluation of Reasoning and Search Methods, 1994.
14. I. Gent and T. Walsh, Easy problems are sometimes hard, *Artificial Intelligence* 70, 335-345, 1994.
15. I. Gent and T. Walsh, The satisfiability constraint gap. *Artificial Intelligence* 81(1-2), 59-80, 1996.
16. M. Ginsberg and D. McAllester, GSAT and dynamic backtracking, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference*, 226-237, 1994.
17. T. Hogg and C. P. Williams, The hardest constraint problems: A double phase transition. *Artificial Intelligence* 69, 359-377, 1994.
18. T. Hogg, B. A. Huberman, C. Williams (eds.), *Artificial Intelligence* 81(1-2), Special issue on Frontiers in Problem Solving: Phase Transitions and Complexity, Elsevier, March 1996.
19. K. Iwama, H. Abeta, E. Miyano, Random generation of satisfiable and unsatisfiable CNF predicates, in *Algorithms, Software, Architecture, Information Processing 92* volume 1, 322-328, 1992.
20. G. Kondrak and P. van Beek, A theoretical evaluation of selected backtracking algorithms, *IJCAI-95*, 541-547, 1995.
21. S.-J. Lee and D. A. Plaisted, Eliminating duplication with the hyper-linking strategy, *Journal of Automated Reasoning* 9, 25-42, 1992.
22. B. Mazure, L. Sais and E. Gregoire, Detecting logical inconsistencies, In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics*, 116-121, 1996.
23. P. Prosser, Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3), 268-299, 1993.
24. A. Rauzy, *On the Random Generation of 3-SAT Instances*, Technical Report 1060-95, Laboratoire Bordelais de Recherche en Informatique, Universite Bordeaux I, 1995.
25. B. Selman and S. Kirkpatrick, Critical behavior in the satisfiability of random Boolean expressions II: Computational cost of systematic search, *Artificial Intelligence* 81(1-2), 273-295, 1996.
26. B. Selman, H. Levesque, and D. Mitchell, A new method for solving hard satisfiability problems, *AAAI-92*, 440-446, 1992.
27. B. Selman, D. Mitchell, and H. Levesque, Generating hard satisfiability problems, *Artificial Intelligence* 81(1-2), 17-29, 1996.
28. B. A. Smith and S. A. Grant, Sparse constraint graphs and exceptionally hard problems, *IJCAI-95*, 646-651, 1995.
29. B. M. Smith and M. E. Dyer, Locating the phase transition in binary constraint satisfaction problems, *Artificial Intelligence* 81(1-2), 155-181, 1996.
30. R. M. Stallman and G. J. Sussman, Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence* 9, 135-196, 1977.
31. R. D. Vlasie, Systematic generation of very hard cases for graph 3-colorability, In *Proceedings of ICTAI-95*, IEEE Press, 1995.

MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems

Christian Bessière[1] and Jean-Charles Régin[2]

[1] LIRMM (UMR 5506 CNRS), 161 rue Ada, 34392 Montpellier cedex 5, France
e-mail: bessiere@lirmm.fr
[2] ILOG S.A., 9 rue de Verdun BP 85, 94253 Gentilly Cedex, France
e-mail: regin@ilog.fr

Abstract. In the last twenty years, many algorithms and heuristics were developed to find solutions in constraint networks. Their number increased to such an extent that it quickly became necessary to compare their performances in order to propose a small number of "good" methods. These comparisons often led us to consider FC or FC-CBJ associated with a "minimum domain" variable ordering heuristic as the best techniques to solve a wide variety of constraint networks.
In this paper, we first try to convince once and for all the CSP community that MAC is not only more efficient than FC to solve large practical problems, but it is also really more efficient than FC on hard and large random problems. Afterwards, we introduce an original and efficient way to combine variable ordering heuristics. Finally, we conjecture that when a good variable ordering heuristic is used, CBJ becomes an expensive gadget which almost always slows down the search, even if it saves a few constraint checks.

1 Introduction

Constraint satisfaction problems (*CSPs*) occur widely in artificial intelligence. They involve finding *values* for problem *variables* subject to *constraints* on which combinations are acceptable. For simplicity we restrict our attention here to *binary* CSPs, where the constraints involve two variables.

Binary constraints are binary relations. If a variable i has a *domain* of potential values D_i and a variable j has a domain of potential values D_j, the constraint on i and j, R_{ij}, is a subset of the Cartesian product of D_i and D_j. If the pair of values a for i and b for j is acceptable to the constraint R_{ij} between i and j, we will call the values *consistent* (with respect to R_{ij}). Asking whether a pair of values is consistent is called a *constraint check*.

The entity involving the variables, the domains, and the constraints, is called *constraint network*. Any constraint network can be associated to a *constraint graph* in which the nodes are the variables of the network, and an edge links a pair of nodes if and only if there is a constraint on the corresponding variables. $\Gamma(i)$ represents the set of nodes sharing an edge with the node i.

In the last twenty years, many algorithms and heuristics were developed to find solutions in constraint networks [16], [21], [22]. Their number had increased to such an extent that it quickly became necessary to compare their performances in order to designate some of them as being the best methods. In the recent years, many authors worked

in this way [23], [5], [10], [1]. The general inference drawn from these works is that forward checking [16] (denoted by FC) or FC-CBJ (CBJ: conflict directed backjumping [22]) associated with a "minimum domain" variable ordering heuristic is the most efficient strategy to solve CSPs. (It has been so repeated that hard problems are often considered as those that FC-CBJ cannot solve [24]). This can be considered as a surprising conclusion when we know that the constraint programming community uses full arc consistency at each step of the search algorithms [32] and claims that it is the only practicable way to solve large real world problems in reasonable time [26]. The contradiction comes from the fact that in the CSP community, the sample problems used for the comparisons were often very particular (especially small or easy [16], [21], [23]), and the way the algorithms were compared was sometimes incomplete (no procedure maintaining full arc consistency involved in the comparisons [5], [10]) or unsatisfactory [1]. But, this apparent contradiction did not give rise to other questions or comments than Sabin and Freuder's paper [28], in which it was pointed out that a procedure Maintaining Arc Consistency during the search (*MAC*) could outperform FC on random problems around the cross-over point.

In this paper, we try to convince the reader that MAC is not only more efficient than FC to solve large practical problems, but it is also really more efficient than FC on hard and large random problems. Afterwards, we introduce an original[3] way to really *combine* different variable ordering heuristics (instead of just using a secondary heuristic to break ties in the main one) and show its efficiency. Finally, we conjecture that when a good variable ordering heuristic is used, CBJ becomes an expensive gadget which almost always slows down the search, even if it saves a few constraint checks.

The paper is organized as follows. Section 2 contains an overview of the main previous works on algorithms and heuristics to solve CSPs. Section 3 describes the instance generator and the experimental method used in the rest of the paper. We show the good behavior of MAC in Sect. 4. The new way to combine variable ordering heuristics is presented in Sect. 5. Section 6 shows that CBJ loses its power when high level look-ahead is performed during the search. Finally, Sect. 7 summarizes the work presented in this paper.

2 Previous Work

It has been noted by several authors (e.g. [15]) that there are four choices to be made when searching solutions in constraint networks: what level of filtering to do, which variable to instantiate next, what value to use as the instantiation, what kind of look-back scheme to adopt.

In fact, a wide part of the CSP community has been working for twenty years to answer these questions.

To the question of the level of filtering to perform before instantiating a variable, many papers concluded that *forward-checking* (FC) is the good compromise between the pruning effect and the amount of overhead involved ([16], [20], [19], [1]).

[3] This approach is original in the sense that it has never been published before. The only presentation we know of such an approach is given in [2].

It has been shown for a long time that in constraint networks, the order in which the variables are instantiated strongly affects the size of the search space explored by backtracking algorithms. In 1980, Haralick and Elliot already presented the "fail first principle" as a fundamental idea [16]. Following this, a variety of static variable ordering heuristics (*SVO*) were proposed to order the variables such that the most constrained variables are chosen first (thus respecting the Haralick and Elliot's principle). They calculate once and for all an order, valid during all the tree search, according to which variables will be instantiated. They are usually based on the structure of the constraint graph. The *minimum width* ordering (`minw`) is an order which minimizes the width of the constraint graph [9]. The *maximum degree* heuristic (`deg`) orders the variables by decreasing number of neighbors in the constraint graph [5]. The *maximum cardinality* ordering (`card`) selects the first variable arbitrarily, then, at each stage, selects the variable that is connected to the largest set of already selected variables [5]. The heuristic proposed by Haralick and Elliot to illustrate their principle was a dynamic variable ordering heuristic (*DVO*[4]). They proposed the *minimum domain* (`dom`) heuristic, which selects as the next variable to be instantiated a variable that has a minimal number of remaining values in its domain. It is a dynamic heuristic in the sense that the order in which variables are instantiated can vary from branch to branch in the search tree. Papers discussing variable ordering heuristics quickly found that DVO is generally better than SVO. More precisely, `dom` has been considered as the best variable ordering heuristic ([27], [15], [5]).

The question of the choice of the value to use as an instantiation of the selected variable did not catch as much researchers' attention as variable ordering. It has been explored in [15] or [6], but without producing a simple generic method proven efficient and usable in any constraint network. Even the *promise* selection criterion of Geelen [14] did not attract FC users.

The question of the kind of look-back scheme to adopt had remained an open question for a long time. Different approaches had been proposed, but none had been elected as the best one (e.g. *learning* [4], *backjumping* [13], *backmarking* [12], etc.). This state of things seems to have finished with the paper of Prosser [22], which presented *conflict-directed backjumping* (CBJ). Indeed, Prosser showed in [23] that the hybrid algorithm FC-CBJ is the most efficient algorithm (among many hybrid algorithms) to find solutions in various instances of the zebra problem.

That's why, for a few years, FC-CBJ associated with the `dom` DVO (denoted by FC-CBJ-`dom`) has been considered as the most efficient technique to solve CSPs (naturally following the FC domination of the eighties). Moreover, the numerous papers studying "really hard problems" ([24], [30], [7]) often take the implicit definition: "an hard problem is a problem hard to solve with FC-CBJ-`dom`".

Recent Work. Recently, some authors, not satisfied at all by the conclusion of the story of search algorithms in CSPs, tried to improve this winner. This leads to the paper of Frost and Dechter [11], which reveals two important ways to overcome the classical FC-CBJ-`dom` algorithm.

[4] The origin of the name DVO is in [10] to denote what we will call here `dom`. We use DVO in its general meaning, as it is proposed in [1].

First, the dom DVO is not as perfect as it seems. When several domains have the same minimal size, the next variable to be instantiated is arbitrarily selected. When the constraint graph is sparse, many useful information on its structure is lost by dom, which does not deal with the constraint graph. In [11], a solution to these shortcomings is proposed by using the dom+deg DVO: it consists of the dom DVO in which ties are broken[5] by choosing the variable with the highest degree in the constraint graph. Frost and Dechter underlined that "this scheme gives substantially better performance than picking one of the tying variables at random".

Second, in FC-CBJ-dom, once a variable is selected for instantiation, values are picked from the domain in an arbitrary fixed order (usually values are arbitrarily assigned a sequence number and are selected according to this sequence). In [11], Frost and Dechter presented various domain value ordering heuristics (*LVO* for look-ahead value ordering) and experimentally showed that the *min-conflicts*[6] (mc) LVO is the one which improves the most the efficiency of FC-CBJ-dom+deg (denoted by FC-CBJ-dom+deg-mc).

Another, quite different way to improve search by reordering values (or variables and values) after a dead-end has been presented in [17]. Its features making it especially suitable to solve real world problems, we do not discuss it here.

3 A Few Words About the Experiments

Before starting the experimental comparisons between different algorithms, we say a few words about the experimental method we chose.

When we want to work on random problems, the first step is to choose an instance generator. The characteristics of the generated problems will depend on the generator used to create them. The CSP literature has presented several generators, always involving four parameters: N the number of variables, D the common size of all the initial domains, and two other parameters concerning the density of the constraint graph and the tightness of the chosen constraints. Early generators often used a probability p_1 that a constraint exists between two variables, and a probability p_2 that a value pair is forbidden in a given constraint. The number of different networks that could be generated with the same four parameters $\langle N, D, p_1, p_2 \rangle$ was really huge. Networks with quite different features (e.g. a network with a complete constraint graph and one with only one constraint) could be generated with the same set of parameters. One of the consequences of this fact was that a very large number of instances must be solved to predict the behavior of an algorithm with a good statistical validity.

Hence, a new generation of instance generators appeared (beginning with [18]), which replaced the probability p_1 to have a constraint between two variables by a fixed number C of constraints [24]. In the same way, p_2 can be replaced by a number T of forbidden value pairs [11]. In [30], p_1 and p_2 are still used, but they represent "proportions"

[5] The idea of breaking ties in SVOs and DVOs had been previously proposed in [33].

[6] *min-conflicts* considers each value in the domain of the selected variable and associates with it the number of values in domains of future variables with which it is not compatible. The values are then affected to the selected variable in increasing order of this count. This is in fact the first LVO presented in [14, page 32].

and not probabilities (i.e. if N=20 and p_1=0.1, the number of generated constraints is exactly $0.1 * (20 * 19)/2 = 19$). This new method generates more homogeneous networks and then, it is not necessary to solve a huge number of networks for each set of parameters. Nevertheless, a particular care must be taken in order to generate networks with a uniform distribution. Specifically, the distribution must be as follows: out of all possible sets of C variable pairs choose any particular set with uniform probability, and for each constrained pair out of all possible sets of T value pairs choose any particular set with uniform probability[7]. We need an algorithm that generates uniform random permutations of p elements selected among k elements without repetition. Essentially, this is just choosing which of the k elements will be the first, which of the remaining $k-1$ elements will be the second, and so forth.

When we want to perform experiments on randomly generated networks, and when the instance generator has already been chosen, a second step is to select the sets of parameters that will be used to illustrate the behavior of the algorithms tested. Each set of parameters $\langle N, D, C, T \rangle$ determines the type of the networks generated: N variables each having a domain of size D, C constraints out of the $N * (N-1)/2$ possible, and T forbidden value pairs in each constraint among the $D * D$ possible. In this paper, we did not want to make a complete study of which sets of parameters to use to illustrate our claims. Thus, we decided to use sets of parameters already presented in the literature, and quite well-known. We chose the problems presented in [11] (some of them were already used in [10]) and some of the most famous experiments used by Smith and Grant ([30], [31], [29]). In certain experiments, we propose some variations in the parameters (for example, increasing domain size to show the behavior of the algorithms on networks with larger domains). But, when we vary the density (C) or the domain size (D), we want to keep the networks generated as close as possible to the *cross-over point* (set of parameters for which approximately 50% of the problems are satisfiable and 50% are not). So, T is moved in order to stay at the value "T_{co}" which produces 50% satisfiable problems and 50% unsatisfiable. When for given values of N, D and C no value of T (which is an integer) produces exactly 50% satisfiable problems we always take as T_{co} the smallest value for which the number of unsatisfiable problems is greater than 50%. These variations of the distance between T_{co} and the effective cross–over point explain the serrated look of some of the curves reported below. The size of the problems tested in such cases is often rather small, because each point of the curves given (see Fig. 2, 3, 5) requires to solve a large number of networks just to find the right value T_{co}.

In the following sections we report different kinds of measures of performances for the algorithms tested. First, we often present what we call "number of constraint checks". The classical "number of constraint checks" measure is well-adapted for algorithms like FC, but presents some problems when used with MAC, which maintains lists where some of the past constraint checks are recorded. Hence, for MAC, what we name "number of constraint checks" is in fact the number of classical constraint checks plus the number of list checks it performs during the search. The second measure we use is cpu

[7] Prosser's generator [24] does not choose all the possible sets of C constraints with a uniform probability. Frost and Dechter's generator, while being better than Prosser's one, is not completely uniform [8]. Although it is not extensively described, Smith's generator seems to be uniform [29] (while Smith and Grant's one is not [30]).

time, and the third one is the number of backtracks performed, i.e. the number of times the algorithm goes backwards in the search tree.

In all the tables below, we generated and solved 100 instances for each tested Frost and Dechter's set of parameters. In all the figures (curves), we limited this number to 50 instances for each tested value of the varying parameter.

We always report mean performances on the number of instances solved for a set of parameters. Indeed, we think that reporting the median cost is questionable when the set of parameters is near the cross-over point: unsatisfiable instances are generally harder to solve than satisfiable ones, so the median will appear in a region where few problems fall into, involving a low representativity of this measure. In the extreme case, we can imagine a set of parameters for which 50 problems are found satisfiable in 1 second and 50 are found unsatisfiable in 10 seconds: what is the median cost of this experiment?

LVOs being outside the scope of the present paper, we just checked that mc was a significant improvement in our experiments compared to the versions of the algorithms written without LVO. Hence, in the results presented in the next sections, mc has always been used, even if on some instances the *promise* LVO of Geelen can have a slight more interesting behavior than mc. However, after a very rough comparison, we could not select a winner.

Finally, we want to point out that the programs used to perform the experiments of this paper are available via the ftp site `ftp.lirmm.fr`.

4 MAC is Better than FC-CBJ

We said in Sect. 2 that FC-CBJ is considered as the best algorithm to find solutions in constraint networks. In fact, in the papers that have compared algorithms with different levels of filtering during search and that have concluded that FC performs the right amount of filtering it is often specified that this claim is stated with respect to the tested problems [16], [20], [23]. The tested problems were often the n-queens, very small random problems not necessarily chosen in the phase transition, or the zebra problem. Therefore, we can conclude that on very easy or very small problems FC is probably the algorithm which performs the right amount of filtering (pure look-back algorithms are probably definitively overcome [23], [1]).

But, Dechter and Meiri already said that "it is conceivable that on larger, more difficult instances, intensive preprocessing algorithms may actually pay off" [5]. A first confirmation appeared in the paper of Sabin and Freuder [28], in which they showed that MAC can outperform FC on hard instances of CSPs. The good performances of MAC on large radio link frequency assignment problems (where FC was thrashing) provide another confirmation [3].

Recently, Smith agreed that "exceptionally hard problems ought more properly to be called problems which the particular search algorithm we are using finds exceptionally hard". This led her and Grant to study the behavior of MAC on problems found exceptionally hard with FC-dom [31]. Their conclusion is that "in most cases, the MAC algorithm can show that the problem is arc inconsistent, and so detects that it is insoluble without searching it".

Finally, [1] is the only paper which clearly gives the advantage to FC-CBJ against algorithms performing arc consistency at each node of the search tree after an experimentation on non-easy problems. But, after discussion with Bacchus, it appears that in his paper, the algorithm that performs arc consistency at each stage of the search uses a kind of AC-0 algorithm, i.e. an AC-1 algorithm which does not take care of the structure of the constraint graph, checking all the variable pairs, as if the network was always a complete graph. So, we cannot take these results into account.

We showed in Sect. 2 that the behavior of FC-CBJ can be improved by using a DVO proposed by Frost and Dechter, dom+deg, and by using a good LVO as mc. In this section, we will show that, even associated to the dom+deg DVO and the mc LVO, FC-CBJ can no longer be considered as the best algorithm to solve CSPs. We will experimentally show that FC has a too weak pruning effect to be the most efficient on relatively hard problems. A search procedure as MAC, with a more intensive filtering mechanism, is more efficient to find solutions on hard and large problems, in which the overhead due to arc consistency is outweighed by its gain.

The experiments of this section are limited to the comparison of FC-CBJ-dom+deg-mc and MAC-dom+deg-mc. FC-CBJ-dom+deg-mc is the algorithm stated to be the best in Sect. 2. MAC-dom+deg-mc is here a classical MAC procedure [28] in which the arc consistency algorithm used is AC-7 [3]. The DVO and the LVO used are the same in the two algorithms.

Table 1. FC-CBJ-dom+deg-mc and MAC-dom+deg-mc performances on problems generated with Frost and Dechter's sets of parameters [11]. "arc-inc" in the backtrack ratio column means that all the problems generated for a given set of parameters were arc-inconsistent, implying an infinite ratio (MAC detects arc-inconsistency without any backtrack).

	Parameters	#constraint checks			cpu seconds			#backtracks
	$N, D, C, T/D * D$	FC-CBJ	MAC	ratio	FC-CBJ	MAC	ratio	ratio
#1	35,6,501,4/36	506,265	330,717	1.53	6.83	2.66	2.56	7.45
#2	35,9,178,27/81	248,414	156,131	1.59	3.26	1.00	3.25	14.29
#3	50,6,325,8/36	412,505	152,197	2.71	5.81	1.29	4.50	17.35
#4	50,20,95,300/400	565,330	273,537	2.07	7.11	1.62	4.39	37.02
#5	100,12,120,110/144	243,766	15,709	15.52	3.79	0.14	25.99	870.28
#6	125,3,929,1/9	271,557	44,862	6.05	4.51	1.52	2.96	12.08
#7	250,3,391,3/9	19,636	2,686	7.31	0.55	0.05	11.26	arc-inc
#8	350,3,524,3/9	820,368	3,558	230.53	31.04	0.07	476.31	arc-inc
#9	350,3,2292,1/9	426,713	51,176	8.34	9.40	4.35	2.16	9.68

A first set of experiments (in which parameters are taken from [11]) is given in Table 1. The columns "ratio" represent how much MAC-dom+deg-mc was better than FC-CBJ-dom+deg-mc with respect to the associated measure (mean number of constraint checks, mean cpu time, mean number of backtracks). On this first set of experiments we can stress that the ratio of the number of constraint checks is less advantageous for MAC than the cpu time ratio. An explanation is that, for any search algorithm that performs some look-ahead filtering, each backtrack point involves restoring the previous state, and running again the variable-value selection. In spite of being free of any constraint check, this process is time consuming. MAC-dom+deg-mc being better and better than FC-CBJ-dom+deg-mc in number of backtracks (see the last column of Table 1) saves

a lot of time in addition to the time saved by constraint checks savings. Anyway, MAC-dom+deg-mc significantly overcomes FC-CBJ-dom+deg-mc on these problems.

We performed a second set of experiments on the now classical $\langle 50, 10, 0.1, p_2 \rangle$ set of parameters of Smith and Grant [30], [31]. In our formalism, it consists of the set of parameters $\langle 50, 10, 123, T \rangle$. Figure 1 gives the results, which corroborate those obtained in Table 1. MAC is slightly worse than FC-CBJ on easy problems (under– and over–constrained) while being much better around the cross–over point.

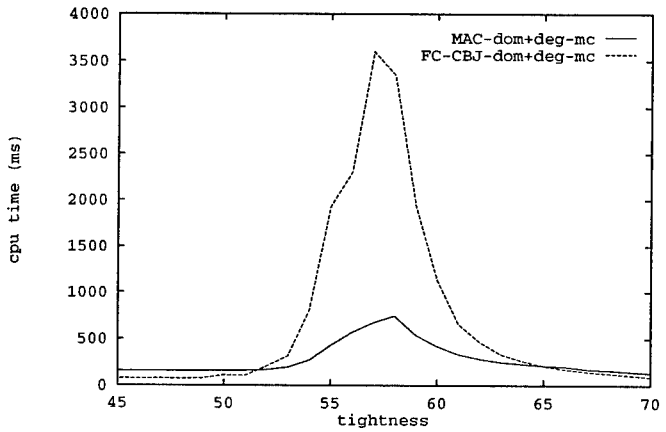


Fig. 1. FC-CBJ-dom+deg-mc and MAC-dom+deg-mc time performances on the $\langle 50, 10, 123, T \rangle$ experiment of Smith and Grant [30].

Frost–Dechter and Smith–Grant's parameters being limited to small domain sizes, we took the $\langle 50, 20, 95, 300 \rangle$ set of parameters in Frost and Dechter's sample, and changed domain sizes while keeping N and C fixed at 50 and 95 respectively, T varying to stay at T_{co} (see Fig. 2-(left)). We note that the more D grows, the more MAC-dom+deg-mc outperforms FC-CBJ-dom+deg-mc, going from 3 times faster when D is smaller than 10 to 26 times faster when D reaches 40.

Finally, we wanted to see the behavior of MAC when the density of the constraint graph increases. Figure 2-(right) presents the FC-CBJ-dom+deg-mc to MAC-dom+deg-mc cpu time ratio when the number C of constraints increases in the $\langle 30, 10, C, T_{co} \rangle$ set of parameters. MAC efficiency increases till the constraint graph contains approximately a third of the possible number of constraints. Afterwards, FC-CBJ becomes less and less worse as the number of constraints grows till the complete graph[8]. This phenomenon was pointed out by Sabin and Freuder.

[8] These cpu times ratios, despite showing the advantage of MAC, do not go higher than 3. The reason is that 30 variables is not enough to generate hard problems on which MAC would show its real efficiency.

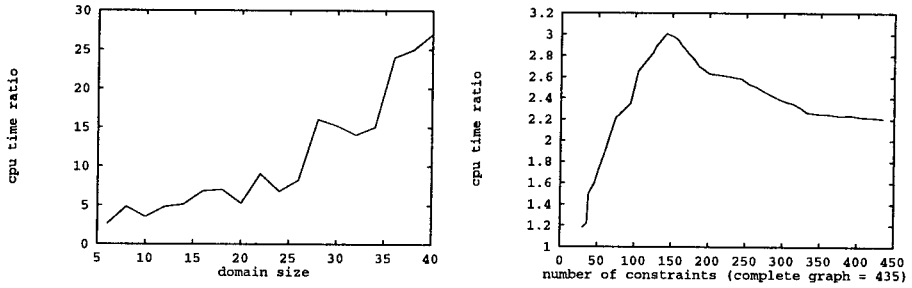


Fig. 2. FC-CBJ-dom+deg-mc to MAC-dom+deg-mc cpu time ratio on the $\langle 50, D, 95, T_{co} \rangle$ (left), D growing from 6 to 40; and on the $\langle 30, 10, C, T_{co} \rangle$ (right), where C grows from 29 to 435 (complete graph).

5 Combined DVOs: dom/deg

In Sect. 2 we presented different kinds of variable ordering heuristics and said that the dom DVO had been considered for a long time as the best one. However, when the constraint graph is sparse, many useful information is lost by this heuristic while it is caught by the SVOs based on the structure of the constraint graph.

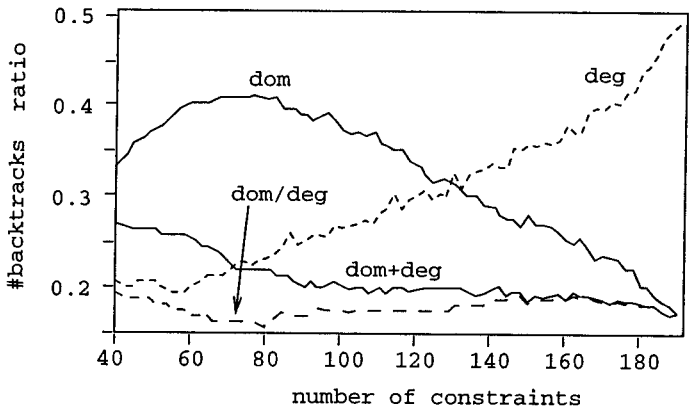


Fig. 3. Different variable ordering heuristics tested with MAC on the $\langle 20, 10, C, T_{co} \rangle$, where C grows from 40 to 190 (complete graph). Each graph represents the ratio of the mean number of backtracks of MAC with the given heuristic to the sum of the mean number of backtracks of the four algorithms tested (absolute results would have given unreadable graphs since the difficulty of the problems significantly grows when C grows).

In Fig. 3, where random problems with increasing density are solved by different ver-

sions of MAC (i.e. using different variable ordering heuristics[9]), it is shown that dom can be a very poor heuristic at low densities, while deg is very efficient on the same problems. Inversely, when the constraint graph becomes dense, deg goes blind while dom becomes clever. dom+deg, which breaks ties in dom by using the degree of the tying variables is shown in this Fig. 3 to improve dom on problems where it was bad. But, in dom+deg, the size of the domains clearly have the main influence on the ordering, the degree of variables being only used in cases where ties are found. To avoid this drawback, which prevents dom+deg from being as good as deg in sparse constraint networks, we propose to really combine dom and deg to obtain a new DVO in which deg is as influent as dom. This new DVO, dom/deg, selects as the next variable to be instantiated a variable that has the smallest ratio: size of the remaining domain to degree of the variable (i.e. a variable v minimizing $|D_v|/|\Gamma(v)|$). In Fig. 3 we have a first idea of its behavior: it has the behavior of dom+deg in networks where dom was good, and the one of deg in networks where deg was better. These first results being promising, we give in Table 2 and Fig. 4 a more complete set of experiments in which we compare MAC-dom+deg-mc and MAC-dom/deg-mc. Once again, the characteristics of the problems tested are taken from [11] and [30]. Results obtained in Table 2 show that with small domain sizes ($D < 10$) the two DVOs have similar behaviors, with a little advantage for dom/deg. The difference is slightly perceptible on the $\langle 35, 9, 178, 27\rangle$ and the $\langle 100, 12, 120, 110\rangle$ experiments. It is significant on the $\langle 50, 20, 95, 300\rangle$. This is explained by the fact that when D is very small, dom/deg and dom+deg are quite similar criteria, the variations of $|D_v|$ –for a given variable v– dominating those of $|\Gamma(v)|$ in dom/deg.

Table 2. MAC-dom+deg-mc versus MAC-dom/deg-mc. Only ratios are given (real values can be obtained from these ratios and Table 1). Values greater than 1 mean dom/deg is better, values smaller than 1 mean dom+deg is better.

	Parameters	ratios		
	$N, D, C, T/D * D$	#constraint checks	time	#backtracks
#1	35,6,501,4/36	1.00	1.01	1.35
#2	35,9,178,27/81	1.24	1.23	1.63
#3	50,6,325,8/36	1.11	1.12	1.53
#4	50,20,95,300/400	3.45	3.05	7.01
#5	100,12,120,110/144	1.11	1.10	3.20
#6	125,3,929,1/9	1.02	0.98	1.42
#7	250,3,391,3/9	1.00	1.00	arc-inc
#8	350,3,524,3/9	1.00	1.00	arc-inc
#9	350,3,2292,1/9	1.00	0.97	1.56

To be convinced that dom/deg is more advantageous when domains are larger, we tested the two heuristics on instances of problems with increasing domain size. In Fig. 5-(left), the domain sizes vary while N and C are fixed to 50 and 95 respectively. T changes so that problems are always on the cross-over point. The more the size of the domains increases, the more MAC-dom/deg-mc overcomes MAC-dom+deg-mc, going from once to 7 times faster when D grows from 6 to 40. Furthermore, Prosser (per-

[9] The LVO used is mc in all these versions. Without LVO, we remarked that the differences increase between good and bad algorithms.

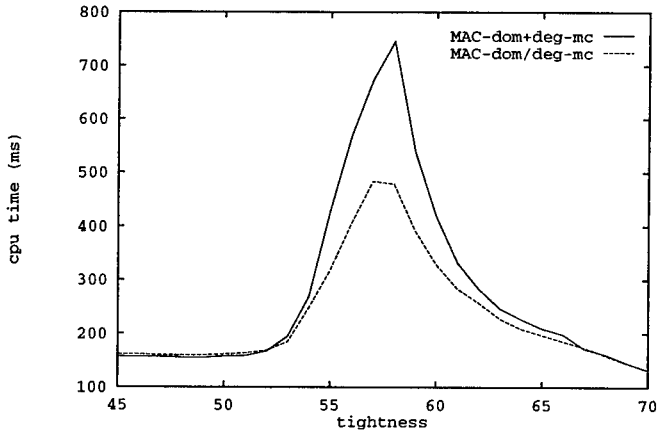


Fig. 4. MAC-dom+deg-mc and MAC-dom/deg-mc on the $\langle 50, 10, 123, T \rangle$.

sonal communication) has pointed out that when initial domain sizes are not all equal, dom (or dom+deg) can be fooled by these initial differences. We suppose that in these cases dom/deg would be even more interesting.

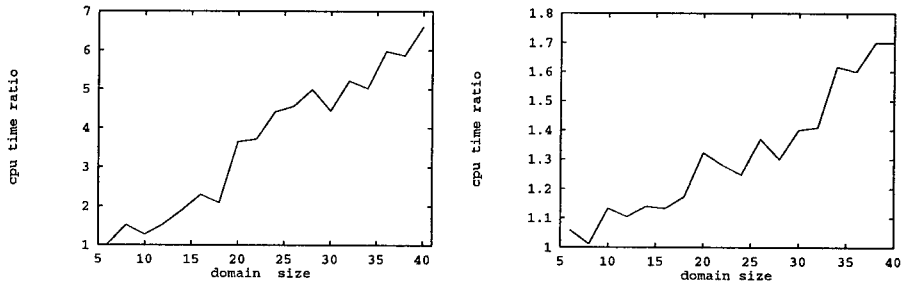


Fig. 5. MAC-dom+deg-mc versus MAC-dom/deg-mc on the $\langle 50, D, 95, T_{co} \rangle$ (left), and MAC-CBJ-dom/deg-mc versus MAC-dom/deg-mc on the $\langle 50, D, 95, T_{co} \rangle$ (right).

Thus, we can conclude that combining different DVOs is a promising approach. We have tested other combined DVOs not presented in this paper. The one that can be named dom/card, in which the number of previously assigned neighbors of the variable replaces the total number of neighbors in the ratio seems to be quite worse than dom/deg (when card alone was considered as a better SVO than deg alone [5]). On the other hand, when the ratio involves the number of not yet assigned neighbors of the variable, the performances are roughly similar to those obtained with dom/deg, sometimes bet-

ter, sometimes worse. dom/deg has been also implemented in FC-CBJ. We saw an improvement with respect to dom+deg, but smaller than the one observed on MAC.

6 CBJ Becomes Useless

We have shown that using MAC instead of FC as the filtering scheme was worthwhile on hard and large problems. If we follow the evolution of FC in FC-CBJ we should now use MAC-CBJ [25]. But, let us recall a sentence found in [16]: "Look ahead to the future in order not to worry about the past". In fact, some authors remarked that if we use a good variable ordering heuristic "CBJ is unlikely to generate large backjumps, and its savings are likely to be minimal" because "variables that have conflicts with past assignments are likely to be instantiated sooner" [1]. In [30], Smith and Grant said that "for most problems, the ordering given by dom ensures that chronological backtracking usually results in backtracking to the real culprit for a failure, so that informed backtracking does not add very much".

These statements, done in the case of FC-dom were probably too optimistic since a non negligible number of problems are easily solved by FC-CBJ-dom when FC-dom is thrashing [31]. But, as it is suggested by Haralick and Elliot's sentence, the more we will perform look-ahead, the less we will have to worry about looking back. CBJ was a strong improvement on BT (simple backtracking), FC-CBJ can be an improvement on FC on hard problems, MAC-CBJ cannot simply be claimed to be an improvement on MAC. In [31], while a lot of problems were found on which FC-CBJ-dom outperformed FC-dom by at least one order of magnitude, only one instance was found on which MAC-CBJ-dom significantly outperformed MAC-dom. If we consider now the DVO dom/deg in place of dom, there are even more reasons to think that CBJ becomes useless (since dom/deg has been shown smarter than dom). Furthermore, the more the amount of filtering involved in a search procedure is high, the more the overhead caused by CBJ is heavy [25]. CBJ was cheap to incorporate in BT, it was not prohibitive in FC, but it palpably slows down the search in MAC. Hence, a significant number of constraint checks must be saved to outweigh this overhead.

Table 3. MAC-CBJ-dom/deg-mc versus MAC-dom/deg-mc.

	Parameters	ratios		
	$N, D, C, T/D * D$	#constraint checks	time	#backtracks
#1	35,6,501,4/36	0.99	1.17	0.99
#2	35,9,178,27/81	0.99	1.32	0.99
#3	50,6,325,8/36	0.99	1.21	0.99
#4	50,20,95,300/400	0.99	1.33	0.99
#5	100,12,120,110/144	0.98	0.99	0.96
#6	125,3,929,1/9	0.97	1.08	0.96
#7	250,3,391,3/9	arc-inc	arc-inc	arc-inc
#8	350,3,524,3/9	arc-inc	arc-inc	arc-inc
#9	350,3,2292,1/9	0.64	0.70	0.61

Table 3 gives the comparison of MAC-CBJ-dom/deg-mc and MAC-dom/deg-mc on the Frost and Dechter's problems. On the problems #1 to #8 the result is easy to read: CBJ leads to a few constraint checks savings which are not sufficient to make good

the loss of time. But, on the set of parameters #9, there is a significant gain for MAC-CBJ-dom/deg-mc. If we focus on the 100 instances which form this experiment we see that on 99 instances MAC-dom/deg-mc and MAC-CBJ-dom/deg-mc have almost the same behavior, solving the problem in less than 1 second with a number of backtracks smaller than 1000. But on one of the 100 instances MAC-dom/deg-mc needs 137 seconds and 41,639 backtracks to find a solution when MAC-CBJ-dom/deg-mc only needs 73 seconds and 20,069 backtracks. The mean performances are strongly influenced by this single instance which seems to match with the definition of "exceptionally hard problems" (*ehps*) [30]. Indeed, it occurs in the region where almost all problems are soluble (2547 constraints are necessary to be at the cross–over point in the $\langle 350, 3, C, 1\rangle$ set of parameters [11]). But, as opposed to the ehps found in [31], where FC-CBJ or MAC-CBJ were orders of magnitude faster than FC or MAC, MAC-CBJ-dom/deg-mc is only twice faster than MAC-dom/deg-mc on our ehp. Further experiments should probably be done to see whether ehps could be found on which MAC-CBJ-dom/deg-mc is really better than MAC-dom/deg-mc, though we did not find any in all the experiments we performed on smaller networks (50 variables).

Finally, we want to recall that the more domain sizes increase, the more the length of the jumps performed by CBJ decreases while CBJ time overhead increases (see the CBJ mechanism in [22]). This is confirmed in Fig. 5-(right) where MAC-CBJ-dom/deg-mc and MAC-dom/deg-mc are compared on the $\langle 50, D, 95, T_{co}\rangle$ experiment with increasing D.

Therefore, except on sparse networks with small domain sizes where more studies should be done, we think we can conclude that including CBJ in MAC-dom/deg-mc has more chances to slow down the search of at least 20% cpu time than to speed it up.

7 Conclusion

After a recall of the story of search procedures in constraint networks, this paper has shown how MAC can outperform FC and FC-CBJ on relatively hard and large randomly generated instances of constraint networks. Once the superiority of MAC has been proven, we have proposed a new kind of variable ordering heuristic, dom/deg, which really combines information on domain sizes and constraint graph structure. We proved its efficiency when compared with dom+deg, the most efficient previous heuristic. The total gain involved by these two techniques (MAC and dom/deg) is summarized in Table 4. The ratios of the mean performances of FC-CBJ-dom+deg-mc to the mean performances of MAC-dom/deg-mc are presented. The tested problems are again Frost and Dechter's problems. The benefit is always significant. Furthermore, we must have in mind that with larger domains the gain is greater and greater.

Therefore, we can conclude that on relatively hard and large instances of random problems, MAC and our new variable ordering heuristic are more efficient than FC-CBJ and classical dom or dom+deg DVOs.

Finally, we have shown in the last section that performing CBJ is almost always useless when combined with a procedure achieving as much look-ahead as MAC-dom/deg-mc. The time overhead is too heavy to be outweighed by the small number of constraint checks and backtracks saved.

Table 4. FC-CBJ-dom+deg-mc versus MAC-dom/deg-mc.

	Parameters	ratios		
	$N, D, C, T/D * D$	#constraint checks	time	#backtracks
#1	35,6,501,4/36	1.54	2.58	10.03
#2	35,9,178,27/81	1.97	3.98	23.33
#3	50,6,325,8/36	3.00	5.03	26.55
#4	50,20,95,300/400	7.13	13.38	259.64
#5	100,12,120,110/144	17.29	28.69	2785.20
#6	125,3,929,1/9	6.15	2.91	17.15
#7	250,3,391,3/9	7.31	11.26	arc-inc
#8	350,3,524,3/9	230.53	476.31	arc-inc
#9	350,3,2292,1/9	8.35	2.10	15.10

Acknowledgments. We want to thank Dan Frost and Stuart Grant for their help concerning instance generators, Olivier Dubois, who is at the origin of our comments on previous instance generators, Dan Sabin for the fruitful discussions we had on MAC, and Gene Freuder for his advice on ordering heuristics.

References

1. F. Bacchus and P. van Run. Dynamic variable ordering in csps. In *Proceedings CP'95*, pages 258–275, Cassis, France, 1995.
2. C. Bessière. Systèmes à contraintes évolutifs en intelligence artificielle. Phd thesis, LIRMM, University of Montpellier II, September 1992. (in French).
3. C. Bessière, E.C. Freuder, and J.C. Régin. Using inference to reduce arc consistency computation. In *Proceedings IJCAI'95*, pages 592–598, Montréal, Canada, 1995.
4. R. Dechter. Learning while searching in constraint satisfaction problems. In *Proceedings AAAI'86*, pages 178–183, Philadelphia PA, 1986.
5. R. Dechter and I. Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 68:211–241, 1994.
6. R. Dechter and J. Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.
7. M.J. Dent and R.E. Mercer. Using local topology to model hard binary constraint satisfaction problems. In *Proceedings of the workshop –Studying and Solving Really Hard Problems–, CP'95*, pages 52–61, Cassis, France, 1995.
8. O. Dubois. Private communication, 1995.
9. E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.
10. D. Frost and R. Dechter. In search of the best constraint satisfaction search. In *Proceedings AAAI'94*, pages 301–306, Seattle WA, 1994.
11. D. Frost and R. Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings IJCAI'95*, pages 572–578, Montréal, Canada, 1995.
12. J. Gaschnig. A general backtrack algorithm that eliminates most redundant tests. In *Proceedings IJCAI'77*, page 457, Cambridge MA, 1977.
13. J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie-Mellon University, Pittsburgh PA, 1979.
14. P.A. Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *Proceedings ECAI'92*, pages 31–35, Vienna, Austria, 1992.

15. M.L. Ginsberg, M. Frank, M.P. Halpin, and M.C. Torrance. Search lessons learned from crossword puzzles. In *Proceedings AAAI'90*, pages 210–215, Boston MA, 1990.
16. R.M. Haralick and G.L. Elliot. Increasing tree seach efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
17. W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *Proceedings IJCAI'95*, pages 607–613, Montréal, Canada, 1995.
18. P.D. Hubbe and E.C. Freuder. An efficient cross product representation of the constraint satisfaction problem search space. In *Proceedings AAAI'92*, pages 421–427, San José CA, 1992.
19. V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13(1):32–44, Spring 1992.
20. B.A. Nadel. Tree search and arc consistency in constraint satisfaction algorithms. In L.Kanal and V.Kumar, editors, *Search in Artificial Intelligence*, pages 287–342. Springer-Verlag, 1988.
21. B.A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.
22. P. Prosser. Domain filtering can degrade intelligent backtracking search. In *Proceedings IJCAI'93*, pages 262–267, Chambéry, France, 1993.
23. P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, August 1993.
24. P. Prosser. Binary constraint satisfaction problems: some are harder than others. In *Proceedings ECAI'94*, pages 95–99, Amsterdam, The Netherlands, 1994.
25. P. Prosser. Mac-cbj: maintaining arc consistency with conflict-directed backjumping. Technical Report 95-177, Department of Computer Science, University of Starthclyde, 1995.
26. J.F. Puget. Ilog solver. In J. Gensel, editor, *Journées Contraintes et Objets*, Grenoble, France, November 1992. (in French).
27. P.W. Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.
28. D. Sabin and E. Freuder. Contradicting conventional wisdom in constraint satisfaction. In Alan Borning, editor, *PPCP'94: Second Workshop on Principles and Practice of Constraint Programming*, Seattle WA, May 1994.
29. B. Smith. The phase transition in constraint satisfaction problems: A closer look at the mushy region. In *Proceedings ECAI'94*, pages 100–104, Amsterdam, The Netherlands, 1994.
30. B. Smith and S.A. Grant. Sparse constraint graphs and exceptionally hard problems. In *Proceedings IJCAI'95*, pages 646–651, Montréal, Canada, 1995.
31. B. Smith and S.A. Grant. Where the exceptionally hard problems are. In *Proceedings of the workshop –Studying and Solving Really Hard Problems–, CP'95*, pages 172–182, Cassis, France, 1995.
32. Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, 1989.
33. R.J. Wallace and E.C. Freuder. Conjunctive width heuristics for maximal constraint satisfaction. In *Proceedings AAAI'93*, pages 762–768, Washington D.C., 1993.

The Independence Property of a Class of Set Constraints

Witold Charatonik* Andreas Podelski

Max-Planck-Institut für Informatik
Im Stadtwald, D-6123 Saarbrücken, Germany
{witold;podelski}@mpi-sb.mpg.de

Abstract

We investigate a class of set constraints that is used for the type analysis of concurrent constraint programs. Its constraints are inclusions between first-order terms (without set operators) interpreted over non-empty sets of finite trees. We show that this class has the independence property. We give a polynomial algorithm for entailment. The independence property is a fundamental property of constraint systems. It says that the constraints cannot express disjunctions, or, equivalently, that negated conjuncts are independent from each other. As a consequence, the satisfiability of constraints with negated conjuncts can be directly reduced to entailment.

1 Introduction

In this paper we show that a class of set constraints that is used for the type analysis of concurrent constraint programs has the independence property and give a polynomial entailment algorithm for this class. Below we introduce the property and the class, and then we are able to state the results more precisely.

Independence property. A constraint system has the independence property if the constraints cannot express disjunctions. Formally, given the constraints γ and $\varphi_1, \ldots, \varphi_n$, the implication $\gamma \rightarrow \varphi_1 \vee \ldots \vee \varphi_n$ is valid iff one of the n implications $\gamma \rightarrow \varphi_1, \ldots, \gamma \rightarrow \varphi_n$ is valid. An equivalent formulation of the property states that the satisfiability problem of a conjunction with any number of negated constraints can be reduced to "independent" sub-problems with exactly one negated constraint. Namely, the conjunction $\gamma \wedge \neg\varphi_1 \wedge \ldots \wedge \neg\varphi_n$ is satisfiable iff the n conjunctions $\gamma \wedge \neg\varphi_1, \ldots, \gamma \wedge \neg\varphi_n$ are satisfiable.[1] As a direct algorithmic consequence, the satisfiability problem of the conjunction of γ with n negated constraints can be reduced to n entailment problems (*i.e.*, the dual of the validity of the n implications $\gamma \rightarrow \varphi_1, \ldots, \gamma \rightarrow \varphi_n$).

The independence property is a fundamental notion. In the context of constraint logic programming (CLP), it has made possible the manipulation of inequations [9].

*On leave from University of Wrocław, Poland. Partially supported by KBN grant 8 S503 022 07 and by Foundation for Polish Science.

[1] Constraints are closed under conjunction; thus, we may use the constraint γ for the conjunction of all positive constraints.

The property also characterizes (and is characterized by) the semantics of bottom-up and top-down computations [22]. A general study of the property shows its importance in various symbolic computation areas [20]. In constraint data bases, the property allows the efficient containment test between constraint relations [17]. The property is necessary for the inference of constrained functional dependencies in polynomial time [23].

Examples of constraint systems with the independence property are: universal closures of the definite Horn clauses of a logic programming language [20], term equations over finite or infinite trees [9], linear equations over the real numbers [19], various constraint classes over feature trees [5, 4, 28], infinite Boolean algebras with positive constraints [15], and various simple subclasses of constraints (*e.g.*, inequations over numbers where each variable always appears on the same side of the inequation) which may be useful for the application considered in [23].

Set constraints. Generally, set constraints, *i.e.*, inclusions between terms formed by first-order function symbols and set operators and interpreted over sets of finite trees (see [1, 3, 6, 7, 8, 10, 12, 13, 14, 18]), do not have the independence property. For example, if f is binary and 0 denotes the empty set, then the equivalence

$$f(x,y) \subseteq 0 \leftrightarrow (x \subseteq 0) \vee (y \subseteq 0)$$

holds but neither of the implications $f(x,y) \subseteq 0 \rightarrow x \subseteq 0$ and $f(x,y) \subseteq 0 \rightarrow y \subseteq 0$ holds.[2] In fact, every inclusion between two terms with the same function symbol of arity $n \geq 2$ expresses a disjunction, since the equivalence

$$f(u_1,\ldots,u_n) \subseteq f(v_1,\ldots,v_n) \leftrightarrow (u_1 \subseteq 0) \vee \ldots \vee (u_n \subseteq 0) \vee (u_1 \subseteq v_1 \wedge \ldots \wedge u_n \subseteq v_n)$$

holds (the inclusion $f(u_1,\ldots,u_n) \subseteq f(v_1,\ldots,v_n)$ entails none of the disjuncts if and only if the arity n is strictly greater than 1).

The two counter examples cited above can be avoided if we restrict the domain of interpretation to only non-empty sets of trees. The question arises for which natural syntactic subclasses of set constraints this semantic restriction is sufficient to obtain the independence property.[3] We start investigating the question by considering the possibly simplest class of set constraints. This class has been introduced as the constraint system INÈS and investigated in [25].

INÈS. We obtain the constraint system INÈS if we take as constraint formulas conjunctions of inclusions $t_1 \subseteq t_2$ between first-order terms (without set operators) and interpret them over the domain of non-empty sets of trees. INÈS is motivated by, and used for, type analysis problems in concurrent constraint programming CCP languages [21, 26], in particular Oz [27]. The non-emptiness is required for that application. We borrow the explanation for this fact from [25]: In CCP languages, execution proceeds by adding conjuncts to a global constraint store. This store thus grows monotonically during the whole execution of a concurrent program (*i.e.*, there

[2] As Aiken et.al. remark in [1], "this [equivalence] allows the system to encode nondeterministic choices, which raises the complexity to NEXPTIME." In fact, we associate an algorithmic intuition with the *presence* of the independence property. Namely, that it is the "deep" reason for the existence of a "deterministic" test of satisfiability (for example the one in [25]).

[3] We observe that if union is one of the set operators, we can find counter examples. Indeed, if $t_1,\ldots,t_n$ are ground terms, the inclusion $x \subseteq t_1 \cup \ldots \cup t_n$ entails the disjunction $t_1 \subseteq x \vee \ldots \vee t_n \subseteq x$ (but none of the disjuncts).

is no backtracking on this level). Therefore, the addition of an inconsistent conjunct amounts to a programming error. The detection of potential errors of this kind at compile time is possible by approximating the constraint store with an INÈS constraint φ and testing φ for satisfiability. For example, the execution of the following Oz program

```
proc {P Y} Y=a end
proc {Q Z} Z=b end
{P X}
{Q X}
```

leads to the inconsistent constraint store $X = Y \wedge Y = a \wedge X = Z \wedge Z = b$, and thus to an error. We infer from the program the set constraint $X \subseteq Y \wedge Y = a \wedge X \subseteq Z \wedge Z = b$, which is unsatisfiable if *and only if* the interpretation domain is restricted to non-empty sets. This part of the type analysis system is implemented and used experimentally for Oz programs [24]. As explained in [25], INÈS has a potential interest also for the analysis of CLP and functional programming languages.

Result. The main result of this paper is that INÈS has the independence property. In order to avoid pathological cases, we assume that the set of function symbols is infinite, as is common in the context of independence results (see, *e.g.* [5, 4, 28]). The practical reading of the assumption is that, when using the constraints for program analysis, the analysis refers to all possible extensions of the program, with an extensible set of identifiers. In a system with incremental compilation, and for modular program analysis, the assumption seems natural.

We also derive a polynomial algorithm for testing entailment between INÈS constraints. Thus, we obtain a polynomial test of the satisfiability of conjunctions of positive and negated constraints.[4]

The only algorithms known previously for the two problems of entailment and satisfiability with negated constraints are for the general case of set constraints with set operators and with negation [2, 11, 7]; their complexity is NEXPTIME [7].

Structure of the paper. In the next section, we give some intuition about the proof of our main result. We then fix the syntax and the semantics of our class of set constraints formally and state the main theorems. In Section 4 we present the full proof. We end with a conclusion section.

2 Overview of the proof

The proof of our independence result is interesting in its own right. To give some intuition, an independence proof may generally work according to the following pattern (as do the ones in [5, 4, 28]). First, one finds syntactic conditions for two constraints γ and φ (in a normal form for the entailment problem) which characterize when the entailment $\gamma \rightarrow \varphi$ holds. Then, given γ and $\varphi_1, \ldots, \varphi_n$ such that none of the n implications $\gamma \rightarrow \varphi_1, \ldots, \gamma \rightarrow \varphi_n$ holds, one adds constraints $\gamma_1, \ldots, \gamma_n$ to γ such that the n implications $\gamma \wedge \gamma_i \rightarrow \neg\varphi_i$ hold, for $i = 1, \ldots, n$.

For example, given $\gamma \approx x = f(u) \wedge f(v) = y$ and $\varphi_1 \approx x = y$, we could set

[4] Such conjunctions are strictly more expressive than conjunctions of only positive constraints. The proof of this fact is by easy modification of the proof of the corresponding statement for a different class of set constraints (Corollary 3.2) in [2].

$\gamma_1 \approx u = a \wedge v = b$ (clearly, $\gamma \rightarrow \varphi_1$ is not valid over the domain of finite trees, but the implication $\gamma \wedge \gamma_1 \rightarrow \neg\varphi_1$ is valid).

Then one uses the syntactic characterization of entailment in order to show that the addition of the constraints $\gamma_1, \ldots, \gamma_n$ to γ is consistent. Now we have that $\gamma \wedge \gamma_1 \wedge \ldots \wedge \gamma_n$ is satisfiable and entails the conjunction $\neg\varphi_1 \wedge \ldots \neg\varphi_n$. This means that the implication $\gamma \rightarrow \varphi_1 \vee \ldots \vee \varphi_n$ does not hold.

The previous description shows that, globally speaking, the art of proving independence is the art of finding a "good" syntactic characterization of entailment (*i.e.*, one that exhibits which conjuncts one has to add to obtain the disentailment of non-entailed constraints). This, however, is not evident for the case of inclusion constraints, for the following intuitive reason. An implication of the form $\gamma \wedge x \subseteq t \wedge s \subseteq y \rightarrow x \subseteq y$, for two given terms s and t, holds if, but generally *not* only if, $\gamma \rightarrow t \subseteq s$ holds. This is different for equations: The implication $\gamma \wedge x = t \wedge s = y \rightarrow x = y$ holds if and only if $\gamma \rightarrow t = s$ holds. Thus, in order to make the entailment of $\neg\, x = y$ (called the "disentailment" of $x = y$) hold, it is sufficient to "make different" the two terms s and t (as done in the example above). This will not work if $=$ is replaced by $\subseteq$, since forcing $t \not\subseteq s$ does not force $x \not\subseteq y$.

The situation becomes even more gloomy when we consider implications like

$$\gamma \wedge x \subseteq t_1 \wedge \ldots \wedge x \subseteq t_n \wedge s_1 \subseteq y \wedge \ldots \wedge s_m \subseteq y \rightarrow x \subseteq y.$$

These are valid if (but, again, not only if) γ entails the inclusion $t_1 \cap \ldots \cap t_n \subseteq s_1 \cup \ldots \cup s_m$. Such an inclusion, however, is not an INÈS constraint.

Even checking of inclusions between intersections and unions (as in the previous example), however, may not suffice to derive the validity of an implications, as our last example shows:

$$x \subseteq f(y, \_) \wedge x \subseteq f(a, \_) \wedge z \subseteq a \rightarrow z \subseteq y.$$

The first step of our proof is to construct an algorithm for testing entailment. This algorithm in particular overcomes the above-mentioned limitations of the syntax with respect to union and intersection. One ingredient of the algorithm is to combine the upper bounds $t_1, \ldots, t_n$ of the variable x by taking their "shuffles," *i.e.*, the terms $f(u_1, \ldots, u_k)$ where $f(u_1, \_, \ldots, \_,)$ and ... and $f(\_, \ldots, \_, u_k)$ are among x's upper bounds. In the example: $x \subseteq f(u_1, u_2) \wedge x \subseteq f(v_1, v_2) \wedge f(u_1, v_2) \subseteq y$, this is already sufficient to derive that $x \subseteq y$ is entailed. Another ingredient is the special treatment of ground terms. We use information that is contained (explicitly or *implicitly*) in a constraint to infer that a ground term is a lower bound of a variable (which is what is needed in the last example).

Since the entailment algorithm does not yield a "good" syntactic characterization of entailment, we abandon the technique of forcing disentailment by adding corresponding conjuncts. Instead, we will use a certain "minimal" solution ν of a given constraint γ such that $\nu(x)$ is not a subset of $\nu(y)$ whenever the inclusion $x \subseteq y$ is not entailed by γ. How do we get this solution ν? The standard canonical solution of γ corresponding to its solved form [25] is *not* a candidate for ν since it is the maximal solution of γ, assigning the set of all trees to every "unconstrained" variable x (*i.e.*, where $x \subseteq \tau$ does not occur for any term τ). The minimal solution of an INÈS constraint γ does generally not exist. We will, however, "complete" the constraint γ

such that for each of its variables x it contains a conjunct $t \subseteq x$ where t is a ground term. Then we can show that a minimal solution does exist (and is unique on the variables that occur in γ). The remaining, technically somewhat involved problem is to complete the constraint such that (1) the constraint remains consistent, and (2) for all pairs of variables x and y where the inclusion $x \subseteq y$ is not entailed, there exists at least one ground term t below x (*i.e.*, $t \subseteq x$ occurs) such that $t \subseteq y$ is not entailed. Then, the value of y under the minimal solution will not contain the tree t. Hence we obtain a solution of the completed constraint (and thus of the original one) which satisfies $\neg\, x \subseteq y$.

3 Formal statement of the result

We assume given an infinite signature Σ fixing the arity $n \geq 0$ of its function symbols $f, g, a, b, \ldots$ and infinite set $\mathcal{V}$ of variables $x, y, z, u, v, w, \ldots$. We use $t, s, \ldots$ as meta-variables for ground terms, $\tau, \tau_1, \tau_2, \ldots$ for (possibly non-ground) terms of depth ≤ 1 and $\theta, \theta_1, \theta_2$ for (possibly non-ground) terms of arbitrary depth. We write $\bar{u}$ for the tuple $(u_1, \ldots, u_n)$ of variables and $\bar{t}$ for the tuple $(t_1, \ldots, t_n)$ of ground terms, where $n \geq 0$ is given implicitly (*e.g.*, in $x \subseteq f(\bar{u})$ by the arity of the function symbol f). We write $\bar{u} \subseteq \bar{v}$ for $\{u_1 \subseteq v_1, \ldots, u_n \subseteq v_n\}$. The notation $\Sigma(\gamma)$ stands for set of symbols occurring in γ, and $\mathcal{V}(\gamma)$ for its variables. We use $\theta[u, \bar{z}]$ for a term containing possibly occurrences of variables $u, z_1, \ldots, z_n$; in the context of $\theta[u, \bar{z}]$, the term $\theta[v, \bar{z}]$ denotes the term θ with the occurrence of u replaced by occurrence of v at the same position. We use $\theta[u/u']$ to denote the term θ with an occurrence of u replaced by u'.

We are interested in inclusions between arbitrary first-order terms. However, to simplify the presentation, we may assume (wlog. and without changing the complexity measure) that our constraint formulas γ are finite sets of inclusions of a restricted form, namely either $\tau_1 \subseteq \tau_2$ *i.e.*, between variables or flat terms, or $t \subseteq \tau$ *i.e.*, between a ground term on the left-hand side and a variable or a flat term on the right-hand side. As is usual, we identify a conjunction of constraints with the set of all conjuncts.

Our interpretation domain is the set of all non-empty sets of finite trees (*i.e.*, ground terms) over the signature Σ. A valuation is a mapping assigning non-empty sets of finite trees to variables. Each valuation ν can be extended in canonical way to a mapping ν from terms to non-empty sets of finite trees by putting

$$\nu(f(\tau_1, \ldots, \tau_n)) = \{f(t_1, \ldots, t_n) \mid t_i \in \nu(\tau_i) \text{ for } i = 1, \ldots, n\}.$$

A valuation ν satisfies γ if for all constraints $\tau_1 \subseteq \tau_2$ in γ we have $\nu(\tau_1) \subseteq \nu(\tau_2)$. We say that γ is satisfiable if there exists valuation satisfying γ, and that γ entails φ, written $\gamma \models \varphi$, if all valuations satisfying γ satisfy φ.

The following remark says that we may restrict ourselves to satisfiability problems where all negated constraints are of the form $x \not\subseteq y$.

Remark 1 Let γ be a constraint, and x_1, x_2 fresh variables, *i.e.*, $x_1, x_2 \notin \mathcal{V}(\gamma)$. Then $\gamma \cup \{\theta_1 \not\subseteq \theta_2\}$ is satisfiable iff $\gamma \cup \{x_1 \subseteq \theta_1, \theta_2 \subseteq x_2, x_1 \not\subseteq x_2\}$ is satisfiable. □

We thus may state our main result as follows.

Theorem 1 (Independence) Let γ be a constraint. Then $\gamma \cup \bigcup_i \{x_i \not\subseteq y_i\}$ is satisfiable iff $\gamma \cup \{x_i \not\subseteq y_i\}$ is satisfiable for all i.

Note our assumption about the infinite signature. In the case of a finite one, the independence property does generally not hold. For a counter example, consider $\Sigma = \{a, f\}$ with a being a constant and f being unary, and the constraint $\gamma = \{f(a) \subseteq y, f(y) \subseteq y\}$. Then γ implies $a \subseteq x \vee x \subseteq y$, but it implies neither $a \subseteq x$ nor $x \subseteq y$.

We will now refer to the axioms in Table 1. Axiom 1 needs to come in two versions since a ground term may be the lower bound of a term. As for the restriction in the formulation of Axiom 2, note that without it, the closure of, for example, the constraint $a \subseteq x \wedge f(x) \subseteq x$ under the consequences of the axiom would be an infinite set of constraints.

Axiom 3 could be stated in the more formal (but less readable) way:

$$x \subseteq f(u_{11}, \ldots, u_{1n}), \ldots, x \subseteq f(u_{m1}, \ldots, u_{mn}) \rightarrow x \subseteq f(u_{j_1 1}, \ldots, u_{j_n n})$$

where $j_1, \ldots, j_n \in \{1, \ldots m\}$. Similarly, the first part of Axiom 4 can be stated as $\gamma \rightarrow \tau \subseteq \tau$ if τ occurs in γ, and $\tau_1 \subseteq \tau_2 \wedge \tau_2 \subseteq \tau_3 \rightarrow \tau_1 \subseteq \tau_3$, and $t \subseteq \tau_1 \wedge \tau_1 \subseteq \tau_2 \rightarrow t \subseteq \tau_2$. The other axioms use a notation that is introduced in the following definition.

Definition 2 (implicit occurrence: "$\widehat{\in}$") Given the ground term t, we say that $x \subseteq t$ *occurs implicitly* in γ, and write $x \subseteq t \mathrel{\widehat{\in}} \gamma$, if or the following, inductively defined condition, holds:

- either t is a constant symbol and $x \subseteq t \in \gamma$, or
- $t = f(t_1, \ldots, t_n)$ and $x \subseteq f(x_1, \ldots, x_n) \in \gamma$ and $x_i \subseteq t_i \mathrel{\widehat{\in}} \gamma$ for all i.

We say that a term t occurs implicitly in γ if $x \subseteq t$ occurs implicitly in γ for some variable x.

Remark 3 The axioms in Table 1 are valid over non-empty sets of finite trees.

Proof. The proof is done by inspection of each axiom. For the Axiom 5, note that if ν satisfies $x \subseteq t$ then $\nu(x) = \{t\}$. For the last axiom, the constraint $x \subseteq f(\ldots, v, \ldots)$ and implicit occurrence of $v \subseteq t$ in γ implies that if ν is a solution of γ then all trees in $\nu(x)$ must have the subtree t on the respective position; the constraint $x \subseteq f(\ldots, u, \ldots)$ implies then that $t \in \nu(u)$. □

Definition 4 (closed) We say that a constraint γ is *closed* if it contains all its consequences according to the axioms in Table 1.

As a direct consequence of Remark 3 (formulated as a corollary below), we may restrict our logical investigation of the entailment problem to closed constraints. Also, in the following we will concentrate on constraints that are satisfiable. Satisfiability can be tested in cubic time [25].

Remark 5 Let γ' be the closure of the constraint γ under of consequences of the axioms in Table 1. Then $\gamma \models x \subseteq y$ iff $\gamma' \models x \subseteq y$. □

Theorem 2 (Entailment) If γ is a satisfiable and closed constraint, then

$$\gamma \models x \subseteq y \text{ iff } x \subseteq y \in \gamma.$$

1. $f(\bar{u}) \subseteq f(\bar{v}) \rightarrow \bar{u} \subseteq \bar{v}$
   $f(\bar{t}) \subseteq f(\bar{u}) \rightarrow \bar{t} \subseteq \bar{u}$
2. $\bar{u} \subseteq \bar{v} \rightarrow f(\bar{u}) \subseteq f(\bar{v})$
   $\gamma \wedge \bar{t} \subseteq \bar{u} \rightarrow f(\bar{t}) \subseteq f(\bar{u})$ if $f(\bar{t})$ occurs in γ
3. $x \subseteq f(u_1, \ldots) \wedge \ldots \wedge x \subseteq f(\ldots u_n) \rightarrow x \subseteq f(u_1, \ldots u_n)$
4. the relation $\subseteq$ is reflexive and transitive
   $\gamma \wedge t \subseteq y \rightarrow x \subseteq y$ if $x \subseteq t \,\hat{\in}\, \gamma$
5. $\gamma \rightarrow t \subseteq x$ if $x \subseteq t \,\hat{\in}\, \gamma$
6. $\gamma \wedge x \subseteq f(\ldots, u, \ldots) \wedge x \subseteq f(\ldots, v, \ldots) \rightarrow t \subseteq u$ if $v \subseteq t \,\hat{\in}\, \gamma$

Table 1: Axioms for inclusion constraints over non-empty sets of finite trees

Entailment algorithm. We define an algorithm by fixed point iteration. At each iteration step, we add the consequences under the axioms in Table 1 of the set of constraints derived so far. After termination, the obtained set of constraints is equivalent to the initial one and it is closed. Hence, by Remark 5 and Theorem 2, entailment is tested by checking membership (for inclusions between variables, which, according to Remark 1, is sufficient).

How many applications of each rule are possible before a fixed point is reached, if n is the size of γ? Axiom 5 can be applied only on pairs (x, t) such that γ entails that $x \subseteq t$. Since γ is assumed to be consistent, there are at most n such pairs. Similarly, the number of applications of Axiom 6 is bounded by the number of variables y times the number of triples (x, t, j) where j is the argument position containing, say the variable u_j, where $u_j \subseteq t$ occurs implicitly in γ for some ground term t. The number of such triples is bounded by $k \cdot n$, where k is the (assumed a priori fixed) maximal arity of all function symbols f occurring in γ. This is because the conjunction of $x \subseteq f(\ldots, t, \ldots)$ and $x \subseteq f(\ldots, t', \ldots)$ is unsatisfiable (over the domain of nonempty sets of trees!) for $t \neq t'$. Thus, the number of consequences under Axioms 5 and 6 is bounded by $n + k \cdot n^2$, and at most $n + k \cdot n$ new explicitly occurring ground terms t can be introduced. There are at most $O(n^2)$ many consequences of Axiom 1, since the consequences are inclusions between variables or between ground terms and variables, and we do not introduce new variables, and we introduce at most $n + k \cdot n$ new ground terms. There are at most $O(n^{2(k+1)})$ consequences of Axiom 2, Axiom 3 and Axiom 4, since the consequences are inclusions between variables, explicitly occurring ground terms and terms $f(\bar{u})$. There are at most n^{k+1} terms of the form $f(\bar{u})$ (since the number of function symbols occurring in γ is bounded by n).

Since the work at each iteration step can clearly be done in polynomial time, this rough approximation of the complexity of the entailment algorithm shows that it is polynomial. We believe that a cubic algorithm can be derived from the refined analysis of data structures and the execution strategy. We leave this for future work.

4 Formal proof

In order to prove Theorems 1 and 2 (which we complete at the end of this section), we need a "minimal" solution of a given solved satisfiable constraint γ. Unfortunately, such a solution generally does not exist. In order to be able to construct such a solution, which we will do in the second part of this section, we have to "complete" γ by adding a ground lower bound for each variable occurring in γ.

Completing with ground lower bounds. The following two notions are used in Definition 8 and in subsequent (inductive) proofs.

Definition 6 (chain of proper upper bounds) We say that a variable x has a *proper upper bound* in γ if γ contains a constraint of the form $x \subseteq f(x_1, \ldots, x_n)$. A *chain of proper upper bounds* for x in γ is a sequence of constraints of the form $x_i \subseteq f_i(\ldots, x_{i+1}, \ldots)$, with $x_0 = x$.

Remark 7 Every chain of proper upper bounds in a satisfiable constraint γ is finite.

Proof. If the variable u has an infinite chain of proper upper bounds in γ, then there exists a term θ such that $\gamma \models u \subseteq \theta[u]$.

Let ν be a solution of γ and let s be a tree of minimal depth in $\nu(u)$. Then s must be of the form $\theta[s']$ where $s' \in \nu(u)$. This contradicts the minimality of s. □

Definition 8 (lower bound completion) Let γ be a satisfiable and closed constraint. We say that γ' is an *m-level lower bound completion* of γ if the following conditions hold:

1. $\gamma' = \gamma \cup \{t_1 \subseteq x_1, \ldots, t_n \subseteq x_n\}$ where all t_i's are ground terms not occurring in γ, and all x_i's are variables occurring in γ;
2. γ' is satisfiable;
3. $\gamma \models u \subseteq v$ iff $\gamma' \models u \subseteq v$ for any two variables u, v;
4. if x_i has no proper upper bound in γ, then t_i is a constant;
5. if $f(\bar{u})$ is a proper upper bound of x_i in γ, then t_i is of the form $t_i = f(\bar{s})$ and $\bar{s} \subseteq \bar{u} \in \gamma'$;
6. if $t_i = f(\bar{s})$ and $\bar{s} \subseteq \bar{y} \in \gamma'$, then $x_i \subseteq f(\bar{y}) \in \gamma$;
7. if the maximal chain of proper upper bounds for x in γ is of length $\leq m$, and if $\gamma \not\models x \subseteq t$ for any ground term t, then $x \in \{x_1, \ldots, x_n\}$.

We say that γ' is a *lower bound completion* of γ if

- γ' is an m-level lower bound completion of γ for all m;
- for every variable x, either there exists a ground term t such that $\gamma \models x \subseteq t$, or there exists a term t_x such that $t_x \subseteq x \in \gamma'$, and t_x is unique for x, *i.e.*, $t_x \neq t_y$ for $x \neq y$;
- if $x \subseteq y \notin \gamma$, then $t_x \subseteq y \notin \gamma'$.

The next two lemmas lead to Proposition 11.

Lemma 9 Given a satisfiable constraint γ and a variable u without a proper upper bound in γ and a constant a_u not occurring in γ, the constraint γ' defined by $\gamma' = \gamma \cup \{a_u \subseteq u\}$ is satisfiable, and $\gamma \models x \subseteq y$ iff $\gamma' \models x \subseteq y$ (for all variables x and y).

Proof. We extend an arbitrary solution ν of γ to a valuation ν' as follows. We first introduce a new variable u' and put $\nu(u') = \nu(u) \cup \{a_u\}$. Then, for $v \in \mathcal{V}(\gamma)$ we set

$$\nu'(v) = \nu(v) \cup \bigcup\{\nu(\theta[u', \bar{z}]) \mid \gamma \models \theta[u, \bar{z}] \subseteq v\}.$$

Since $\gamma \models u \subseteq u$, we have that $\nu'(u) \supset \nu(u) \cup \{a_u\}$. If $\gamma \not\models \theta \subseteq v$ for any term θ containing an occurrence of u, then $\nu'(v) = \nu(v)$.

Clearly, ν' satisfies the constraint $a_u \subseteq u$. All other conjuncts of γ' are also in γ. We consider their different forms, namely

- $t \subseteq x$: $t \in \nu(x)$ and $\nu(x) \subseteq \nu'(x)$ imply that ν' satisfies $t \subseteq x$;
- $x \subseteq y$: Since $\nu(x) \subseteq \nu(y)$, and $\gamma \models \theta \subseteq x$ implies $\gamma \models \theta \subseteq y$, also $\nu'(x) \subseteq \nu'(y)$;
- $f(x_1, \ldots, x_n) \subseteq y$: Let $f(t_1, \ldots, t_n)$ be an arbitrary tree in $\nu'(f(x_1, \ldots, x_n)) = f(\nu'(x_1), \ldots, \nu'(x_n))$. We will define n terms $\theta_1, \ldots, \theta_n$ such that $\gamma \models f(\theta_1, \ldots, \theta_n) \subseteq y$ and $f(t_1, \ldots, t_n) \in \nu(f(\theta_1, \ldots, \theta_n)[u/u'])$.

  If $t_i \in \nu(x_i)$ then let $\theta_i = x_i$. Otherwise, if $t_i \in \nu'(x_i) - \nu(x_i)$, then let θ_i be such a term that $\gamma \models \theta_i \subseteq x_i$ and $t_i \in \nu(\theta_i[u/u'])$.

  Thus, $\gamma \models f(\theta_1, \ldots, \theta_n) \subseteq f(x_1, \ldots, x_n)$. Since $f(x_1, \ldots, x_n) \subseteq y \in \gamma$, we have that $\gamma \models f(\theta_1, \ldots, \theta_n) \subseteq y$. Thus, $f(t_1, \ldots, t_n) \in \nu(f(\theta_1, \ldots, \theta_n)[u/u'])$.
- $x \subseteq f(y_1, \ldots, y_n)$: Let $t \in \nu'(x)$. If $t \in \nu(x)$, then $t \in \nu(f(y_1, \ldots, y_n)) \subseteq \nu'(f(y_1, \ldots, y_n))$, and we are done.

  Suppose $t \in \nu'(x) - \nu(x)$. Then there exists a term $\theta[u, \bar{z}]$ such that $\gamma \models \theta[u, \bar{z}] \subseteq x$ and $t \in \nu(\theta[u', \bar{z}])$. Since $\gamma \models \theta[u, \bar{z}] \subseteq f(y_1, \ldots, y_n)$ and u has no proper upper bounds, $\theta[u, \bar{z}]$ cannot be equal to u. Since $t \notin \nu(x)$, $\theta[u, \bar{z}]$ must be a composed term of the form $f(\theta_1[u, \bar{z}], \ldots \theta_n[u, \bar{z}])$. Now $\gamma \models f(\theta_1[u, \bar{z}], \ldots, \theta_n[u, \bar{z}]) \subseteq f(y_1, \ldots, y_n)$, which implies $\gamma \models \theta_i[u, \bar{z}] \subseteq y_i$.

  This means that $\nu(\theta_i[u', \bar{z}]) \subseteq \nu'(y_i)$ and $\nu(\theta[u', \bar{z}]) \subseteq f(\nu'(y_1), \ldots, \nu'(y_n))$. Hence $t \in \nu'(f(y_1, \ldots, y_n))$.
- $f(\bar{u}) \subseteq f(\bar{v})$ (or $f(\bar{t}) \subseteq f(\bar{u})$): Since this conjunct is redundant in the closed constraint γ, its satisfaction follows from the satisfaction of the constraints $x \subseteq y$ (respectively $t \subseteq x$) shown above.

It is easy to see that $\gamma \models x \subseteq y$ iff $\gamma' \models x \subseteq y$. Namely, if $\gamma \models x \subseteq y$ then $\gamma' \models x \subseteq y$. If $\gamma \not\models x \subseteq y$ then there exists ν satisfying $\nu(x) \not\subseteq \nu(y)$, and, thus, $\nu'(x) \not\subseteq \nu'(y)$. □

Lemma 10 For every satisfiable and closed constraint γ and all natural numbers k and m, there exists an m-level lower bound completion γ' of γ such that:

- if the chain of upper bounds for x is of length $\leq m$, and if $\gamma \not\models x \subseteq t$ for any ground term t, then there exist k different ground terms $t_1, \ldots t_k$ such that $t_i \subseteq x \in \gamma'$;
- if the ground term t occurs in γ and $t \subseteq y \in \gamma'$, then also $t \subseteq y \in \gamma$.

Proof. The proof goes by induction on m. For $m = 0$, we obtain γ' by successively applying k times Lemma 9 to each variable in γ without a proper upper bound.

For the induction step, let u be a variable with the maximal chain of proper upper bounds of length equal to $m+1$ and such that $\gamma \not\models u \subseteq t$ for any ground term t.

Consider all proper upper bounds $u \subseteq f(\bar{v})$ for u. Note that by assumption u does have such bounds, and since γ is satisfiable, all of them must have the same function symbol f. Let $\bar{v}_u = (v_{u1}, \ldots, v_{un})$ be a sequence of fresh variables of the length equal to the arity of f, and let γ_c be the closure under the axioms in Table 1 of

$$\gamma \cup \{\bar{v}_u \subseteq \bar{v} \mid u \subseteq f(\bar{v}) \in \gamma\}.$$

It is easy to see that γ_c is satisfiable. Namely, if ν is a solution of γ, then we get solution of γ_c by putting $\nu(\bar{v}_u) = \bigcap_{u \subseteq f(\bar{v}) \in \gamma} \nu(\bar{v})$.

All variables in $\bar{v}_u$ have the chain of proper upper bounds of length at most m, and there exists a variable v_{uj} in the sequence $\bar{v}_u$ such that $\gamma_c \not\models v_{uj} \subseteq t$ for any ground t.

Thus, by the induction hypothesis, there exists an m-level lower bound completion φ of γ_c with k different lower bounds for v_{uj}.

If there exists a ground t such that $\gamma \models v_{ui} \subseteq t$, set $t_i = t$; otherwise, let t_i be any lower bound for v_{ui} in φ (for $i = 1, \ldots, n$). Now we prove that φ' is satisfiable, where

$$\varphi' = \varphi \cup \{f(t_1, \ldots, t_n) \subseteq u\}.$$

Let ν be any solution of φ. Similarly as we did it in Lemma 9, we introduce a new variable u' and put $\nu(u') = \nu(u) \cup \{f(t_1, \ldots t_n)\}$. We then define

$$\nu'(y) = \nu(y) \cup \bigcup \{\nu(\theta[u', \bar{z}]) \mid \gamma \models \theta[u, \bar{z}] \subseteq y\}.$$

The proof that ν' satisfies the constraints in φ' is essentially the same as in Lemma 9, except for the case of constraints of the form $x \subseteq f(y_1, \ldots y_n)$, where $\theta[u, \bar{z}]$ can be equal to u. But then $t \in \nu(u') - \nu(u)$, that is, $t = f(t_1, \ldots t_n)$. Since $\varphi \models \theta[u, \bar{z}] \subseteq x$ and $\theta[u, \bar{z}] = u$ and $x \subseteq f(\bar{y}) \in \varphi$, we have $\varphi \models u \subseteq f(\bar{y})$. Hence $f(\bar{y})$ is an upper bound for u, and by condition 5 of Definition 8, $(t_1, \ldots, t_n) \in \nu(\bar{y})$. Therefore $t = f(t_1, \ldots, t_n) \in \nu(f(\bar{y})) \subseteq \nu'(f(\bar{y}))$. This means that the constraints of the form $x \subseteq f(\bar{y})$ are satisfied under ν'.

Now, to obtain the constraint γ' satisfying the induction statement, we first repeat the same construction for k different lower bounds t_j of v_{uj} (which exist by the induction hypothesis) and for all variables u with the chain of proper upper bounds of length $m+1$. We then "remove" all occurrences of the variables $\bar{v}_u$ which we have added for the construction of γ_c. Namely, first, for each such variable v_{ui}, each term t and each variable y, if $t \subseteq v_{ui} \in \gamma'$ and $v_{ui} \subseteq y \in \gamma'$, then we add to γ' the constraint $t \subseteq y$. Second, we remove from γ' all constraints with an occurrence of this variable.

We still have to prove that γ' satisfies the conditions in Definition 8. All of them except 3 and 6 are either easy or already proven.

Ad 3. Clearly, if $x, y \in \mathcal{V}(\gamma)$ then $\gamma \models x \subseteq y$ iff $\varphi' \models x \subseteq y$. This is because if $\nu(x) \not\subseteq \nu(y)$ then $\nu'(x) \not\subseteq \nu'(y)$, and if $\gamma \models x \subseteq y$ then $\varphi' \models x \subseteq y$. It is also easy to see that $\gamma' \models x \subseteq y$ iff $\varphi' \models x \subseteq y$.

Ad 6. If $f(\bar{t}) \subseteq u$ was added to γ' then $\bar{t} \subseteq \bar{y} \in \gamma'$ iff $\bar{v}_u \subseteq \bar{y} \in \gamma_c$. This implies that each variable in $\bar{y}$ occurs in some upper bound $u \subseteq f(\bar{v})$ and, thus, $u \subseteq f(\bar{y}) \in \gamma$ by Axiom 3.

To prove the second statement of the lemma, let t occur in γ and $t \subseteq y \in \gamma'$. The only possibility that $t \subseteq y$ occurs in $\gamma' - \gamma$ is the following. There must exist a

variable u such that $t \subseteq v_{u_i} \in \gamma_c$ for some i and $v_{u_i} \subseteq y \in \gamma_c$. This is possible only if $u \subseteq f(\ldots,t,\ldots)$ occurs implicitly in γ and $u \subseteq f(\ldots,y,\ldots) \in \gamma$. But then, by Axiom 6, $t \subseteq y \in \gamma$. □

Proposition 11 There exists a lower bound completion for every satisfiable and closed constraint γ.

Proof. Let m be the maximal length of a chain of proper upper bounds in γ. We modify the m-level lower bound completion constructed in Lemma 10 in order to satisfy the second and the third condition in the definition of a lower bound completion.

We can choose a term t_x that is a unique lower bound for x if we choose the parameter k in the induction statement greater or equal to the number of variables occurring in γ.

In order to obtain that $t_x \subseteq y \notin \gamma'$ if $x \subseteq y \notin \gamma$, we remove all constraints from γ' that are of the form $t \subseteq y$ where $t \notin \{t_z \mid z \subseteq y \in \gamma\}$ and t is not a subterm of t_z for some other variable z. This is possible if we choose the parameter k in the induction statement greater or equal to the square of the number of variables occurring in γ. □

Constructing a minimal solution. The next lemma explains why the completion of a constraint γ with ground lower-bounds applies only to variables x where $\gamma \not\models x \subseteq t$ (see Condition 7 in Definition 8) if γ is closed (in particular under Axiom 5).

Lemma 12 Let γ be a satisfiable and closed constraint and let t be a ground term. If $\gamma \models x \subseteq t$, then $x \subseteq t \,\widehat{\in}\, \gamma$.

Proof. Note that if x has no proper upper bound, then $\gamma \not\models x \subseteq t$ (this follows from the results in [25], saying that the maximal solution for γ assigns the set of all trees to x, or from Proposition 9). Therefore, x does have proper upper bounds in γ, and since γ is satisfiable, the head function symbol in t must coincide with the head function symbol of all proper upper bounds of x.

Now the proof goes by induction on the structure of t. If t is a constant symbol, then the only possible proper upper bound of x is $x \subseteq t$, and we are done.

If $t = f(t_1,\ldots,t_n)$ then let $f(u_{11},\ldots,u_{1n}),\ldots,f(u_{m1},\ldots,u_{mn})$ be the list of all proper upper bounds of x. For each argument position $i \in \{1,\ldots,n\}$ let us consider all the variables $u_{1i},\ldots,u_{mi}$. For at least one of them, say $u_{j_i i}$, we have $\gamma \models u_{j_i i} \subseteq t_i$ (if this is not the case, then $\gamma \not\models x \subseteq t$). By the induction hypothesis we have that $u_{j_i i} \subseteq t_i$ occurs implicitly in γ. By Axiom 3, $x \subseteq f(u_{j_1 1},\ldots,u_{j_n n}) \in \gamma$ and, thus, also $x \subseteq t \,\widehat{\in}\, \gamma$. □

Definition 13 Let γ' be a lower bound completion of the satisfiable and closed constraint γ, let γ'' be a closure of γ' under transitivity. Let ν be the minimal (under pointwise inclusion) valuation satisfying, for all finite trees t, the condition: $t \in \nu(x)$ iff

- $t \subseteq x \in \gamma''$, or
- $t = f(t_1,\ldots,t_n)$, $f(x_1,\ldots,x_n) \subseteq x \in \gamma''$, and $t_i \in \nu(x_i)$ for all i.

We call ν *the minimal solution* of γ''.

Our terminology is justified by the lemma below.

Lemma 14 In the situation of Definition 13, the valuation ν satisfies γ''.

Proof. First note that ν assigns nonempty sets of terms to variables. This is because if $\gamma \models x \subseteq t$ for some ground term t then, by Lemma 12 and Axiom 5, $t \in \nu(x)$, and if $\gamma \not\models x \subseteq t$ for any ground term t then, by Definition 8, γ' contains constraint $t_x \subseteq x$.

The constraints in $\gamma'' - \gamma$ are of the form $t \subseteq \tau$ for a ground term t. Thus, they are satisfied by the definition of ν. We now show that all constraints in γ are satisfied.

- If $x \subseteq y \in \gamma$, then, for $t \in \nu(x)$, there are two possibilities:
  - $t \subseteq x \in \gamma''$. Then $t \subseteq y \in \gamma''$ since γ'' is closed under transitivity of $\subseteq$, and, thus, $t \in \nu(y)$;
  - t is of the form $t = f(t_1, \ldots, t_n)$ and $f(x_1, \ldots, x_n) \subseteq x \in \gamma$ and $t_i \in \nu(x_i)$. Thus, again by transitivity, $f(x_1, \ldots, x_n) \subseteq y \in \gamma''$, and $t \in \nu(y)$.
- If $f(x_1, \ldots, x_n) \subseteq x \in \gamma$, then $t_i \in \nu(x_i)$ implies, by the definition of ν, that $f(t_1, \ldots, t_n) \in \nu(x)$.
- If $x \subseteq f(x_1, \ldots, x_n) \in \gamma$, then a tree $t \in \nu(x)$ cannot be of the form $g(t_1, \ldots, t_m)$ with $f \neq g$, since then γ would not be satisfiable. Thus, t must be of the form $f(t_1, \ldots, t_n)$. There are two possibilities:
  - $t \subseteq x \in \gamma''$. If $t \subseteq x \in \gamma$ then by closeness of γ we have $t_i \subseteq x_i \in \gamma$. If $t \subseteq x \in \gamma'' - \gamma$, then $t_i \subseteq x_i \in \gamma''$ by Condition 5 of Definition 8. Thus, in both cases $t \in f(\nu(x_1), \ldots, \nu(x_n))$.
  - $f(y_1, \ldots, y_n) \subseteq x \in \gamma''$ and $t_i \in \nu(y_i)$. Then, since $f(y_1, \ldots, y_n)$ is not a ground term, $f(y_1, \ldots, y_n) \subseteq x \in \gamma$. Since γ is closed, $y_i \subseteq x_i \in \gamma$. Thus, $t_i \in \nu(x_i)$ and $t \in f(\nu(x_1), \ldots, \nu(x_n))$.
- If $f(\bar{u}) \subseteq f(\bar{v}) \in \gamma$ (or, if $f(\bar{t}) \subseteq f(\bar{u}) \in \gamma$), then, since this conjunct is redundant in the closed constraint γ, its satisfaction follows from the satisfaction of the constraints $x \subseteq y$ (respectively $t \subseteq x$) shown above. □

Lemma 15 In the situation of Definition 13, if t is a ground term occurring in $\gamma' - \gamma$ and $t \in \nu(y)$, then either $x \subseteq y \in \gamma$ and $t \subseteq x \in \gamma'$, or $t \subseteq y \in \gamma'$.

Proof. We first recall some basic facts. All constraints in $\gamma'' - \gamma$ have ground terms on the left-hand side of the inclusion. Thus, if $\tau \subseteq \tau' \in \gamma''$ and τ is not ground, then $\tau \subseteq \tau' \in \gamma$. Furthermore, the constraints in $\gamma' - \gamma$ have ground terms on the left-hand side of the inclusion and variables (*i.e.*, not composed terms) on the right-hand side.

The proof of the lemma goes by induction on the definition of the minimal solution ν of γ''. Let t be a ground term occurring in $\gamma' - \gamma$ such that $t \in \nu(y)$. Then, by the definition of ν, either $t \subseteq y \in \gamma''$ or $t = f(t_1, \ldots, t_n)$, $t_i \in \nu(x_i)$ and $f(x_1, \ldots, x_n) \subseteq y \in \gamma''$.

Induction base: $t \subseteq y \in \gamma''$. Then, since γ'' is a transitive closure of γ', either $t \subseteq y \in \gamma'$ (and we are done), or there exists a variable x such that $t \subseteq x \in \gamma'$ and $x \subseteq y \in \gamma''$. Since x is a variable, $x \subseteq y \in \gamma$.

Induction step: $t = f(t_1, \ldots, t_n)$, where $t_i \in \nu(x_i)$ and $f(x_1, \ldots, x_n) \subseteq y \in \gamma''$. Since $f(x_1, \ldots, x_n)$ is not ground, $f(x_1, \ldots, x_n) \subseteq y \in \gamma$. By the induction hypothesis,

for all i, we have either $t_i \subseteq x_i \in \gamma'$ or there exists a variable u_i such that $t_i \subseteq u_i \in \gamma'$ and $u_i \subseteq x_i \in \gamma$.

Let x be a variable such that $t \subseteq x \in \gamma'$. By Condition 6 of Definition 8, if $t_i \subseteq y_i \in \gamma'$ then $f(y_1, \ldots, y_n)$ is an upper bound for x in γ. Let $y_i = x_i$ if $t_i \subseteq x_i \in \gamma'$, and $y_i = u_i$ if $t_i \subseteq u_i \in \gamma'$. Then, for all i, we have $y_i \subseteq x_i \in \gamma$. Thus, by Axiom 2, $f(y_1, \ldots, y_n) \subseteq f(x_1, \ldots, x_n) \in \gamma$, and by transitivity $x \subseteq y \in \gamma$. □

Corollary 16 In the situation of Definition 13, we have: $t_x \in \nu(y)$ iff $x \subseteq y \in \gamma$.

Lemma 17 In the situation of Definition 13, if t is a ground term that occurs in γ and $t \in \nu(x)$, then $t \subseteq x \in \gamma$.

Proof. The proof goes by induction on the structure of t. If t is a constant symbol then $t \subseteq x \in \gamma''$ from the definition of ν, and the thesis of the lemma follows from Lemma 10.

Now suppose $t = f(t_1, \ldots, t_n)$. If $t \subseteq x \in \gamma''$ then, again by Lemma 10, $t \subseteq x \in \gamma$ and we are done. Otherwise, $t_i \in \nu(x_i)$ and $f(x_1, \ldots, x_n) \subseteq x \in \gamma$. Then, by the induction hypothesis, $t_i \subseteq x_i \in \gamma$. Since $f(t_1, \ldots, t_n)$ occurs in γ and γ is closed (in particular under Axiom 2), $f(t_1, \ldots, t_n) \subseteq f(x_1, \ldots, x_n) \in \gamma$. Thus, by transitivity, $f(t_1, \ldots, t_n) \subseteq x \in \gamma$. □

Proof of Theorem 2 (Entailment). The "if" direction of the proof is obvious. For the "only if" direction, let γ', γ'' and ν be as in Definition 13. We assume $x \subseteq y \notin \gamma$.

- If there exists a ground term t such that $\gamma \models x \subseteq t$, then $x \subseteq t \,\hat{\in}\, \gamma$ by Lemma 12. Since $x \subseteq y \notin \gamma$, by Axiom 4 we have $t \subseteq y \notin \gamma$. This implies $t \subseteq y \notin \gamma''$. Now Lemma 17 yields that $t \notin \nu(y)$. Thus, $\nu(x) \not\subseteq \nu(y)$. Since ν is a solution of γ, $\gamma \not\models x \subseteq y$.

- Otherwise (*i.e.*, there exists no ground term t such that $\gamma \models x \subseteq t$), we have $t_x \subseteq x \in \gamma''$. Corollary 16, together with our assumption $x \subseteq y \notin \gamma$, yields that $t_x \in \nu(x) - \nu(y)$. This means that $\gamma'' \not\models x \subseteq y$. Since each solution of γ'' is also a solution of γ, we have $\gamma \not\models x \subseteq y$. □

Proof of Theorem 1 (Independence). The "only if" direction is obvious. For the proof of the "if" direction, let γ', γ'' and ν be as in Definition 13. By Remark 5 and Theorem 2 we have that γ'' does not contain $x_i \subseteq y_i$ for any i. Thus, $\nu(x_i) \not\subseteq \nu(y_i)$ for all i. That is, ν is a solution of $\gamma \cup \bigcup_i \{x_i \not\subseteq y_i\}$. □

5 Conclusion

We have shown, for the first time, the independence property for a natural (and practically used) class of set constraints. We have also given a polynomial entailment test. Together, this yields a polynomial satisfiability test for conjuncts with negation.

The main interest of our results is a fundamental one. As for applications, we still have to investigate how exactly our results can help to make more precise the analysis of concurrent constraint programs, which is from where this class of set constraints originates.

As pointed out in [25], a potential application of the constraint system INÈS lies in its use as an instance of the CLP($\mathcal{X}$) scheme [16]. Our results are relevant for the semantics of CLP(INÈS) programs (see [22]) and imply that one can manipulate negated constraints in the same way as inequations in Prolog-II [9]. Having presented an incremental entailment test, we may conceive to also use INÈS in a CCP language, *e.g.*, in Oz.

We believe that there are many other interesting classes of set constraints with the independence property. Let us consider, for example, the extension of INÈS constraints where terms may be formed with the union operator (see Footnote 3). We conjecture that if we exclude the cases where a variable x is contained in a finite union of terms with ground subterms, then the independence property holds.

References

[1] A. Aiken, D. Kozen, M. Vardi, and E. L. Wimmers. The complexity of set constraints. In *1993 Conference on Computer Science Logic*, LNCS 832, pp. 1–17, Sept. 1993.

[2] A. Aiken, D. Kozen, and E. L. Wimmers. Decidability of systems of set constraints with negative constraints. Technical Report 93-1362, Computer Science Department, Cornell University, June 1993.

[3] A. Aiken, E. Wimmers, and T. Lakshman. Soft typing with conditional types. In *Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, Jan. 1994.

[4] H. Aït-Kaci and A. Podelski. Entailment and disentailment of order-sorted feature constraints. In A. Voronkov, editor, *Fourth International Conference on Logic Programming and Automated Reasoning*, Springer LNAI 698, pp. 1–18. Springer-Verlag, July 1993.

[5] H. Aït-Kaci, A. Podelski, and G. Smolka. A feature-based constraint system for logic programming with entailment. *Theoretical Computer Science*, 122(1–2):263–283, 1994.

[6] L. Bachmair, H. Ganzinger, and U. Waldmann. Set constraints are the monadic class. In *Eighth Annual IEEE Symposium on Logic in Computer Science*, pp. 75–83, 1993.

[7] W. Charatonik and L. Pacholski. Negative set constraints with equality. In *Ninth Annual IEEE Symposium on Logic in Computer Science*, pp. 128–136, 1994.

[8] W. Charatonik and L. Pacholski. Set constraints with projections are in NEXPTIME. In 35^{th} *Symposium on Foundations of Computer Science*, pp. 642–653, 1994.

[9] A. Colmerauer. Equations and inequations on finite and infinite trees. In *2nd International Conference on Fifth Generation Computer Systems*, pp. 85–99, 1984.

[10] T. Frühwirth, E. Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Sixth Annual IEEE Symposium on Logic in Computer Science*, pp. 300–309, July 1991.

[11] R. Gilleron, S. Tison, and M. Tommasi. Solving systems of set constraints with negated subset relationships. In 34^{th} *Symposium on Foundations of Computer Science*, pp. 372–380, 1993.

[12] N. Heintze. Set based program analysis. PhD thesis, School of Computer Science, Carnegie Mellon University, 1992.

[13] N. Heintze. Set based analysis of ML programs. Technical Report CMU–CS–93–193, School of Computer Science, Carnegie Mellon University, July 1993.

[14] N. Heintze and J. Jaffar. A decision procedure for a class of set constraints (extended abstract). In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pp. 42–51, 1990.

[15] R. Helm, K. Marriott, and M. Odersky. Constraint-based query optimization for spatial databases. In *Tenth ACM Symposium on the Principles of Database Systems*, pp. 181–191, Denver, CO, May 1991.

[16] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *14th ACM Conference on Principles of Programming Languages*, pp. 111–119, Munich, Germany, Jan. 1987. ACM.

[17] P. C. Kanellakis, G. M. Kuper, and P. Z. Revesz. Constraint query languages. *Journal of Computer and System Sciences*, 51:26–52, 1995.

[18] D. Kozen. Logical aspects of set constraints. In *1993 Conference on Computer Science Logic*, LNCS 832, pp. 175–188, Sept. 1993.

[19] J. Lassez and K. McAloon. Applications of a canonical form for generalized linear constraints. In *International Conference on 5th Generation Computer Systems*, pp. 703–710, Tokyo, Japan, Dec. 1988.

[20] J.-L. Lassez and K. McAloon. A constraint sequent calculus. In *Fifth Annual IEEE Symposium on Logic in Computer Science*, pp. 52–61, June 1990.

[21] M. J. Maher. Logic semantics for a class of committed-choice programs. In J.-L. Lassez, editor, *Logic Programming, Fourth International Conference*, pp. 858–876, Cambridge, MA, 1987. The MIT Press.

[22] M. J. Maher. A logic programming view of CLP. In D. S. Warren, editor, *10th International Conference on Logic Programming*, pp. 737–753, Budapest, Hungary, June 1993. The MIT Press.

[23] M. J. Maher. Constrained dependencies. In U. Montanari, editor, *First International Conference on Principles and Practice of Constraint Programming (CP'95)*, LNCS 976, pp. 170–185, Cassis, France, 19–22 Sept. 1995. Springer-Verlag.

[24] M. Müller. *Type Analysis for a Higher-Order Concurrent Constraint Language.* Doctoral Dissertation. Universität des Saarlandes, Technische Fakultät, 66041 Saarbrücken, Germany, 1996. In preparation.

[25] M. Müller, J. Niehren, and A. Podelski. Inclusion constraints over non-empty sets of trees. Submitted for publication.

[26] V. A. Saraswat. *Concurrent Constraint Programming.* The MIT Press, Cambridge, MA, 1993.

[27] G. Smolka. The Oz Programming Model. In *Volume 1000 of LNCS.* Springer-Verlag, Berlin, Germany, 1995.

[28] G. Smolka and R. Treinen. Records for Logic Programming. *The Journal of Logic Programming*, 18(3):229–258, Apr. 1994.

Speeding Up Constraint Propagation By Redundant Modeling

B.M.W. Cheng, J.H.M. Lee, and J.C.K. Wu

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
Email: {mwcheng,jlee,ckwu}@cs.cuhk.edu.hk
Fax: (+852) 26035024

Abstract. The paper describes a simple modeling and programming approach for speeding up constraint propagation. The idea, although similar to redundant constraints, is based on the concept of redundant modeling. We define CSP model and model redundancy formally, and show how mutually redundant models can be combined and connected using channeling constraints. The combined model contains the original but redundant models as sub-models. Channeling constraints allow the sub-models to cooperate during constraint-solving by propagating constraints freely amongst the sub-models. This extra level of pruning and propagation activities becomes the source of execution speedup. We apply our method to the design and construction of a real-life nurse rostering system. Experimental results provide empirical evidence in line with our prediction.

Keywords: Constraint Propagation, Redundant Modeling, Nurse Rostering

1 Introduction

The problem at hand is that of *constraint satisfaction problems* (CSP) defined in the sense of Mackworth [16], which can be stated briefly as follows:

> We are given a set of variables, a domain of possible values for each variable, and a conjunction of constraints. Each constraint is a relation defined over a subset of the variables, limiting the combination of values that the variables in this subset can take. The goal is to find a *consistent* assignment of values to the variables so that all the constraints are satisfied simultaneously.

CSP's are, in general, NP-complete and some are even NP-hard [5]. Thus, a general algorithm designed to solve any CSP will necessarily require exponential time in problem size in the worst case. One widely-adopted approach in solving CSP's features various degrees of combinations of backtracking tree search and constraint propagation [19, 12]. This framework is realized in the constraint logic programming languages CHIP [8] and C++ class library ILOG SOLVER [11],

as many mutually redundant models as one can dream up. One must, however, take into account the time and memory overhead of larger network size in the combined model. In addition, implementing an alternate model is not a task that can be taken lightly.

Another common and important question to ask is how to generate a redundant model. We observe that most real-life scheduling and resource allocation problems can be modeled reciprocally in the sense that if a problem can be modeled as assigning objects of type X to objects of type Y, then the same problem can also be modeled as assigning objects of type Y to those of type X. This principle is general and widely applicable. At the beginning of this section, we have already given an example of applying the principle to a generic job-shop scheduling problem. Similarly, we can model a train driver rostering problem as either assigning trips to drivers or assigning drivers to trips. We can also model a school timetabling problem as either assigning courses to time slots in the timetable or assigning a set of time slots to courses. We can find more examples of this types. It may be argued that the above resembles the resource-centered view and the job-centered view used in the scheduling community. The novelty of our proposal is not on suggesting multiple perspectives on the same problem. The novelty is on suggesting to connect these perspectives to form a combined perspective which exhibits more efficient pruning behaviour.

Using reciprocal views of problems is just one way of obtaining redundant models but it is not the only way. The general guideline is to study the problem at hand from different angles and perspectives. For example, Tsang [21] suggests two ways of modeling the 8-queens problem. The first and familiar model consists of eight variables, each of which denotes the position of a queen in a different row of the chess board. Every domain contains the eight possible positions of a queen. The alternative model consists also of eight variables, each of which denotes the position of the queen on either one of the sixty-four squares on the chess board. Thus, the domain of each variable becomes $\{1, \ldots, 64\}$. It is not difficult to come up with yet another model consisting of sixty-four variables, each of which has domain $\{0, 1\}$. Each variable denotes a square on the board. A value of 0 denotes an empty square and a value of 1 denotes a square occupied by a queen.

Channeling constraints for models M_1 and M_2 must be able to propagate constraints from M_1 to M_2 and vice versa. Suppose M_1 is modeled after assigning objects of type X to those of type Y and M_2 is modeled after assigning objects of type Y to those of type X. An effective channeling constraint is of the form:

> The variable associated with object x of type X has object y of type Y as value *if and only if* the variable associated with y has x as value.

This form of constraint is simple and can be generated systematically.

Note that redundant constraints and redundant models are orthogonal concepts although their working principles are similar. Given two models $M_1 = \langle \mathcal{X}_1, \mathcal{F}_{\mathcal{X}_1}, \mathcal{C}_{\mathcal{X}_1} \rangle$ and $M_2 = \langle \mathcal{X}_2, \mathcal{F}_{\mathcal{X}_2}, \mathcal{C}_{\mathcal{X}_2} \rangle$ of a problem P. The constraints $\mathcal{C}_{\mathcal{X}_1}$ and $\mathcal{C}_{\mathcal{X}_2}$ have no relationship to each other at all since they do not even share

variables. Within a model, say M_1 (or M_2), we can add a redundant constraint c such that c is entailed by $\mathcal{C}_{\mathcal{X}_1}$ (or $\mathcal{C}_{\mathcal{X}_2}$).

3 A Case Study

Our task is to design and implement a nursing staff rostering system for the Ambulance and Emergency Unit (AEU) of the Tang Shiu Kin Hospital (TSKH) in Hong Kong. The AEU provides daily 24-hour emergency services to the general public seven days a week. Therefore, AEU nurses have to work in shifts. The main function of the rostering system is to roster the nurses in such a way that steady and high-quality services are provided to the community, taking into account (1) professional rules, (2) preferential rules, (3) pre-arranged duties, and (4) pre-arranged preferred shifts. Other than having to obey all professional rules, fairness is an important measure of the quality of the generated roster. Every nurse must be assured of equal chance in taking night shifts, having day-offs on weekends or the actual public holidays, *etc*, although most fairness rules are in the form of soft constraints. The main difficulties of the process arise from pre-arranged duties, vacation leaves, and pre-arranged preferred shifts requested by the nurses, which often break the regularities of wanted (or unwanted) duties. In the following, we give an overview of the nurse rostering system. Readers interested in how we handle fair rotation of want and unwanted shifts and soft constraints are referred to [4].

There are three basic shifts in a day: namely AM shift (A), PM shift (P), and night shift (N). An evening shift (E) is essentially a PM shift with a slightly different duty time. An irregular shift (I) has special working hours either arranged by the nursing officer or requested by individual nurses. Other shift types concern holidays and special duties. Nurses can take several different types of holiday. These include day-off (O), compensation-off (CO), public holiday (PH) and vacation leave (VL). In addition, nurses can be pre-assigned some special work shifts such as study day (SD) and staff-on-loan (SOL). In total, there are eleven shift types.

A weekly duty roster for week i should be generated about two weeks prior to i. The rostering process assigns, for each nurse, a work shift for Monday to Sunday, taking into account of the nurse's past rostering history. A sample roster duty sheet is shown in figure 1. There are two steps for a duty planner to complete a duty roster. First, the planner has to collect information about pre-assigned shifts and shift requests from the nurses. Second the planner generates the remaining shift slots, observing all planning rules and nurse preference rules.

There are two types of planning rules: imperative planning rules and preference planning rules. Imperative planning rules are rules that must be respected in any timetable. Therefore, in generating the roster, the duty planner must ensure that every planning decision made is coherent with these hard rules. There are totally six imperative rules. Some examples rules are:

Rule 1 *Each staff is required to work one shift per day.*

Nurse	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Nurse-01	E	A	O	P	A	N	PH
Nurse-02	VL	VL	VL	VL	VL	VL	O
Nurse-03	A	CO2	P	A	N	O	E
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Fig. 1. A Sample Roster Duty Sheet

Rule 2 *Each staff get one day-off per week.*

Preference planning rules are optional in the sense that they should be satisfied as much as possible. Violating preference rules, however, does not destroy the validity of a roster. Therefore, preference rules corresponds to soft constraints. A soft constraint c can be viewed as a disjunctive constraint $c \cup \texttt{true}$. Following Van Hentenrcyk [22], we model disjunctive constraints as choices. While choice is easy to implement in a constraint programming language, accumulation of soft constraints can result in a huge search tree. While the preference rules are not professional rules, they do reflect the preference of nurses in general. Therefore, the number of preference rules satisfied is a good measure of the quality of a generated roster. There are totally ten preference rules[3]. Some examples rules are:

Rule 3 *A nurse should not work for the same shift for two consecutive days. In particular, a nurse prefer alternating A shift or P/E shift. If such an arrangement is impossible, a nurse also accept two consecutive A or P shift. However, under no circumstance will three consecutive A or P/E shifts be accepted by a nurse.*

Rule 4 *Apart from all holiday schedule, the number of A shifts and P shifts allocated to each nurse each week should be balanced.*

4 Modeling

In this section, we describe two models for the nurse rostering problem. The first model is based on the format of the roster sheet, which suggests allocation of

[3] Two of the ten preference rules cannot be expressed as constraints. They are implemented algorithmically. The remaining eight rules are numbered C2.1, C2.2, C2.3a, C2.5, C2.6, C2.7, C2.8, and C2.9 respectively. Interested readers are referred to [4] for details.

shifts to nurses. Implementation of the first model performs well in general but fails to return answer in a timely manner for some difficult cases. This prompts our work on the second model, which is an allocation of nurses to shifts. Implementation of the second model again exhibits deficiency on some problem instances while performing well in general. Having two mutually redundant models in hand, we proceed to connect the models using channeling constraints.

4.1 Model One

In a roster sheet, each row consists of seven slots, holding the work shifts assigned to a nurse in a scheduled week. Each nurse occupies a row in the roster sheet. It is thus natural to model the slots *Nurse-DayOfWeek* on the sheet as constrained variables, each of which is associated with a domain of eleven possible shift types {A,P,N,E,I,O,CO,PH,VL,SD,SOL}. If there are n nurses in the AEU, then there will be $7n$ variables in the rostering system.

Under this formulation, we have rule 1 satisfied for free since the rule is implicitly satisfied by requiring each constrained variable to take on exactly one value. Rule 2 can be modeled using a counting constraint [11] as follows:

For each nurse `Nurse`*, the number of variables in the set*

$$\{\texttt{Nurse-Mon}, \texttt{Nurse-Tue}, \ldots, \texttt{Nurse-Sun}\}$$

assigned with the day-off (O) shift must be equal to one.

Preference planning rules, or soft constraints, are expressed in the same manner as imperative planning rules. There is, however, one significant difference in how they are posted to the constraint-solving engine. For each soft constraint c, we set up a choice point. In the first branch, the constraint c is told to the solver. The other branch, one without c, is tried upon backtracking.

4.2 Model Two

Model two regards the rostering process as assigning nurses to serve in the eleven shifts in each day of a week. Since there may be more than one nurse working in one shift of a day, the variables take on sets of nurses as values. This kind of variables are called *constrained set variables* [11, 9]. Thus, we model each shift of a day *Shift-DayOfWeek* as constrained variables, each of which has as domains the power set of the set of all nurses. Regardless of the number of nurses in the AEU, the rostering system contains 77 (11×7) variables. If there are n nurses in the AEU, the size of the domain of each variable is 2^n.

The expression of the planning rules is now based on set operations and constraints. For example, rule 1 is now expressed as the constraints:

For each day `Day`*, consider the set of variables*

$$V = \{\texttt{A-Day}, \texttt{P-Day}, \texttt{N-Day}, \texttt{E-Day}, \ldots, \texttt{VL-Day}, \texttt{SD-Day}, \texttt{SOL-Day}\}.$$

The following two constraints must be satisfied: (1) Elements of V are pairwise disjoint and (2) the cardinality of $\bigcup V$ is equal to the number of nurses.

Similarly, rule 2 is expressed as:

Consider the set

$$V = \{\texttt{O-Mon}, \texttt{O-Tue}, \ldots, \texttt{O-Sun}\}.$$

The following two constraints must be satisfied: (1) Elements of V are pairwise disjoint and (2) the cardinality of $\bigcup V$ is equal to the number of nurses.

Again, the soft constraints in this model is represented similarly as imperative constraints, and are posted to the solving engine using choice points.

4.3 Combining the Models

The combined model contains model one and model two as sub-models. In addition, a set of channeling constraints are used to relate variables in the two models so that constraints can be propagated between the two models. The channeling constraints are of the following form:

`Nurse-DayOfWeek == Shift` *if and only if* `Nurse` $\in$ `Shift-DayOfWeek`.

It is worth noting that the constraints are generated mechanically using a simple double `for`-loop in our implementation. In general, there are $7mn$ channeling constraints for m nurses and n shift types for a weekly roster.

5 Implementation and Preliminary Results

To verify the correctness and effectiveness of our modeling, we have implemented the three models on personal computer (PC), which is chosen over workstation for PC's wide availability. Our prototypes consist of a user-interface and a rostering engine. The former is implemented in Microsoft Visual Basic 3.0 while the latter is realized in C++ with ILOG Solver library 3.0 [11]. Class constraints [11] of ILOG Solver facilitate our object-oriented design and greatly reduce our coding effort.

Implementations of model one, model two and the combined model consist of 6015, 4523, and 8656 lines of C++ code respectively. This little difference in code size is a result of the fact that constraint expressions occupy relatively few lines as compared to coding for I/O, data structure definition, *etc.* Therefore, much coding can be shared and combined in implementing the combined model, resulting in no significant increase in code size. *Most important of all, we implement model two and the combined model in four man-weeks, whereas model one is implemented in four man-months.* It takes longer time in the first implementation since we spend much time in problem understanding, correspondence

with users, experimentation with ideas and modification. The time used for the second and third implementations is purely devoted to coding.

Our real-life benchmark data contains 27 nurses and 11 shift types. The rostering system produces weekly roster sheet[4]. The implementation of model one generates 3884 constraints, out of which 266 are imperative, during execution. Executing the model two implementation results in 1827 constraints, out of which 1675 are imperative. These unbalanced figures reflect two phenomena. First, some types of constraints are more expressive and concise than others. Second, some planning rules that are easier to express in one model can be tedious to formulate in another model. Finally, the combined model consists of, in addition to the constraints from models one and two, 2079 channeling constraints.

We employ the simple smallest-domain-first variable-ordering heuristics and the smallest-first value-ordering heuristics in the variable labeling process. In most case, the prototypes can return a solution within 5 seconds running on a Pentium PC. In the following, we present the timing result of the three models for a particularly difficult problem instance. Figure 2 shows the preset requests of this particular week's roster, which should partially explain why this problem instance is difficult. In this week's roster assignment, 17 out of the 27 nurses request for preset shifts. Among them, 5 nurses request the day-off (O) shift on Sunday, which is most wanted by other nurses as well. 2 nurses request vacation leave (VL), which extends across the entire week. Last but not least, one nurse is assigned special duty (SD) from Monday to Friday and requests for O on Sunday. As a result, 47 out of 189 slots are preset before the roster can be generated. Some of the preset slots are filled in with the most undesirable (from the rostering point of view) patterns, making this week's roster exceptionally hard to generate.

The number of preference rules posted and the nature of the preference rules posted also affects the difficulty of the problem. For the same set of preset requests, we vary the combination of preference rules posted and show the performance of the three models in figure 3. Column one of each row specifies the combination of preference rules imposed. Column two and three contains the timings of models one and two respectively. The string "short" means that execution finishes within 1 minute. The string "fast" means an almost instantaneous response and "long" means that the system does not return an answer within 30 mins. Since the combined model contain models one and two as sub-models, labeling variables of either model will instantiate also variables in the other model. Column four shows the timing result of the combined model by labeling only variables in model one. Alternatively, we can label only variables in model two and yield the results in column five. Experimental results confirm that the combined model exhibits significant speedup over either model one or two.

[4] Generating more than one week's roster can be achieved by running the system multiple times.

Nurse	Mon	Tue	Wed	Thu	Fri	Sat	Sun
Nurse-01							
Nurse-02	A	O					
Nurse-03		A	P			A	
Nurse-04							
Nurse-05		P	SD	SD			O
Nurse-06							A
Nurse-07							
Nurse-08		A		A			O
Nurse-09	VL	VL	VL	VL	VL	VL	
Nurse-10					O	P	
Nurse-11							
Nurse-12						A	O
Nurse-13	SD	SD	SD	SD	SD		O
Nurse-14			O	SD		A	
Nurse-15		A					
Nurse-16	VL	VL	VL	VL	VL	VL	
Nurse-17							
Nurse-18							
Nurse-19							
Nurse-20						O	
Nurse-21		SD	A				
Nurse-22							
Nurse-23							
Nurse-24		SD					
Nurse-25						O	
Nurse-26							
Nurse-27	A		SD	SD	A		

Fig. 2. A Difficult Rostering Problem

6 Concluding Remarks

The contribution of this paper is three-fold. First, we define formally the notions of modeling and model redundancy of constraint satisfaction problems. These definitions serve as the foundation for the systematic study of the use of redundant modeling to speed up constraint-solving. Second, we introduce channeling constraints for combining mutually redundant models of the same problem. We further suggest guidelines for constructing alternate models for a problem and

Soft Constraints	Model One	Model Two	Combined Model	
			One Vars	Two Vars
—	short	short	short	short
C2.1+C2.3a	long	long	fast	1 min
C2.1+C2.2	long	long	1 min	long
C2.1+C2.2+C2.3a	long	long	fast	long
C2.1+C2.2+C2.3a+2.6	long	long	fast	long
C2.1+C2.3a+2.6	long	long	fast	long
C2.1+2.6	long	long	long	long
C2.1	8 min	5 min	1 min	2 min
C2.2	1 min	long	fast	long
C2.3a	long	fast	fast	fast
C2.5	1 min	fast	fast	fast
C2.6	long	fast	long	fast
C2.7	long	fast	fast	fast
C2.8	long	long	fast	long
C2.9	fast	fast	fast	fast

Fig. 3. Benchmark Results

for how the models can be connected by a form of channeling constraints. In the nurse rostering application, the channeling constraints are generated mechanically, if not automatically, using a simple double `for`-loop. Third, we apply our method in a real-life nurse rostering problem and verify empirically that the combined model does exhibit significant speedup over either of the mutually redundant models.

We believe that the redundant modeling method will be a valuable tool for the constraint community. The method, although simple, is systematic enough, allowing programmers to generate channeling constraints almost mechanically. Our approach also requires no special insight into the problem domain or the problem itself. This is not true for the case of introducing redundant constraints. Redundant models and channeling constraints are different from redundant constraint although they are based on a similar working principle. Programmers can still exploit full knowledge of the problem or the problem domain to inject redundant constraints into one or all of the redundant models to further speed up constraint-solving. Another nice property of our method is that it implies no modification to the underlying constraint solver or labeling heuristics.

A potential problem of our method is the memory overhead introduced by the accommodation of more than one model (variables and constraints) of a problem. There is always a tradeoff between time and space in the design of algorithms.

The choice is always dictated by either limitation on hardware or urgency for time-efficiency. With the advent of cheaper and more massive memory, we do not foresee memory overhead as a major obstacle for the adoption of our method.

Results in this paper are preliminary. Many open problems remain. We definitely need more experience in applying the method to other real-life problems. In particular, we should investigate automatic or semi-automatic method of creating alternate models from an existing model. Such exercise is useful even if we can automate model generation for only problems of a restricted problem domain. Another open problem is the code maintenance of the combined model: how modifications in one model can be "propagated" to the other models automatically. To solve these problems, we may benefit from a formal model description language or notation so that models can be described fully and formally. Such a language helps in extracting and formally reasoning with properties of models.

On the empirical side, it is also interesting to study and experiment with variable-ordering heuristics that label variables in the sub-models alternatively. Another important direction of work is to understand and analyze the propagation behaviour of constraints among sub-models in a combined model. One conjecture is that propagation among sub-models achieves a higher consistency level of the combined network.

Acknowledgement

We are indebted to Mr. Vincent Tam of the Tang Shiu Kin Hospital, who spared great patience in explaining to us the planning rules and rostering techniques. Our work benefitted immensely from many fruitful discussions with Mr. Tam and his often timely response to our queries. Constructive comments from the anonymous referees help improve the quality of the final version of the paper. Last but not least, this paper would not have come into existence without Ho-Fung Leung's confirmation that the redundant modeling idea is indeed new.

References

1. P. Baptiste and C. Le Pape. Disjunctive constraints for manufacturing scheduling: Principles and extensions. In *Proceedings of the Third International Conference on Computer Integrated Manufacturing*, Singapore, 1995.
2. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 17(7):57–73, 1994.
3. J. Bitner and E.M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18:651–655, 1985.
4. B.M.W. Cheng, J.H.M. Lee, and J.C.K. Wu. A constraint-based nurse rostering system using a redundant modeling approach. Technical report (submitted to the 8th IEEE International Conference on Tools with Artificial Intelligence), Department of Computer Science and Engineering, The Chinese University of Hong Kong, 1996.

5. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.
6. M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the car-sequencing problem in constraint logic programming. In *Proceedings of the European Conference on Artificial Intelligence*, pages 290–295, 1988.
7. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, and T. Graf. Applications of CHIP to industrial and engineering problems. In *Proceedings of the 1st International Conference on Industrial & Engineering Applications of Artificial Intelligence & Expert Systems*, pages 887–892, 1988.
8. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, Japan, December 1988.
9. C. Gervet. Conjunto: Constraint logic programming with finite set domains. In *Logic Programming: Proceedings of the 1994 International Symposium*, pages 339–358, 1994.
10. R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
11. ILOG. *ILOG: Solver Reference Manual Version 3.0*, 1995.
12. V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI Magazine*, 13:32–44, 1992.
13. C. Le Pape. Implementation of resource constraints in ILOG SCHEDULE: A library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3:55–66, 1994.
14. C. Le Pape. Resource constraints in a library for constraint-based scheduling. In *Proceedings of the INRIA/IEEE Conference on Emerging Technologies and Factory Automation*, Paris, France, 1995.
15. J. Liu and K. Sycara. Emergent constraint satisfaction through multi-agent coordinated interaction. In *Proceedings of MAAMAW'93*, Neuchatel, Switzerland, 1993.
16. A.K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99–118, 1977.
17. P. Marti and M. Rueher. A distributed cooperating constraints solving system. *International Journal on Artificial Intelligence Tools*, 4(1&2):93–113, 1995.
18. S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling. *Artificial Intelligence*, 58:161–205, 1992.
19. B.A. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, 5:188–224, 1989.
20. M. Perrett. Using constraint logic programming techniques in container port planning. *ICL Technical Journal*, 7(3):537–545, May 1991.
21. E.P.K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.
22. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

A Constraint-Based Interactive Train Rescheduling Tool

C.K. Chiu, C.M. Chou, J.H.M. Lee, H.F. Leung, and Y.W. Leung

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
Email: {ckchiu,cmchou,jlee,lhf,ywleung}@cs.cuhk.edu.hk

Abstract. In this paper, we formulate train rescheduling as constraint satisfaction problem and describe a constraint propagation approach to tackle it. Algorithms for timetable verifications and train rescheduling are designed under a coherent framework. We define two optimality criteria that correspond to minimizing passenger delay and the number of station visit modifications respectively for rescheduling. Two heuristics are then proposed to speed up and direct the search towards the optimal solutions. The feasibility of our proposed algorithms and heuristics are confirmed with experimentation using real-life data.

Keywords: Rescheduling, Constraint Propagation, Variable and Value Ordering

1 Introduction

The **PRaCoSy** (**P**eople's Republic of China **Ra**ilway **Co**mputing **Sy**stem) project [10] is undertaken by the International Institute for Software Technology, United Nations University (UNU/IIST). The aim of the project is to develop skills in software engineering for automation in the Chinese Railways. A specific goal of the project is the automation of the preparation and updating of the *running map*[1], for dispatching trains along the 600 kilometer long railway line between Zhengzhou and Wuhan in the People's Republic of China. The Zhengzhou to Wuhan section has been chosen as a case study because it is along the busy Beijing-Guangzhou line, the arterial north-south railway in China. The rate of running trains, both goods and passengers, of this section is high and present management procedures are not adequate with the dramatic development of domestic economy.

A running map [9] contains information regarding the topology of the railway, train number and classification, arrival and departure time of trains at each station, arrival and departure paths, *etc.* A computerized running map tool should read in stations and lines definition from a descriptor file, allow segments (subsets of all stations) and time intervals to be defined, allow train timetable to

[1] A running map is a method of monitoring the movement of trains and rescheduling their arrivals and departures to satisfy operational constraints.

be read, and finally display graphically the projection of the timetable against a given segment and a given interval. A sample running map is shown in figure 6. Train dispatchers, users of the tool, have to modify the timetable when trains in some sections cannot run according to the map, possibly due to accidents and/or train delays. The modification to the map should be performed in such a way that certain *scheduling rules* (laid down by the local railway bureau) are not violated. Therefore, a computer running map tool should check users' modifications against possible violation of scheduling rules, and warn users of such violations. In addition, the tool should also assist the user in repairing, either automatically or semi-automatically, an infeasible timetable so that the least train service disruption is made. We call this process *rescheduling*. Scheduling and rescheduling are different in two aspects. First, while scheduling creates a timetable from scratch, rescheduling assumes a feasible timetable and user modifications, which may introduce inconsistencies to the timetable, as input. Second, optimality criteria used in scheduling, such as minimum operating cost, are usually defined in the absolute sense. In rescheduling, however, the quality of the output is measured with respect to the original timetable.

The **PRaCoSy** project has resulted in a running map tool capable of train timetable verification [7]. Our task at hand is to enhance the **PRaCoSy** tool to perform automatic rescheduling, which can be considered as constraint resatisfaction. A major problem with the **PRaCoSy** implementation is that constraints are used only passively to test possible violation of scheduling rules. In view of this limitation, we decided to re-create the running map tool from scratch using a constraint programming approach. In this paper, we give algorithms for timetable verification and train rescheduling used in our tool, and show that constraint programming allows us to perform constraint checking and solving (or propagation) in a coherent framework. We study two notions of optimality for the rescheduled timetable with respect to the original timetable. These notions provide a measure of the quality of the rescheduling operation. We also present two heuristics that direct and speed up the search towards optimal rescheduled timetables.

The rest of the paper is organized as follows. Section 2 defines basic terminology and discusses related work. Section 3 explains the timetable verification algorithm. In section 4, we show how to formulate train rescheduling as a constraint satisfaction problem and give an associated algorithm. We also discuss two heuristics that help to direct and speed up the search towards optimal solutions. In section 5, we describe our prototype implementation and sample runs of the tool. We summarize our contribution and shed light on future work in section 6.

2 Preliminaries

In the following, we provide informal definitions of necessary terminology according to [9] to facilitate subsequent discussions. Names not defined should be self-explanatory or clear from the context. Interested readers can refer to [9] for

formal definitions of the same terminology.

The *topology* of a railway system is defined by a collection of named stations and identified lines. We differentiate between the lines within a station and those connecting two stations by referring the former as *lines* and the latter as *tracks*. A train *journey* is a sequence of visits to connected stations. Each *visit* to a station is represented by the arrival time and the departure time of the train. A *timetable* is an association from trains to journeys to be made. Scheduling rules is a set of temporal constraints to restrict the arrival time and departure time of each visit in order to prevent such undesirable events as train crash. A timetable is *valid* (or *feasible*) if no scheduling rules are violated under the associations. Otherwise, the timetable is *invalid* (or *infeasible)*.

Given a feasible timetable with n station visits, which is represented by a set of assignments (or equality constraints) of the form $AT_i = t_i^a$ and $DT_i = t_i^d$, where AT_i and DT_i denote the arrival and departure time respectively, for $1 \leq i \leq n$, the *modifications* that we can make to the timetable are to replace some assignments $AT_j = t_j^a$ (or $DT_j = t_j^d$) by $AT_j = t_j'^a$ (or $DT_j = t_j'^d$) where $t_j^a \leq t_j'^a$ (or $t_j^d \leq t_j'^d$) for $j \in \{1 \ldots n\}$. In other words, we can only delay the arrival time or departure time of visits. Given an infeasible timetable, *rescheduling* is the process of modifying the timetable so as to make the timetable feasible.

2.1 Problem Statement

There are six types of scheduling rules [9] in our railway system: the speed rule, the station occupancy rule, the station entry rule, the station exit rule, the line time rule, and the stopover rule. Let there be two trains 1 and 2 and two adjacent stations A and B. The variables AT_{XY} and DT_{XY} denote train Y's arrival and departure time at/from station X respectively. The above scheduling rules can be formulated as the following types of *scheduling constraints*.

The Speed Constraint

$$(lng/(AT_{A1} - DT_{B1})) \leq sp$$

The constant lng denotes the distance between station A and station B. This constraint enforces that the average train speed traveling between the two stations cannot exceed sp.

The Station Occupancy Constraint

$$(DT_{A2} + ctr \leq AT_{A1}) \vee (DT_{A1} + ctr \leq AT_{A2})$$

This constraint enforces that there is at least ctr time units between two trains occupying a track.

The Station Entry Constraint

$$(AT_{A1} - AT_{A2} \geq cen) \vee (AT_{A2} - AT_{A1} \geq cen)$$

This constraint enforces that there is at least cen time units between two trains entering a station via a line.

The Station Exit Constraint

$$(DT_{A1} - DT_{A2} \geq cex) \vee (DT_{A2} - DT_{A1} \geq cex)$$

This constraint enforces that there is at least cex time units between two trains departing from a station via a line.

The Line Time Constraints

$$((DT_{B1} < DT_{B2}) \wedge (AT_{A1} < AT_{A2})) \vee ((DT_{B1} > DT_{B2}) \wedge (AT_{A1} > AT_{A2}))$$
$$AT_{B1} < (DT_{B2} - cenx) \vee AT_{A2} < (DT_{A1} - cenx)$$

The line time rule is split into two constraints. The first constraint enforces that no train overtakes another train if they are traveling in the same direction on a line. The second constraint enforces that if there are two journeys on a line in opposing directions, the line must be unoccupied for at least $cenx$ time units.

The Stopover Constraint

$$DT_{A1} - AT_{A1} \geq cst$$

This constraint enforces that a train will stay in a station for at least cst time units.

Given the topology of a railway system with a valid train timetable, due to unexpected events, the users of the running map tool may want to modify the timetable. Our work is to first check the feasibility of the modified timetable. If it is feasible, the previous timetable is replaced by the modified one. Otherwise, we reschedule the infeasible timetable to generate a new feasible timetable. Note that efficiency should be a critical concern in designing the verification and rescheduling algorithms since in real-life situations, rescheduling must be performed in a timely manner. The notion of "efficiency" may vary according to situations. Ten minutes, however, should be a tolerable bound in general [2].

Optimal solutions are not required usually. In most cases, it is impractical to generate optimal solutions within a given (usually small) time bound. Criteria for optimality, however, should be defined. Such definitions can serve as guidelines for designing various variable-ordering and value-ordering heuristics to generate "good" answers. A precise notion of optimality also enables us to measure the "quality" of the rescheduled timetable. In the following, we present two optimality criteria.

A rescheduled timetable is *minimum-changes optimal* with respect to the original timetable if the least number of station visits are modified. This criterion can be satisfied easily in general since in most cases, we can simply delay the trains in question to the latest possible time. The resulting timetable, however, may introduce unreasonable long delay to some train visits. Thus this criterion should usually be applied with other criteria limiting the maximum delay.

A rescheduled timetable is *minimum-delay optimal* with respect to the original timetable if the longest delay among all train visits is minimum. Let the

tuples $\langle AT_1, DT_1, \ldots, AT_n, DT_n \rangle$ and $\langle AT'_1, DT'_1, \ldots, AT'_n, DT'_n \rangle$ denote the infeasible and the rescheduled timetable respectively. The goal of this criterion is to minimize the following expression:

$$\max(AT'_1 - AT_1, DT'_1 - DT_1, \ldots, AT'_n - AT_n, DT'_n - DT_n).$$

The aims of the two criteria could contradict one another and represent the extremes of a spectrum of other possible definitions of optimality.

2.2 Related Work

Rescheduling is different from traditional scheduling in the sense that the possible solutions of rescheduling are restricted by the original schedule. Zweben *et al* [11] tackle this problem using constraint-based iterative repair with heuristics. The resultant GERRY scheduling and rescheduling system is applied to coordinate Space Shuttle Ground Processing. Our work is based on a propagation-based constraint solver.

Somewhat related to our work is train scheduling. Komaya and Fukuda [5] propose a problem solving architecture for knowledge-based integration of simulation and scheduling. Two train scheduling systems are designed in this architecture. Fukumori *et al* [3] use the tree search and constraint propagation technique with the concepts of time belt in their scheduling system. This approach is claimed to be suitable for double-track line and continuous time unit. Recently, Chiang and Hau [1] attempt to combine repair heuristic with several search methods to tackle scheduling problems for general railway systems.

There are two on-going projects that aim at automating train scheduling for real-life railway ministries. Our work is a direct outgrowth of the **PRaCoSy** project [10] at UNU/IIST. The latest **PRaCoSy** running map tool prototype uses constraints only passively to test for constraint violation in their verification engine. The Train Scheduling System (TSS) designed for Taiwan Railway Bureau (TRB) [6] is a knowledge-based interactive train scheduling system incorporating both an automatic and a manual schedulers. Users and the computer system are thus able to bring complementary skills to the scheduling tasks.

3 Timetable Verification

In the following, we describe a timetable verification algorithm, which examines if a given timetable is valid with respect to a set of scheduling constraints. Violated scheduling rules (or constraints) in an invalid timetable will be located and displayed to the user. The algorithm, shown in figure 1, assumes the existence of a propagation-based constraint solver [8] `propagate()`.

The timetable T can be viewed as a set of constraints for all variables in either the form $AT_i = t_i^a$ or $DT_i = t_i^d$, where AT_i is an arrival time and DT_i is a departure time. After all variables become ground (line `5`), we post the scheduling constraints one by one (lines `7-13`). Having all variables ground, the `propagate()` engine performs essentially constraint checking. It is easy to check

```
1    procedure verify(in C, T, out I)
2    /* C: scheduling constraints, T: timetable, I: violated constraints */
3          /* Initialization */
4          I ← {}
5          S ← propagate(T)          /* Constraint store S is initialized to T */
6          /* Constraint Verification */
7          for each c ∈ C
8                S ← propagate(S ∪ c)
9                if inconsistency found
10                    I ← I ∪ {c}
11                    S ← S\{c}    /* Retract c from the constraint network */
12               endif
13         endfor
14   end
```

Fig. 1. The Timetable Verification Algorithm

that a constraint is violated under the given timetable if inconsistency is found after the constraint is told to the store. Violated constraints are retracted so that the algorithm can proceed to check for other possible constraint violations. Again, the groundness of all variables allows us to retract a constraint by simply removing the undesirable constraint from the constraint store.

4 Rescheduling as Constraint Satisfaction

Scheduling is an instance of constraint satisfaction problem. In the following, we show the same for rescheduling. Given a timetable T. Users modify T by adjusting its arrival and departure times, obtaining T', which can be valid or invalid. If T' is invalid, the *rescheduling* process should attempts to repair T' to make it feasible. By repairing, we mean adjusting the value of the non-modified variables so that (1) the timetable becomes valid again and (2) the new timetable should be reasonably "close" to the original timetable T. By being close to T, we mean that the new timetable should create the least service disruptions. Example optimality criteria are given in section 2. Note that user modified variables must be kept fixed during the rescheduling process since the modifications represent dispatcher requirements.

In order to formulate rescheduling as constraint satisfaction, we have to determine the variables of the problem, the domains associated with the variables, and the constraints of the problem. In rescheduling, the variables are the arrival and departure time of the timetable. Every variable share the integer domain $\{0, \ldots, 1439\}$[2]. There are three types of constraints in the rescheduling problem.

1. Scheduling constraints: The scheduling constraints set forth in section 2.

[2] There are 1440 minutes in 24 hours.

2. Modification constraints: For each arrival or departure time X which is modified by the user to new value t, we have the equality constraint $X = t$. This constraint enforces the user modifications to stay fixed during rescheduling.
3. Forward-labeling constraints: For each non-modified variable X with value t in the original timetable T, we have the constraint $X \geq t$. This constraint is necessary to ensure that we can only *delay* arrival or departure time.

Rescheduling now becomes finding a solution to the above constraint satisfaction problem. A solution is *optimal* if the solution is "closest" to the original timetable.

We are now ready to present the rescheduling algorithm, shown in figures 2 and 3. The algorithm can be divided into three phases. In phase one (line 4), we

```
1    procedure reschedule(in C, T)
2    /* C: scheduling constraints, T: feasible timetable */
3          /* Initialization */
4          S0 ← propagate(C)        /* Save a copy of the constraint store after
5                                        propagating constraints in C */
6          /* Rescheduling */
7          while true
8               Read in user modifications U      /* U is in the same form as T */
9               modify(T, S0, U, R)
10              if R = fail        /* No feasible timetable */
11                   Prompt error messages
12              else
13                   T ← R    /* Display rescheduled timetable if necessary */
14              endif
15         endwhile
```

Fig. 2. The Train Rescheduling Algorithm

post and propagate all scheduling constraints to prune infeasible values in the variables. The pruned constraint network is then saved in S_0. Since the scheduling constraints are the same for any timetable T and user modifications U, the same store S_0 will be reused in every rescheduling step. In real-life situation, rescheduling has to be performed repeatedly for different timetables and user modifications, this saving operation helps to avoid unnecessary invocations of constraint relaxation.

Actual rescheduling takes place in the procedure `modify()` (lines 16-46). In the second phase (lines 19-35) of rescheduling, information is extracted from user modifications and the original timetable to post and propagate the modification constraints (lines 23-27) and the forward-labeling constraints (lines 28-35). If inconsistency is found, rescheduling is halted and failure is reported. In the third phase (lines 36-46), variables that are not modified by the users (extracted by the `vars()` function) are enumerated or labeled using some form of variable-

```
16    procedure modify(in T, S0, U, out R)
17    /* T: previous feasible timetable, S0: saved constraint network state,
18       U: user modifications, R: rescheduled timetable */
19          /* Initialization */
20          O ← {X = t | (X = t) ∈ T, X ∈ vars(U)}
21            /* Assignment constraints associating with user-modified variables */
22          /* Post all user modifications */
23          S ← propagate(S0 ∪ U)       /* First and second type of constraints */
24          if inconsistency found          /* User modifications are inconsistent */
25                R ← fail
26                return
27          endif
28          /* Prune infeasible values: can only delay arrival & departure time */
29          for constraint (X = t) ∈ T\O
30                S ← propagate(S ∪ (X ≥ t))     /* Third type of constraints */
31                if inconsistency found
32                      R ← fail
33                      return
34                endif
35          endfor
36          /* Rescheduling */
37          E ← vars(T\O)                    /* Set of variables for rescheduling */
38          A = labeling(E)
39                /* Appropriate variable- and value-ordering should be used.
40                   The function labeling() returns either {} or a set of equality
41                   constraints (or bindings) for each variable X ∈ E */
42          if A = {}
43                R ← fail
44          else
45                R ← U ∪ A
46          endif
```

Fig. 3. The Train Rescheduling Algorithm *(cont.)*

and value-ordering heuristics, which are embedded in the `labeling()` function, to speed up and direct the search towards a near-optimal solution.

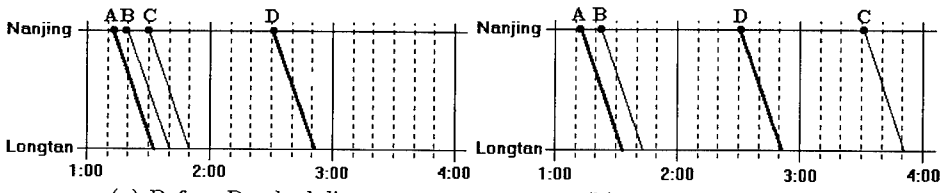
There are two situations in which user modifications lead to a timetable T' that is non-repairable. First, the user modifications are self conflicting. Since user modified variables must be kept fixed during rescheduling, it is impossible to repair other variables to make the timetable valid. Second, user modifications are not self conflicting but there is no room for other variables to adjust to make the timetable valid. Constraint propagation algorithms are well-known to be incomplete [8]. Thus, phase two of the rescheduling algorithm can detect some, but not all, of this kind of conflicts. Theoretically speaking, the enumerating procedure in phase three can guarantee to detect inconsistency but it would usually take impractically long to do so. In cases when the rescheduling algorithm

fails to return an answer within a few minutes, users are advised to abort the current computation, re-adjust the modifications, and restart the rescheduling process.

In the rest of this section, we present two variable labeling heuristics, which are designed to yield rescheduled timetables in the minimum-delay and the minimum-change optimal sense respectively.

Smallest-First Principle Variables are ordered in the ascending order of the lower bound of their domains. Values in the variable domains are also enumerated in ascending order. This principle is founded on the assumption that a short delay on a train visit will cause short delay on the subsequent one, which means that delays propagate in a monotonic fashion. Experimental results confirm that, using actual timetables from **PRaCoSy**, this heuristic usually helps to generate solutions that are minimum-delay optimal efficiently. We construct below an unrealistic artificial example that defeats the heuristic.

Figure 4 (a) shows a small segment of four journeys on a railway running



(a) Before Rescheduling (b) After Rescheduling

Fig. 4. A Non-Optimal Solution by Smallest-First Heuristic

map. The three journeys A, B, and C share the same track in the Nanjing station, while the two journeys C and D take the same line in traveling from Nanjing to Longtan. The journeys A and D are fixed (indicated by thick lines) by the users. Suppose the station occupancy and the station exit rules enforces that at least ten minutes among each of the three points A,B, and C, and sixty minutes between the points C and D respectively. The modified timetable is thus infeasible due to the insufficient long distance between the points A and B.

Figure 4 (b) shows the rescheduled timetable obtained using the smallest-first principle. The rescheduling starts from moving point B ahead in time to achieve the ten-minute requirement between points A and B. The movement in turn causes another conflict between points B and C. Point C is thus forced to move. However, there is no feasible location for point C to move between points B and D since point D is fixed by user. Therefore, we have to move point C one-hour ahead of point D. The maximum delay in this case is two hours. This

solution is not optimal since a better solution can be obtained by simply moving point B twenty minutes ahead in time, as shown in figure 5.

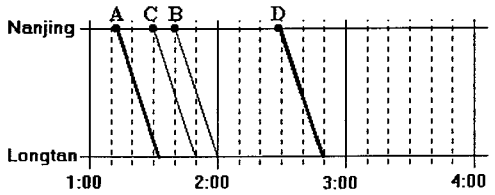


Fig. 5. A Smallest-Changes-Optimal Solution

Consistent-Assignment-First Principle This heuristic suggests to instantiate first those variables that can be instantiated with its time value in the original timetable. We call these variables *non-conflicting*. The other variables are *conflicting*. Therefore, the labeling of the non-conflicting variables will be backtracked into last. Again values in the variable domains are enumerated in ascending order. The idea is to maintain as many variables with its original value as possible. This heuristic direct searching towards a minimum-change optimal solution. Note that a non-conflicting variable may be instantiated with a value other than its original value eventually if the conflicting variables have tried all possible combination of time values and no solution is found.

For efficiency reason, we further classify the non-conflicting variables into two group. The first group contains variables that share journeys with one of conflicting variables. The second group contain the rest of the non-conflicting variables. Our heuristic suggest to label first the second group of non-conflicting variables. This ordering is essential since the labeling of variables sharing journeys with conflicting variables have a higher chance of being backtracked into.

We apply this heuristic to reschedule the infeasible timetable in figure 4 (a). Recall that the points A and D are fixed by users. We classify the variables associated with point B and point C as conflicting and non-conflicting respectively. Thus point C is labeled first to retain its original position in the map and point B is forced to move until it reaches the location ten-minute ahead point C. In this specific case, the minimum-change optimal solution coincides with the minimum-delay optimal solution. This example also shows that, in general, the two heuristics give different first solution.

5 Prototype Implementation

In order to demonstrate the feasibility of our algorithms, we have re-constructed and enhanced the **PRaCoSy** running map tool prototype [7] with rescheduling

capability. The prototype consists of a constraint-based scheduler and a user-interface. The former is implemented in C++ with ILOG Solver library 2.0 [4] while the latter is built using Microsoft Visual Basic 3.0.

In the following, an overview of the running map tool is presented. We then give a sample session of our tool using a segment (from Nanjingxi to Shanghai) of the China railway which amounts to 3005 constraints and 596 variables. The rescheduled timetables generated by the two heuristics are explained. We conclude this section by showing two examples to which our tool fails to respond in a timely manner.

The running map display (figure 6) consists of six columns (regions). Column one and column four show the abbreviated identifiers of stations. Column two shows the number of arrival and departure lines of stations. Column three is the main window for presenting the graphical representation of a timetable, with time and locations as the X and Y axes respectively. Column five shows the cumulative distance from the first station. Column six shows the distance between the current station and the previous station. In the cases where a timetable is too large to fit into the main window, two scrollbars will be enabled.

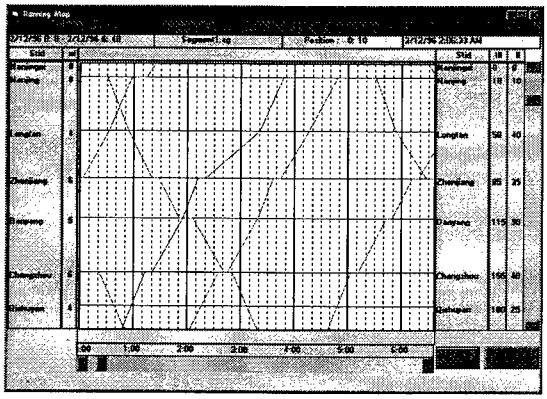


Fig. 6. The Running Map Tool

A user can click-and-drag any lines to modify the corresponding train visits on the map. The modified timetable will be validated using the verification algorithm when the "Check" button is pressed. If it is infeasible, a warning window, such as that shown in figure 7, will pop up to display all constraint violations. At this point, the user can either invoke the rescheduling algorithm by pressing the "Reschedule" button, correct the modifications manually, or restore the original feasible timetable.

Figure 8 (a) shows a segment of a China railway timetable. Due to an acci-

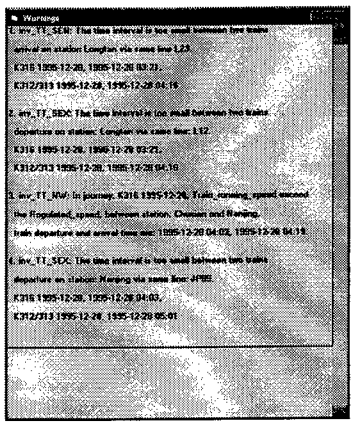
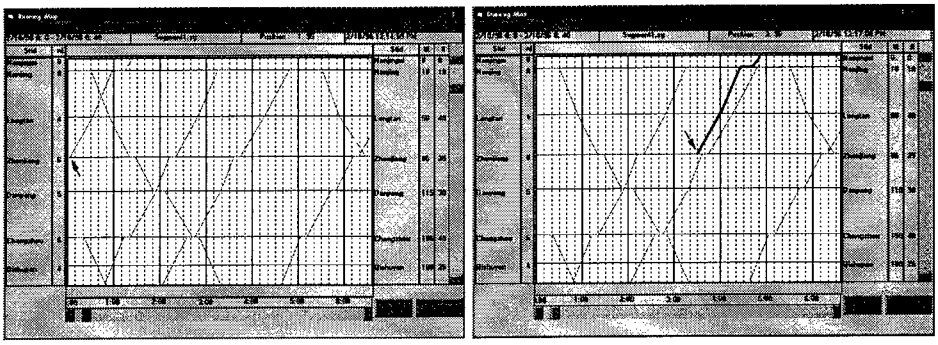


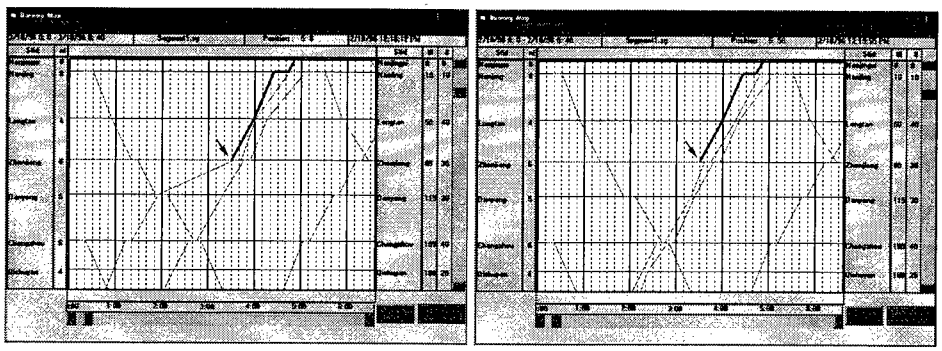
Fig. 7. A Warning Window



(a) Original Feasible Timetable (b) Modified Infeasible Timetable

Fig. 8. A Comparison of Two Heuristic

dent, we have to delay the train departuring from Zhenjiang at 1:05 to 3:30, yielding the map shown in figure 8 (b). The user-modified departure time (pointed by an arrow) and all the subsequent train visits on the journey (the highlighted segment) are then fixed immediately by the running map tool. This user movement incurs seven constraint violations between the modified journey and its left adjacent journey. We reschedule the infeasible timetable with our two heuristics. Both of them succeed in generating a feasible timetable within a few seconds. Figure 9 (a) and figure 9 (b) show the rescheduled timetable generated using the smallest-first principle and the consistent-assignment-first principle respectively.



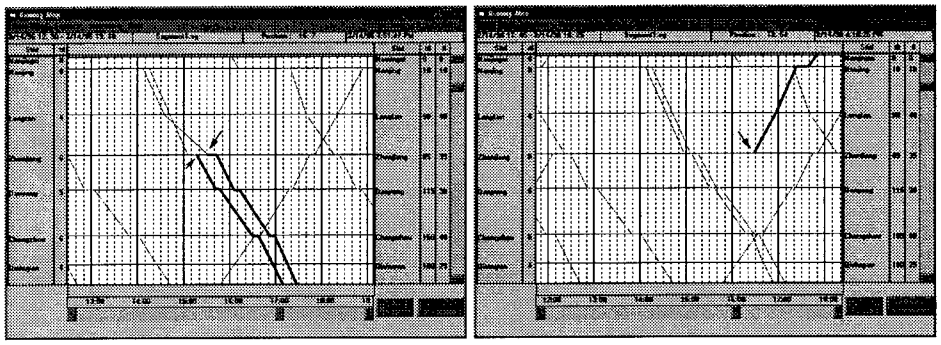
(a) Smallest-First Principle (b) Consistent-Assignment-First Principle

Fig. 9. A Comparison of Two Heuristic *(cont.)*

Applying the smallest-first principle, we process all visits on the map from the left (earliest) to the right (latest). For each visit, if its associated arrival (or departure) time does not violate any scheduling constraints, we preserve the current value. Otherwise, we move the time ahead as little as possible to eliminate the inconsistencies. Thus some visits on a journey may be modified while others remain unchanged. This explains why, visually, a journey is not only shifted right horizontally, but can also be "bent" by the rescheduling process. The movement propagates in the above fashion from left to right. Whenever no further movements are possible, backtracking takes place.

Instead of massaging several journeys to produce a feasible timetable, the consistent-assignment-first principle suggests to modify as few station visits as possible. This goal can be well approximated by first locating station visits that can remain unchanged with respect to the user modifications. These station visits will labeled first. The conflicting variables, having to change their original values, will be labeled last. Our experiments reveal that, in many cases, this heuristic produces timetable which is "almost identical" to the original timetable. As seen in figure 9 (b), most of the journeys retain their original locations. Even for the right-shifted journey, its shape is mostly preserved.

Experimental results confirm that rescheduling can usually be completed within seconds. This is not always the case. Figure 10 (a) and figure 10 (b) provide two such examples. The infeasible timetable in figure 10 (a) can be rescheduled using the consistent-assignment-first principle in a few seconds, but the smallest-first principle fails to return an answer within five minutes. Figure 10 (b) is simply a non-repairable timetable. Neither heuristics can return promptly to confirm the unsolvability of the problem.



(a) Infeasible Timetable 1 (b) Infeasible Timetable 2

Fig. 10. Poor Performance Example

6 Concluding Remarks

The contribution of this paper is three-fold. First, we define formally train rescheduling as a constraint satisfaction problem. Two algorithms for railway timetable verifications and rescheduling are then derived based on a propagation-based constraint solver. We define two optimality criteria, which are used to measure the "quality" of the rescheduled timetable. It is important to note that the optimality criteria are defined with respect to the original timetable. Second, based on the domain knowledge learned from domain analysis, we propose two heuristics to speed up and direct the search towards minimum-delay optimal and minimum-change optimal solutions respectively. The feasibility of our proposed algorithms and heuristics are confirmed with experimentation using real-life data. Third, we have re-constructed and enhanced the **PRaCoSy** running map tool prototype.

It would be an interesting future work to study different stochastic methods for train rescheduling. Work is also in progress to experiment our rescheduling method on larger-scale real-life railway timetables.

Acknowledgement

We acknowledge, with pleasure, the interaction we have had with Fellows and Staff of UNU/IIST, the United Nations University, International Institute for Software Technology, Macau. In particular, we are indebted to Prof. Dines Bjørner for inviting our participation in the **PRaCoSy** project. We also had numerous fruitful discussion and working sessions with Søren Prehn, Chris George, Yulin Dong, Liansuo Liu, and Dong Yang. Last but not least, we thank the anonymous referees for their constructive comments, which help to improve the final version of the paper.

References

1. T.W. Chiang and H.Y. Hau. Railway scheduling system using repair-based approach. In *Proceedings: Seventh International Conference on Tools with Artificial Intelligence*, pages 71–78, 1995.
2. Y. Dong. The Zhengzhou ↔ Wuhan train dispatch system. UNU/IIST PRaCoSy Document DYL/1/3, International Institute for Software Technology, United Nations University, July 1994.
3. K. Fukumori, H. Sano, T. Hasegawa, and T. Sakai. Fundamental algorithm for train scheduling based on artificial intelligence. *Systems and Computers in Japan*, 18(3):52–63, 1987.
4. ILOG. *ILOG: Solver Reference Manual Version 2.0*, 1994.
5. K. Komaya and T. Fukuda. A knowledge-based approach for railway scheduling. In *Proceedings: Seventh IEEE Conference on Artificial Intelligence Applications*, pages 405–411, 1991.
6. H.C. Lin and C.C. Hsu. An interactive train scheduling workbench based on artificial intelligence. In *Proceedings: Sixth International Conference on Tools with Artificial Intelligence*, pages 42–48, 1994.
7. X. Liu. A simple running map display tool. UNU/IIST PRaCoSy Document lx/tool/02, International Institute for Software Technology, United Nations University, August 1994.
8. A.K. Mackworth. Consistency in networks of relations. *AI Journal*, 8(1):99–118, 1977.
9. S. Prehn. A railway running map design. UNU/IIST PRaCoSy Document SP/12/3, International Institute for Software Technology, United Nations University, July 1994.
10. S. Prehn and D. Bjørner. PRaCoSy: An executive overview. UNU/IIST PRaCoSy Document par/02/09, International Institute for Software Technology, United Nations University, June 1994.
11. M. Zweben, E. Davis, B. Daun, and M.J. Deale. Scheduling and rescheduling with iterative repair. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6):1588–1596, November/December 1993.

Local Search and the Number of Solutions

David A. Clark[1], Jeremy Frank[2], Ian P. Gent[1],
Ewan MacIntyre[1], Neven Tomov[3], Toby Walsh[4]

[1] Department of Computer Science, University of Strathclyde,
Glasgow G1 1XH, Scotland.
Tel: +44 141 552-4400. Fax: +44 141 553 4101.
E-mail: {dac,ipg,em}@cs.strath.ac.uk
[2] Department of Computer Science, University of California at Davis,
Davis, CA. 95616, USA.
Tel: +1 916 758-5925. E-mail: frank@cs.ucdavis.edu
[3] Department of Computing & Electrical Engineering, Heriot-Watt University,
Edinburgh EH14 4AS Scotland.
Tel: +44 131 449 5111. Fax: +44 131 451 3431. E-mail: neven@cee.hw.ac.uk
[4] IRST, I38100 Trento & DIST, I16145 Genova, Italy.
Tel: +39 461 314438. Fax: +39 461 810851. E-mail: toby@itc.it

Abstract. There has been considerable research interest into the solubility phase transition, and its effect on search cost for backtracking algorithms. In this paper we show that a similar easy-hard-easy pattern occurs for local search, with search cost peaking at the phase transition. This is despite problems beyond the phase transition having fewer solutions, which intuitively should make the problems harder to solve. We examine the relationship between search cost and number of solutions at different points across the phase transition, for three different local search procedures, across two problem classes (CSP and SAT). Our findings show that there is a significant correlation, which changes as we move through the phase transition.

Keywords: **computational complexity, constraint satisfaction, propositional satisfiability, search**

1 Introduction

Local search has been proposed as a good candidate for solving the "hard" but soluble problems that turn up at the phase transition in solubility for satisfiability and constraint satisfaction problems. The position of such a phase transition appears to be strongly determined by the expected number of solutions [21, 19, 7, 8]. Recent theoretical analysis has shown that large variances in the number of solutions can occur at the phase transition [11]. In addition, empirical analysis has shown phase transitions can occur when the expected number of solutions is significantly larger than 1 [19]. These results may be important for understanding performance of local search procedures as they might be expected to be strongly influenced by the number of solutions. If there are many solutions,

local search may stumble on one easily. On the other hand, local search may also be led in conflicting directions by different solutions.

In this paper we show that, across a range of problem classes and local search procedures, the hardest problems occur (as for complete systematic algorithms) at the phase transition in solubility. This is despite the fact that problems beyond the phase transition can have fewer solutions. Problem difficulty across the phase transition is, however, affected by the number of solutions. We identify a correlation between number of solutions and problem hardness for local search. We show this correlation is robust across problem class and types of local search procedure, and across the phase transition. The number of solutions is not the only factor, since we identify significant variation in problem hardness when this is held constant. These results are likely to be of considerable importance for understanding phase transition behaviour in local search procedures and for benchmarking such procedures.

2 Background

2.1 SAT

Propositional satisfiability (or SAT) is the problem of deciding if there is an assignment of truth values for the variables in a propositional formula that makes the formula true using the standard interpretation for logical connectives. We will consider SAT problems in conjunctive normal form (CNF); a formula, Σ in CNF is a conjunction of clauses, where a clause is a disjunction of literals, and a literal is a negated or un-negated variable. In k-SAT problems, all clauses contain exactly k literals. Both SAT and k-SAT (for $k \geq 3$) are NP-complete [6]. As is usual [14], we will generate random k-SAT problems with n variables and l clauses, by picking k variables out of the n possible for each clause, and then negating each variable with probability $\frac{1}{2}$.

2.2 CSP

A constraint satisfaction problem (CSP) consists of a set of n variables and a set of constraints. Each variable v has a domain, M_v of size m_v. Each k-ary constraint restricts a k-tuple of variables, $(v_1, ...v_k)$ and specifies a subset of $M_1 \times ... \times M_k$, each element of which are values that these variables cannot simultaneously take. We consider here binary CSPs in which constraints are only between pairs of variables. As in previous studies [19, 7], we will generate binary CSPs with n variables each with domain m, constraint density p_1, and constraint tightness p_2, by picking exactly $p_1 n(n-1)/2$ out of the $n(n-1)/2$ possible binary constraints between variables. For each selected constraint, we disallow exactly $p_2 m^2$ of the m^2 possible pairs of values of the two variables.

2.3 Phase Transitions

Many NP-complete problems like satisfiability and constraint satisfaction, display a rapid transition in solubility as we increase the constrainedness of random

problem instances. This phase transition is associated with problems which are typically hard to solve for backtracking procedures [2]. Problems that are under-constrained tend to have many solutions. It is usually therefore very easy to guess one of the solutions. Problems that are over-constrained tend not to have any solutions. As there are many constraints, any possible solution is usually quickly ruled out. At an intermediate point, problems are critically constrained: out of a random sample some will be soluble and some not, and it is usually hard to either find a solution or to prove that none exists. Many investigations have studied phase transition behaviour in backtracking algorithms, in problems such as SAT (e.g. [14, 3]), CSP [7, 16, 19], Hamiltonian circuits [2, 5], and the traveling salesman problem (e.g. [8]).

A uniform treatment of phase transitions in combinatorial problems has recently been presented in [8], formalising the notion of 'constrainedness'. Given an ensemble of problems, the constrainedness is defined by,

$$\kappa =_{\text{def}} 1 - \frac{\log_2(< Sol >)}{\mathcal{N}} \tag{1}$$

where $< Sol >$ is the expected number of solutions for a problem in the ensemble, and $\mathcal{N}$ is the number of bits needed to write down a solution, i.e. the base 2 logarithm of the size of the state space. κ lies in the range $[0, \infty)$. If $\kappa \ll 1$ problems are under-constrained and likely to soluble, if $\kappa \gg 1$ problems are over-constrained and likely to be insoluble, and if $\kappa \approx 1$ problems are critically constrained and may be soluble or insoluble.

As in [8] we plot most of our results against the constrainedness, κ of problem instances. This allows phase transitions in different classes such as SAT and CSP to be directly compared. Furthermore, such comparisons are directly related to the number of solutions since,

$$\log_2(< Sol >) = \mathcal{N}(1 - \kappa) \tag{2}$$

For random k-SAT problems, $\mathcal{N} = n$ and $\kappa = -\log_2(1 - 2^{-k})l/n$ [8]. This is a constant multiplied by the familiar parameter l/n [14]. For the familiar case of $k = 3$, i.e. 3-SAT, the multiplier is $\log_2(8/7) = 0.192\ldots$ Note that the prediction of a phase transition at $\kappa = 1$ is equivalent to $l/n = 5.19\ldots$ The fact that the actual phase transition is observed at $l/n \approx 4.3$, i.e. $\kappa \approx 0.83$, is indicative of the fact that in 3-SAT the expected number of solutions at the phase transition grows as approximately $2^{0.17n}$. For random CSPs, $\mathcal{N} = n\log_2(m)$ and $\kappa = \frac{n-1}{2} p_1 \log_m(\frac{1}{1-p_2})$ [7].

Despite the extensive literature of phase transitions in backtracking search, there has been little analysis of phase transitions in local search. This is perhaps because phase transition behaviour is usually associated with the transition from soluble to insoluble problems, and local search procedures can only solve soluble problems. It might therefore appear that phase transitions will not be observed with local search procedures. We can, however, conduct experiments on the *soluble phase* of the ensemble. By 'soluble phase', we mean those problems in an ensemble which are soluble, no matter what the generation parameters are. To

study the soluble phase, we generated problems at random as described above, and then used a complete backtracking algorithm to eliminate insoluble problems from the ensemble.

2.4 Local Search Procedures

Local search procedures start with an initial assignment of values to the variables. They then explore their "local neighbourhood" for "better" assignments. The local neighbourhood usually consists of those assignments where the value of one variable is changed. A "score" function is applied to determine which neighbour to move towards. In SAT, we use the number of satisfied clauses. In CSP, we use the number of satisfied constraints. Hill-climbing is used in the GSAT and min-conflicts procedures to maximize the score. Other procedures use more complex procedures for selecting neighbours. For example, the MC-LOG procedure chooses a neighbour probabilistically according to the relative ranking of the scores. Local procedures can, of course, be trapped in local maxima. Various techniques have been developed to overcome this. For example, GSAT simply restarts from a new point in the state space. By comparison, procedures like MC-LOG and simulated annealing allow score-decreasing moves with a certain probability. The experiments in this paper use three local search procedures: GSAT, a CSP analogue of GSAT called GCSP, and a min-conflicts algorithm for CSPs called MC-LOG.

GSAT [18] is a local search procedure for SAT which begins with a random generated initial truth assignment, then hill-climbs by reversing or "flipping" the assignment of the variable which increases the number of satisfied clauses the most. After a fixed number, MaxFlips, of moves, search is restarted from a new random truth assignment. Search continues until we find a model or we have performed a fixed number, MaxTries, of restarts.

GCSP [20] is an analogous procedure to GSAT for CSPs. It begins with a random generated assignment of values to variables, then hill-climbs by finding a new variable-value assignment which increases the number of satisfied constraints the most. After a fixed number, MaxChanges, of moves (exactly analogous to MaxFlips in GSAT), search is restarted from a new random assignment. Search continues until we find a solution or we have performed a fixed number, MaxTries, of restarts.

MC-LOG is based on min-conflicts hill-climbing [13] but with an ability to escape local maxima. Unlike GCSP, MC-LOG does not consider all variables, but instead selects randomly a variable in conflict with some constraint. The local neighbourhood for this variable consists of alternative values for it. Unlike min-conflicts hill-climbing, MC-LOG does not select the neighbour which minimizes the number of conflicts, but ranks all neighbours according to their min-conflicts 'score', and selects one probabilistically. Changing the value of this variable is called a 'repair'. The selection function is logarithmic, so the 'best' value is chosen most often, but not exclusively as with min-conflicts[5]. The number of

[5] More precisely, we pick the ith ranked value where $i = \text{int}(\log_2(1/r)/w)$ and r is a random number in $[0, 1]$ and w is some fixed weighting.

conflicts can therefore occasionally increase, enabling the procedure to escape from local maxima.

3 Phase Transitions and Local Search

We first investigate the performance of local search as we vary the number of solutions for a fixed size of problem. Naively, one might think that problems will get monotonically harder as we decrease the number of solutions since we must search for an ever smaller number of needles in a haystack. However, even though all the problems tested are soluble, behaviour is affected by the solubility phase transition. Indeed, the hardest problems for local search seem to occur at the same point as the hardest problems for complete search, namely at the phase transition in solubility. In this paper, we take this to be the point in the phase space where 50% of problems are soluble and 50% insoluble. This point is often associated with the hardest mean search cost for backtracking algorithms [3].

3.1 MC-LOG

In Fig 1 (left), we present results for MC-LOG on 1000 soluble CSPs with $n = 20$, $m = 10$, and $p_1 = 0.5$, and p_2 varying from 0.32 to 0.42, corresponding to a range of κ from 0.80 to 1.12. We plot search cost (the number of repairs) against the constrainedness, κ. The phase transition in solubility starts at $\kappa = 0.89$ where 99.1% of problems generated are soluble[6], the nearest point to 50% solubility is at $\kappa = 0.95$ where 57.0% are soluble, and our graphs extend to regions where very few problems are soluble. At $\kappa = 1.12$, only 1.2% of problems had solutions. The peak in median search cost is at $\kappa = 0.99$ while the peak in mean search cost is slightly earlier at $\kappa = 0.95$. Surprisingly, at larger values of κ, the search cost decreases even though the average number of solution is declining. As with complete procedures, the peak average search cost is associated with the solubility phase transition.

Similar results are obtained if we study CSPs with different constraint densities. In Fig 1 (right), we vary p_2 and plot median search cost for $p_1 = 0.30$ to $p_1 = 1.00$, i.e. complete constraint graphs. As p_1 increases the phase transition occurs at smaller values of p_2. In all cases the peak median search cost is within 0.01 of the value of p_2 where 50% of problems are soluble. For comparison of different problem classes, we also plot our data against κ instead of p_2, as in Fig 1. As the constraint density, p_1 increases the peak search cost (and solubility transition) occurs nearer to the expected value of $\kappa = 1$. As p_1 increases, the mean search cost at the phase transition increases, and by (2), the expected number of solutions decreases. This suggests a correlation between the average number of solutions and search cost at the phase transition. However the picture is not clear, since at a fixed κ there is a fixed expected number of solutions, yet the search cost varies by a large factor for different p_1.

[6] Recall, that we simply discard insoluble problems.

We draw two conclusions from this data. First, we observe an 'easy-hard-easy' pattern of problem difficulty. The hard region is associated with the solubility phase transition, despite the fact that as we make problems more constrained, they have fewer solutions. This is consistent with the results of [14, 19] for backtracking algorithms applied to soluble problems. Second, there is some correlation between peak search cost and expected number of solutions at the phase transition.

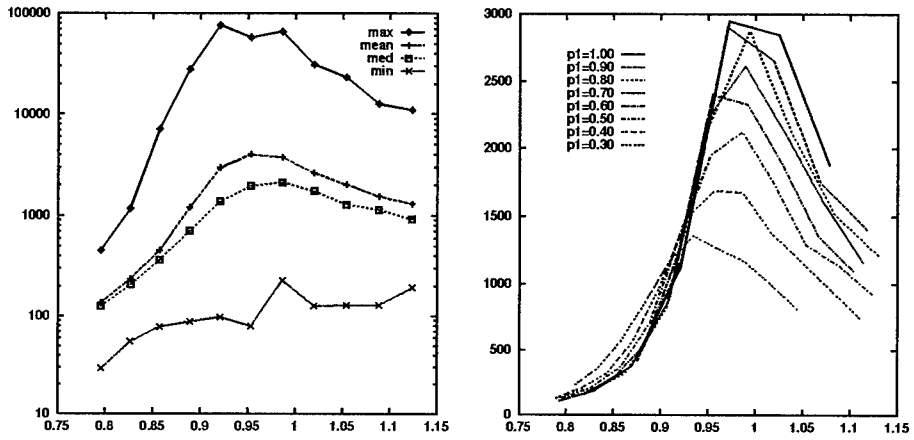


Fig. 1. (left) Search cost for MC-LOG on (20,10,0.5) problems as p_2 varies, plotted against kappa. (right) Median search cost for MC-LOG on (20,10,p_1) problems.

3.2 GCSP

To determine if the results of Section 3.1 are specific to the MC-LOG procedure, we re-ran the experiment reported in Fig 1 (left) using the local search procedure GCSP. To minimise variation, we tested GCSP with the identical problems used with MC-LOG. We set MaxChanges to 500 and ran until problems where solved, i.e. we effectively set Max-Tries to infinity.[7] We plot total changes used by GCSP against κ in Fig 2 (left). Total changes is calculated as MaxChanges times the number of failed tries, plus the number of Changes on the final try, and gives a measure of search cost for GCSP.

Behaviour is broadly similar to that seen with MC-LOG even though these procedures have significant differences. GCSP uses restarts instead of probabilistic acceptance to avoid local maxima. In addition, GCSP makes a global choice of the best variable-value assignment rather than a local choice of value for a selected variable.

[7] Two of the authors implemented GCSP independently and observed very similar results.

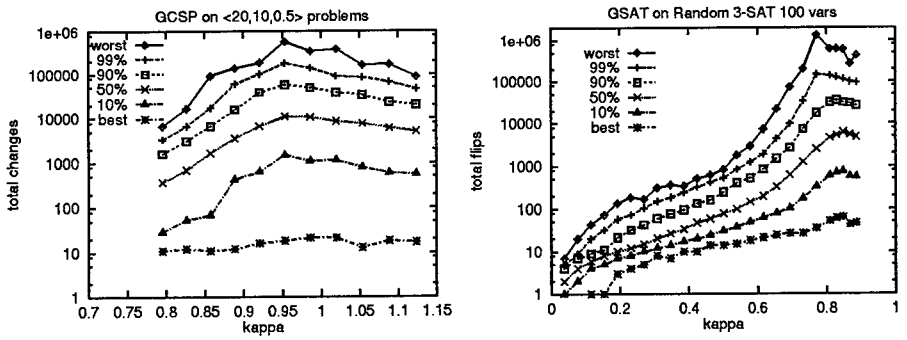


Fig. 2. (left) GCSP phase transition behaviour (right) GSAT phase transition behaviour

This suggests that other local search procedures for binary CSPs will display an easy-hard-easy pattern with the peak in search cost occurring at the solubility phase transition.

3.3 GSAT

It is likely that these results for CSPs will apply to other NP-complete problem classes like SAT. To test this hypothesis, we performed a similar experiment with 100 variable random 3-SAT problems. We varied the number of clauses from 20 to 420 in steps of 20, and from 420 to 460 in steps of 10. To aid comparison with CSPs we will plot our results against κ. For the 3-SAT problems studied here, the value of l/n can be read off from the relation $l/n \approx 5.19\kappa$. The end point corresponds to a value of $\kappa \approx 0.89$. At each point we tested GSAT on 1000 soluble problems. We set MaxFlips to 342, the optimal value for the middle of the phase transition reported by [10]. As a measure of search cost we use the total flips used, calculated analogously to total changes in GCSP. The phase transition in solubility starts at 380 clauses, $\kappa \approx 0.73$ where 99.3% of problems generated are soluble. The point nearest 50% solubility is 430 clauses, $\kappa \approx 0.83$, where 48.2% of problems were soluble. At the end point of our experiment, 9.9% of problems were soluble. The fact that κ is smaller at the phase transition than in previous graphs indicates that there are more solutions at this point than for comparably sized CSPs.

Fig 2 (right) shows different percentiles of search cost plotted against κ. The peak in median is at the next point beyond the solubility transition, and indeed the peak in the 90th percentile of search cost is at 430 clauses, the 50% solubility point. Even best case behaviour seems to get easier as we increase κ.

To conclude, for both random CSPs and SAT problems, we see an easy-hard-easy pattern in the cost of local search procedures with the peak in search cost occurring at the phase transition in solubility.

4 Search Cost and Number of Solutions

In the last section, we suggested that naively one expects problems to get monotonically harder for local search as the number of solutions decreases. To determine how true this is, in Fig 3 we give scatter plots of search effort for MC-LOG ($\log_{10}$(repairs) to find a solution) against $\log_{10}$(number of solutions). Inevitably we must investigate comparatively small problem sizes such as $n = 20$ as otherwise the exhaustive search to find all solutions becomes prohibitive. Each data point reported is the mean of 10 runs of MC-LOG to solve an individual problem.

In Fig 3 (left), we give a scatter plot for all problems in the soluble phase. At large numbers of solutions, this is a close correlation between the number of solutions and the search cost, and the spread in search costs is relatively small. The overall shape of Fig 3 (left) suggests a linear correlation between the log number of solutions and log search cost. We performed linear regression on this data, finding a best fit gradient of -0.31 with a correlation coefficient r of -0.79. At small number of solutions, however, the spread is huge, up to nearly 3 orders of magnitude. This suggests that search cost is not simply a function of the number of solutions.

We obtain a better picture of behaviour if we look at fixed points in the phase space. In Fig 3 (right), we give a scatter plot for problems with a low constraint tightness ($p_2 = 0.32$), in Fig 4 (left), problems from the middle of the phase transition ($p_2 = 0.37$), and in Fig 4 (right), problems with a large constraint tightness ($p_2 = 0.42$). At each point, there is less overall spread in the search cost for a given number of solutions than seen in Fig 3 (left). We performed regression analysis at each point separately, and the resulting lines are shown. At $p_2 = 0.32$ we estimated a gradient of -0.36 with $r = -0.56$, at $p_2 = 0.37$, we estimated a gradient of -0.61 with $r = -0.70$, and at $p_2 = 0.42$, we estimated a gradient of -0.29 with $r = -0.28$. Notice that the gradient is steepest at the solubility phase transition. Problems with a low constraint tightness have a large number of solutions and search cost is relatively uniform and small. At the phase transition, there is a large variation in the number of solutions. For problems with few solutions at the phase transition, search cost tends to be large. Although problems with a larger constraint tightness have a smaller number of solutions, search cost for problems with the same number of solutions tends to be less than at the phase transition. As seen earlier, the overall cost is greatest at the phase transition. Although there is still considerable variability, the hardest problems are those from the phase transition with a few solutions.

4.1 GCSP

To determine if these results are specific to the MC-LOG procedure, we also made scatter plots for the logarithm of search cost of GCSP against the logarithm of the number of solutions. Each data point is the cost of solving an individual problem once. Fig 5 (left) gives the plot for the middle of the phase transition. The regression line has gradient -0.56 and $r = -0.40$. Results are very similar to MC-LOG. Again there is considerable variability, problems with more solutions

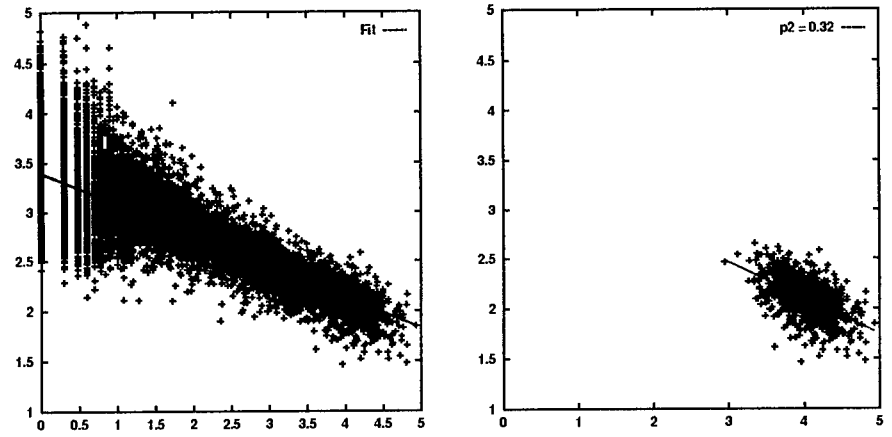


Fig. 3. $\log_{10}(repairs)$ for MC-LOG (y-axis) plotted against $\log_{10}(solutions)$ (x-axis). (left) All p_2 , (right) $p_2 = 0.32$

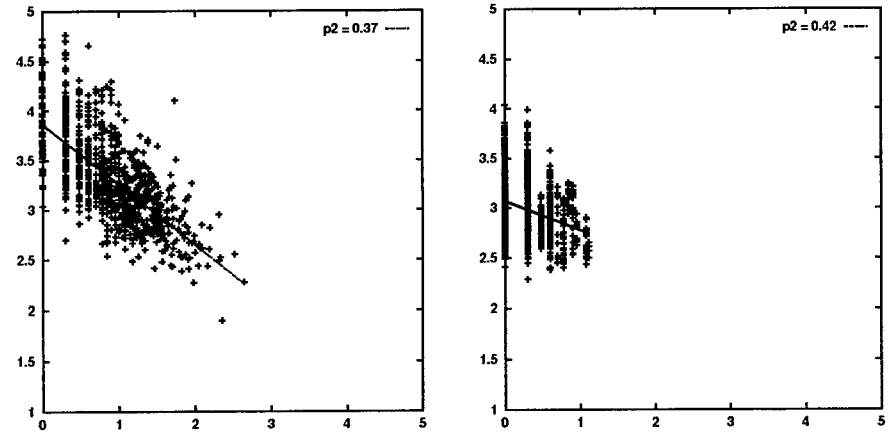


Fig. 4. $\log_{10}(repairs)$ for MC-LOG (y-axis) plotted against $\log_{10}(solutions)$ (x-axis). (left) $p_2 = 0.37$, (right) $p_2 = -0.42$

are generally much easier than problems with very few solutions. The hardest problems are again those from the phase transition with a single or very few solutions. The greater noise in this figure than seen with MC-LOG is probably due to only reporting a single rather than an average of 10 runs.

4.2 GSAT

Once again, we wished to see if our results extend beyond CSPs to SAT. To test this, we randomly generated 1000 soluble 3-SAT problems with 100 variables and 430 clauses, the point nearest the solubility phase transition as reported above.

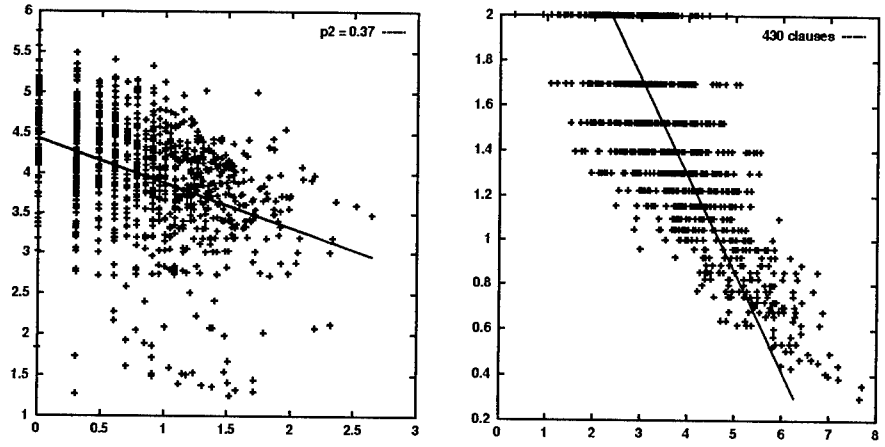


Fig. 5. $\log_{10}(cost)$ (y-axis) plotted against $\log_{10}(solutions)$ (x-axis) (left) GCSP at $p_2 = 0.37$ (right) GSAT at $n = 100$, $l = 430$

We ran GSAT on 100 tries and used this to give an estimate of search cost as simply 100 divided by the number of successes for each problem.[8] We also found the exact number of solutions that each problem has. The relationship between these two measures is shown in Fig 5 (right). Again, there is a strong tendency for problems with more solutions to be easier, although again there is a lot of noise. We modeled this by a regression line with gradient -0.44 and $r = -0.77$. The granularity in the search cost in the figure is due to the measure we used, and the use of this measure may make the regression less accurate. We intend further experiments to investigate this.

To summarise, we have shown a strong if noisy correlation between search cost and number of solutions. The correlation seems to be linear on a log-log scale. The dependency between search cost and number of solutions changes as we change the constrainedness of problems, with the hardest problems being those at the solubility phase transition with very few solutions.

5 Regression Analysis Through Phase Transitions

We have shown that the correlation between search cost and the number of solutions varies at different points in the solubility phase transition. We now look more closely at this variation.

Fig 6 shows the regression lines for MC-LOG for all values of constraint tightness p_2 across the phase transition. There is a distinct pattern in these regression lines; at low constraint tightness, the magnitude of the gradient is low, but increases to reach a peak when $p_2 = 0.36$ ($\kappa = 0.92$), which is at the phase transition. The gradient then decreases as constraint tightness increases.

[8] Thus we are using a different cost measure to that used above for GSAT in Fig 2.

This is shown more clearly in Fig 7 (left), which shows the values of the gradient against κ. This makes clear that the steepest gradient is at the solubility phase transition. We interpret this as suggesting that the number of solutions has most influence on search cost at the solubility phase transition.

Fig 7 (right) shows how the regression fit gradient changes against κ, for $p_1 = 0.30$ to $p_1 = 1.00$. In each case, the minimum gradient (maximum absolute gradient) occurs at or very close to the 50% solubility point on the phase transition. Once again, this suggests that the number of solutions has most influence on search cost at the phase transition.

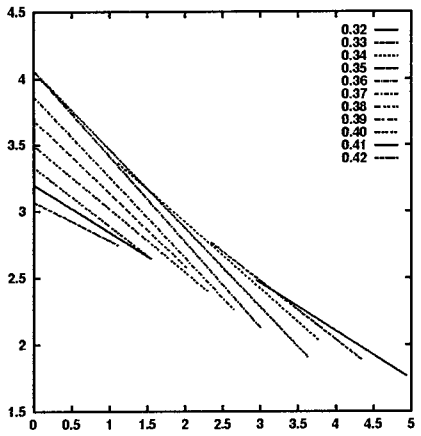


Fig. 6. Regression fits for log search effort (x-axis) vs log solutions (y-axis), as p_2 changes (20, 10, 0.50, p_2)

5.1 GCSP

We again continued our investigations by seeing if our results would also be seen in GCSP. Accordingly, we plotted the regression gradients obtained from analysis of experiments on problems from the classes (20,10,0.5) with varying p_2. The results are seen in Fig 8 (left). The same pattern emerges as in Fig 7 (left). The steepest gradient is at exactly the same point. Some noise can be seen in this graph, perhaps due as we noted earlier to not averaging GCSP results over many runs. Nevertheless, it seems that for a range of CSP local search methods, the number of solutions matters most at the phase transition.

5.2 GSAT

To conclude our investigations of how regression gradients changed through the phase transitions, we analysed our data for GSAT with 100 variables from 410 to 480 clauses. Again the gradient of the regression between number of solutions

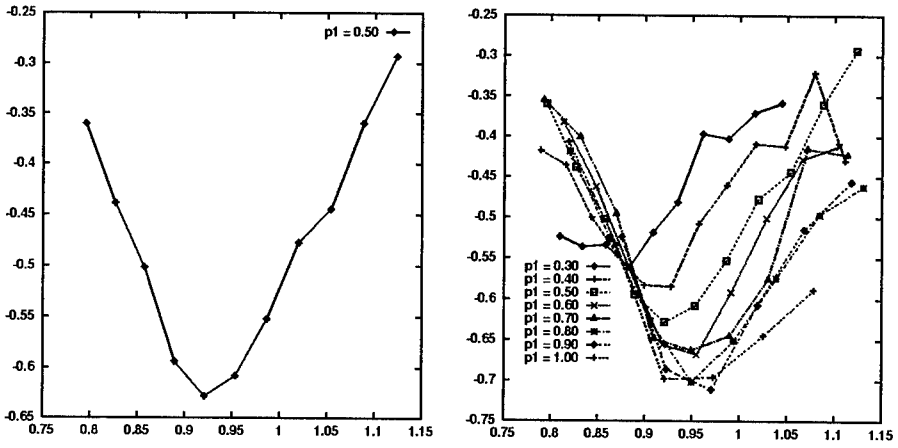


Fig. 7. Regression fit gradients (y-axis) plotted against κ (x-axis) for (left) $p_1 = 0.50$, and (right) for p_1 from 0.30 to 1.00.

and search cost changes as we vary the constrainedness of problems. Recalling that the solubility phase transition is at 430 clauses, in this case we do not find the steepest gradient at this point. As yet we are unclear as to the reasons for this change from our results on CSPs. In particular, it must for the moment remain open whether this is due to an inherent difference between 20 variable CSP instances and 100 variable 3-SAT instances, or to some more mundane feature such as the differing measures of search costs used. One significant difference between the CSP and 3-SAT classes we have studied here may be that the solubility transition in 3-SAT occurs at lower values of κ, i.e. where more solutions occur in relation to problem size.

6 Comparison with Complete Algorithms

Systematic backtracking procedures will also be affected by the number of solutions. If there are many solutions, then it may not to be hard to find a path that leads to one. If there are few, it may be hard to find a path that ends in a solution. In Fig 9, we give scatter plots for the search effort for the FC-CBJ-BZ algorithm (measured in consistency checks) against the number of solutions. We give plots for both the search effort to find the first solution and to find all solutions. FC-CBJ-BZ is a forward checking algorithm with conflict-directed backjumping [15] using the Brelaz heuristic for dynamic variable ordering [1]. This is currently one of the best algorithms for CSPs.

There is a tendency for search cost to find the first solution to decrease with the number of solutions, and this can be regressed by a line with gradient -0.18 and $r = -0.21$. However the tendency is very weak and even noisier than the plots seen earlier for local search. For finding all solutions, there seems to be no tendency for increasing numbers of solutions to reduce cost. This is not surprising

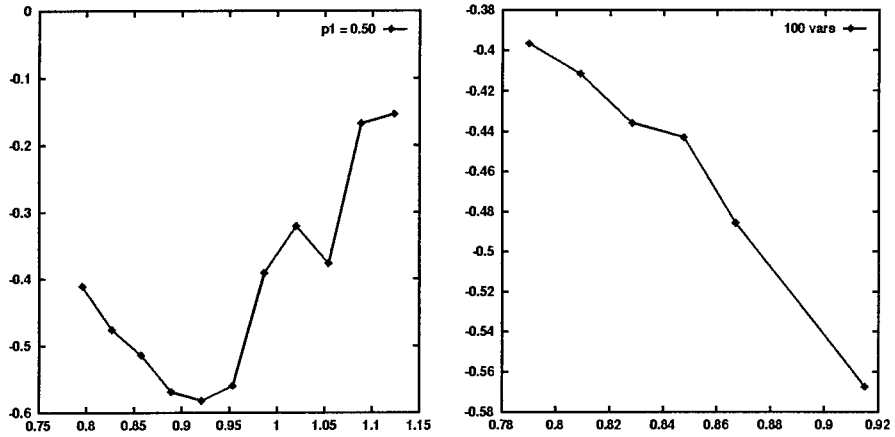


Fig. 8. Regression fit gradients (y-axis) plotted against κ (x-axis). (left) GCSP on (20,10,0.5) problems. (right) GSAT on $n = 100$ problems

since to find all solutions, all parts of the search space must be explored in full. It seems that new features of the solution space will have to be investigated to predict the search cost for complete algorithms.

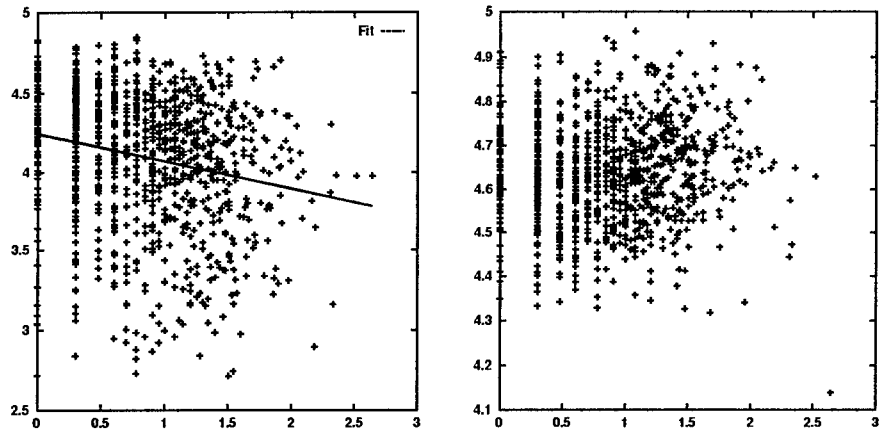


Fig. 9. $\log_{10}(\textit{checks})$ (y-axis) for FC-CBJ-BZ against $\log_{10}(\textit{solutions})$ (x-axis). (left) checks until first solution found (right) is checks to find all solutions

7 Related Work

Phase transitions in backtracking algorithms have been the subject of enormous study in recent years, but comparatively little attention has been paid to the

behaviour of local search algorithms as the position in the phase space changes. Experimental results for GENET and GSAT applied to random 3-colouring instances are reported by [4]. These results are consistent with those reported in Section 3 above, in that search cost reduces beyond the phase transition, and further suggest that the results reported here are unlikely to be dependent on the particular algorithms or classes studied.

Gent and Walsh showed that a range of variants of GSAT applied to non-random problems gave good performance on problems with many solutions such as the n-queens problem, and poor performance on problems with few solutions, for example quasigroup construction problems [10]. They speculated that the number of solutions in relation to problem size would be critical in understanding local search cost. In this paper we have shown that this speculation is confirmed when random problems are studied, although there are other features which must be taken into account such as position in the phase space. This still does not yield a full explanation of behaviour, so we wish to research the topology of local search in more depth, following studies such as [9, 12].

In this paper we have studied three local search algorithms for two different problem classes. The fact that we see similar results in each case suggests that our results may well apply to a large number of similar algorithms. However, it remains an interesting question if these results will apply to algorithms such as WSAT [17] which may explore the search space in different ways.

8 Conclusions

We have investigated in depth the relationship between the number of solutions and search cost for local search procedure. Although there is no single simple story (for example, search cost is inversely proportional to the solution density), we have identified some important connections. In particular, the hardest problems tend to have few solutions and usually occur (as with complete, systematic algorithms) at the solubility phase transition. We have shown that there is a significant correlation between the number of solutions and problem hardness for local search. This correlation is robust across problem class and types of local search procedure, and across the phase transition. The number of solutions is, however, not the only factor determining problem hardness since there is significant variation in problem hardness when the solution density is held constant. These results improve our understanding of phase transition behaviour and of the factors affecting the performance of local search methods.

References

1. D. Brelaz. New methods to color the vertices of a graph. *Comms. ACM*, 22:251–256, 1979.
2. P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proceedings, IJCAI-91*, pages 331–337, 1991.

3. J.M. Crawford and L.D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings, AAAI-93*, pages 21–27, 1993.
4. A. Davenport. A comparison of complete and incomplete algorithms in the easy and hard regions. In *Proceedings, Workshop on Studying and Solving Really Hard Problems, CP-95*, pages 43–51, 1995.
5. J. Frank and C. Martel. Phase transitions in random graphs. In *Proceedings, Workshop on Studying and Solving Really Hard Problems, CP-95*, 1995.
6. M. R. Garey and D. S. Johnson. *Computers and intractability : a guide to the theory of NP-completeness.* W. H. Freeman, 1979.
7. I.P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. Scaling effects in the CSP phase transition. In *Proceedings, CP-95*, pages 70–87, 1995.
8. I.P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Proceedings, AAAI-96*, 1996.
9. I.P. Gent and T. Walsh. An empirical analysis of search in GSAT. *Journal of Artificial Intelligence Research*, 1:47–59, September 1993.
10. I.P. Gent and T. Walsh. Unsatisfied variables in local search. In *Hybrid Problems, Hybrid Solutions*, pages 73–85. IOS Press, 1995. Proceedings, AISB-95.
11. A. Kamath, R. Motwani, K. Palem, and P. Spirakis. Tail bounds for occupancy and the satisfiability threshold conjecture. *Randomized Structure and Algorithms*, 7:59–80, 1995.
12. S. A. Kauffman. *The Origins of Order.* Oxford University Press, 1993.
13. S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings, AAAI-90*, pages 17–24, 1990.
14. D. Mitchell, B. Selman, and H. Levesque. Hard and easy distributions of SAT problems. In *Proceedings, AAAI-92*, pages 459–465, 1992.
15. P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.
16. P. Prosser. Binary constraint satisfaction problems: Some are harder than others. In *Proceedings, ECAI-94*, pages 95–99, 1994.
17. B. Selman, H. Kautz, and B. Cohen. Noise Strategies for Improving Local Search. In *Proceedings, AAAI-94*, pages 337–343, 1994.
18. B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings, AAAI-92*, 1992.
19. B.M. Smith. Phase transition and the mushy region in constraint satisfaction problems. In *Proceedings, ECAI-94*, pages 100–104, 1994.
20. N. Tomov. Hill-climbing heuristics for solving constraint satisfaction problems. 4th year project report, Department of Artificial Intelligence, University of Edinburgh, 1994.
21. C.P. Williams and T. Hogg. Exploiting the deep structure of constraint problems. *Artificial Intelligence*, 70:73–117, 1994.

Derivation of Constraints and Database Relations

David Cohen[1], Marc Gyssens[2] and Peter Jeavons[1],

[1] Department of Computer Science, Royal Holloway, University of London, UK
e-mail: p.jeavons@dcs.rhbnc.ac.uk
[2] Department WNI, University of Limburg, B-3590 Diepenbeek, Belgium
e-mail: gyssens@charlie.luc.ac.be

Abstract. In this paper we investigate which constraints may be derived from a given set of constraints. We show that, given a set of relations $\mathcal{R}$, all of which are invariant under some set of permutations P, it is possible to derive any other relation which is invariant under P, using only the projection, Cartesian product, and selection operators, together with the effective domain of $\mathcal{R}$, provided that the effective domain contains at least three elements. Furthermore, we show that the condition imposed on the effective domain cannot be removed. This result sharpens an earlier result of Paredaens [13], in that the union operator turns out to be superfluous. In the context of constraint satisfaction problems, this result shows that a constraint may be derived from a given set of constraints containing the binary disequality constraint if and only if it is closed under the same permutations as the given set of constraints.

Keywords: Relational database, relational algebra, constraint derivation

1 Introduction

In a constraint satisfaction problem [11, 12] some of the constraints are explicit, whilst others are generally present only as *implicit* or *derived* constraints. In this paper we investigate which constraints it is possible to derive using a given collection of explicit constraint types.

The approach taken is interdisciplinary, drawing on the results of relational database theory. The close relationship between the theory of constraint satisfaction problems and relational databases has been pointed out by several authors [2, 4, 6, 15]. In this paper we present a result concerning the algebraic properties of relations which sharpens an earlier result of Paredaens [13] in database theory, and thereby allows an application to constraint satisfaction problems.

The result obtained by Paredaens states that a database relation can be obtained from a set of database relations using a relational algebra expression if and only if the relation is invariant under all permutations of the database

domain under which all relations of the given set are invariant[3].

Paredaens actually proved more than he claimed: he showed that a relation satisfying his invariancy condition can be derived using only the monotonic projection, Cartesian product, equality and disequality selection, and union operators. Thus, the non-monotonic difference operator of the relational algebra is not required.

In this paper, we shall show that, when the effective domain of the given set of relations is available, we can also dispense with the union operator *if this effective domain contains at least three elements*. We shall give both a non-constructive and a constructive proof of this result, and we shall also show that the union operator cannot generally be dispensed with if the effective domain contains fewer than three elements.

In those cases where the union operator has been shown to be superfluous, the result may be applied to the question of deriving constraints in the context of constraint satisfaction problems. It demonstrates that a constraint (of any arity) may be "derived" from a given set of constraints which includes the binary disequality constraint, if and only if it is invariant under the same permutations as that set. In other words, if we are given a set of constraints, S, which contains the binary disequality constraint, then we may obtain *any* other constraint, C, as a projection of the set of solutions to some constraint satisfaction problem involving constraints from S, provided that C is invariant under any relabelings of the domain which leave all the constraints in S unchanged.

The consequences of this result are quite striking. It implies, for example, that any finite constraint over the integers, of any arity, may be obtained as an implicit constraint in a constraint network containing only binary constraints of the form $x \neq y$, and $x = y + 1$.

Even when we know that it is *possible* to derive a constraint from some set of given constraints, it is often far from obvious how this derivation may be carried out. For example, the reader is invited to attempt to design a constraint network over positive integers less than 100 which contains an implicit constraint of the form $x^2 + y^2 = z^2$, using only binary constraints of the form $x \neq y$ and $x = y + 1$. In Section 5 we describe explicitly how to construct a suitable constraint satisfaction problem to derive any given constraint which belongs to the set of possible derived constraints.

2 Definitions and terminology

Relations play a central role in both relational databases and constraint satisfaction problems.

Definition 1. Let D be a set, called the *domain*, and let n be a number. An n-ary *relation* over the domain D is a finite subset of D^n.

[3] Independently, Bancilhon [1] proved a similar result for the equivalent relational calculus. This property subsequently became known as the BP-completeness of the relational algebra and calculus.

Following a common practice in constraint satisfaction problems, we shall refer to the elements of the domain D as *colors*.

A constraint satisfaction problem [11, 12] is simply a hypergraph in which the edges are ordered sets and each edge is labelled with a relation over some fixed domain D. These relations are called constraints. A solution to the problem is a mapping from the vertices to the domain such that the image of the vertices in each edge of the hypergraph is an element of the corresponding constraint. In a binary constraint satisfaction problem (often called a constraint network) all of the constraints are relations of arity 2.

In the context of constraint satisfaction problems, the domain D is usually finite. In the context of relational databases, the domain D is usually infinite. In order to overcome this apparent mismatch between these two related areas, we make use of the following notation, which is well-established in database theory [14].

Definition 2. Let $\mathcal{R}$ be a set of relations over a domain D. The *effective domain* of $\mathcal{R}$, denoted $D_{\mathcal{R}}$, is defined to be the smallest subset of D over which (as domain) $\mathcal{R}$ is a set of relations.

Note that, by Definition 1, $D_{\mathcal{R}}$ is finite whenever $\mathcal{R}$ is finite.

In relational database theory, a standard set of operators on relations is defined, which is called the *relational algebra* [14].

Definition 3. The relational algebra is a set of operators on relations consisting of the binary operators *"union"* ($\cup$), *"difference"* ($-$), and *"Cartesian product"* ($\times$), and the unary operators *"selection"* (σ) and *"projection"* (π). The binary operators are defined in the usual, set-theoretic way. The unary operators are defined as follows:

- Let r be an n-ary relation over a domain D. Let $1 \leq i, j \leq n$. The *equality selection* $\sigma_{i=j}(r)$ is defined to be the n-ary relation

$$\sigma_{i=j}(r) = \{t \in r \mid t[i] = t[j]\}.$$

  The *disequality selection* $\sigma_{i \neq j}(r)$ is defined to be the n-ary relation

$$\sigma_{i \neq j}(r) = \{t \in r \mid t[i] \neq t[j]\}.$$

- Let r be an n-ary relation over a domain D. Let $i_1, \ldots, i_k$ be a subsequence of $1, \ldots, n$. The *projection* $\pi_{i_1,\ldots,i_k}(r)$ is defined to be the k-ary relation

$$\pi_{i_1,\ldots,i_k}(r) = \{\langle t[i_1], \ldots, t[i_k]\rangle \mid t \in r\}.$$

Using the selection operator, two special relations over a domain D can be defined:

1. the *binary equality relation*, $=_D$, is defined by $\sigma_{1=2}(D^2)$; and
2. the *binary disequality relation*, $\neq_D$, is defined by $\sigma_{1\neq 2}(D^2)$.

We now define a much more restricted algebra which contains only three operators.

Definition 4. The *SPJ-algebra*[4] consists of the relational algebra operators Cartesian product, equality selection, and projection.

Note that *intersection* ($\cap$), defined in the usual set-theoretic way, can be expressed in the SPJ-algebra, by using a sequence of Cartesian product, equality selection and projection operations. Similarly, *generalized projection*, in which the indices $i_1, \ldots, i_k$ are only required to be in the range 1–n (their order being arbitrary, and repetition being allowed) can also be expressed in the SPJ-algebra.

Most importantly of all, for our purposes, the *join* operator ($\bowtie$), [14], can be expressed in the SPJ-algebra. Now, it is well-known that the set of solutions to a constraint satisfaction problem (expressed as a relation) can be obtained by performing a join operation on the constraints [2, 6]. The possible *derived* constraints are the projections of these sets of solutions, as the following definition indicates.

Definition 5. A constraint can be *derived* from a set of relations $\mathcal{R}$ if it is equal to some projection of the set of solutions to some constraint satisfaction problem with constraints chosen from $\mathcal{R}$.

Lemma 6. *Let $\mathcal{R}$ be a set of constraints over a domain D such that the binary equality relation, $=_D$, may be derived from $\mathcal{R}$.*

A constraint C can be derived from $\mathcal{R}$ if and only if C can be expressed in the SPJ-algebra over $\mathcal{R}$.

In other words, whenever the binary equality relation is available as a constraint, the notion of a derived constraint corresponds precisely to the notion of a relation which may be expressed in the SPJ-algebra.

We shall establish in Section 3 that in order to determine which constraints it is possible to derive from a given set of relations $\mathcal{R}$, it is helpful to consider arbitrary *operations* on the domain D, defined as follows.

Definition 7. Let D be a domain and let m be a natural number. An m-ary *operation* on D is a total mapping from D^m to D.

Let $f : D^m \to D$ be an m-ary operation on D. Then f can be extended to relations over D, as follows. Let r be an n-ary relation over D, and let $t_1, \ldots, t_m$ be tuples of r (not necessarily distinct). We define $f(t_1, \ldots, t_m)$ to be the n-ary tuple $\langle f(t_1[1], \ldots, t_m[1]), \ldots, f(t_1[n], \ldots, t_m[n])\rangle$ and define $f(r)$ to be the n-ary relation $\{f(t_1, \ldots, t_m) \mid t_1, \ldots, t_m \in r\}$.

Definition 8. Let D be a domain, let f be an m-ary operation on D, and let r be an n-ary relation over D. The relation r is *closed under* f if $f(r) \subseteq r$.

[4] From "Select-Project-Join."

In the sequel, we shall be concerned particularly with operations which only depend on a single argument. We therefore define the following properties of an operation.

Definition 9. Let $f : D^m \to D$ be an m-ary operation on D. The operation f is called *essentially unary* if there exists i, $1 \leq i \leq m$, and $g : D \to D$, a unary operation on D, such that, for all $d_1, \ldots, d_m$ in D, $f(d_1, \ldots, d_m) = g(d_i)$. If g is a permutation, then f is said to be *essentially a permutation.*

Relations which are closed under the same operations have many common properties, so we introduce the following notation.

Notation 1 *Let $\mathcal{R}$ be a set of relations over a domain D.*

- *The set $\overline{\mathcal{R}}$ is the set of all relations over $D_{\mathcal{R}}$ which are closed under* all permutations *under which all the relations of $\mathcal{R}$ are closed.*
- *The set $\overline{\overline{\mathcal{R}}}$ is the set of all relations over $D_{\mathcal{R}}$ which are closed under* all operations *under which all the relations of $\mathcal{R}$ are closed.*

3 Derivation of relations

We now quote two results from the literature involving derivation of relations, the first having been stated in the context of constraint satisfaction, and the second in the context of relational databases. The main result of this paper relies on combining these two results. In order to state the results concisely, we introduce the following notation.

Notation 2 *If $\mathcal{R}$ is a set of relations over a domain D, then*

- *$\mathcal{R}^*$ denotes the set of relations over $D_{\mathcal{R}}$ that can be obtained from $\mathcal{R}$ in the relational algebra.*
- *$\mathcal{R}^+$ denotes the set of relations over $D_{\mathcal{R}}$ that can be obtained from $\mathcal{R}$ in the SPJ-algebra.*

***Theorem 10** [8]. Let $\mathcal{R}$ be a set of relations over a finite domain D which contains the binary equality relation. Then $\mathcal{R}^+ = \overline{\overline{\mathcal{R}}}$.*

We note here that the inclusion $\mathcal{R}^+ \supseteq \overline{\overline{\mathcal{R}}}$ is proved in [8] in a non-constructive way.

***Theorem 11** [13]. Let $\mathcal{R}$ be a finite set of relations over a domain D. Then $\mathcal{R}^* = \overline{\mathcal{R}}$.*

The inclusion $\mathcal{R}^* \supseteq \overline{\mathcal{R}}$ is proved in [13] in a constructive way. In preparation for the construction to be given later in this paper, we briefly sketch the construction in [13].

Let D be the effective domain of $\mathcal{R}$, and let r_1 be the Cartesian product of all non-empty relations in $\mathcal{R}$. Clearly, r_1 is closed under exactly those permutations

under which each relation in $\mathcal{R}$ is closed. Let n_1 be the arity of r_1, and let s_1 be the number of tuples in r_1. Consider the Cartesian product $r_1^{s_1}$, which is an $s_1 n_1$-ary relation, and let t_1 be a tuple of $r_1^{s_1}$ containing all tuples of r_1 as a subtuple. In particular, all values of D occur in t_1. Now, for all $i, j = 1, \ldots, s_1 n_1$, $i \neq j$, perform the equality selection $\sigma_{i=j}$ if $t_1[i] = t_1[j]$ and the disequality selection $\sigma_{i \neq j}$ if $t_1[i] \neq t_1[j]$. Then choose $1 \leq i_1, \ldots, i_{|D|} \leq s_1 n_1$ such that $D = \{t_1[i_1], \ldots, t_1[i_{|D|}]\}$, and perform the projection $\pi_{i_1,\ldots,i_{|D|}}$. Call the resulting relation r_2. Let $t_2 = t_1[i_1, \ldots, i_{|D|}]$. Now it is easy to show that r_1 is closed under a permutation $f : D \to D$ if and only if there exists a tuple t in r_2 such that, for all $i = 1, \ldots, |D|$, $f(t_2[i]) = t[i]$. This statement remains true if t_2 is an arbitrary tuple of r_2. Hence $\overline{\mathcal{R}} = \overline{\{r_1\}} = \overline{\{r_2\}}$

Now let r be any relation in $\overline{\mathcal{R}}$, and let n be the arity of r. Let t be an arbitrary tuple of r. Choose $1 \leq j_1, \ldots, j_n \leq |D|$ such that, for $i = 1, \ldots, n$, $t_2[j_i] = t[i]$. Consider the generalized projection $r_t = \pi_{j_1,\ldots,j_n}(r_2)$. We have $\{t\} \subseteq r_t \subseteq r$, the latter inclusion because r is closed under all permutations under which r_2 is closed. Hence $r = \bigcup_{t \in r} r_t$ and $r \in \mathcal{R}^*$.

This proof sketch clearly shows that the non-monotonic difference operator is not required to construct any relation in $\overline{\mathcal{R}}$. This construction, however, relies heavily on the use of the union operator.

4 Main result

In this section, we prove our main result: in Theorem 11, not only the difference operator but also the union operator is shown to be superfluous, provided

1. we have the effective domain $D_{\mathcal{R}}$ at our disposal as a unary relation, and
2. $D_{\mathcal{R}}$ contains at least three elements.

The proof we shall give in this section is non-constructive and relies on the following lemma.

Lemma 12. *Let D be a finite domain containing at least three colors and let $f : D^m \to D$ be an m-ary operation on D. Then the binary disequality relation, $\neq_D$, is closed under f if and only if f is essentially a permutation.*

Proof. Omitted. See [3] for details.

We draw the reader's attention to the fact that the condition that the domain D contain at least three colors is used repeatedly in the proof of this result. Without this condition, Lemma 12 does not hold. To see this, it suffices to observe that the binary disequality relation on a bi-valued domain D is closed under the so-called *majority operation* [10], which is the ternary operation on D returning, on each triple of colors in D, the unique color in that tuple occurring more than once. Clearly, the majority operation is not essentially unary.

We now prove the main result.

Theorem 13. *Let D be a finite domain and let $\mathcal{R}$ be a finite set of relations over D, containing the binary disequality relation $\neq_D$. If D contains at least three colors, then $\mathcal{R}^* = \mathcal{R}^+$.*

Proof. Since $\mathcal{R}$ contains the relation $\neq_D$, a simple derivation shows that $\mathcal{R}^+$ contains the equality relation, $=_D$. Hence, by Theorem 10, $\mathcal{R}^+ = \overline{\overline{\mathcal{R}}}$. By Lemma 12, all the operations under which the relation $\neq_D \in \mathcal{R}$ is closed are essentially permutations, so $\overline{\overline{\mathcal{R}}} = \overline{\mathcal{R}}$. By Theorem 11, $\overline{\mathcal{R}} = \mathcal{R}^*$.

The theorem fails without the condition that the domain contain at least three colors. To see this, let D be a domain containing just two colors and let $\mathcal{R}$ be the singleton set consisting of the binary relation $\neq_D$. When $|D| = 2$, the relation $\neq_D$ is closed under the majority operation on D, as described above. Since closure is preserved by the Cartesian product, equality selection and projection operations, it follows that every relation in $\mathcal{R}^+$ is also closed under the majority operation on D. Now consider the relation $R_0 = \sigma_{2\neq 3}(D^3) \cup \sigma_{1\neq 3}(D^3) \cup \sigma_{1\neq 2}(D^3)$, which is the complement of the ternary equality relation. This relation clearly belongs to $\mathcal{R}^*$, but it is not closed under the majority operation on D and hence does not belong to $\mathcal{R}^+$.

We now restate Theorem 13 in terms of relational database theory.

Corollary 14. *Let $\mathcal{R}$ be a finite set of relations over D. If $\mathcal{R}$ contains the effective domain $D_{\mathcal{R}}$ as a unary relation, and $|D_{\mathcal{R}}| \geq 3$, then any relation in $\mathcal{R}^*$ can be obtained without using the union or difference operators.*

Note that the condition that $\neq_D \in \mathcal{R}$ is no longer required, since the relational algebra includes the disequality selection operator.

5 Constructive proof of the main result

The proof of Theorem 13 exhibited in Section 4 is not entirely satisfactory, because it is non-constructive. Unlike the proof of Theorem 11, it does not provide a clue as to the size of the expression required to derive a relation, or, in terms of constraint satisfaction problems, the size of the network required to derive a constraint. In this section, we provide a constructive proof of Theorem 13.

What we have to show is how any relation in $\mathcal{R}^*$ can be obtained from $\mathcal{R}$ in the SPJ-algebra. The proof of Theorem 11 in [13], which was outlined briefly above, indicates how to construct an expression which generates any given relation in $\mathcal{R}^*$ using projection, selection, Cartesian product and union operators. Because of the presence of the binary disequality relation $\neq_D$, disequality selection can easily be simulated in the SPJ-algebra. Thus the only operator which presents any difficulty is the union operator.

Since the union of two n-ary relations is equal to the complement (with respect to D^n) of the intersections of their complements, it will be sufficient to construct an expression which yields the complement of a given relation. This is the purpose of the remainder of this proof.

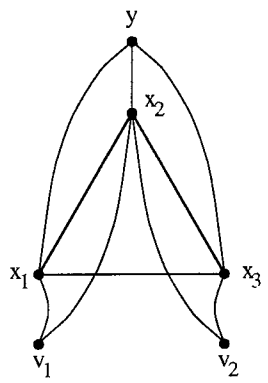


Fig. 1. The construction of a witness y for the variables v_1 and v_2 in the case that $|D| = 4$. All edges are labeled with the binary disequality constraint $\neq_D$.

We shall proceed in steps. A construction which will re-occur in several of the steps is the following. Given variables v_1 and v_2, and a domain D with $|D| \geq 3$, construct a constraint network as follows. First, add a complete graph on $|D|-1$ new variables $x_1, \ldots, x_{|D|-1}$. Then, connect v_1 to each of $x_1, \ldots, x_{|D|-2}$ (but not to $x_{|D|-1}$) and v_2 to each of $x_2, \ldots, x_{|D|-1}$ (but not to x_1). Next, add a variable y and connect y to each of $x_1, \ldots, x_{|D|-1}$. Finally, label each edge with the binary disequality constraint $\neq_D$. (Figure 1 illustrates this construction for $|D| = 4$.) The constraint satisfaction problem corresponding to the constructed network has the following properties:

(i) whenever v_1 and v_2 are assigned the same color in a solution, y must also be assigned that color; and
(ii) whenever v_1 and v_2 are assigned different colors, every assignment of color to y is further extendible to a solution.

We shall call y a *witness* of v_1 and v_2. (Observe that the construction fails for $|D| < 3$.)

More generally, we define the following notion.

Definition 15. Let D be a finite domain. In a constraint network, a variable y is called a *witness* for the variables $v_1, \ldots, v_n$ if the following properties hold:

(i) whenever all of $v_1, \ldots, v_n$ are assigned the same color in a solution, y must also be assigned that color; and
(ii) whenever not all of $v_1, \ldots, v_n$ are assigned the same color, every assignment of color to y is further extendible to a solution.

The first step of our construction is contained in the following lemma.

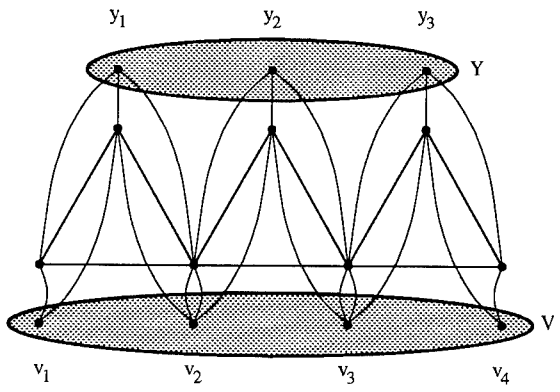


Fig. 2. The recursive step in the constructions in the proofs of both Lemma 16 and 17 for $|D| = 4$ and $n = 4$. All binary edges are labeled with the binary disequality constraint $\neq_D$.

Lemma 16. *Let D be a finite domain containing at least three colors and let n be a number, $n \geq 1$. Let $v_1, \ldots, v_n$ be distinct variables. A constraint network containing $v_1, \ldots, v_n$ in which some variable y is a witness for $v_1, \ldots, v_n$ can be effectively constructed.*

Proof. If $n = 1$, the isolated node v_1 is the desired network, since an individual variable is always its own witness.

For larger values of n, we proceed recursively. First, construct a constraint network containing, besides the variables $v_1, \ldots, v_n$, variables $y_1, \ldots, y_{n-1}$ such that, for $i = 1, \ldots, n-1$, y_i is a witness of v_i and v_{i+1}. (Figure 2 illustrates this construction for $|D| = 4$ and $n = 4$.) The constraint satisfaction problem corresponding to the constructed network has the following properties, which can readily be deduced from the analogous properties of the witness construction for $n = 2$:

(i) whenever all of $v_1, \ldots, v_n$ are assigned the same color in a solution, all of $y_1, \ldots, y_{n-1}$ must also be assigned that color; and

(ii) whenever not all of $v_1, \ldots, v_n$ are assigned the same color, there exists i, $1 \leq i \leq n-1$ such that every assignment of color to y_i is further extendible to a solution.

The above properties guarantee that, if we add a constraint network in which y is a witness for $y_1, \ldots, y_{n-1}$ to the one constructed above, y will be a witness for $v_1, \ldots, v_n$.

The next step of the construction is contained in Lemma 17, which has a very similar proof to Lemma 16.

Lemma 17. *Let D be a finite domain containing at least three colors and let n be a number, $n \geq 2$. Let $v_1, \ldots, v_n$ be distinct variables. A constraint network containing $v_1, \ldots, v_n$, in which the constraint induced on $V = \{v_1, \ldots, v_n\}$ is the complement of the n-ary equality relation over D, can be effectively constructed.*

Proof. Of course, the complement of the binary equality relation is the binary disequality relation. Thus, if $n = 2$, the network consisting of the nodes v_1 and v_2 linked by an edge labeled with the binary disequality relation $\neq_D$ is the desired network.

For larger values of n, we proceed recursively. First, as in the proof of Lemma 16, construct a constraint network containing, besides the variables v_1, $\ldots, v_n$, variables $y_1, \ldots, y_{n-1}$ such that, for $i = 1, \ldots, n-1$, y_i is a witness of v_i and v_{i+1}. (Figure 2 illustrates this construction for $|D| = 4$ and $n = 4$.) Then, add a constraint network which includes the variables in the set $Y = \{y_1, \ldots, y_{n-1}\}$ and induces the constraint on these variables which is the complement of the $n-1$-ary equality relation. By properties (i) and (ii) of the previous construction, described in the proof of Lemma 16, the constraint induced on $V = \{v_1, \ldots, v_n\}$ is the complement of the n-ary equality relation.

The n-ary equality relation imposed on the variables $v_1, \ldots, v_n$ is a special case of a constraint of the form $\bigwedge_{1 \leq i \neq j \leq n} v_i \triangle_{ij} v_j$, with "$\triangle_{ij}$" either "$=$" or "$\neq$." We next show that the complements of all constraints of this form can be computed in the SPJ-algebra from the binary equality and disequality constraints.

Lemma 18. *Let D be a finite domain containing at least three colors and let n be a number, $n \geq 1$. Let $v_1, \ldots, v_n$ be distinct variables. Let r be any n-ary relation over D on $v_1, \ldots, v_n$ of the form $\bigwedge_{1 \leq i \neq j \leq n} v_i \triangle_{ij} v_j$, with "$\triangle_{ij}$" either "$=$" or "$\neq$." A constraint network containing $v_1, \ldots, v_n$, in which the constraint induced on $V = \{v_1, \ldots, v_n\}$ is the complement of r, can be effectively constructed.*

Proof. First, verify whether, for all i, j, k, $1 \leq i < j < k \leq n$, "$\triangle_{ik}$" equals "$=$" each time "$\triangle_{ij}$" and "$\triangle_{jk}$" both equal "$=$." If this is not the case, r is the empty relation, whence its complement is D^n, for which, of course, a constraint network can be constructed. Otherwise, let $\{V_1, \ldots, V_m\}$ be the partition induced on V by the equivalence relation $\equiv$ defined by $v_i \equiv v_j$ if and only if "$\triangle_{ij}$" equals "$=$." If $m = 1$, then r is the n-ary equality relation, whence the lemma holds by Lemma 17. Thus suppose $m > 1$. Construct a constraint network containing, besides the variables $v_1, \ldots, v_n$, variables $y_1, \ldots, y_m$ such that, for $l = 1, \ldots, m$, y_l is a witness of V_l. (Such a network can be effectively constructed by Lemma 16.) Add to the network a complete graph with $|D| - m + 1$ variables, $x_1, \ldots, x_{|D|-m+1}$, and label each of its edges with the binary disequality constraint $\neq_D$. Finally, connect each of $y_1, \ldots, y_m$ to each of $x_1, \ldots, x_{|D|-m+1}$, also by an edge labeled with the binary disequality constraint $\neq_D$. (Figure 3 illustrates this construction for $|D| = 5$ and $m = 3$.)

We now prove that the constraint induced on V by the constructed network is indeed the complement of r. First, suppose that a certain assignment of colors to $v_1, \ldots, v_n$ satisfies the constraint imposed by the complement of r on V.

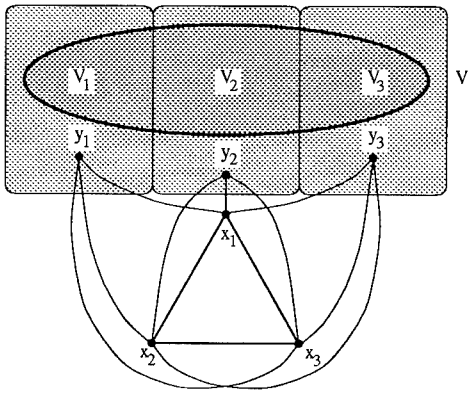


Fig. 3. Construction of the constraint network in the proof of Lemma 18 for $|D| = 5$ and $m = 3$. The heavily shaded edge is V; the lightly shaded areas represent the sub-networks required to construct witnesses for the classes into which V is partitioned.

Hence there exist i, j, $1 \leq i < j \leq n$, such that $v_i \ \triangle_{ij} \ v_j$ is not satisfied. We distinguish two cases. If "$\triangle_{ij}$" equals "$=$," then let V_l, $1 \leq l \leq m$, be the class of the partition of V to which v_i and v_j both belong. Since v_i and v_j are assigned different colors, the witness point y_l can be assigned any color. Now, extend the color assignment to $v_1, \ldots, v_n$ validly to $y_1, \ldots, y_{l-1}, y_{l+1}, \ldots, y_m$. (If choices are possible, make them arbitrarily.) Then, assign to y_l any color used to color the other witness points. If, on the other hand, "$\triangle_{ij}$" equals "$\neq$," then let V_{l_1} and V_{l_2}, $1 \leq l_1, l_2 \leq m$, be the distinct classes of the partition of V to which v_i respectively v_j belong. Since v_i and v_j are assigned the same color, that color can also be used to color both y_{l_1} and y_{l_2}. Now, extend the color assignment validly to the remaining witness points. (Again, if choices are possible, make them arbitrarily.) In both cases, at most $m - 1$ distinct colors have been used to color the witness points. Hence, at least $|D| - m + 1$ distinct colors remain to color $x_1, \ldots, x_{|D|-m+1}$, and the original assignment of colors to $v_1, \ldots, v_n$ has been extended to a solution of the constraint satisfaction problem on the network. Conversely, assume the constraint satisfaction problem on the constructed network has a solution. Since $|D|-m+1$ distinct colors are required to color $x_1, \ldots, x_{|D|-m+1}$, the points $y_1, \ldots, y_m$ are colored with at most $m - 1$ distinct colors. Thus at least two of these variables, say y_{l_1} and y_{l_2}, $1 \leq l_1, l_2 \leq m$, are assigned the same color. We distinguish two cases. If there exists l, $1 \leq l \leq m$, and v_i and v_j in V_l, $1 \leq i < j \leq n$, such that v_i and v_j are assigned different colors, then this color assignment violates the conjunct $v_i = v_j$ in r, whence the assignment to $v_1, \ldots, v_n$ satisfies the constraint imposed by the complement of r on V. If, on the other hand, for all l, $1 \leq l \leq m$, all variables in V_l are assigned the same color, then they must have been assigned the same color as y_l. Hence, for all v_i in V_{l_1} and v_j in V_{l_2}, $1 \leq i, j \leq n$, the color assignment

to v_i and v_j violates the conjunct $v_i \neq v_j$ or $v_j \neq v_i$ in r (whichever of the two occurs), whence also in this case the assignment to $v_1, \ldots, v_n$ satisfies the constraint imposed by the complement of r on V.

Observe that the construction for the general case in the proof of Lemma 18 does not work for the case $m = 1$, which is why this case was treated separately.

For the last-but-one step in the construction, we introduce the following notation.

Notation 3 *Let D be a finite domain and let r be an n-ary relation over D. Then $\tilde{r}$ denotes the $2n$-ary relation over D consisting of all $2n$-ary tuples t over D for which $\langle t[1], \ldots, t[n] \rangle$ is in r and for which there exists i, $1 \leq i \leq n$, such that $t[i] \neq t[n+i]$.*

Lemma 19. *Let D be a finite domain containing at least three colors and let n be a number, $n \geq 1$. Let $v_1, \ldots, v_n, w_1, \ldots, w_n$ be distinct variables. Let r be an n-ary relation over D on $v_1, \ldots, v_n$ of the form $\bigwedge_{1 \leq i \neq j \leq n} v_i \triangle_{ij} v_j$, with "$\triangle_{ij}$" either "$=$" or "$\neq$." A constraint network containing $v_1, \ldots, v_n, w_1, \ldots, w_n$, in which the constraint induced on $W = \{v_1, \ldots, v_n, w_1, \ldots, w_n\}$ is $\tilde{r}$, can be effectively constructed.*

Proof. First, construct a constraint network containing, besides the variables $v_1, \ldots, v_n, w_1, \ldots, w_n$, variables $y_1, \ldots, y_n$, such that, for $i = 1, \ldots, n$, y_i is a witness for v_i and w_i. (Such a network can be effectively constructed by Lemma 16.) Finally, augment the constraint network by adding an n-ary edge $V = \{v_1, \ldots, v_n\}$ labeled with the relation r (which is induced by a complete graph on $v_1, \ldots, v_n$ the edges of which are appropriately labeled with the binary equality and disequality relations $=_D$ and $\neq_D$) and an n-ary edge $Y = \{y_1, \ldots, y_n\}$ labeled with the complement of r (which is induced by some constraint network by Lemma 18). (Figure 4 illustrates this construction for $n = 4$.)

We now prove that the constraint induced on W by the constructed network is indeed $\tilde{r}$. First, suppose that a certain assignment of colors to $v_1, \ldots, v_n$, $w_1, \ldots, w_n$ satisfies the constraint imposed by the relation $\tilde{r}$ on W. Obviously, the assignment of colors to $v_1, \ldots, v_n$ satisfies the constraint imposed by the relation r on V. By assumption, there exists i, $1 \leq i \leq n$, such that v_i and w_i are assigned different colors. Hence, the witness point y_i can be assigned any color. Now, extend the color assignment to $v_1, \ldots, v_n, w_1, \ldots, w_n$ validly to $y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_m$. (If choices are possible, make them arbitrarily.) Finally, assign to y_i any color such that the color assignment to $y_1, \ldots, y_n$ satisfies the constraint imposed by the complement of r on Y. (For instance, if r satisfies the constraint $v_i = v_j$, then assign to y_i another color than to y_j; if, on the other hand, r satisfies the constraint $v_i \neq v_j$, then assign to y_i the same color as to y_j.) Then the original assignment of colors to $v_1, \ldots, v_n, w_1, \ldots, w_n$ has been extended to a solution of the constraint satisfaction problem on the whole network. Conversely, consider a solution of the constraint satisfaction problem on the constructed network. By definition, the assignment of colors to $v_1, \ldots, v_n$

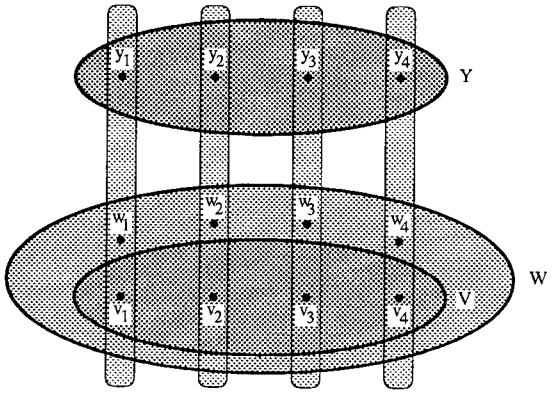


Fig. 4. Construction of the constraint network in the proof of Lemma 19 for $n = 4$. The lightly shaded areas represent the sub-networks required to construct witnesses for v_1 and w_1, v_2 and w_2, v_3 and w_3, and v_4 and w_4.

satisfies the constraint imposed by the relation r on V. Similarly, the assignment of colors to $y_1, \ldots, y_n$ satisfies the constraint imposed by the complement of r on Y. Hence there exists i, j, $1 \leq i \neq j \leq n$, such that y_i and y_j are assigned different colors whereas v_i and v_j are assigned the same color, or vice-versa. In both cases, the assumption that w_i is assigned the same color as v_i (and hence the same color as y_i) *and* w_j is assigned the same color as v_j (and hence the same color as y_j) readily leads to a contradiction. So, either v_i and w_i or v_j and w_j have different colors, whence the assignment of colors to $v_1, \ldots, v_n, w_1, \ldots, w_n$ satisfies the constraint imposed by $\tilde{r}$ on W.

We are now ready to make the last step in the construction.

Theorem 20. *Let D be a finite domain containing at least three colors and let n be a number, $n \geq 1$. Let $v_1, \ldots, v_n$ be distinct variables. Let r be any n-ary relation on $v_1, \ldots, v_n$. A constraint network containing $v_1, \ldots, v_n$ in which the constraint induced on $V = \{v_1, \ldots, v_n\}$ is the complement of r can be effectively constructed.*

Proof. We re-use a technique applied in the proof of Theorem 11. Let s be the number of tuples in r. Consider the Cartesian product r^s, which is an sn-ary relation, and let t be a tuple of r^s containing all tuples of r as a subtuple. Now, for all $i, j = 1, \ldots, sn$, $i \neq j$, perform the equality selection $\sigma_{i=j}$ if $t[i] = t[j]$ and the disequality selection $\sigma_{i \neq j}$ if $t[i] \neq t[j]$, and let $\hat{r}$ be the final result. Each tuple in the sn-ary relation $\hat{r}$ over D is a concatenation of tuples of r into a single sn-ary tuple. Not all $s!$ possible concatenations need occur, however.

We start the actual construction by a constraint network containing, besides the variables $v_1, \ldots, v_n$, variables $w_1, \ldots, w_{sn}$, and single edge $W = \{w_1, \ldots, w_{sn}\}$

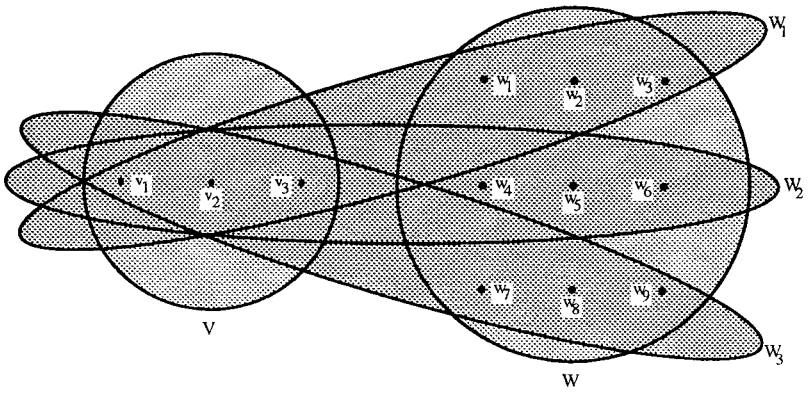


Fig. 5. Construction of the constraint network in the proof of Theorem 20 for $n = 3$ and $s = 3$.

which is labeled with $\hat{r}$. Next, for $k = 1, \ldots, s$, let r_k be the n-ary relation over D defined on $w_{(k-1)n+1}, \ldots, w_{kn}$ by $\bigwedge_{1 \leq i \neq j \leq n} w_{(k-1)n+i} \ \triangle_{ij} \ w_{(k-1)n+j}$, with "$\triangle_{ij}$" being "$=$" if $t[(k-1)n+i] = t[(k-1)n+j]$, and "$\neq$" if $t[(k-1)n+i] \neq t[(k-1)n+j]$. By construction, t above may be replaced by any tuple in $\hat{r}$ without altering the definitions of $r_1, \ldots, r_s$. We complete our construction by adding to the network, as follows: for $k = 1, \ldots, s$, we add the $2n$-ary edge $W_k = \{w_{(k-1)n+1}, \ldots, w_{kn}, v_1, \ldots, v_n, \}$, labeled with $\tilde{r}_k$ (which is induced by some constraint network, by Lemma 19). (Figure 5 illustrates this construction for $n = 3$ and $s = 3$.)

We now prove that the constraint induced on V by the constructed network is indeed the complement of r. First, suppose that a certain assignment of colors to $v_1, \ldots, v_n$ satisfies the complement of r on V. Assign to $w_1, \ldots, w_{sn}$ any tuple of $\hat{r}$. It is readily verified that the color assignment thus obtained is a solution of the constraint satisfaction problem on the constructed network. Conversely, consider a solution of the constraint satisfaction problem on the constructed network. In this solution, the assignment of colors to $w_1, \ldots, w_{sn}$ is a concatenation of the tuples of r, since the edge W is constrained by $\hat{r}$. Let $t_1, \ldots, t_s$ be the tuples of r in the order they occur in the present coloring. In particular, $t_1, \ldots, t_s$ satisfy $r_1, \ldots, r_s$, respectively. For each $k = 1, \ldots, s$, the assignment of colors to $v_1, \ldots, v_n$ yields an n-ary tuple different from t_k. Hence the assignment of colors to $v_1, \ldots, v_n$ satisfies the constraint imposed by the complement of r.

6 Conclusion

The results in this paper highlight once again how fruitful the interaction between constraint satisfaction problems and database theory can be. In previous papers [6, 9] we applied results from database theory to obtain new results about

constraint satisfaction problems. In contrast, the results in this paper were obtained by linking recent results on constraint satisfaction to relational database theory.

These results also show that the expressive power of a set of constraints is very closely related to the closure properties of the constraints as described in Definition 8. The authors are now working on generalizing the construction exhibited in this paper to arbitrary sets of constraints (with or without disequality).

References

1. F. Bancilhon. On the completeness of query languages for relational data bases. In *Proceedings 7th International Symposium on Mathematical Foundations of Computer Science* (Zakopane, Poland), *Lecture Notes in Computer Science*, 64, Springer-Verlag, Berlin/New York, 1978, pp. 112–123.
2. W. Bibel. Constraint satisfaction from a deductive viewpoint. *Artificial Intelligence*, 35, 1988, pp. 401–413.
3. D. Cohen, P. Jeavons, M. Gyssens. Derivation of constraints and database relations. *Technical Report* CSD-TR-96-01, Royal Holloway, Univ. of London, January 1996.
4. R. Dechter. Decomposing a relation into a tree of binary relations. *Journal of Computer and System Sciences*, 41, 1990, pp. 2–24.
5. E.C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32, 1985, pp. 755–761.
6. M. Gyssens, P.G. Jeavons, and D.A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66, 1994, pp. 57–89.
7. R.M. Haralick and L.G. Shapiro. The consistent labeling problem: Part I. *IEEE Trans. Pattern. Anal. Mach. Intell.*, 1, 1979, pp. 173–184.
8. P.G. Jeavons. On the algebraic structure of combinatorial problems. *Technical Report* CSD-TR-95-15, Royal Holloway, Univ. of London, October 1995.
9. P. Jeavons, D. Cohen, and M. Gyssens. A structural decomposition for hypergraphs. In *Proceedings Jerusalem Combinatorics '93*, H. Barcelo and G. Kalai, eds. *Contemporary Mathematics*, 178, 1994, pp. 161–177.
10. P. Jeavons, D. Cohen, and M. Gyssens. A unifying framework for tractable constraints. In *Proceedings CP '95*, *Lecture Notes in Computer Science*, 976, Springer-Verlag, Berlin/New York, 1995, pp. 276–291.
11. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8, 1977, pp. 99–118.
12. U. Montanari. Networks of constraints: fundamental properties and applications to picture processing. *Information Sciences*, 7, 1974, pp. 95–132.
13. J. Paredaens. On the expressive power of the relational algebra. *Information Processing Letters*, 7:2, 1978, pp. 107–111.
14. J.D. Ullman. *Principles of Database and Knowledge Base Systems*, Vols. I and II. Computer Science Press, Rockville, Maryland, 1988 and 1989.
15. Y. Zhang and A.K. Mackworth. Parallel and distributed algorithms for finite constraint satisfaction problems. In *Proceedings 3rd IEEE Symposium on Parallel and Distributed Computing* (Dallas, Texas), 1991, pp. 394–397.

Constraint Programming: an Efficient and Practical Approach to Solving the Job-Shop Problem

Yves Colombani

Laboratoire d'Informatique de Marseille - URA CNRS 1787
Faculté des sciences de Luminy, case 901
163 av. de Luminy 13288 Marseille Cedex 9, France
Tel: (33) 91 26 93 58 - Fax: (33) 91 26 92 75
Yves.Colombani@lim.univ-mrs.fr

Abstract. Recent improvements in constraint programming have made it possible to tackle hard problems in a practical way. Before this, these problems were solved only by specialized programs often complex to implement. Scheduling problems and more especially the job-shop problem belong to this class. In this paper we explain a relatively simple constraint system, which enables us to solve 10×10 problems efficiently. The method described here, based on evaluations which come as close as possible to release and due dates of jobs to be scheduled, requires no prior knowledge of the problem being processed, in particular, no bounds over optimum value (consequently no specific algorithm to find approximate solutions). We also comment on the results of experiments on known problems. As far as we know, the system outlined here is the only one that, using just constraint solving and an exhaustive enumeration strategy, can completely solve orb3[AC91] in less than half an hour computational time.

Keywords: Job-Shop Scheduling, Constraint Programming, Efficiency.

Introduction

The disjunctive scheduling problem or job-shop problem is an NP-hard problem of which the MT10[MT63] is an example has been solved for the first time 20 years after its presentation[CP89]. If at first, this type of problem was considered to be an operational research (OR) speciality, improvements in constraint programming have lent impetus to other developments around languages of artificial intelligence[AB92, CL94, BLPN95]. The latter, even if they do not always present performances equal to the ones of OR products[CP94], have on the other hand the advantage of being easier to implement and to adapt because of the expressive powers of constraint programming. In order to reach performances of specific developments, some languages possess specialized procedures and/or use approximate techniques (simulated annealing, stochastic algorithms...). A drawback is that the former generality of these languages is restricted when using these extensions.

In this paper we describe several mechanisms that provide a standard constraint solver with the means to tackle these scheduling problems. The method suggested rests on bounding, as exact as possible, of values of variables and on the use of immediate selections[CP89]. Moreover, the enumeration strategy which we have used allows us to completely solve 10×10 problems. We then use some well known examples that highlight the efficiency of our approach. Although relatively simple, our algorithm offers performances which are comparable and, indeed even superior, to other achievements in this field.

The paper is organized as follows: the first part briefly presents principles of constraint programming and describes in detail different components of a scheduling problem of the job-shop type. The second part explains several properties of the problem intended to complete the initial constraint system. The third part presents the constraint solver used, its employment and the selected enumeration strategy. Finally, experimental results are discussed in the last part.

1 Job-Shop and Constraint System

1.1 Constraint Solving

Constraint programming consists essentially in describing problems by means of relations between variables[VH89]. The constraint solver has to make the constraint system consistent, in order to ensure that a solution is *possible* for a given set of constraints (arc consistency[Mac77]). Thus, it can give a solution only if the constraint system is strong enough. Consequently, the difficulty implied by such a technique is in defining a constraint system capable of leading the solver to a solution. This description may be divided into three main parts:

1. an initial system of elementary constraints describing problem data, which is directly built from the problem definition;
2. a global constraint system not directly implied by the initial system, involving exploitation of the properties of the problem;
3. an enumeration algorithm and its heuristics to isolate effective solutions.

The last part is needed because whatever the quality of the constraint system may be, some variables cannot be found. Indeed, in most cases, several equivalent solutions can be exhibited.
Therefore, the efficiency of a solving algorithm in this particular case essentially depends both on the "power" of the global system and on the quality of the enumeration strategy. Obviously, we want such a strategy to construct the smallest possible search tree (in terms of the total number of nodes).

1.2 The Job-Shop Scheduling Problem

A job-shop scheduling problem is defined by a set of n *jobs* which has to be executed on m *machines*. Each job consists of a sequence of m *operations* assigned to the m *machines*. The objective is to find the shortest possible schedule (i.e.

a processing order for all operations) considering that each machine can handle, at most, one job at a time and preemption is not allowed (each operation which is begun must be ended without interruption).
A $n \times m$ problem will designate a job-shop of n jobs for m machines that represents $n \times m$ operations to schedule.

Each of n jobs T_i has a specified processing order $\sigma_i = (\sigma_i^1, ..., \sigma_i^m)$ on the m machines. A job T_i is represented for each machine k by the operation i^k. Each operation i^k is defined by its duration p_i^k, the date from which it is ready to be executed r_i^k (release date) and the date after which it must be finished d_i^k (due date). All the r_i^k, d_i^k, p_i^k are integers. The p_i^k are given whereas the r_i^k, d_i^k are unknown.

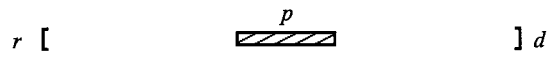


Fig. 1. An operation

In the following, $i^k \prec j^k$ will mean operation i^k precedes operation j^k on a given machine k. Moreover, to simplify notation, references to machines will be omitted where there no ambiguity, so $r_i + p_i \leq r_j$ should be understood as $r_i^k + p_i^k \leq r_j^k$ for any k.

The job-shop problem is easy to explain with relations between various initial data:

- relation between release date and due date of the same operation;

$$\forall k \in \{1..m\}, \forall i \in \{1..n\},\ r_i^k + p_i^k \leq d_i^k \tag{1}$$

- order between operations belonging to a given job;

$$\forall i \in \{1..n\}, k \in \{1..m-1\},\ d_i^{\sigma_i^k} \leq r_i^{\sigma_i^{k+1}} \tag{2}$$

- mutual exclusion between operations belonging to a given machine.

$$\forall k \in \{1..m\}, \forall i \in \{1..n\}, j \neq i \in \{1..n\},\ [d_i^k \leq r_j^k] \oplus [d_j^k \leq r_i^k] \tag{3}$$

Although it expresses all problem data, this constraint set is not sufficient for the constraint solver to find a solution: at the most it makes it possible to validate a proposal. This occurs for two main reasons. On the one hand, the local nature of solving mechanisms (arc consistency) makes it possible to propagate information only between variables directly linked through relations. The following example illustrates what this particularity involves: if two operations $o1$ and $o2$ not yet ordered precede a third $o3$, the release date lower bound of the latter will be raised to $\max(r_{o1} + p_{o1}, r_{o2} + p_{o2})$. As in every case these two operations will be executed before $o3$, it is obvious that $r_{o_3} \geq \min(r_{o_1}, r_{o_2}) + p_{o1} + p_{o2}$. As long

as the disjunction $o1/o2$ is not decided, the proposed constraint system cannot deduce this relation so, the release date lower bound may be easily improved. On the other hand, for a given cost (i.e. scheduling duration) several solutions can be exhibited. Indeed, shifting or swapping some operations can produce different solutions with the same cost. In this case, the work of the solver has to be completed by an enumeration step because it is unable to make choices (cf. 3.4). However, solver precision can be improved using constraints over groups of variables. In the following section, we shall study some properties of this particular problem in order to define these *global constraints*. These constraints belong to two categories: the first one improves values of release and due dates making them more precise whereas the second one detects which disjunctions accept only one order (immediate selections).

2 The Global System

In this part we present several properties of the initial problem which are not implied by the solver system. These properties are relations shared by groups of variables associated with concurrent operations over a given machine. Three levels of the global constraint system are concerned by this:

1. improvement of precision of problem variables;
2. detection of configurations that do not accept any solutions;
3. deciding between disjunctions that accept only one direction.

Thus, we assume problem data is examined for a partially established schedule.

2.1 Processing Time Estimation

As for a single operation, a set J of operations (over a given machine) is forced to be executed in a determined space of time. In the first estimation, this space is bounded by the *release date* r_J of the set and its *due date* d_J.

$$r_J = \min_{i \in J}(r_i)$$

$$d_J = \max_{i \in J}(d_i)$$

The objective is to calculate the minimal duration a machine needs to execute this operation set in order to improve these two dates.

Function $\mathcal{E}$. The activity duration is at least equal to the sum of durations of the operations to be treated. From this we deduce a natural lower bound of the schedule of J.

$$d_J \geq \min_{i \in J}(r_i) + \sum_{j \in J} p_j$$

In this case the machine is assumed to handle all operations without interruption; nevertheless some operations, not yet released, force it to stop for waiting

periods during which it is inactive. Adding these pause periods to the sum of the durations gives a better estimation for the total treatment duration. This second estimation is obtained executing the operations in increasing release date order without ignoring constraints over these dates.

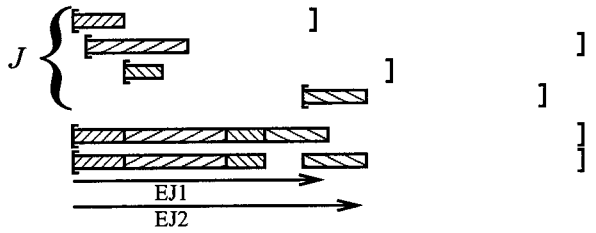


Fig. 2. Schedule at the earliest

In the example of Fig.2, EJ1 is the sum of operation durations and EJ2 the scheduling "at the earliest". The fourth operation forces the machine to stop for a waiting period which is taken into account by EJ2 but not by EJ1.
The function $\mathcal{E}(J)$ computes the schedule "at the earliest".

Function $\mathcal{E}(J)$:**Date**
{
$J' \leftarrow \varnothing$
$t \leftarrow 0$
While $(J \backslash J' \neq \varnothing)$
{
i such as $r_i = \min_{j \in J \backslash J'} (r_j)$
$t \leftarrow \begin{cases} r_i \leq t : t + p_i \\ r_i > t : r_i + p_i \end{cases}$
$J' \leftarrow J' \cup \{i\}$
}
}
Return (t)

Computation of $\mathcal{E}(J)$

Property 1 *The function $\mathcal{E}(J)$ is a lower bound of the set J release date.*

$$d_J \geq \mathcal{E}(J)$$

Function $\mathcal{D}$. The release date of J has an immediate upper bound built in the same way as the due date lower bound:

$$r_J \leq \max_{i \in J}(d_i) - \sum_{j \in J} p_j$$

The calculus of the schedule "at the latest" gives a better bound of this date. It is computed processing operations in decreasing due date order respecting these date constraints.

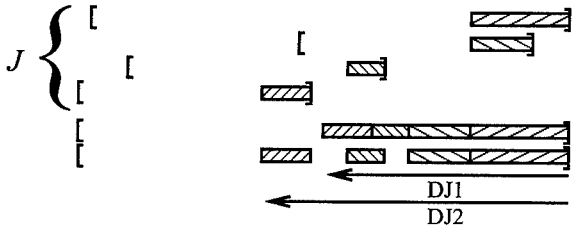


Fig. 3. Schedule at the latest

This second example (Fig.3) has two waiting periods in its schedule "at the latest" (DJ2). Once again, a simple sum of operation durations is not precise enough (DJ1).
The function $\mathcal{D}(J)$ processes the schedule "at the latest".

Function $\mathcal{D}(J)$:**Date**
{
$J' \leftarrow \varnothing$
$t \leftarrow \infty$
While $(J \backslash J' \neq \varnothing)$
{
i such as $d_i = \max_{j \in J \backslash J'} (d_j)$
$t \leftarrow \begin{cases} d_i \geq t : t - p_i \\ d_i < t : d_i - p_i \end{cases}$
$J' \leftarrow J' \cup \{i\}$
}
}
Return (t)

Computation of $\mathcal{D}(J)$

Property 2 *The function $\mathcal{D}(J)$ is an upper bound of the set J release date.*

$$r_J \leq \mathcal{D}(J)$$

2.2 Maximum Delay

The *maximum delay* δ of a set J denotes the difference between the total amount of time allotted to the machine to process all operations of J and the estimated

duration of this treatment[1].

$$\delta_J = (d_J - r_J) - \sum_{i \in J} p_i$$

The maximum delay can be likened to a measure of degree of freedom of the machine, the greater δ is, the greater the freedom to the machine is to start its processing. Mutually the less δ is, the more the machine can be considered as *constrained.* In the same way that release/due dates have been bounded, the value of δ can be estimated more precisely. To do this, we have to first compare, the valuation of the schedule "at the earliest" with the release date of J then, the valuation of the schedule "at the latest" with the due date of J.

$$\delta_J \leq \max_{i \in J}(d_i) - \mathcal{E}(J)$$

$$\delta_J \leq \mathcal{D}(J) - \min_{i \in J}(r_i)$$

Consequently we select for the value of δ the minimum of these two estimations:

$$\delta_J \leq \min(\max_{i \in J}(d_i) - \mathcal{E}(J), \mathcal{D}(J) - \min_{i \in J}(r_i))$$

In the example of Fig.4, we have $RJ1 > RJ2 > RJ3$ thus we deduce $\delta_J \leq RJ3$.

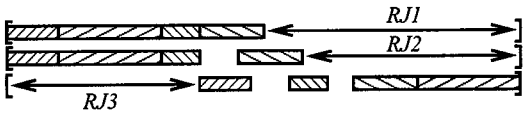


Fig. 4. The three valuations of the maximum delay

Property 3 *If the maximum delay is negative, then the current configuration does not accept any solution.*

2.3 Immediate Selections

In this section we recall part of the immediate selection algorithms from Carlier & Pinson[CP89]. The general method consists in considering each machine individually and trying to determine disjunctions that accept only one direction. Furthermore, at each step we verify the existence of feasible schedules with current data in order to identify configurations which cannot provide a solution.

Four Words of Vocabulary... An operation is said to be *totally ordered* if there exists no disjunction over this operation not yet decided.
A *clique* denotes a set of operations not totally ordered attached to a given machine (or sharing a given resource). The *input* of a clique is the operation that begins the schedule of the clique and the *output* the one that finishes it.

[1] It is also called "slack"

Input and Output of Clique Detection. For a given clique I we have to select which operation must be the input of the clique and/or which operation has to be the output.
With an aim to identifying operations that cannot take the lead of clique I; we compute for each operation i a lower bound of the due date of the set if i is the first processed operation. If this valuation exceeds the latest due date of I (except i because it is assumed to be at the head) the operation i can't precede the set.

$$r_i + \sum_{j \in I} p_j > \max_{j \in S_I \setminus \{i\}} d_j \tag{4}$$

Given set E_I initialized with I and reduced with test (4). E_I describes the set of operations candidates for the input of clique I. Similarly, we construct the set S_I of task candidates for the output of the clique I evaluating upper bounds of the release date of I for each i when this one is the last processed operation and comparing it with the earliest release date of I (except i). The relation (5) qualifies inappropriate candidates for the output of I.

$$d_i - \sum_{j \in I} p_j < \min_{j \in E_I \setminus \{i\}} r_j \tag{5}$$

Property 4 *If the set E_I amounts to a single element, then this element must be the input of clique I for any solution. Similarly, if the set S_I amounts to a single element, this element must be the output of I for any solution.*

Property 5 *If one of the sets E_I or S_I is empty ($I \neq \varnothing$) then no solution can be built with current configuration.*

3 Implementation

The properties mentioned above have led us to describe a constraint set which has been realized by means of a solver system based on interval arithmetic[BOV94]. This makes it possible to link variables through ordinary relations such as $z = x + y$, $x < y$...
The representation of boolean expressions is provided by intervals $[0, 1]$ which authorize a logic of three values (0:false, 1:true, $[0, 1]$: indeterminate).
As well as the classical functionalities for this kind of tool, our constraint solver prototype also offers the means to describe sets and constraints over sets. This particularity simplifies appreciably the implementation of immediate selections.

This section describes the job-shop problem in terms of constraints and gives the enumeration strategy used in experimentations.

3.1 Basic Constraints

All the relations described in section 1 are copied into the constraint system as they are. Nevertheless it should be noted that disjunctions (3) are represented by

three logic expressions associated with variables having their truth value. Thus, for each pair of operations i and j, we have:

$$\begin{cases} D_{ij} = [d_i \leq r_j] \\ D_{ji} = [d_j \leq r_i] \\ D_{ij} = \neg D_{ji} \end{cases}$$

If D_{ij} takes the *true* value, then D_{ji} becomes *false* and operation i is processed before operation j.

$$D_{ij} \Leftrightarrow i \prec j$$

3.2 Updating Release and Due Dates

Let us consider an operation i and build the set P_i of its predecessors and the set S_i of its successors. Relations of type (3) shared between S_i and i then P_i and i are converted into precedence relations:

$$\forall j \in S_i,\ d_i \leq r_j$$

$$\forall j \in P_i,\ d_j \leq r_i$$

The combined action of these two constraint groups enables us to verify the following relations:

$$r_i \geq \max_{j \in P_i}(d_j)$$

$$d_i \leq \min_{j \in S_i}(r_j)$$

Such a constraint system is not able to find good bounds for due and release dates; consequently, to improve these evaluations, we complete it with constraints built with the help of functions $\mathcal{E}$ and $\mathcal{D}$. These last two give much better estimations of the dates (Prop. 1 and 2):

$$r_i \geq \mathcal{E}(P_i) \tag{6}$$

$$d_i \leq \mathcal{D}(S_i) \tag{7}$$

3.3 The Sets

Using sets of integers is an efficient and practical way to implement immediate selection algorithms. Indeed, each element of a set (an integer) is associated with a particular task and its properties over the studied resource are stored as vectors such as r (release dates) or d (due dates). Partial orders are taken from a boolean matrix D (see above 3.1).

In this section we assume all information to be attached to a given machine, thus the following statements have to be done for each machine individually and are sufficient.

Sets Definition. Given the set *neo* of not yet ordered operations (i.e. a clique):

$$neo : \{1..n\} \backslash \{i \in \{1..n\} | \forall j \neq i \in \{1..n\}, (i \prec j) \vee (j \prec i)\}$$

Definitions set forth in the previous section do not take into account the fact that a part of the disjunctions of an operation set may already be decided. According to this information, we can add two complementary subsets of *neo*: the set e_1 describes operations that could be the input of the clique because there is no operation to process before; and s_1 characterizes operations that could be the output because they have no successor.

$$e_1 : neo \backslash \{i \in neo | \exists j \neq i \in neo, j \prec i\}$$

$$s_1 : neo \backslash \{i \in neo | \exists j \neq i \in neo, i \prec j\}$$

Moreover, the making up of sets of candidates for the input/output of the clique is based on the estimation of time required for treatment of the whole of the clique. Actually other operations, not belonging to the clique because they are totally ordered, have to be processed in the same span of time. To obtain good valuations of execution duration we complete *neo* with these operations. The set a_t is composed of operations to be processed in the time allotted to the machine to treat *neo*. It is built adding to *neo* all operations with at least one predecessor belonging to $e_1 \backslash s_1$ and at least one successor belonging to $s_1 \backslash e_1$.

$$a_t : neo \cup \{i \in \{1..n\} \backslash neo | \exists j \in e_1 \backslash s_1, \exists k \in s_1 \backslash e_1, (j \prec k) \wedge (j \prec i) \wedge (i \prec k)\}$$

Intermediate sets e_1, s_1 and a_t are used as a base to make up the sets of candidates for input and output *ee* and *es*.

$$ee : e_1 \backslash \{i \in e_1 | r_i + \sum_{j \in a_t} p_j > \max_{k \in es \backslash \{i\}} (d_k)\}$$

$$es : s_1 \backslash \{i \in s_1 | d_i - \sum_{j \in a_t} p_j < \min_{k \in ee \backslash \{i\}} (r_k)\}$$

Constraints Over Sets. While *neo* is not empty, sets *ee* and *es* have to contain at least one element (Prop. 5).

$$neo \neq \varnothing \wedge (ee = \varnothing \vee es = \varnothing) \Rightarrow \text{Fail}$$

The *ee* or *es* unique element determines clique input or output (Prop. 4).

$$\text{card}(ee) = 1 \Rightarrow \forall i \in ee, \forall j \in neo \backslash ee, i \prec j$$

$$\text{card}(es) = 1 \Rightarrow \forall i \in es, \forall j \in neo \backslash es, j \prec i$$

Maximum delay estimation must be positive for each set *neo* (Prop. 3).

$$\delta_{neo} < 0 \Rightarrow \text{Fail}$$

3.4 Enumeration Strategy

For a job-shop scheduling problem, a solution can be characterized by one of two variable sets: first, we can choose directly the release dates in which case, the solution is the list of beginning dates of each operation; or we can examine mutually exclusive constraints and the solution is described by a list of decided disjunctions. For this last case, the one we have selected here, the solution is a passage order of tasks defined by the list of ordered operation pairs for each machine (ex: $2 \prec 1, 2 \prec 4...$). This list is embodied by boolean variables which all receive a value when a solution is reached. For a problem of $n \times m$, $m \times \frac{n \times (n-1)}{2}$ variables must be found. This gives the search space a dimension of $2^{m \times \frac{n \times (n-1)}{2}}$ combinations[2].
At each choice point, a pair of non-ordered operations must be selected (i.e. choose one of not found boolean variables) next a passage order has to be picked (i.e. fix a truth value to the variable). It is worth noting that because of the size of the search space, the behavior of the enumeration procedure is determining. Two criteria are consequently needed. The first one is to elect the machine on which a decision has to be made and the second is to choose an operation pair to order and a mean to fix the direction of the disjunction (however other approaches could be used [Col96]).

Choice of a Machine. The aim of this first criterion is to designate the machine which will have, a priori, the strongest incidence over the rest of the system. In other words the machine for which the schedule should involve the greatest number of deductions (from the constraint system). This is why we first choose machines with lower maximum delay δ_{neo}. In case of a tie, the machine which has the smallest set *neo* is chosen.

Choice of a Pair of Operations. On the selected machine, a disjunction must be chosen. This choice is done in the set *neo* (assigned to the selected machine) among non-ordered disjunctions. To "help" the system, we choose first a pair which seems to be difficult to decide between. Thus, for a given set J we take the pair (i, j) that minimizes the difference between the release date of the first and the due date of the second:

$$(i, j) / \min(r_i - d_j)$$

The disjunction will take the direction $i \prec j$.

Optimality Search. For this kind of problem, two search techniques can be employed regarding optimality:

[2] An important amount of combinations are inconsistent (such as $1 \prec 2, 2 \prec 3$ with $3 \prec 1$) and are not explored.

- **Min-Max**: for each found solution S with cost C_S, the enumeration procedure is re-initialized with the new constraint: $C < C_S$;
- **Minimize**: we use a specialized predicate *minimize*(C) which allows, each time a solution S is found, for backtracking into the search tree until the constraint $C < C_S$ is accepted (i.e. it does not cause the system to fail) and then added. Next the enumeration continues with the new constraint system.

This second solution gives better results with the selected enumeration. It should also be noted that whatever the technique employed is, an initial upper bound may be fixed (i.e. an initial cost C_I) in order to reduce the search time. In this case, another algorithm is needed to find this initial cost...

4 Experimental Results

The test-set is made up of 18 problems taken from literature and reputed to be hard (orb1-10, la16-20, abz5-6, mt10) added to a collection of 50 randomly generated problems all of size 10×10. The resolution of each problem has been entirely submitted to our system. That means that no upper bound is fixed a priori and no lower bound is required. Optimum search and proof of optimality are performed in only one execution of the enumeration procedure. The following table summarizes results for the 10 best known problems. Times are expressed in seconds and have been obtained on a SparcStation 5. The first three columns refer to respectively the problem's usual name, its optimal cost (Opt) and the first found solution cost (Sol1). The next two columns give the number of choice points (PcOpt) and the time (Topt) needed to reach an optimal solution. Finally, the two last columns give the total number of choice points (PcTot) and the total amount of time (Ttot) to find the best solution and to prove its optimality.
Note that the first solution is reached in less than one second in all cases.

Table 1. Resolution for 10 well known problems

Problem	Opt	Sol1	PcOpt	Topt(s)	PcTot	Ttot(s)
mt10	930	1028	5178	55	10991	116
orb1	1059	1335	43592	431	48842	495
orb2	888	991	5709	49	8032	71
orb3	1005	1194	120064	1221	135394	1393
orb4	1005	1193	2626	21	6538	64
orb5	887	989	5878	50	6936	62
abz5	1234	1468	169430	325	172162	350
abz6	943	1053	1140	11	1536	15
la19	842	946	3023	21	5809	45
la20	902	964	14337	109	15434	120

Among the 50 random problems, only 2 need more than 2 minutes of computational time; problems generated by these means seem to be clearly easier!

Moreover, computational times required by all other problems, not mentioned in the above table, belong to a range from 3s to 140s.
Although no particular technique is employed with this aim, the quality of the first solution cost is often about 10% of the optimum. Also, we have observed that the search is performed step by step, i.e. intermediate solutions are clustered and each group of solutions needs a certain amount of computational time but all solutions of a given group are found quickly. This phenomenon is particularly significant for the abz5 problem: after 5s there is a solution at 1392; then at 30s a second stage stops at 1357; the next step is reached after 300s for a cost of 1346 which is converted to 1242; the last step begins 40s later for a cost of 1239 immediately followed by the optimum 1234. From this remark, we deduce that the quality of an upper bound depends on the group of solutions it appears in (two values extracted from the same cluster certainty give the same results): for the abz5 example, there are four groups. First over 1391, then between 1391 and 1357, then between 1356 and 1242 and to finish between 1241 and 1234. Of course the nearer to optimum the upper bound is the better the result is but differences between values of a given group are not significant.

In order to compare our algorithm to that of Applegate & Cook, we have also executed resolutions employing such upper bounds as put forth in [AC91].

Table 2. Resolution for 10 well known problems with upper bounds[AC91]

Problem	Opt	Bsup	PcTot	Ttot(s)	PcTot[AC91]	Ttot[AC91](s)
mt10	930	930	7105	74	16055	372
orb1	1059	1070	25896	306	71812	1482
orb2	888	890	4539	40	153578	2484
orb3	1005	1021	113549	1248	130181	2297
orb4	1005	1019	4509	47	44547	1013
orb5	887	896	5483	52	23113	526
abz5	1234	1245	8371	68	57848	951
abz6	943	943	527	5	1269	90
la19	842	848	3549	28	93807	1462
la20	902	911	12195	89	81918	1402

Even if times cannot be seriously compared (the computers used are probably too different), the number of choice points can be taken as criterion and over these examples, our system offers better performances.

Table 3 shows the times required to completely solve (optimum+proof) the 10 problems by IlogSchedule[BLPN95] and Task Intervals system[CL95]. The times, expressed in seconds, were obtained on a RS6000 computer for the first column and on a Pentium 90 for the second column. The last column of the table shows our performances on SparcStation 5.

Table 3. Compared performances

Problem	[BLPN95]	[CL95]	LocPerf
mt10	235	151	116
orb1	407	189	495
orb2	507	31	71
orb3	606	588	1393
orb4	213	371	64
orb5	210	89	62
abz5	282	127	350
abz6	100	8	15
la19	269	100	45
la20	496	78	120

These two other techniques, also based on constraint solving, use additional approximate methods to compute good bounds before starting enumeration in order to reduce the search tree. This is why the numbers of choice points are not indicated, indeed this measure is significant only in the enumeration procedure. These pre-treatments (one step for IlogSchedule and two steps for Task Intervals) obviously need developments much more complex than the ones suggested in this paper. Our algorithm is nevertheless still competitive.

5 Conclusion

We have presented a relatively simple algorithm based exclusively on constraint programming which is capable of solving 10×10 job-shop problems efficiently. Based on the estimation of release/due dates of operations, this method does not depend on the constraint solver used. Moreover, the power of the suggested system is emphasized by the rudimentary nature of the enumeration strategy which is sufficient, both, for finding optimal solution of the problem, and for proving its optimality without any prior knowledge. Although appreciably simpler, this method often proves to be more efficient than other techniques in the same field (constraint solving); and remains comparable to combined techniques (such as approximate + enumerative).

In spite of these encouraging results, this method can't tackle larger problems. In order to process larger problems, we are at present working on an extension of the set system and we are aiming toward the development of a more *intelligent* enumeration strategy.

References

[AB92] A. Aggoun and N. Beldiceanu. Extending CHIP in Order To Solve Complex Scheduling and Placement Problems. In *Journées Francophones de Programmation Logique*, 1992.

[AC91] David Applegate and William Cook. A Computational Study of the Job-Shop Scheduling Problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[BLPN95] Philippe Baptiste, Claude Le Pape, and Win Nuijten. Constraint-Based Optimisation and Approximation for Job-Shop Scheduling. In *Proceedings of the AAAI-SIGMAN Workshop on Intelligent Manifacturing Systems, IJCAI-95*, 1995.

[BOV94] Frédéric Benhamou, William J. Older, and André Vellino. Constraint Logic Programming on Boolean, Integer and Real Intervals. Accepted for publication in Journal of Symbolic Computation, 1994.

[CL94] Yves Caseau and François Laburthe. Improved CLP Scheduling with Task Intervals. In *International Conference on Logic Programming*, 1994.

[CL95] Yves Caseau and François Laburthe. Disjunctive Scheduling with Task Intervals. Technical Report 95-25, LIENS, 1995.

[Col96] Yves Colombani. Stratégies d'énumération pour le problème du job-shop. Technical Report 141, LIM, 1996.

[CP89] Jacques Carlier and Éric Pinson. An Algorithm for Solving the Job-Shop Problem. *Management Science*, 35(2):164–176, 1989.

[CP94] Jacques Carlier and Éric Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994.

[Mac77] A.K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[MT63] J.F. Muth and G.L. Thompson. *Industrial scheduling.* Prentice Hall, 1963.

[VH89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

An Instance of Adaptive Constraint Propagation

Hani El Sakkout, Mark G. Wallace, and E. Barry Richards

IC-Parc, Imperial College, London SW7 2AZ, United Kingdom.
Email: {hhe,mgw,ebr} @doc.ic.ac.uk

Abstract. Constraint propagation algorithms vary in the strength of propagation they apply. This paper investigates a simple configuration for *adaptive propagation* – the process of varying the strength of propagation to reflect the dynamics of search. We focus on two propagation methods, Arc Consistency (AC) and Forward Checking (FC). AC-based algorithms apply a stronger form of propagation than FC-based algorithms; they invest greater computational effort to detect inconsistent values earlier. The relative payoff of maintaining AC during search as against FC may vary for different constraints and for different intermediate search states. We present a scheme for Adaptive Arc Propagation (AAP) that allows the flexible combination of the two methods. Meta-level reasoning and heuristics are used to dynamically distribute propagation effort between the two. One instance of AAP, Anti-Functional Reduction (AFR), is described in detail here. AFR achieves precisely the same propagation as a pure AC algorithm while significantly improving its average performance. The strategy is to gradually reduce the scope of AC propagation during backtrack search to exclude those arcs that may be subsequently handled as effectively by FC. Experimental results confirm the power of AFR and the validity of adaptive propagation in general.

1 Introduction

1.1 Background

The strategy of utilizing meta-level knowledge inference to reduce the computation required to achieve full Arc Consistency in a constraint network has been shown to be successful [15, 5, 2]. However, the algorithms suggested assume that constraint characteristics are known at the start of the search.

Depth-first backtracking tree search enhanced with Forward Checking (FC) or maintained Arc Consistency (AC) comprises the repeated application of two alternating steps: labelling (the assignment to a variable of a value from its domain) and constraint propagation (the deletion of inconsistent values from the domains of unassigned variables) [6, 8, 9]. Hence a single application of the two-step label-propagate process results in a transformation of the original problem to a sub-problem with one or more reduced domains. Any search decisions (variable assignments) may render initial meta-knowledge (about structure, tightness, density, etc.) obsolete, since sub-problems have different characteristics.

An instance demonstrating the effectiveness of using up-to-date meta-knowledge to improve the performance of the search algorithm is found in the popular first fail heuristic, often used in conjunction with FC [8]. Other dynamic meta-level heuristics have been found to achieve significant performance gains [13, 7]. All these have adapted algorithm behaviour by focussing search on promising sub-problems through variable and value ordering. As far as we know little or no research has been conducted into the utility of dynamically altering the extent of propagation deployed by the algorithm. The purpose of this paper is to demonstrate that meta-level knowledge about the problem may be updated to reflect the changes brought about by label-update steps and used locally to select appropriate constraint propagation methods and heuristics for remaining sub-problems. We focus in particular on a meta-knowledge propagation scheme Adaptive Arc Propagation (AAP) that has a choice of propagation methods at its disposal. The search algorithm initially decides a status for each binary constraint arc that determines how changes to the "source" variable's domain are propagated onto the "destination" variable. In the case of full AC and FC, this means that full propagation occurs on any change and on instantiation respectively. As the labelling and propagation take place, the algorithm updates meta-level knowledge and uses it to change the status of these constraint arcs.

A number of algorithms implementing a hybrid FC and AC search already exist. These hybridizations have been static in the sense that they apply FC and AC in specific phases or to specific sets of constraints, and do not update their configuration according to the dynamics of the search. Such hybridizations seem to be less effective than pure AC for hard problems [11].

One simple instance of the general framework, Anti-Functional Reduction (AFR), is described in detail here. The meta-level reasoning applied allows AAP to achieve precisely the same propagation as a pure AC algorithm while significantly improving its average performance. This is achieved by gradually reducing the scope of AC propagation during search to exclude those arcs that are handled as effectively by FC propagation. Experimental results confirm large performance gains for AFR, indicating the validity of adaptive propagation and the viability of more sophisticated AAP instances.

1.2 Paper Outline

The paper is structured in three descriptive sections followed by experimental results and a conclusion. Section 2 details the difference between AC and FC Propagation to suggest how the appropriate method may be selected for particular constraint arcs. Adaptive Arc Propagation, a scheme that allows the dynamic integration of AC and FC propagation, is described in Sect. 3. Section 4 then gives a simple instance of this scheme – Anti-Functional Reduction (AFR) – that has been tested experimentally. The results for AFR (Sect. 5) confirm large performance gains when applied to hard problems. Conclusions are drawn and opportunities for further research are outlined in Sect. 6.

2 Arc Consistency and Forward Checking

Arc Consistency is more powerful than Forward Checking in that it provides earlier detection of inconsistent values. We describe the exact conditions under which Arc Consistency is superior, restricting our analysis to binary constraint arcs.

2.1 Basic Notation

For a definition of the Constraint Satisfaction Problem (CSP) see [14]. If X and Y are CSP variables we define the following notation:

- $dom(X)$ represents the domain of X;
- $|X|$ is the size of $dom(X)$;
- $x_i, 1 \leq i \leq |X|$ are the elements in $dom(X)$;
- $r(X, x_i)$ is a reduction operation that removes x_i from the domain of X such that the new domain $dom'(X) = dom(X) \backslash \{x_i\}$;
- $r(X, R)$ represents the set of reduction operations given by $\{r(X, x_i) | x_i \in R\}$;[1]
- a binary constraint $constr_{XY}$ on X and Y is represented by the two directed arcs arc_{XY} and arc_{YX} corresponding to the two directions in which the constraint can propagate.

2.2 FC- and AC-monitoring

We compare below the behaviour of AC and FC on a directed arc arc_{XY} by counting the number of different possible reductions on X for which AC would yield a reduction on Y, but FC would not. We first introduce the terminology used:

- An arc_{XY} is said to propagate a reduction $r(X, R)$ of X when the domain of Y is reduced to exclude those values inconsistent with X's new domain and $constr_{XY}$ ($constr_{YX}$).
- An arc_{XY} is said to be *FC-monitored* in a propagation sequence if it is used only to propagate those reductions of X that are instantiations.
- An arc_{XY} is said to be *AC-monitored* if it propagates any reduction of X.

2.3 When Does AC Produce More Reductions than FC?

Constraints are relations and may be represented in terms of a relation matrix. These matrices are useful for analysing the effects of reductions.

[1] An instantiation of X – when its domain is reduced to one value through labelling or propagation – thus results from applying any set of reductions $r(X, I)$ such that $|I| = |X| - 1$.

Example 1. A constraint $constr_{XY}$ that has the tuple representation

{(a,a),(a,b),(a,c),(b,c),(c,b),(c,c)}

is equivalent to the relation matrix of Table 1.

In the table Boolean values indicate whether a given pair of values hold in the constraint relation. In addition the example matrix is annotated with the quantities of **ones** in a given row. These are usually referred to in the Arc Consistency literature as the *support* and denoted here by $S_X(y_i)$ [10]. In our comparison of Arc Consistency and Forward Checking the important quantity is the number of **zeros**; we call this quantity the *support-complement.* The support-complement $\overline{S}_X(y_i)$ may be obtained in terms of $S_X(y_i)$ and $|X|$:

$$\overline{S}_X(y_i) = |X| - S_X(y_i) \tag{1}$$

Consider the impact of reductions of X on Y resulting from the propagation of arc_{XY}. AC-monitoring is superior to FC-monitoring when a reduction of X that is not an instantiation results in a reduction of Y. Those values of Y that have two or more zeros in their row are the only candidates for deletion in such a situation. This is because values that have no zeros are always consistent with the constraint arc, while those with a single zero might be deleted only on the instantiation of X. The condition $\overline{S}_X(y_i) > 1$ thus determines whether AC-monitoring of arc_{XY} might produce a reduction of y_i where FC-monitoring would not. The final column of the matrix reflects this analysis by listing the minimum amount of propagation required to maintain the Arc Consistency of the domain of Y with respect to the arc-value pair (arc_{XY}, y_i); some arc-value pairs need more propagation than others to maintain an Arc Consistent Y domain.

Table 1. An example relation matrix

	X a	b	c	ones	zeros	Minimum Monitoring of arc_{XY} Required for Achieving AC
Y a	1	0	0	1	2	AC
b	1	0	1	2	1	FC
c	1	1	1	3	0	None

An *anti-functional arc* arc_{XY} has $\forall y_i : \overline{S}_X(y_i) \leq 1$. The class of constraints $constr_{XY}$ that have anti-functional arc_{XY} and arc_{YX} are known as *anti-functional constraints* [15]. The disequality constraint $\neq$ is an example of an anti-functional constraint.

2.4 AC-superiority

For values having $\overline{S}_X(y_i) > 1$, the number of possible reduction sets $r(X, R)$ that are not instantiations and result in the deletion of y_i in AC-monitoring but

not in FC-monitoring is given by:

$$n_1 = \sum_{r=0}^{\overline{S}_X(y_i)-2} \binom{\overline{S}_X(y_i)}{r} \tag{2}$$

while the total number of possible reductions that are not instantiations and are not the empty set is given by:

$$n_2 = \sum_{r=0}^{|X|-2} \binom{|X|}{r} - 1 \tag{3}$$

Assuming that all possible reduction sets $r(X, R)$ are equally probable in a propagation sequence we may obtain figures for the probability of AC-monitoring achieving a reduction where FC-monitoring would not; we call this probability *AC-superiority*. AC-superiority depends on the support-complement of y_i and $|X|$.

$$AC\text{-}superiority(y_i) = \frac{n_1}{n_2} \tag{4}$$

Figure 1 shows how AC-superiority varies for a variable X of domain size 10 according to the support-complement of y_i.

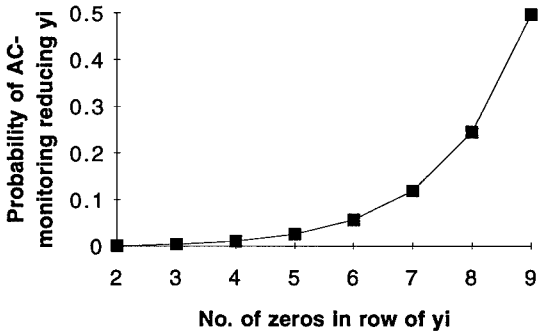


Fig. 1. AC-superiority vs. support-complement ($|X| = 10$)

The criterion used here to measure Arc Consistency superiority vis-à-vis Forward Checking was the number of propagated reductions. The analysis might be used to identify arcs with a low probability of achieving superior propagation with AC-monitoring. AC-superiority figures could be obtained at run-time for particular values in variable domains with the aid of a preprocessed table indexed by domain size and support-complement. A decision procedure could then use these figures to switch arcs propagating onto domains with low AC-superiority values from AC-monitoring to FC-monitoring.

Anti-Functional Reduction (AFR), the technique described in detail in Sect. 4 does not apply such a sophisticated decision procedure. No effort is made to monitor AC-superiority; only those arcs that are *guaranteed* to be handled as effectively by FC-monitoring are switched from AC-monitoring to FC-monitoring.

3 Adaptive Arc Propagation

Adaptive Arc Propagation (AAP) is an adaptive propagation scheme that makes use of arc tags to maintain recommended propagation methods for each arc in a constraint network. These propagation methods are selected by meta-level knowledge inference or heuristics. Anti-Functional Reduction is chosen to demonstrate AAP in Sect. 4, but other instances could also have been applied.[2]

In the following, all procedures and functions that need to be defined by AAP instances are subscripted by inst. These include the two meta-knowledge maintenance procedures, $\mathsf{InitList}_{\mathsf{inst}}$ and $\mathsf{ReviseList}_{\mathsf{inst}}$ that create and modify the set of arc tag lists $\{l(X,Y)|(X,Y) \in arcs(G)\}$.

$\mathsf{InitList}_{\mathsf{inst}}$ initializes each $l(X,Y)$; it might assign it a default list or take into account existing knowledge about the arc in question. During backtrack labelling and propagation, whenever an arc is selected to propagate a variable reduction the first *active* method in the list is applied – a method is active when its triggering conditions are satisfied. The arc's list is then revised using the second function $\mathsf{ReviseList}_{\mathsf{inst}}$ to take into account the changes resulting from the propagation step. Hence each arc propagation step involves also a list revision step. This gives meta-knowledge mechanisms a fine grained control over propagation.

In the interests of simplicity, the general AC-5 scheme has been adopted in AAP. An even more general scheme such as AC-Inference could have been used as a basis for AAP; it allows the possibility of the lazy evaluation of constraints to minimize constraint checks [2]. However, we do not explore this possibility here. This is justified from our point of view because our aim is to reduce the time spent on backtrack search and arc consistency maintenance while solving large-scale problems, and our focus has been on applying algorithms that are AC-4 instances of the AC-5 scheme [15]. While AC-4's over-eager constraint checking has drawn deserved criticism, when used for *maintaining* arc consistency (MAC-4) it performs all constraint checking in the preprocessing phase avoiding the need for explicit constraint checks during the label-update steps of backtracking search [16, 1, 10, 11]. This is clearly advantageous when trying to minimize the backtrack search time.

3.1 AAP Initialization

In addition to initialising the network and the propagation method data structures the initialization scheme of AAP applies the list creation function $\mathsf{InitList}_{\mathsf{inst}}$

[2] Sect. 6.2 gives one other example.

for each arc (see Fig. 2). The functions $\mathsf{InitMethod}_{\mathsf{inst}}$, Remove and Enqueue are similar their equivalents in the AC-5 scheme.[3] $\mathsf{InitMethod}_{\mathsf{inst}}$ calls the appropriate initialization routine of the algorithm instance. Remove deletes values found to be inconsistent by $\mathsf{InitMethod}_{\mathsf{inst}}$ from the relevant domain and Enqueue adds the new reductions to the queue of processable reductions. $\mathsf{InitList}_{\mathsf{inst}}$ uses the domains of the variables after propagation and the arc attributes to decide the contents of the arc's initial method list.

```
begin AAP-Initialization
      Q = {};
      for each (X, Y) ∈ arc(G) do
         begin
               InitMethod_inst (X, Y, Δ_out);
               Remove(Δ_out, D_Y);
               Enqueue(Y, Δ_out, Q);
               I(X, Y) = InitList_inst (X, Y)
         end;
end AAP-Initialization
```

Fig. 2. AAP-Initialization

3.2 AAP Maintenance

AAP maintenance allows the revision of the lists created at initialization by $\mathsf{InitList}_{\mathsf{inst}}$ to reflect the search dynamics (Fig. 3). Dequeue obtains an arc and a domain reduction that requires propagation. One difference to the AC-5 scheme is that multiple reductions on a single arc are presented together by Dequeue.[4]

The propagation procedure used is given in Fig. 4. ApplyActiveMethod traverses the list of methods associated with the arc applying the first method in the list for which the Boolean function $\mathsf{Active}_{\mathsf{inst}}$ returns **true**. This enforces the ordering of the applicable methods implicit in the list sequence. The ordering is important since propagation methods have different computational overheads and various triggering conditions (see Sect. 4). The method chosen returns a possibly empty reduction Δ_{out} on the destination variable Y. The domain of Y is updated before the meta-knowledge function $\mathsf{ReviseList}_{\mathsf{inst}}$ is called to revise the list of recommended methods. Any reductions are added to the queue as usual by Enqueue.

[3] In the pseudo-code AC-5's arc directions are reversed to be consistent with other notation.

[4] This is a source of efficiency since reductions on a single arc are handled best together, minimizing queue operations and allowing the quantity of reductions to be measured.

```
begin AAP-Maintenance
      while not EmptyQueue(Q) do
            begin
                  Dequeue(Q, X, Y, Δ_in);
                  ApplyActiveMethod(l(X, Y), X, Y, Δ_in, Δ_out);
                  Remove(Δ_out, D_Y);
                  l(X, Y) = ReviseList_inst(X, Y);
                  Enqueue(Y, Δ_out, Q)
            end
end AAP-Maintenance
```

Fig. 3. AAP-Maintenance

```
begin ApplyActiveMethod(l, X, Y, Δ_in, Δ_out))
      while not-empty(l) do
            begin
                  method = head(l);
                  if Active_inst (method, X, Y)
                    then apply method(X, Y, Δ_in, Δ_out);
                  l = tail(l)
            end
end ApplyActiveMethod
```

Fig. 4. AAP propagation procedure

4 Anti-Functional Reduction

Anti-Functional Reduction (AFR) is a simple instance of AAP that aims to reduce the number of AC reduction operations during backtracking search. The importance of reducing propagation operations (e.g. decrementing a support counter in AC-4) arises from the mechanisms applied by the backtracking search process; the savings are not primarily due to the time saved performing a basic instruction such as subtraction. Every change that occurs to a data structure (such as a support counter) in a propagation sequence must be recorded to enable restoration of state on backtracking. The act of recording and restoring state consumes time and space, and an unnecessary change to a data structure incurs an overhead and should be avoided. In the worst propagation sequence AC-4 performs ea^2 support decrement propagation operations, where e is the number of constraints and a is the maximum domain size.

For a single constraint arc, FC has far fewer recordable (restorable) operations than AC. There is no need to record multiple support counter decrements,

only a single domain reduction. Section 2 examined AC and FC propagation and described how it is possible to estimate the probability of achieving hits (reduction of the destination variable) using AC-monitoring but not in FC-monitoring. However, as mentioned earlier the instance of AAP described here takes the approach that any possibility of a domain reduction is worth pursuing through the application of AC-monitoring. AFR identifies only anti-functional arcs that guarantee no loss in propagation when their monitoring status is switched from AC-monitoring to FC-monitoring, thus minimizing the amount of backtrack restoration. Key to the efficient detection of anti-functional arcs in AFR is the commonality between the data structures of AC-4, the AC propagation method deployed in AFR, and those required to detect anti-functionality; the support-complement is obtained in terms of the support and the domain size of the source variable by (1).

As well as switching detected anti-functional arcs to FC-monitoring, AFR activates FC instead of AC for reduction sets that are instantiations. These two enhancements reduce the number of restorable propagation operations in the average case, albeit with a small meta-knowledge maintenance overhead. The experimental results of Sect. 5 show that using FC to support AC in this way is not only viable but entirely justified, especially for hard problems.

4.1 AFR Initialization

As mentioned above, it is preferable to give priority to FC over AC when propagating reductions that are instantiations at all times. A sensible initial ordering of applicable methods is thus {fc,ac} because it forces the use of Forward Checking in preference to Arc Consistency whenever its triggering conditions are met ($\mathsf{Active}_{\mathsf{afr}}(\mathsf{fc}, X, Y)$ returns true on instantiation). Hence the function $\mathsf{InitList}_{\mathsf{afr}}$ defined in Fig. 5 assigns {fc,ac} as the initial recommended list for all arcs that are not anti-functional.

```
begin InitList_afr(X,Y)
      |X|' = |X| − 1;
      D_Y^AC = {y_i | S_X(y_i) < |X|'}
      if D_Y^AC = φ
        return {fc}
      else
        return {fc,ac}
end
```

Fig. 5. AFR method list initialization

4.2 AFR Maintenance and Propagation

The task of the revision procedure of AFR is to identify when arc tags may be switched from mixed FC- AC-monitoring ({fc,ac}) to FC-monitoring ({fc}). $\mathsf{ReviseList}_{\mathsf{afr}}$ revises the method list according to the contents of a shadow domain D_Y^{AC} (Fig. 6). D_Y^{AC} contains all those values that are not handled as effectively by FC-monitoring of X. Note that D_Y^{AC} is monotonically reduced; once a value has been removed from D_Y^{AC} there is never a cause to reintroduce it to D_Y^{AC} in remaining sub-problems. When D_Y^{AC} is empty the switch can occur. D_Y^{AC} is in fact a data structure that must be restored on backtracking, however it should be noted that any value that is not a member of D_Y^{AC} need never have its support counter $S_X(y_i)$ decremented by the AC-4 propagation procedure. The AC-4 procedure is focussed only on reducing the support counters of members of D_Y^{AC}, resulting in a net reduction in the number of restorable propagation operations. Completeness of propagation is restored because the FC procedure always activates on instantiation, determining the status of those values not in D_Y^{AC}.

---

begin $\mathsf{ReviseList}_{\mathsf{afr}}(X, Y)$
  $|X|' = |X| - 1$;
  $D_Y^{AC} = \{y_i : y_i \in D_Y^{AC} \wedge y_i \in D_Y \wedge S_X(y_i) < |X|'\}$;
  if $D_Y^{AC} = \phi$
    return {fc}
  else return {fc,ac}
end

Fig. 6. AFR method list maintenance

5 Empirical Results

5.1 Experimental Setup

Fifty thousand experiments were used to compare FC, AC, and AFR. A fixed variable ordering was employed to remove extraneous influences on algorithm performance. The experiments were conducted in groups of trials. The following parameters were varied over groups: no. of variables (10, 20), domain size (5, 10), and P_{density} (0.2, 0.4, 0.6, 0.8, 1.0). In addition, structure was introduced into the problem constraints with two tightness parameters, $P_{\text{applicability}}$ (0.1,0.3,0.5,0.7,0.9) and $P_{\text{constriction}}$ (0.1, 0.3, 0.5, 0.7, 0.9). The three probabilities are described below:

- P_{density}
  Problem variable pairs (X, Y) have a constraint defined on them with probability P_{density}.
- $P_{\text{applicability}}$ This parameter is used to control the proportion of values in variable domains to which constraints apply. For each constraint $constr_{XY}$, two *applicability sets* (App_X and App_Y) are created. These contain the values in the domains of X and Y that the constraint applies to. A value is *not* a member of its variable's applicability set with probability $P_{\text{applicability}}$.
  Low values for $P_{\text{applicability}}$ imply tight constraints.
- $P_{\text{constriction}}$
  $P_{\text{constriction}}$ controls tightness of a constraint $constr_{XY}$ given that it applies only to those values in App_X and App_Y. Two *tuple sets*, Tup_X and Tup_Y, are created. The intersection of Tup_X and Tup_Y yields the tuples of the constraint. Each pair constructed by matching a value from App_X and any value from $dom(Y)$ is a member of Tup_X with probability $P_{\text{constriction}}$. In addition, pairs constructed by matching values not in App_X to any value from $dom(Y)$ are members of Tup_X. Tup_Y is constructed symmetrically.
  Hence low values for $P_{\text{constriction}}$ imply tight constraints.

$P_{\text{applicability}}$ and $P_{\text{constriction}}$ are more complex than the conventional tightness probability often applied in experimental studies. They were used instead because they enable the configuration of constraints to be close to – or far from – anti-functionality. For example, binary constraints on variables with a domain of size 10 would tend to be anti-functional when $P_{\text{applicability}} = 0$ and $P_{\text{constriction}} = 0.9$. Arguably these two parameters are a fairer representation of real world constraints since they are more structured and allow constraints to be relevant to only subsets of the values in their variables' domains, rather than the more uniform distribution over tuples created by the conventional tightness parameter.

Each possible combination of problem parameters was tested 100 times. All trials were conducted on a Sun Sparcstation 20 with 100 CPU seconds timeout per trial. Almost all timeouts were caused by FC solving relatively unconstrained problems (cf. exceptionally hard problems [12]). Timeouts were not penalized. The algorithms were implemented in the ECL^{i}PS^e Constraint Logic Programming environment [3].

5.2 Results

Figures 7–11 compare AC, FC and AFR in terms of average execution time in CPU seconds. AFR shows the smallest increase with increasing domain size and number of variables (Figs. 7, 8). Figure 9 demonstrates that AFR significantly improves AC performance across all densities, and is generally better than FC for all but the highest value. The graphs for $P_{\text{applicability}}$ and $P_{\text{constriction}}$ show again that AFR is superior to AC, and is better than or close to FC performance.

Finally, Fig. 12 shows that AFR achieves its best improvements over AC when solving hard problems with a large number of backtracks. Each point

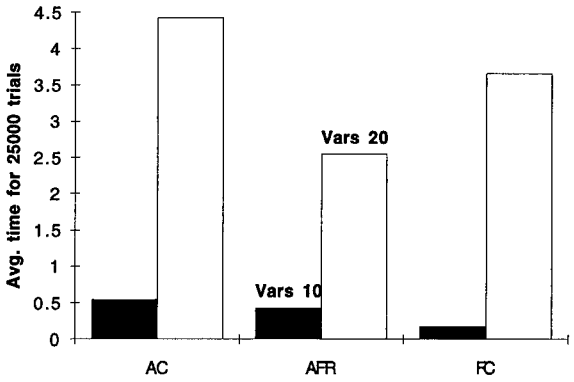


Fig. 7. Average time (25,000 trials) versus no. of variables

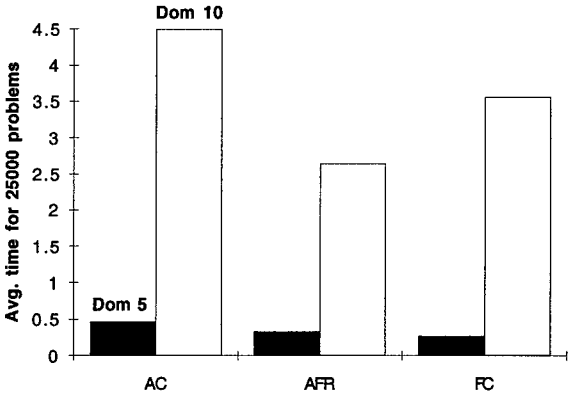


Fig. 8. Average time (25,000 trials) versus domain size

marked on the graph represents the average of 100 trials or more in the case of overlapping points. The AFR improvement was measured both in the average absolute time and in the average number of restorable operations; the figure demonstrates the close relationship between the two and shows that AFR reduces AC timings by up to 60% for hard problems.

6 Conclusion

6.1 Summary

Adaptive propagation is not only viable but an effective means for improving performance. Propagation may be adapted usefully during backtracking search according to dynamically changing search parameters and problem meta-knowledge. An adaptive propagation algorithm switches between a number of

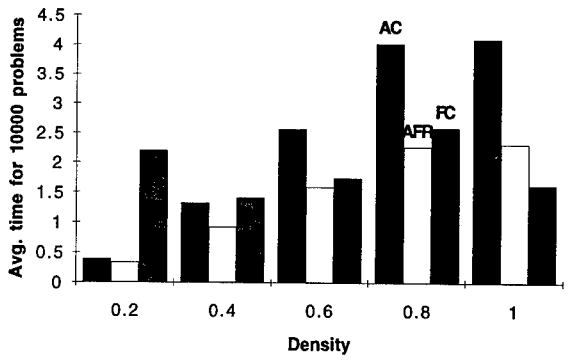


Fig. 9. Average time (10,000 trials) versus P_{density}

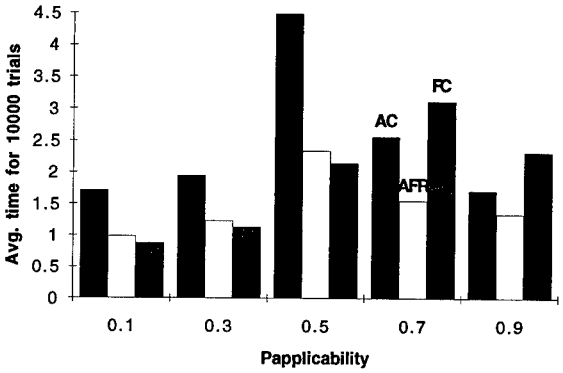


Fig. 10. Average time (10,000 trials) versus $P_{\text{applicability}}$

propagation methods depending on these monitored parameters. In this paper we focussed on two arc propagation methods, AC and FC. The adaptive propagation scheme AAP was defined to allow the flexible interchange between these two arc propagation methods. A particular instance of this scheme, AFR, reduced the scope of AC propagation to exclude arcs that were handled as effectively by FC. Experimental evidence confirmed that AFR performed better than a pure AC maintenance algorithm, and was especially effective for hard problems.

6.2 Future Work

Another instance of AAP is currently being refined to reduce redundant propagation on repair. Repair involves the reassignment of variables from old values to new ones in their domain. Assuming that old values are mutually consistent, variables that include their old values may remain unchanged; constraint arcs from such variables may remain inactive until an update in the form of a variable reassignment or domain reduction deletes the old value from their domain.

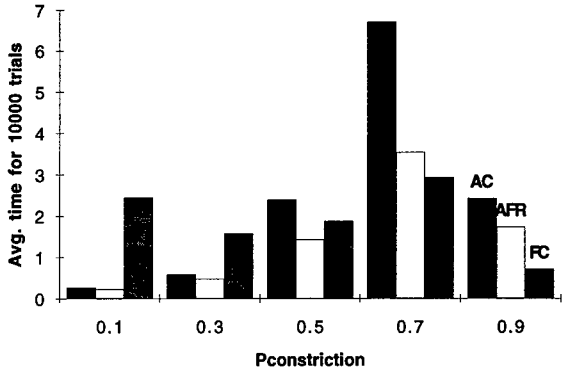


Fig. 11. Average time (10,000 trials) versus $P_{\text{constriction}}$

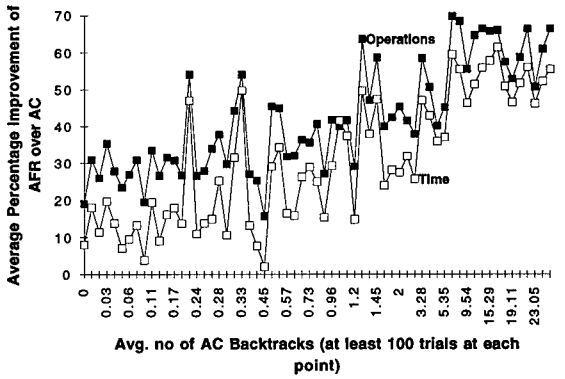


Fig. 12. Percentage improvement in time/operations versus average backtracks

Arcs from such variables then propagate onto their destination variables possibly causing other variables and arcs to "wake up". This is another example of propagation realigning to reflect the dynamics of search, in this case the search for consistent repairs to an existing solution [4].

Acknowledgements. Many thanks to all those at IC-Parc who have contributed to the writing of this paper through their helpful discussions and criticism. Carmen Gervet and Robert Rodošek deserve particular thanks for their useful feedback. This work has been supported by the Overseas Research Students Awards Scheme and British Airways collaboration projects.

References

1. C. Bessiere. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994.

2. C. Bessiére, E. Freuder, and J-C. Régin. Using inference to reduce arc consistency computation. In *IJCAI-95*, pages 592–598, Montréal, August 1995.
3. ECLiPSe version 3.4 user manual, July. Technical report, ECRC, 1995.
4. H. El Sakkout. Extending finite domain propagation for repair. Technical report, IC-Parc, 1995.
5. E.C. Freuder. Using metalevel knowledge to reduce constraint checking. In *Constraint Processing: Selected Papers from the ECAI'94 Workshop*. Springer, 1995.
6. S.W. Golomb and L.D. Baumert. Backtrack programming. *Journal of the ACM*, 12:516–524, 1965.
7. O. Hansson and A. Mayer. A decision-theoretic scheduler for space telescope applications. In *Intelligent Scheduling*. Morgan Kaufmann, 1994.
8. R.M. Haralick and G.L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–314, October 1980.
9. A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
10. R. Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.
11. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *ECAI-94. 11th European Conference on Artificial Intelligence*, pages 125–129, Amsterdam, August 1994.
12. B.M. Smith and S. A. Grant. Sparse constraint graphs and exceptionally hard problems. In *IJCAI-95*, pages 646–651, Montréal, August 1995.
13. Andrew B. Philips Steven Minton, Mark D. Johnston and Philip Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992. Minconfict heuristic, CSPs.
14. Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
15. P. Van Hentenryck, Y. Deville, and C. Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57:291–321, 1992.
16. R.J. Wallace. Why ac-3 is almost always better than ac-4 for establishing arc consistency in csps. In *IJCAI-95*, pages 592–598, Montréal, August 1995.

An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem*

Ian P. Gent[1], Ewan MacIntyre[1], Patrick Prosser[1], Barbara M. Smith[2] and Toby Walsh[3]

[1] Department of Computer Science, University of Strathclyde, Glasgow G1 1XH, Scotland. E-mail: {ipg,em,pat}@cs.strath.ac.uk
[2] Division of Artificial Intelligence, School of Computer Studies, University of Leeds, Leeds LS2 9JT, England. E-mail: bms@scs.leeds.ac.uk
[3] IRST, I38100 Trento & DIST, I16145 Genova, Italy. E-mail: toby@itc.it

Abstract. The constraint satisfaction community has developed a number of heuristics for variable ordering during backtracking search. For example, in conjunction with algorithms which check forwards, the Fail-First (FF) and Brelaz (Bz) heuristics are cheap to evaluate and are generally considered to be very effective. Recent work to understand phase transitions in NP-complete problem classes enables us to compare such heuristics over a large range of different kinds of problems. Furthermore, we are now able to start to understand the reasons for the success, and therefore also the failure, of heuristics, and to introduce new heuristics which achieve the successes and avoid the failures. In this paper, we present a comparison of the Bz and FF heuristics in forward checking algorithms applied to randomly-generated binary CSP's. We also introduce new and very general heuristics and present an extensive study of these. These new heuristics are usually as good as or better than Bz and FF, and we identify problem classes where our new heuristics can be orders of magnitude better. The result is a deeper understanding of what helps heuristics to succeed or fail on hard random problems in the context of forward checking, and the identification of promising new heuristics worthy of further investigation.

1 Introduction

In the constraint satisfaction problem (CSP) we are to assign values to variables such that a set of constraints is satisfied, or show that no satisfying assignment exists. This may be done via a systematic search process, such as depth first search with backtracking, and this amounts to a sequence of decisions, where a decision is a choice of variable and value to assign to that variable. The order in which decisions are made can have a profound effect on search effort. Dechter and Meiri's study of preprocessing techniques [3] shows that dynamic search rearrangement (DSR), i.e. a variable ordering heuristic that selects as next variable

* This research was supported by HCM personal fellowship to the last author, by a University of Strathclyde starter grant to the first author, and by an EPSRC ROPA award GR/K/65706 for the first three authors. Authors listed alphabetically. We thank the other members of the APES group, and our reviewers, for their comments.

the one that has minimal number of values in its domain, dominated all other static orderings. Here, we present three new dynamic variable ordering (dvo) heuristics, derived as a result of our studies of phase transition phenomena of combinatorial problems, and compare these against two existing heuristics.

Tsang, Borrett, and Kwan's study of CSP algorithms [22] shows that there does not appear to be a universally best algorithm, and that certain algorithms may be preferred under certain circumstances. We carry out a similar investigation with respect to dvo heuristics in an attempt to determine under what conditions one heuristic dominates another.

In the next section we give a background to the study. We then go on to describe four measures of the constrainedness of CSP's, and in Section 4 describe five heuristics, based on these measures. The empirical study is reported in Section 5, the heuristics are then discussed with respect to previous work in Section 6, and conclusions are drawn in Section 7.

2 Background

A constraint satisfaction problem consists of a set of n variables V, each variable $v \in V$ having a domain of values M_v of size m_v, and a set of constraints C. Each constraint $c \in C$ of arity a restricts a tuple of variables $\langle v_1, \ldots, v_a \rangle$, and specifies a subset of $M_1 \times M_2 \times \ldots \times M_a$, each element of which is a combination of values the variables are forbidden to take simultaneously by this constraint. In a binary CSP, which the experiments reported here are exclusively concerned with, the constraints are all of arity 2. A solution to a CSP is an assignment of a value to every variable satisfying all the constraints. The problem that we address here is the decision problem, i.e. finding one solution or showing that none exists.

There are two classes of complete search algorithm for the CSP, namely those that check backwards and those that check forwards. In algorithms that checks backwards, the current variable v_i is instantiated and checking takes place against the (past) instantiated variables. If this is inconsistent then a new value is tried, and if no values remain then a past variable is reinstantiated. In algorithms that check forwards, the current variable is instantiated with a value and the (future) uninstantiated variables are made consistent, to some degree, with respect to that instantiation. Chronological backtracking (BT), backmarking (BM), backjumping (BJ), conflict-directed backjumping (CBJ), and dynamic backtracking (DB) are algorithms that check backwards [11, 5, 6, 10], whereas forward checking (FC) and maintaining arc-consistency (MAC) are algorithms that check forwards [13, 18]. This study investigates only forward checking algorithms, and in particular forward checking combined with conflict-directed backjumping (FC-CBJ) [15].

Algorithm FC instantiates variable v_i with a value x_i and removes from the domains of future variables any values that are inconsistent with respect to that instantiation. If the instantiation results in no values remaining in the domain of a future variable, then a new value is tried for v_i and if no values remain for v_i (i.e. a dead end is reached) then the previous variable is reinstantiated (i.e. chronological backtracking takes place). FC-CBJ differs from FC; on reaching a

dead end the algorithm jumps back to a variable that is involved in a conflict with the current variable [15].

In selecting an algorithm we will prefer one that takes less search effort than another, where search effort is measured as the number of times pairs of values are compared for compatibility, i.e. consistency checks. Generally, checking forwards reduces search effort, as does jumping back.

The order in which variables are chosen for instantiation profoundly influences search effort. Algorithms that check backwards tend to use variable ordering heuristics that exploit topological parameters, such as width, induced width or bandwidth, and correspond to static instantiation orders (i.e. they do not change during search) [21]. Algorithms that check forwards have additional information at their disposal, such as the current size of the domains of variables. Furthermore, since domain sizes may vary during the search process, forward checking algorithms may use dynamic variable ordering (dvo) heuristics [17], and it is this class of heuristics that is investigated here.

3 Constrainedness

Many NP-complete problems display a transition in solubility as we increase the constrainedness of problem instances. This phase transition is associated with problems which are typically hard to solve [2]. Under-constrained problems tend to have many solutions and it is usually easy to guess one. Over-constrained problems tend not to have solutions, and it usually easy to rule out all possible solutions. A phase transition occurs in between when problems are "critically constrained". Such problems are usually difficult to solve as they are neither obviously soluble or insoluble. Problems from the phase transition are often used to benchmark CSP and satisfiability procedures [22, 9]. Constrainedness can be used both to predict the position of a phase transition in solubility [23, 20, 16, 7, 19] and, as we show later, to motivate the construction of heuristics.

In this section, we identify four measures of some aspect of constrainedness. These measures all apply to an *ensemble* of random problems. Such measures may suggest whether an individual problem from the ensemble is likely to be soluble. For example, a problem with larger domain sizes or looser constraints is more likely to be soluble than a problem with smaller domains or tighter constraints, all else being equal. To make computing such measures tractable, we will ignore specific features of problems (like the topology of the constraint graph) and consider just simple properties like domain sizes and constraint tightness.

One simple measure of constrainedness can be derived from the *size* of problems in the ensemble. Size is determined by both the number of variables and their domain sizes. Following [7, 8], we measure problem size via the size of the state space being explored. This consists of all possible assignments of values to variables, its size is simply the product of the domain sizes, $\prod_{v \in V} m_v$. We define the size ($\mathcal{N}$) of the problem as the number of bits needed the number of bits needed to describe a point in the state space, so we have:

$$\mathcal{N} =_{\text{def}} \sum_{v \in V} \log_2 m_v \qquad (1)$$

A large problem is likely to be less constrained and has a greater chance of being soluble than a small problem with the same number of variables and constraints of the same tightnesses.

A second measure of constrainedness is the *solution density* of the ensemble. If the constraint c on average rules out a fraction p_c of possible assignments, then a fraction $1 - p_c$ of assignments are allowed. The average solution density, ρ is the mean fraction of assignments allowed by all the constraints. The mean solution density over the ensemble is,

$$\rho = \prod_{c \in C} (1 - p_c) \tag{2}$$

Problems with loose constraints have high solution density. As noted above, all else being equal, a problem with a high solution density is more likely to be soluble than a problem with a low solution density.

A third measure of constrainedness is derived from the size and solution density. $E(N)$, the expected number of solutions for a problem within an ensemble is simply the size of the state space times the probability that a given element in the state space is a solution. That is,

$$E(N) = \rho 2^{\mathcal{N}} = \prod_{v \in V} m_v \times \prod_{c \in C} (1 - p_c) \tag{3}$$

If problems in an ensemble are expected to have a large number of solutions, then an individual problem within the ensemble is likely to be loosely constrained and to have many solutions.

The fourth and final measure of constrainedness, κ is again derived from the size and solution density. This has been suggested as a general measure of the "constrainedness" of combinatorial problems [8]. It is motivated by the randomness with which we can set a bit in a solution to a combinatorial problem. If κ is small, then problems typically have many solutions and a given bit can be set more or less at random. For large κ, problems typically have few or no solutions and a given bit is very constrained in how it can be set. κ is defined by,

$$\kappa =_{\text{def}} 1 - \frac{\log_2(E(N))}{\mathcal{N}} \tag{4}$$

$$= -\frac{\log_2(\rho)}{\mathcal{N}}$$

$$= \frac{-\sum_{c \in C} \log(1 - p_c)}{\sum_{v \in V} \log(m_v)} \tag{5}$$

If $\kappa \ll 1$ then problems have a large expected number of solutions for their size. They are therefore likely to be under-constrained and soluble. If $\kappa \gg 1$ then problems have a small expected number of solutions for their size. They are therefore likely to be over-constrained and insoluble. A phase transition in solubility occurs inbetween where $\kappa \approx 1$ [8]. This is equivalent for CSPs to the prediction made in [19] that a phase transition occurs when $E(N) \approx 1$.

4 Heuristics for Constrainedness

Many heuristics in CSPs branch on what can often be seen as an estimate of the most constrained variable [8]. Here, we describe two well known heuristics for CSPs and three new heuristics. We use the four measures of constrainedness described above. These measures were defined for an ensemble of problems. Each measure can be computed for an individual problem, but will give only an estimate for the constrainedness of an individual problem. For example, an insoluble problem has zero solution density and this may be very different from the measured value of ρ. Even so, such measures can provide both a good indication of the probability of a solution existing and, as we show here, a heuristic estimate of the most constrained variable.

Below, we adopt the following conventions. When a variable v_i is selected as the current variable and instantiated with a value, v_i is removed from the set of variables V, constraint propagation takes place, and all constraints incident on v_i, namely C_i, are removed from the set of constraints C. Therefore V is the set of future variables, C is the set of future constraints, m_j is the actual size of the domain of $v_j \in V$ after constraint propagation, p_c is the actual value of constraint tightness for constraint $c \in C$ after constraint propagation, and C_j is the set of future constraints incident on v_j. All characteristics of the future subproblem are recomputed and made available to the heuristics as local information.

4.1 Heuristic FF

Haralick and Elliott [13] proposed the fail-first principle for CSPs as follows: *"To succeed, try first where you are most likely to fail."* The reason for attempting next the task which is most likely to fail is to encounter dead-ends early on and prune the search space. Applied as a constraint ordering heuristic this suggests that we check first the constraints that are most likely to fail and when applied as a variable ordering heuristic, that we choose the most constrained variable. An estimate for the most constrained variable is the variable with the smallest domain. That is we choose $v_i \in V$ such that m_i is a minimum.

An alternative interpretation of this heuristic is to branch on v_i such that we maximize the size of the resulting subproblem, without considering the constraint information on that variable. That is, choose the variable $v_i \in V$ that maximizes

$$\sum_{v \in V - v_i} \log(m_v) \tag{6}$$

where $V - v_i$ is the set of future variables with v_i removed, and is the same as selecting the variable v_i which maximizes the denominator of equation (5).

4.2 Heuristic Bz

The Brelaz heuristic (Bz) comes from graph colouring [1]; we wish to find a colouring of the vertices of a graph such that adjacent vertices have different colours. Given a partial colouring of a graph, the *saturation* of a vertex is the number of differently coloured vertices adjacent to it. A vertex with high saturation will have few colours available to it. The Bz heuristic first colours a vertex of

maximum degree. Thereafter Bz selects an uncoloured vertex of maximum saturation, tie-breaking on the degree in the uncoloured subgraph. Bz thus chooses to colour next what is estimated to be the most constrained vertices.

When applying Bz to a CSP we choose the variable with smallest domain size and tie-break on degree in the future subproblem. That is, choose the variable with smallest m_i and tie-break on the variable with greatest future degree $|C_i|$. In a fully connected constraint graph, Bz will behave like FF, because all variables have the same degree.

4.3 Heuristic Rho

The Rho (ρ) heuristic branches into the subproblem that maximizes the solution density, ρ. The intuition is to branch into the subproblem where the greatest fraction of states are expected to be solutions. To maximize ρ, we select the variable $v_i \in V$ that maximizes

$$\prod_{c \in C - C_i} (1 - p_c) \tag{7}$$

where $C - C_i$ is the set of future constraints that do not involve variable v_i, and $(1 - p_c)$ is the *looseness* of a constraint. If we express (7) as a sum of logarithms, $\sum_{c \in C - C_i} \log(1 - p_c)$, then this corresponds to selecting a variable that minimizes the numerator of (5). Expression (7) gives an estimate of the solution density of the subproblem after selecting v_i. More concisely (and more computationally efficient), we choose the future variable v_i that minimizes

$$\prod_{c \in C_i} (1 - p_c) \tag{8}$$

This is the variable with the most and/or tightest constraints. Again, we branch on an estimate of the most constrained variable.

4.4 Heuristic E(N)

The E(N) heuristic branches into the subproblem that maximizes the expected number of solutions, $E(N)$. This will tend to maximize both the subproblem size (the FF heuristic) and its solution density (the Rho heuristic). Therefore, we select a variable $v_i \in V$ that maximizes

$$\prod_{v \in V - v_i} m_v \times \prod_{c \in C - C_i} (1 - p_c) \tag{9}$$

where $V - v_i$ is the set of future variables with v_i removed, and $C - C_i$ is the set of future constraints that do not involve variable v_i. This can be more succinctly (and efficiently) expressed as choose the variable $v_i \in V$ that minimizes

$$m_i \prod_{c \in C_i} (1 - p_c) \tag{10}$$

The E(N) heuristic has an alternative, intuitively appealing, justification. Let N be the number of solutions to the current subproblem. At the root of the tree, N is the total number of solutions to the problem. If N=0, the current subproblems has no solutions, and the algorithm will at some point backtrack.

If N=1, the current subproblem has exactly one solution, and N will remain constant on the path leading to this solution, but be zero everywhere else. As we move down the search tree, N cannot increase as we instantiate variables. The obvious heuristic is to maximize N in the future subproblem. We use E(N) as an estimate for N, so we branch into the subproblem that maximizes E(N). And this is again an estimate for the most constrained variable, as loosely constrained variables will tend to reduce N most. Consider a loosely constrained variable v_i that can take any value in its domain. Branching on this variable will reduce N to N/m_i. Tightly constrained variables will not reduce N as much.

4.5 Heuristic Kappa

The Kappa heuristic branches into the subproblem that minimizes κ. Therefore, select a variable $v_i \in V$ that minimizes

$$\frac{-\sum_{c \in C - C_i} \log(1 - p_c)}{\sum_{v \in V - v_i} \log(m_v)} \tag{11}$$

Let α be the numerator and β be the denominator of equation (5), the definition of κ. That is, $\alpha = -\sum_{c \in C} \log(1 - p_c)$ and $\beta = \sum_{v \in V} \log(m_v)$. Then we select a variable $v_i \in V$ such that we maximize the following

$$\frac{\alpha + \sum_{c \in C_i} \log(1 - p_c)}{\beta - \log(m_i)} \tag{12}$$

This heuristic was first suggested in [8] but has not yet been tested extensively on a range of CSPs, and depends on the proposal in [8] that κ captures a notion of the constrainedness of an ensemble of problems. We assume that κ provides an estimate for the constrainedness of an individual in that ensemble. We again want to branch on a variable that is estimated to be the most constrained, giving the least constrained subproblem. We estimate this by the subproblem with smallest κ. This suggests the heuristic of minimizing κ.

4.6 Implementing the heuristics

We use all the above heuristics with the forward checking algorithm FC-CBJ. After the current variable has successfully been assigned a value (i.e. after domain filtering all future variables have non-empty domains), the constraint tightness is recomputed for any constraint acting between a pair of variables, v_j and v_k, such that values have just been removed from the domain of v_j or v_k, or both. To compute constraint tightness p_c for constraint c acting between variables v_j and v_k we count the number of conflicting pairs across that constraint and divide by the product of the new domain sizes. This counting may be done via consistency checking and will take $m_j \times m_k$ checks. Constraint tightness will then be in the range 0 (all pairs compatible) to 1 (all pairs are conflicts). When computing the sum of the log looseness of constraints (i.e. the numerator of equation (5)), if $p_c = 1$ a value of $-\infty$ is returned. Consequently, the Kappa heuristic will select variable v_j or v_k next, and the instantiation will result in a dead end.

In the FF heuristic the first variable selected is the variable with smallest domain size, and when all variables have the same domain size we select first the lowest indexed variable v_1. For the Bz heuristic *saturation* is measured as the inverse of the domain size; i.e. the variable with smallest domain size will have largest saturation. Consequently, when the constraint graph is a clique FF and Bz will have identical behaviours.

Search costs reported in this paper do not include the cost in terms of consistency checks of recomputing the constraint tightness. This overhead makes some of the heuristics less competitive than our results might suggest. However, our main concern here is to establish sound and general principles for selecting variable ordering heuristics. In the future, we hope to develop book-keeping techniques and approximations to the heuristics that reduce the cost of re-computing or estimating the constraint tightness but which still give good performance.

5 The Experiments

The experiments attempt to identify under what conditions one heuristic is better than another. Initially, experiments are performed over *uniform* randomly generated CSP. In a problem $\langle n, m, p_1, p_2 \rangle$ there will be n variables, with a uniform domain of size m, $\frac{p_1 . n . (n-1)}{2}$ constraints, and exactly $p_2 m^2$ conflicts over each constraint [16, 19]. This class of problem is then modified such that we investigate problems with non-uniform domains and constraint tightness.

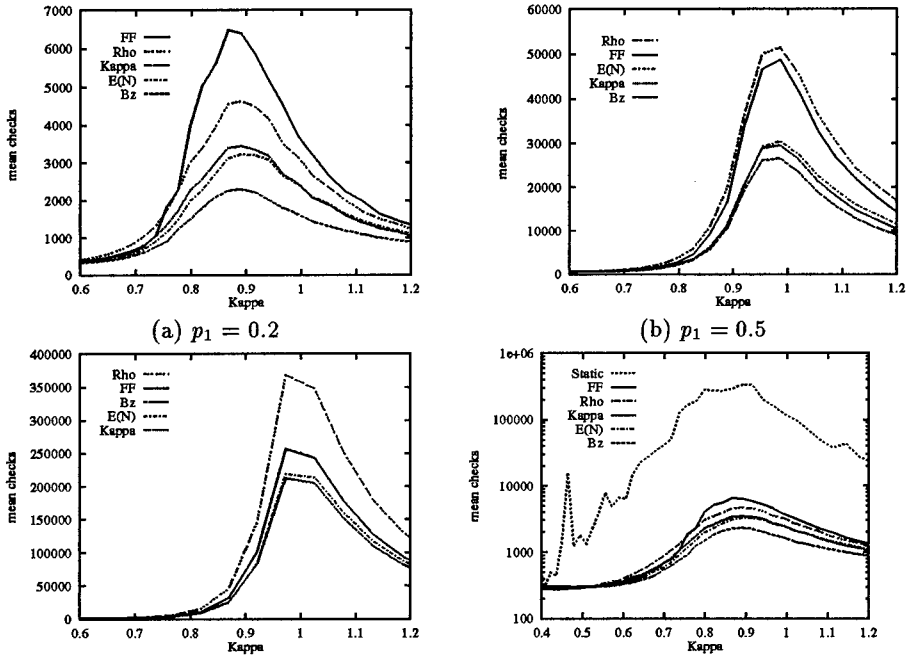
When plotting the results, problems will be measured in terms of their constrainedness, κ. This is because in some experiments we vary the number of variables and keep the degree of variables γ constant, vary the tightness of constraints p_2, and so on. By using constrainedness we hope to get a clear picture of what happens. Furthermore, in non-uniform problems constrainedness appears to be one of the few measures that we can use. It should be noted that in the experiments the complexity peak does not always occur exactly at $\kappa = 1$, and that in sparse constraints graphs the peak tends to occur at lower values of κ, typically in the range 0.6 to 0.9. This has been observed empirically in [16], and an explanation is given by Smith and Dyer [19].

In all of the graphs we have kept the same line style for each of the heuristics. The labels in the graphs have then been ordered, from top to bottom, to correspond to the ranking of the heuristics in the phase transition. The best heuristic will thus appear first.

5.1 Uniform Problems, Varying Constraint Graph Density p_1

The aim of this experiment is to determine how the heuristics are affected as we vary the number of constraints within the constraint graph. The experiments were performed over problems with 20 variables, each with a domain size of 10. In Figure 1, we plot the mean performance for sparse constraint graphs[4] with $p_1 = 0.2$, maximally dense constraint graphs with $p_1 = 1.0$ and constraint graphs of intermediate density $p_1 = 0.5$. At each density 1,000 problems were generated at each possible value of p_2 from 0.01 to 0.99 in steps of 0.01.

[4] Disconnected graphs were not filtered out since they had little effect on performance.



(a) $p_1 = 0.2$ (b) $p_1 = 0.5$

(c) $p_1 = 1.0$ (d) $p_1 = 0.2$, also with static order

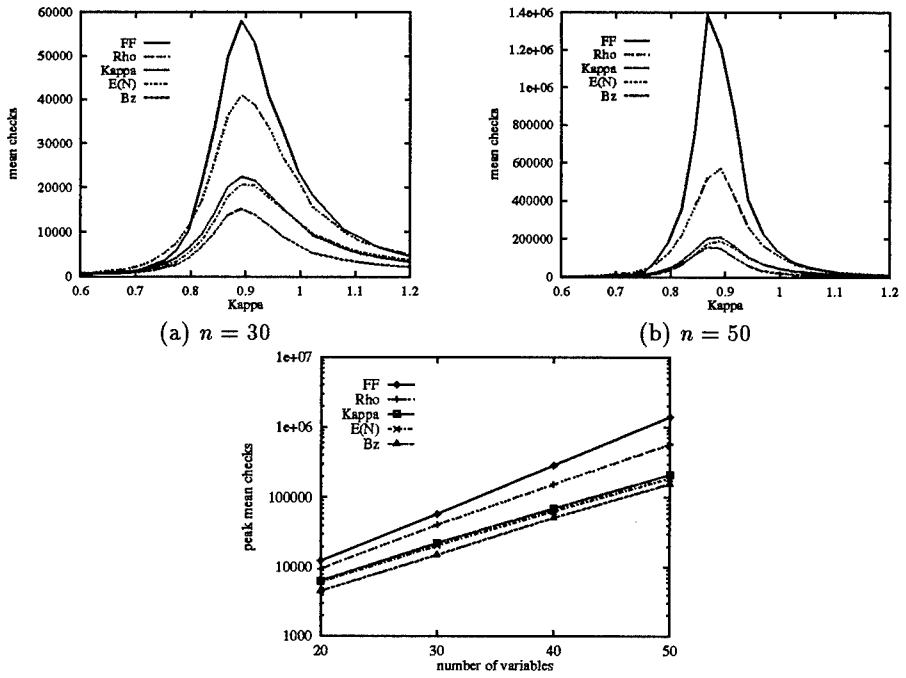
Fig. 1. Mean performance of heuristics for $\langle 20, 10 \rangle$

For sparse constraint graphs (see Figure 1(a)), Bz performs best, whilst E(N) and Kappa are not far behind. Rho is significantly worse and FF even more so. Analysing the distribution in performance (graphs are not shown) e.g. the median, 95% and higher percentiles, we observed a similar ranking of the heuristics with the differences between the heuristics opening up in the higher percentiles in the middle of the phase transition. As problems become more dense at $p_1 = 0.5$ (see Figure 1(b)) Kappa dominates E(N). Rho and FF continue to perform poorly, although FF does manage to overtake Rho.

For complete graphs with $p_1 = 1.0$ (see Figure 1(c)), Bz and FF are identical, as expected. (The contour for FF overwrites the Bz contour.) For uniform and sparse problems, Bz seemed to be best, whilst for uniform and dense problems, Kappa or E(N) would seem to be best.

For comparison with the dynamic variable ordering heuristics, in Figure 1(d) we also plot the mean performance of FC-CBJ with a static variable ordering: variables were considered in lexicographic order. Performance is much worse with a static ordering than with any of the dynamic ordering heuristics, even on the relatively easy sparse constraint graphs. The secondary peaks for the static variable ordering at low κ occur as a result of ehps [20], occasional "exceptionally hard" problems that arise following poor branching decisions early in search [9]. The worst case outside the phase transition was more than 14 million checks at $\kappa = 0.46$, in a region where 100% of problems were soluble. This was 5 orders of magnitude worse than the median of 288 checks at this point.

5.2 Uniform Problems, Varying Number of Variables n



(a) $n = 30$ (b) $n = 50$

(c) Peaks of mean performance over n

Fig. 2. Mean performance for FC-CBJ + heuristics for $\langle n, 10\rangle$ with $\overline{\gamma} = 5$

The aim of this experiment is to determine how the heuristics scale with problem size. At first sight, this can be simply done by increasing the number of variables n, while keeping all else constant. However, if n increases while p_1 is kept constant the degree γ of a variable (i.e. the number of constraints incident on a variable) also increases. To avoid this, we vary p_1 with n such that average degree $\overline{\gamma}$ remains constant at 5, similar to [12]. To observe a phase transition, 1,000 problems were then generated at each possible value of p_2 from 0.01 to 0.99 in steps of 0.01.

In Figure 2, we plot the performance of each heuristic as we increase n. In Figures 2(a) and (b), we show the mean performance for $n = 30$ and $n = 50$ respectively. The ranking of the heuristics remains the same as in the previous experiment for constraint graphs of intermediate density. Though not shown, we observed similar behaviour in the distribution of performance (*e.g.* median, 95% and higher percentiles). As before, the differences between the heuristics tend to open up in the higher percentiles in the middle of the phase transition.

In Figure 2(c) we plot the peak in average search effort in the phase transition region for each value of n. This then gives a contour showing how search cost increases with n, for this class of problem. The Figure suggests that Bz, Kappa and E(N) scale in a similar manner. Using a least square linear fit on the limited data available, we conjecture that E(N) would become better than Bz when $n > 90$, and Kappa would do likewise when $n > 164$. Further empirical studies on larger problems would be needed to confirm this. However, Rho and FF appear to scale less well. The gradients of Figure 2(c) suggests that FF and Rho scale with larger exponents than Bz, Kappa and E(N).

5.3 Problems with Non-Uniform Constraint Tightness

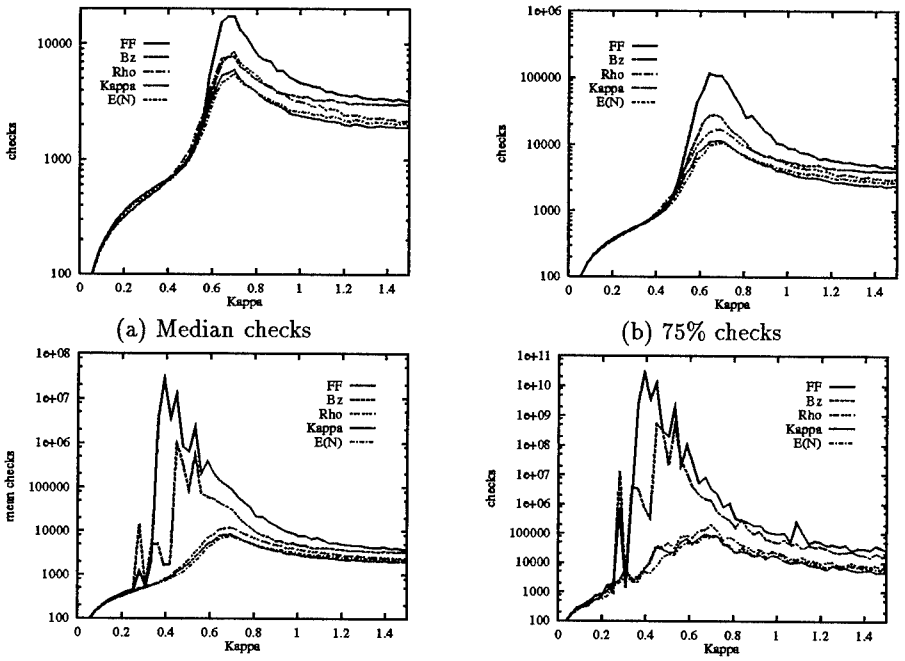
All experiments considered above have constraints generated uniformly. That is, a single value of p_2 describes the tightness of *every* constraint. At the start of search, every constraint is equally tight, so a good measure of the constrainedness of a variable is simply the number of constraints involving this variable (i.e. the variable's degree), together with its domain size. Even as we progress through search and tightnesses vary, this measure should still be reasonably accurate. This might explain why Bz has never been significantly worse in earlier experiments than Kappa or $E(N)$ which undertake the computationally heavy overhead of measuring exact constraint tightnesses.

If we are given a problem with significantly varying constraint tightnesses we must take account of this to measure constrainedness accurately. We therefore expect that Bz and FF may perform poorly on problems with varying constraint tightnesses, while the other heuristics should perform well, because they do take account of constraint tightness. To test this hypothesis, we generated problems with mainly loose constraints, but a small number of very tight constraints. We did this by generating problems with a multiple of 5 constraints, and choosing exactly 20% of these constraints to have tightness $p_2 = 0.8$ (i.e. tight constraints) and the remainder tightness $p_2 = 0.2$ (i.e. loose constraints). We expect Bz to perform poorly on these problems as it will tie-break on the number of constraints and not the tightness of those constraints (the more significant factor in this problem class).

We set $n = 30$ and $m = 10$, and to observe a phase transition we varied the constraint graph density, p_1 from $\frac{1}{87}$ to 1 in steps of $\frac{1}{87}$. Results are plotted in Figure 3. The 50% solubility point is at $\kappa \approx 0.64$ when $p_1 = \frac{23}{87}$.

Median performance, Figure 3(a), shows that as predicted Kappa and E(N) do well. Most significantly, Bz is dominated by all except FF. This is the first of our experiments so far where Bz has been shown to perform relatively poorly.

Figure 3(b) shows the 75th percentiles for the five heuristics (i.e. 75% of problems took less than the plotted amount of search effort) and Figure 3(d) shows worst case. We see that at the 75th percentile there is a greater difference between the heuristics, suggesting a more erratic behaviour from FF and Bz. Mean performance (Figure 3(c)) and worst case performance (Figure 3(d)) shows the existence of exceptionally hard problems for FF and Bz. The worst case for FF was 26,545 million consistency checks at $\kappa \approx 0.39$, in a region where 100% of



(a) Median checks (b) 75% checks

(c) Mean checks (d) Maximum checks

Fig. 3. Performance of heuristics on $n = 30$ and $m = 10$, with $p_2 = 0.2$ for 80% of the constraints, and $p_2 = 0.8$ for the remainder. Note the different y-scales.

problems were soluble. This was 8 orders of magnitude worse than the median of 659 checks at this point, and took 87 hours on a DEC Alpha $200^{4/166}$.

5.4 Problems with Non-Uniform Domain Size

Unlike the other four heuristics, Rho completely ignores the domain sizes and its contribution to problem constrainedness. We therefore expect that the Rho heuristic will do poorly on problems with mixed domain sizes. To test this hypothesis, we generated 20 variable problems, giving each variable a domain of size 10 with probability 0.5 and a domain of size 20 otherwise. We denote this as $m = \{10, 20\}$. To observe a phase transition, we fixed the constraint density p_1 at 0.5 and varied p_2 from 0.01 to 0.99 in steps of 0.01, generating 1,000 problems at each point. We plot the results for mean checks for each of the heuristics in Figure 4. As predicted, the Rho heuristic performs worse than in the previous problem classes. This seems to reaffirm the worth of exploiting information on domain sizes.

6 Discussion

Theory-based heuristics for the binary CSP are presented by Nudel [14], based on the minimization of a complexity estimate, namely the number of compound

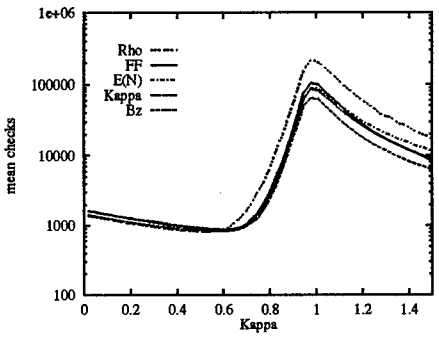


Fig. 4. Performance of FC-CBJ, with $n = 20$, $m = \{10, 20\}$ and $p_1 = 0.5$

labels at a given depth in the search tree. Two classes of heuristic are presented, global and local. Global heuristics fix the instantiation order at the start of search, whereas local heuristics take account of information made available during search, such as actual domain sizes and constraint tightness. Nudel's local heuristics are thus dynamic variable ordering (dvo) heuristics. Three dvo heuristics are presented, IO_2, IO_3, and IO_4. IO_2 chooses *"next below a node, that variable with minimum number m_i of surviving labels after forward checking at the node"*, and is equivalent to FF. Heuristic IO_3 tie-breaks IO_2 by choosing the variable (with smallest domain) that most constrains future variables, and has much in common with Bz. IO_4 stops when any future constraint disallows all tuples across that constraint. As Nudel says, this is not so much a heuristic but an algorithmic step. IO_4 is implicit in heuristics Rho, E(N), and Kappa.

It is interesting to contrast our approach with Nudel's as both give theory-based variable ordering heuristics. Nudel gives measures that estimate the size of the remaining search tree, and then constructs heuristics which seek to minimize these estimates. We have not related our measures directly to the search tree. Instead we have sought to move into areas of the search tree likely to be unconstrained and therefore have solutions. When one makes certain simplifications, both approaches can result in the same heuristic such as FF. However, the detailed relationship between the approaches has not yet been fully analysed.

Feldman and Golumbic [4] applied Nudel's heuristics to real-world constraint satisfaction problems. Three heuristics are presented, one for a backward checking algorithm (BT), and two for a forward checking algorithm (FC1 and FC2). All three heuristics were applied as global/static orderings. Heuristic FC1 selects v_i with minimum $m_i \prod_{i<j}(1-p_{i,j})$, where $p_{i,j}$ is tightness of the constraint acting between v_i and future variable v_j. This corresponds to a global E(N) ordering. Heuristic FC2 takes into consideration all constraints, and selects variable v_i with minimum $m_i \prod_{j\neq i, k\neq i}(1 - p_{j,k})$. As far as we can see, there is no correspondence between FC2 and the heuristics presented here. In their experiments heuristic FC1 dominated FC2 on hard problems.

The new dvo heuristics presented here may be used as global/static vari-

able ordering heuristics. When we have uniform constraint tightness, Rho will correspond to a reverse maximum cardinality ordering [3], suitable for forward checking algorithms. If all variables have the same constraint tightness then E(N) maximizes $\mathcal{N}$ (the FF heuristic), and if all variables have the same domain size E(N) simplifies to maximizing ρ (the Rho heuristic). Like the E(N) heuristic, the Kappa heuristic simplifies to maximizing $\mathcal{N}$ (the FF heuristic) if all variables have the same constraint tightness and to maximizing ρ (the Rho heuristic) if all variable have the same domain size. Clearly, FF and Bz can be considered as low cost surrogates of the minimize Kappa heuristic; both attempt to minimize (11) by maximizing the denominator, and Bz tie-breaks by estimating the numerator of (11) by assuming all constraints are of the same tightness.

7 Conclusions

Three new variable ordering heuristics for the CSP have been presented, namely E(N), Rho, and Kappa. These new heuristics are a product of our investigations into phase transition phenomena in combinatorial problems. The new heuristics have two properties in common. Firstly, they all attempt to measure the constrainedness of a subproblem, and secondly, they attempt to branch on the most constrained variable giving the least constrained subproblem. The heuristics differ in how they measure constrainedness, and what information they exploit.

The new heuristics have been tested alongside two existing heuristics, namely Fail-First (FF) and Brelaz (Bz), and on a variety of uniform and non-uniform problems, using a forward checking algorithm FC-CBJ. On uniform problems, the new heuristics perform similarly to each other and dominate FF. Bz was consistently better on sparse and moderately dense constraint graphs, and was easier to calculate. As constraint graph density increased to the point of becoming a clique, Bz performance degraded to be the same as FF. With respect to problem size, the new heuristics appear to scale better than FF and Bz.

Problems with non-uniform constraint tightnesses exposed poor behaviour from Bz. This was expected, because Bz exploits information from the domain sizes and topology of the constraint graph, but ignores the tightness of constraints. Experiments on problems with non-uniform domains demonstrated that ignoring information of domain sizes results in poor performance.

In some respects the work reported here might be considered as a first foray into a better understanding of what makes heuristics work. Further work could include determining the importance of tie-breaking in the heuristic Bz, compared to simply choosing the *first* variable sensibly. Faster substitutes for the heuristics would allow us to investigate the hypothesis that the new heuristics scale better than the old. Little has been done to compare the ranking of the new heuristics on an individual problem basis. We would also like to investigate the performance of the new heuristics in problems where there is a very large set of different domain sizes at the start of search.

References

1. D. Brelaz. New methods to color the vertices of a graph. *JACM*, 22(4):251–256, 1979.
2. P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proc. IJCAI-91*, pages 331–337, 1991.
3. R. Dechter and I. Meiri. Experimental evaluation of preprocessing algorithms for constraint satisfaction problems. *Artificial Intelligence*, 68:211–241, 1994.
4. R. Feldman and M.C. Golumbic. Interactive scheduling as a constraint satisfaction problem. *Annals of Mathematics and Artificial Intelligence*, 1:49–73, 1990.
5. J. Gaschnig. A general backtracking algorithm that eliminates most redundant tests. In *Proc. IJCAI-77*, page 457, 1977.
6. J. Gaschnig. Performance measurement and analysis of certain search algorithms. Tech. rep. CMU-CS-79-124, Carnegie-Mellon University, 1979.
7. I.P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. Scaling effects in the CSP phase transition. In *Principles and Practice of Constraint Programming*, pages 70–87. Springer, 1995.
8. I.P. Gent, E. MacIntyre, P. Prosser, and T. Walsh. The constrainedness of search. In *Proc. AAAI-96*, 1996.
9. I.P. Gent and T. Walsh. Easy problems are sometimes hard. *Artificial Intelligence*, 70:335–345, 1994.
10. M.L. Ginsberg. Dynamic backtracking. *JAIR*, 1:25–46, 1993.
11. S.W. Golomb and L.D. Baumert. Backtrack programming. *JACM*, 12:516–524, 1965.
12. S. Grant and B.M. Smith. The phase transition behaviour of maintaining arc consistency. In *Proc. ECAI-96*, pages 175–179, 1996.
13. R.M. Haralick and G.L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.
14. B. Nudel. Consistent-labeling problems and their algorithms: Expected-complexities and theory-based heuristics. *Artificial Intelligence*, 21:135–178, 1983.
15. P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.
16. P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1-2):1–15, 1996.
17. P.W. Purdom. Search rearrangement backtracking and polynomial average time. *Artificial Intelligence*, 21:117–133, 1983.
18. D. Sabin and E.C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proc. ECAI-94*, pages 125–129, 1994.
19. B.M. Smith and M.E. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1-2):1–15, 1996.
20. B.M. Smith and S. Grant. Sparse constraint graphs and exceptionally hard problems. In *Proc. IJCAI-95*, pages 646–651, 1995.
21. E.P.K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.
22. E.P.K. Tsang, J.E. Borrett, and A.C.M. Kwan. An attempt to map the performance of a range of algorithm and heuristic combinations. In *Hybrid Problems, Hybrid Solutions*, pages 203–216. IOS Press, 1995. Proceedings of AISB-95.
23. C.P. Williams and T. Hogg. Exploiting the deep structure of constraint problems. *Artificial Intelligence*, 70:73–117, 1994.

Empirical Studies of Heuristic Local Search for Constraint Solving*

Jin-Kao Hao and Raphaël Dorne

LGI2P
EMA-EERIE
Parc Scientifique Georges Besse
F-30000 Nîmes
France
email: {hao, dorne}@eerie.fr

Abstract. The goal of this paper is twofold. First, we introduce a class of local search procedures for solving optimization and constraint problems. These procedures are based on various heuristics for choosing variables and values in order to examine a general neighborhood. Second, four combinations of heuristics are empirically evaluated by using the graph-coloring problem and a real world application - the frequency assignment problem. The results are also compared with those obtained with other approaches including simulated annealing, Tabu search, constraint programming and heuristic graph coloring algorithms. Empirical evidence shows the benefits of this class of local search procedures for solving large and hard instances.
Keywords: Local search, constraint solving, combinatorial optimization, graph coloring, frequency assignment.

1 Introduction

Constraint problems embodies a class of general problems which are important both in theory and in practice. Well-known examples include constraint satisfaction problems (CSP), maximal constraint satisfaction problems (MCSP) [5] and constraint satisfaction optimization problems (CSOP) [18]. Constraint problems can be considered as search problems, i.e. given a finite search space composed of a set of configurations, we want to find one or more particular configurations which minimize (or maximize) certain pre-defined criteria. Constraint problems have many practical applications related to scheduling, transportation, layout/circuit design, telecommunications and so on. In general, constraint problems are NP-hard. Consequently, there is little hope of finding any deterministic polynomial solution for this class of problems. Given their practical importance, many methods have been devised to tackle these search problems. This paper looks at one class of methods which are based on surprisingly simple, yet powerful local search techniques.

* Work partially supported by the CNET (French National Research Center for Telecommunications) under the grant No.940B006-01.

Local search (LS), also called neighborhood search, constitutes an important class of general heuristic methods based on the notion of neighborhood [14]. Starting with an initial configuration, a typical LS procedure replaces iteratively the current configuration by one of its neighbors, which is often of better quality, until some stop criteria are verified; for example, a fixed number of iterations is reached or a sufficiently good local optimum is found. Well-known examples of LS-based methods include simulated annealing (SA) [11], Tabu search [6] and various forms of hill-climbers [1]. Given that LS uses only a neighborhood function and possibly some other general notions, it can be applied to a large class of problems. Traditionally, LS was used with success to tackle well-known NP-hard combinatorial optimization problems such as TSP [12] and graph partitioning [10]. More recent applications of LS include the graph coloring problem [8, 9, 3], CSPs [13], and the satisfiability problem [7, 16].

Local search is essentially based on three components: a *configuration structure (encoding)*, a *neighborhood function* defined on the configuration structure, and a *neighborhood examination mechanism*. The first component defines the search space S of the application, the second associates a subset of S with each point of the search space while the third defines the way of going from one point to another. The configuration structure is often application-dependent and should be chosen in such a way that it reflects the natural solution space of the problem and facilitates its exploration and exploitation. For a given neighborhood, the way in which neighbors are examined is certainly the most determinant part of the performance of a LS procedure.

In this paper, we present a class of LS procedures for solving optimization and constraint problems. These LS procedures are based on different heuristics for examining a general neighborhood. Some heuristics are well-known and others less so. Computational tests are carried out on two NP-hard problems: graph coloring and frequency assignment in mobile radio networks. Experimental evidence shows the benefits of these procedures for solving large and hard instances. The results are compared with those obtained by two other LS procedures: SA and Tabu search and two other heuristic methods: constraint programming and heuristic graph coloring algorithms.

2 Constraint Problems and Local Search

2.1 Constraint Problems

In order to apply LS to constraint problems, we will consider constraint problems as combinatorial optimization problems, i.e., a constraint problem P is defined by a quadruple $< X, D, C, f >$ where

- $X = \{V_1, V_2 ... V_n\}$ is all the distinct variables in P,
- $D = \{D_1, D_2 ... D_n\}$ is all the domains of variables,
- $C = \{C_i(V_{i_1} ... V_{i_k})\}$ is the set of constraints, each C_i being a relation on $V_{i_1} ... V_{i_k} \in X$,
- f is a cost function to be minimized (maximized).

With this definition, several cases are possible. First, if P is a standard CSP, there will be no associated cost function. Therefore, any assignment of values to the variables satisfying the constraints of C will be a solution. Second, if P is a CSOP, then solving P implies finding assignments such that all the constraints are satisfied and the cost f minimized (or maximized). Third, if P is a MCSP, i.e. the underlying $CSP < X, D, C >$ is not satisfiable, then solving P is to maximize (minimize) the number of satisfied (unsatisfied) constraints.

Note that in this formulation, the *cost* function f is not necessarily the *evaluation* function which is required by any LS search procedure to evaluate the quality of configurations.

2.2 Configuration and Neighborhood

Given a constraint problem $P =< X, D, C, f >$, the configuration structure is defined as follows: a configuration s is any complete assignment such that $s = \{< V_i, v_i > \mid V_i \in X \text{ and } v_i \in D_i\}$. We use $T(s, V_i)$ to note the value of V_i in s, i.e. if $< V_i, v_i > \in s$ then $T(s, V_i) = v_i$. The search space S of the problem P is then defined as the set of all the possible configurations. Clearly the cardinality of S is equal to the product of the size of the domains, i.e. $\prod_{i=1}^{n} |D_i|$.

In general, the configuration structure is application-dependent. However, some configuration structures are general enough to be applied to many applications. The above structure is such an example. In fact, it can be used to model problems such as graph coloring, satisfiability and many CSPs. Another general configuration structure is "permutation" which can be used to model naturally the traveling salesman problem.

Given the configuration structure defined above, the neighborhood function $N : S \rightarrow 2^S$ may be defined in many ways. In this paper, we use the *one-difference* or *one-move* neighborhood[2]. Formally, let $s \in S$ be a configuration, then $s' \in N(s)$ if and only if there exists one and only one $i \in [1..n]$ such that $T(s, V_i) \neq T(s', V_i)$. In other words, a neighbor of a configuration s can be obtained by changing the current value of a variable in s. Since a variable V_i in s can take any of its $|D_i|$ values, s has exactly $\sum_{i=1}^{n} (|D_i| - 1) = \sum_{i=1}^{n} D_i - n$ neighbors. Note that this neighborhood is a reflexive and symmetric relation.

2.3 Neighborhood Examination

We now turn to different ways of examining the neighborhood, i.e., going from one configuration to another. In this paper, we use a two-step, heuristic-based examination procedure to perform a move.

1. first choose a *variable*,
2. and then choose a *value* for the selected variable.

It is easy to see that many heuristics are possible for both choices. In what follows, we present various heuristics for choosing variables and values.

[2] In general, a k-move neighborhood can be defined.

Heuristics for choosing the variable V_i:

- *var.1 random*: pick randomly a variable V_i from X;
- *var.2 conflict-random*: pick randomly a variable from the *conflict set* defined by $\{V_i \mid V_i \in X$ is implicated in an unsatisfied constraint$\}$;
- *var.3 most-constrained*: pick a most constrained variable, for instance, the one which occurs in the biggest number of unsatisfied constraints (break ties randomly).

Heuristics for choosing the value for V_i:

- *val.1 random*: pick randomly a value from D_i;
- *val.2 best-one (min-conflicts[13])*: pick a value which gives the greatest improvement in the evaluation function (break ties randomly). If no such value exists, pick randomly a value which does not lead to a deterioration in the evaluation function (the current value of the variable may be picked);
- *val.3 stochastic-best-one*: pick a best, different value which does not lead to a deterioration in the evaluation function. If no such value exists, with probability p, take a value which leads to the smallest deterioration;
- *val.4 first-improvement*: pick the *first* value which improves the evaluation function. If no such value exists, take a value which does not lead to a deterioration or which leads to the smallest deterioration[3];
- *val.5 probabilistic-improvement*: with probability p, apply the *val.1 random* heuristics; with $1-p$, apply the *val.2 best-one* heuristic.

Let us note first that *val.2* forbids deteriorative moves. As we will see later, this property will penalize its performance compared with others.

Both *val.3* and *val.5* use a probability in order to accept deteriorative moves. The purpose of accepting such moves is to prevent the search from being stuck in local optima by changing the search direction from time to time. This probability may be static or dynamic. A static probability will not be changed during the search while a dynamic one may be modified by using some pre-defined mathematic laws or may adapt itself during the search. In any case, this probability is determined more often empirically than theoretically.

val.5 is similar to the *random-walk* heuristic used by GSAT [17], but they are different since for the satisfiability problem, there is only one explicit choice, i.e. the choice of the variable to flip. Here the heuristic determines the value for a chosen variable, not the variable itself. Another interesting point is that varying the probability p will lead to different heuristics. If $p = 1$, *val.5* becomes *val.1 random*. If $p = 0$, *val.5* becomes *val.2 best-one*. Finally, both the p part and the $1-p$ part can be replaced by other heuristics.

Evidently, any combination of a *var.x* heuristic and a *val.y* heuristic gives a different strategy for examining the neighborhood. There are 15 possibilities in our case. One aim of this work is to assess the performance of these combinations. Note that an extensive study on the *val.2 min-conflicts* heuristic for solving CSPs has been carried out and conclusions have been drawn [13]. However, that work

[3] As for *val.3*, a probability can be introduced here to control deteriorative moves.

concerns essentially the *value* choice for a given conflict variable. In this work, we study various combinations for choosing both variables and values.

Note finally that in order to efficiently implement the above variable/value choice heuristics, special data structures are indispensable to be able to recognize the conflict or the most constrained variables and the appropriate new value for the chosen variable.

2.4 Heuristic Local Search Template

Using the above variable/value choice heuristics, various heuristic local search (HLS) procedures can be built. The general HLS template is given below:

Procedure Heuristic Local Search (HLS)

Input:
$P =< X, D, C, f >$: the problem to be solved;
f & L: objective function and its lower bound to be reached;
MAX: maximum number of iterations allowed;
Output:
s: the best solution found;
begin
  generate(s); /* generate an initial configuration */
  $I \leftarrow 0$; /* iterations counter */
  while $(f(s) > L)$ **and** $(I < MAX)$ **do**
    choose a variable $V_i \in X$; /* heuristics var.x */
    choose a value $v_i \in D_i$ for V_i; /* heuristics val.y */
    if $T(s, V_i) \neq v_i$ **then**
      $s \leftarrow s - \{< V_i, T(s, V_i) >\} + \{< V_i, v_i >\}$;
    $I \leftarrow I + 1$;
  output(s);
end

This HLS template uses two parameters L and MAX in the stop condition. L fixes the (optimization) objective to be reached and MAX the maximum number of iterations allowed. Therefore, the procedure stops either when an optimal solution has been found or MAX iterations have been performed. The complexity of such a procedure depends on MAX, the size of domains $|D_i|$ and the way in which the neighborhood is examined.

3 Experimentation and Results

In this section, we present empirical results of HLS procedures which are based on some representative combinations of heuristics introduced above for exploiting the neighborhood structure. Tests are carried out on two NP-complete problems: the graph-coloring (COL) and the frequency assignment (FAP). For the

COL problem, test instances come essentially from the archives of the second DIMACS Implementation Challenge[4]. For the FAP problem, instances (60 in total) are provided by the CNET (French National Research Center for Telecommunications)[5]. A limited number of instances for each problem are used to evaluate four combinations of heuristics for choosing variables/values. More instances are then solved to compare these HLS procedures with other approaches including Tabu search, SA, constraint programming, and heuristic graph coloring algorithms.

3.1 Tests and Problem Encoding

Graph Coloring

There are two main reasons to choose the graph-coloring as our test problem. First, it is a well-known reference for NP-complete problems. Second, there are standard benchmarks in the public domain.

The basic COL is stated as a *decision* problem: given k colors and a graph $G =< E, V >$, is it possible to color the vertices of E with the k colors in such a way that any two adjacent vertices of V have different colors. In practice, one is also interested in the *optimization* version of the problem: given a graph G, find the smallest k (the chromatic number) with which there is a k-coloring for G.

Many classic methods for graph-coloring are based on either exhaustive search such as branch-and-bound techniques or successive augmentation heuristics such as Brélaz's DSATUR algorithm, Leighton's Recursive Largest First (RLF) algorithm and the more recent XRLF by Johnson et al. [9]. Recently, local search procedures such as SA [9] and Tabu search [8, 3] were also applied to the coloring problem.

In order to apply our HLS procedures to the graph-coloring problem, the COL must first be encoded. Given a graph $G =< E, V >$, we transform G into the following constraint problem $< X, D, C, f >$ where

- $X = E$ is the the set of the vertices of G,
- D is the set of the k integers representing the available colors,
- C is the set of constraints specifying that the colors assigned to u and v must be different if $\{u, v\} \in V$,
- f is the number of colors used to obtain a proper (conflict-free) coloring.

With this encoding, a (proper or improper) coloring of G will be a complete assignment $\{< u, i > | u \in E, i \in D\}$. Coloring G consists in assigning integers in D (colors) to the vertices in E in such a way that all the constraints of C are satisfied while a minimum number of k colors is used.

In order to minimize k, a HLS procedure solves a series of CSPs (decision problems). More precisely, it begins with a big k and tries to solve the underlying

[4] DIMACS benchmarks are available from ftp dimacs.rutgers.edu.

[5] Another set of FAP instances are available from ftp ftp.cs.city.ac.uk. These tests correspond to sparse graphs and are consequently much less constrained.

$CSP < X, D, C >$. If a proper k-coloring is found, the search process proceeds to color the graph with $k-1$ colors and so on. If no coloring can be found with the current k colors, the search stops and reports on the last proper coloring found. In other words, it tries to solve a harder problem (CSP) with fewer colors if it manages to solve the current one. In order to solve each underlying CSP, our local search procedure needs an *evaluation* function to measure the relative quality of each configuration (which is usually an improper coloring). Several possibilities exist to define this function. In this paper, it is defined simply as the number of unsatisfied constraints.

The DIMACS archives contain more than 50 benchmarks. We have chosen a subset of them for our experiments. Note that the main objective of this work is not to improve on the best known results for these instances. In order to achieve any improvement, local search alone may not be sufficient. It must be combined with special coloring techniques. In this work, we use these instances to study the behavior of our HLS procedures. For this purpose, we have chosen some 16 small and medium size (< 500 vertices) instances from different classes.

Frequency Assignment Problem

Our second test problem concerns a real world application: the frequency assignment problem which is a key application in mobile radio networks engineering. As we will see later, the basic FAP can be easily shown to be NP-complete.

The main goal of the FAP consists in finding frequency assignments which minimize the number of frequencies (or channels) used in the assignment and the electro-magnetic interference (due to the re-use of frequencies). The difficulty of this application comes from the fact that an acceptable solution of the FAP must satisfy a set of multiple constraints, some of these constraints being orthogonal. The most severe constraint concerns a very limited radio spectrum consisting of a small number of frequencies (usually about 60). This constraint imposes a high degree of frequency re-use, which in turn increases the probability of frequency interference. In addition to this frequency constraint, two other types of constraints must be satisfied to ensure communication of a good quality:

1. *Traffic constraints*: the minimum number of frequencies required by each station S_i to cover the communications of the station, noted by T_i.
2. *Frequency interference constraints* belong to two categories: 1. *Co-station constraints* which specify that any pair of frequencies assigned to a station must have a certain distance between them in the frequency domain; 2. *Adjacent-station constraints* which specify that the frequencies assigned to two adjacent stations must be sufficiently separated. Two stations are considered as adjacent if they have a common emission area.

About 60 instances were used in our experiments. Some of them are not only very large in terms of number of stations and in terms of number of interference constraints, but also very difficult to solve. The FAP instances we used in our experiments belong to three different sets which are specified below.

- **Test-Set-No.1** *Traffic constraints*: one frequency per station. Consequently, there is no co-station constraint. *Adjacent constraints*: frequencies assigned to two adjacent stations must be different.
- **Test-Set-No.2** *Traffic constraints*: two frequencies per station. *Co-station constraints*: frequencies assigned to a station must have a minimum distance of 3. *Adjacent constraints*: frequencies assigned to two adjacent stations must have a minimum distance of 1 or 2 according to the station.
- **Test-Set-No.3** *Traffic constraints*: up to 4 frequencies per station. *Co-station and adjacent constraints*: the separation distance between frequencies assigned to the same station or adjacent stations varies from 2 to 4.

In fact, Test-Set-No.1 corresponds to the graph-coloring problem. To see this, frequencies should be replaced by colors, stations by vertices and adjacent constraints by edges. Finding an optimal, i.e., an interference-free, frequency assignment using a minimum number of distinct frequencies is equivalent to coloring a graph with a minimum number of colors.

In order to apply our HLS procedures, the FAP will be encoded as a constraint (optimization) problem $< X, D, C, f >$ such that

- $X = \{L_1, L_2, ..., L_{NS}\}$ where each L_i represents a list of T_i frequencies required by the i^{th} station and NS is the number of stations in the network,
- D is the set of the NF integers representing the NF available frequencies,
- C is the set of co-station and adjacent-station interference constraints,
- f is the number of frequencies used to obtain conflict-free assignments.

It is easy to see that with this encoding, a frequency assignment has the length of $\sum_{i=1}^{NS} T_i$. Fig.1 gives an example where the traffic of the three stations is respectively 2, 1 and 4 frequencies and $f_{i,j}$ represents the j^{th} frequency value of the i^{th} station S_i.

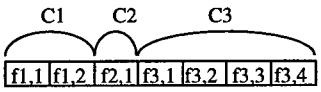


Fig. 1. FAP encoding

Solving the FAP consists in finding assignments which satisfy all the interference constraints in C and minimize NF, the number of frequencies used. To minimize NF, we use the same technique as that explained in the last section.

3.2 Comparison of Strategies for Neighborhood Examination

We have chosen to compare four combinations of heuristics: *var.1 random/val.2 conflict-random, var.2 conflict-random/val.2 min-conflicts, var.2 conflict-random/val.3 stochastic-best-one, var.2 conflict-random/val.5 probabilistic-improvement.* The main reason for choosing these strategies is to have a good sample which combines two important aspects of a search strategy: randomness and guideness.

The *var.1 random/val.1 random* combination, which represents a random search strategy, was also tested. The results of this strategy were so bad that we have decided to omit them from the comparison. The probability p used in *var.2/val.3* is respectively fixed at 0.1 for the COL and 0.2 for the FAP. The probability p used in *var.2/val.5* is fixed at 0.05 for both problems.

Table 1 shows the comparative results of these four strategies for 6 COL instances (3 structured graphs $le450\_15[c-d].col$, $flat\_28\_0.col$ and 3 random graphs). For each instance and each strategy, we give the mean number of colors (Colors) over 5 runs and the mean number of evaluations (Eval.) in thousands, needed to find a coloring. These two criteria reflect respectively the quality of a solution and the efficiency of a strategy.

Table 1. Comparison of heuristics for COL

Problems	Runs	var1/val2		var2/val2		var2/val3		var2/val5	
		Colors	Eval.	Colors	Eval.	Colors	Eval.	Colors	Eval.
R125.1c.col	5	46.00	1141	47.80	2939	47.60	128	46.00	653
R250.5.col	5	72.60	9800	70.60	8500	70.00	5600	70.80	1420
DJSC250.5.col	5	33.20	2000	34.80	2100	31.60	1240	32.00	1280
le450_15c.col	5	22.60	6173	>25	>5100	21.60	2574	21.80	2907
le450_15d.col	5	22.60	3789	>25	>5100	21.40	3094	21.20	3906
flat300_28_0.col	5	35.00	20368	>38	>7100	34.00	2832	34.80	4486

From Table 1, we observe that compared with the other strategies, *var.2 conflict-random/val.2 min-conflicts* gives the worst results. In fact, for 5 out of 6 instances, it requires more colors and more evaluations than the others to find colorings. In particular, it has serious problems coloring the 3 structured graphs even with up to 10 colors more than the minimum. *var.1 random/val.2 conflict-random* is a little better than *var.2/val.2* for 5 instances, but worse than *var.2 conflict-random/val.3 stochastic-best-one* and *var.2 conflict-random/val.5 probabilistic-improvement*. Finally, the results of *var.2/val.3* and *var.2/val.5* are similar with a slightly better performance for the first: *var.2/val.3* performs a little better than *var.2/val.5* for 4 instances. Therefore, for the coloring problem, it seems that the following relation holds: $(var.2/val.3 \approx var.2/val.5) > var.1/val.2 > var.2/val.2$, where $\approx$ and $>$ mean respectively "comparable" and "better than" in terms of the solution-quality/efficiency. This relation was confirmed when we tried to solve other COL instances.

Table 2 shows the comparative results of these four strategies for 5 FAP instances. Each instance is specified by three numbers $nf.tr.nc$ which are respectively the lower bound of the number of distinct frequencies necessary to have an interference-free assignment, the sum total of the traffic (the number of integer variables) of all stations, and the number of co-station and adjacent-station interference constraints. For example, 16.300.12370 means that this instance has a total traffic of 300 and 12,370 interference constraints, and requires at least 16 frequencies to have an interference-free assignment. The same criteria as for the COL are used except that "Colors" criterion is replaced by the mean number of

the frequencies (NF) for having interference-free assignments.

From Table 2, we see once again that *var.2 conflict-random/val.2 min-conflicts* is the worst strategy. With even 10 frequencies more than the lower bound (16), it cannot find a conflict-free assignment for *16.300.12370*. For the other instances, it requires at least 3 to 6 frequencies more than the lower bound. At the other extreme, we notice that *var.2 conflict-random/val.3 stochastic-best-one* dominates the others for the 5 instances. Finally, the performance of *var.2 conflict-random/val.5 probabilistic-improvement* is between that of *var.1 random/val.2 conflict-random* and *var.2/val.3*. Therefore, for the FAP, it seems that the following relation holds: *var.2/val.3* > *var.2/val.5* > *var.1/val.2* > *var.2/val.2*. This relation was confirmed when we tried to solve other FAP instances.

Table 2. Comparison of heuristics for FAP

Problems	Runs	var1/val2		var2/val2		var2/val3		var2/val5	
		NF	Eval.	NF	Eval.	NF	Eval.	NF	Eval.
8.150.3308	5	8.0	916	14.00	1229	8.00	47	8.00	50
15.300.8940	5	16.00	1769	18.00	741	15.00	1804	15.40	2016
16.150.3203	5	17.80	1246	21.80	6731	16.75	10236	17.60	2638
16.300.12370	5	18.75	5978	>26	>1350	17.00	2535	19.20	3120
30.600.45872	5	30.00	6513	33.20	11080	30.00	570	30.00	1700

Based on the data in Tables 1 and 2, we can make several remarks. First, we notice that *var.2 conflict-random/val.3 stochastic-best-one* and *var.2 conflict-random/val.5 probabilistic-improvement* turn out to be winners for both problems compared with *var.1 random/val.2 min-conflicts* and *var.2 conflict-random/val.2 min-conflicts*. As for most procedures of heuristic search, there is no theoretical justification for this. However, intuitive explanations help to understand the result. The tested instances represent hard problems and may contain many deep local optima. For a search procedure to have a chance of finding a solution, it must be not only able to converge efficiently towards optima, but also able to escape from local optima. Both *var.2/val.3* and *var.2/val.5* have this capacity thanks to deteriorative moves authorized by *var.2/val.3* and random moves in *var.2/val.5*. On the contrary, *var.1 random/val.2 min-conflicts* and *var.2 conflict-random/val.2 min-conflicts*, once they have reached a local optimum, is trapped since deteriorative moves are forbidden. Although they both allow side-walk moves by taking values that do not change the value of the evaluation function, this is not sufficient to escape from local optima.

If we trace the evolution of the evaluation function, i.e. the number of unsatisfied constraints as a function of the number of iterations, we observe that all four strategies are able to reduce rapidly the number of unsatisfied constraints after a relatively small number of iterations (descending phase). The difference between these strategies appears during the following phase. In fact, for *var.2/val.3* and *var.2/val.5*, the search, after the descending phase, goes into an oscillation phase composed of a long series of up-down moves, while for *var.1/val.2* and *var.2/val.2*, the search stagnates at plateaus.

In principle, any search strategy must conciliate two complementary aspects:

exploitation and exploration. Exploitation emphasizes careful examinations of a given area while exploration encourages the search to investigate new areas. From this point of view, we can say that *var.2/val.3* and *var.2/val.5* manage to exploit and explore correctly the search space by balancing randomness and guideness. On the contrary, due to the deterministic nature of the *min-conflicts* heuristic, *var.1/val.2* and *var.2/val.2* focus only on the aspect of exploitation.

3.3 Comparisons with Other Methods

This section lists the best results of the HLS procedures with *var.2/val.3* and *var.2/val.5* for the COL and the FAP. The controlling probability p used by *var.2/val.3* varied between 0.1 and 0.2. The probability used by *var.2/val.5* was around 0.05. Whenever possible, the results are compared with those obtained by other methods: Tabu search for the COL, SA, constraint programming (CP) and heuristic coloring algorithms (HCA) for the FAP.

Results are based on 1 to 10 independent runs according to the difficulty of the instance. For each run, a HLS procedure begins with k colors/frequencies (usually 10 colors/frequencies more than the best known value) and tries to find a conflict-free solution for the underlying CSP. For each given color/frequency, $t = 1, 2, 3$ tries are authorized, i.e. if the search cannot find a conflict-free solution within t tries, the current run is terminated. If a conflict-free solution is found (within t tries), the number of colors/frequencies is decreased and the search continues. The maximum number of iterations (moves) is fixed at 50,000 to 500,000 for each try in each run.

Three criteria are used for reporting results: the minimum and the mean number of colors/frequencies (NC/NF), and the mean time for finding a conflict-free solution. The first two criteria reflect the quality of solutions while the third reflects solving efficiency. In order to better assess the robustness of HLS, we also give in brackets how many times a solution with the minimum number of colors/frequencies has been found. The time (on a SPARCstation 10) is the total user time for solving an instance, including the time for solving the intermediate CSPs and the time for failed tries (up to 3 tries for each run). Note that timing is given here only as a rough indication.

Table 3 shows the results of the HLS procedures for some DIMACS benchmarks and two classes ($g100.5.col$ and $g300.5.col$) of 25 random graphs taken from [4]. The instances are specified as follows. For random graphs $gxxx.y.col$ and $Rxxx.y.col$, xxx and y indicate respectively the number of vertices and the density of the graph. For Leighton's graphs $lexxx\_yy[a-d].col$ and structured graphs $flat300\_yy\_0.col$, yy is the chromatic number. $DSJC.1000.5.col(res)$ is the *residual graph* (200 vertices and 9,633 edges) of $DSJC.1000.5.col$ (1,000 vertices and about 500,000 edges). This residual graph was obtained by eliminating 61 independent sets and all the related edges [4]. Table 3 also shows the results of Tabu search (using a faster SuperSPARC 50 machine) [3].

The top part of Table 1 presents results for graphs having up to 300 vertices. From the data in Table 1, we notice that for the 20 $g100.5.col$ instances, HLS finds the same result as that of Tabu search while for the 5 $g300.5.col$ instances,

it needs en average 1.3 more colors. For the *Rxxx.y.col* family, HLS manages to find a best known coloring for 5 out of 6 instances, but has difficulty to color *R*125.5.*col* for which it requires 2 colors more than Tabu. For the three *flat*300_*yy*_0.*col* instances, HLS obtains the same results as Tabu search for 2 out of 3 instances. For *flat*300_26_0.*col*, HLS finds an optimal coloring for one out of 5 runs. For *DSJC*.1000.5.*col*(*res*), HLS manages to color the graph with 24 colors in about half an hour (instead of the best known value of 23 obtained with Tabu after about 20 hours of computing). With 23 colors, HLS usually leaves 1 or 2 unsatisfied constraints at the end of its search.

Table 3. HLS performance for COL

Problems	Edges	HLS				Tabu			
			NC				NC		
		Runs	Min.(nb)	Ave.	T[sec.]	Runs	Min.	Ave.	T[sec.]
g100.5.col (20 inst.)	$\simeq$ 2500	1	14(1)	14.95	275.3	1	-	14.95	$\simeq$ 9.5
g300.5.col (5 inst.)	$\simeq$ 22000	1	34(1)	34.80	4793	1	-	33.50	$\simeq$ 353
R125.1.col	209	10	5(10)	5.00	0.5	10	5	5.00	0
R125.1c.col	7501	10	46(10)	46.00	129	10	46	46.00	4.1
R125.5.col	3838	10	37(1)	38.00	187	5	35	35.60	1380
R250.1.col	867	10	8(10)	8.00	2	10	8	8.00	0
R250.1c.col	30227	10	64(8)	64.20	2946	10	64	64.00	108
R250.5.col	14849	3	69(1)	69.75	6763	5	69	69.00	1664
flat300_20_0.col	21375	5	20(5)	20.00	1997	10	20	20.00	40
flat300_26_0.col	21633	5	26(1)	31.40	6710	3	26	26.00	8100
flat300_28_0.col	21695	5	33(5)	33.00	2402	3	33	33.00	4080
DSJC.1000.5.col(res)	9633	5	24(5)	24.00	1531	5	23	23.00	68400
le450_15a.col	8168	5	16(5)	16.00	354	10	15	15.00	248
le450_15b.col	8169	5	16(5)	16.00	273	10	15	15.00	248
le450_15c.col	16680	5	15(1)	16.00	4376	10	16	16.00	268
le450_15d.col	16750	5	15(2)	15.60	3990	10	16	16.00	791
le450_25a.col	8260	5	25(5)	26.00	61	5	25	25.00	4.0
le450_25b.col	8263	5	25(4)	25.60	27	5	25	25.00	3.9

The lower part of Table 1 presents the results for 6 Leighton graphs. We see that HLS finds an optimal coloring for 4 out of 6 instances. For *le*450_15*c*.*col* and *le*450_15*d*.*col*, HLS requires one less color than Tabu. For *le*450_15*a*.*col* and *le*450_15*b*.*col*, the reverse is true. It should be mentioned that, in order to color these graphs and more generally any graph having more than 300 vertices, Tabu uses the technique of independent sets mentioned above to produce first a much smaller residual graph which is then colored.

Finally, we make a general remark about the HLS procedures used to obtain the above results. The main goal of this work is to study the behavior of various heuristics, but not to improve on the best results for the coloring problem (In fact, this second point constitutes another ongoing work). Consequently, the HLS procedures used are voluntarily general and do not incorporate any specialized technique for the COL. With this fact in mind, the results of the HLS procedures may be considered to be very encouraging.

Table 4 shows the best results for 12 FAP instances obtained with HLS procedures combined with a technique for handling co-station constraints [2]. These instances are taken from a series of 60 instances and represent some of the hardest problems. It should be remembered that each instance *nf.tr.nc* is specified by the lower bound for the number of frequencies, the sum total of the traffic, and the number of interference constraints. As we can see from the table, some instances are very large and have a high density of constraints. Table 4 also shows the best results of SA, CP (ILOG-Solver) and HCA, all reported in [15]. These procedures have been run on HP Stations which are considered to be at least three times faster than the SPARCstation 10 we used. A minus "-" in the table means that the result is not available.

Table 4. HLS performance for FAP

Problems	HLS					HCA	CP		SA	
		NF								
	Runs	Min.(nb)	Ave.	Eval.	T[sec.]	NF	NF	T[sec.]	NF	T[sec.]
8.150.2200	10	8(10)	8.00	812	639	8	8	7200	8	509
15.300.8940	10	15(10)	15.00	1326	1606	20	17	3600	15	4788
15.300.13400	10	15(10)	15.00	2557	3600	27	25	1560	15	2053
16.150.3203	10	**16(1)**	16.9	1070	658	19	18	14400	**17**	1744
16.150.3323	10	17(8)	17.2	3246	2283	19	19	7200	17	1383
30.300.13638	10	30(10)	30.00	18	25	30	30	1	30	1558
30.600.47852	2	**30(1)**	30.50	46666	122734	47	46	4800	**36**	5309
40.335.11058	2	43(2)	43.00	9434	17823	-	-	-	-	-
40.966.35104	10	45(10)	45.00	707	1341	-	-	-	-	-
60.600.47688	10	60(10)	60.00	4234	9288	60	-	-	60	858
60.600.45784	10	60(10)	60.00	2039	3981	60	-	-	60	516

In Table 4, we notice first that HCA gives the worst results for all the solved instances; it requires up to 17 frequencies more than the lower bound. The results of CP are a little better than HCA for 6 out of 7 problems. It is interesting to see that instances which are difficult for HCA remain difficult for CP. On the contrary, the results of HLS and SA are much better than those of HCA and CP: HLS (SA) finds an optimal assignment for 8 (6) instances. In general, the harder the problem to be solved, the bigger the difference: for *15.300.13400* and *30.600.47852*, there is a difference of more than 10 frequencies. Note finally that HLS improves on the result of SA for two instances (in bold). In particular, for *30.600.47852*, HLS finds a conflict-free assignment with only 30 frequencies (36 for SA). This is rather surprising given the similar results of these two approaches for the other instances. The computing time for HLS and SA is generally similar to obtain solutions of the same quality. It is difficult to compare the computing time with CP since they give solutions which are too different.

To sum up, HLS gives the best result for the 12 hardest FAP instances. This remains true for 48 other instances which have been solved, but not reported here. However, we notice that for easier instances, all the methods behave similarly. Moreover, CP and HCA may be faster for some easy instances.

4 Conclusions & Future Work

In this paper, we have presented a class of local search procedures based on heuristics for choosing variables/values. The combinations of these heuristics give different strategies for examining a very general neighborhood. These heuristics can be applied to a wide range of problems.

Four representative combinations of these heuristics out of fifty possibilities have been empirically evaluated and compared using the graph-coloring problem and the frequency assignment problem. Two strategies turn out to be more efficient: *var.2 conflict-random/val.3 stochastic-best-one* and *var.2 conflict-random/val.5 probabilistic-improvement.* In essence, these two strategies are able to find a good balance between randomness and guideness, which allows them to explore and exploit correctly the search space. The controlling probability p used by these two strategies plays an important role in their performance and should be empirically determined. In our experiments, we have used values ranging from 0.1 to 0.2 for *var.2/val.3* and values around 0.05 for *var.2/val.5.* More work is needed to better determine these values. Moreover, the possibility of a self-adaptive p is also worth studying.

We also found that *var.1 random/val.2 min-conflicts*, and especially *var.2 conflict-random/val.2 min-conflicts* are rather poor strategies for both COL and FAP. In fact, these two strategies are too deterministic to be able to escape from local optima. It is interesting to contrast this finding with the work concerning the *min-conflicts* heuristic [13].

To further evaluate the performance of these heuristic LS procedures, they were tested on more than 40 COL benchmarks and 12 hard FAP instances. Although they are not especially tuned for the coloring problem, the HLS procedures give results which are comparable with those of Tabu search for many instances. At the same time, we noticed that HLS alone, like many other pure LS procedures, has difficulty coloring very large and hard graphs. For the FAP problem, the results of HLS on the tested instances are at least as good as those of simulated annealing, and much better than those obtained with constraint programming and heuristic coloring algorithms.

Currently, we are working on several related issues. At a practical level, we want to evaluate other combinations not covered in this paper. Secondly, we are developing specialized coloring algorithms combining the general heuristics of this paper and well-known coloring techniques. Indeed, in order to color hard graphs, all efficient algorithms use specialized techniques. It will be very interesting to see if the combination of our heuristics with these kinds of techniques can lead to better results.

At a more fundamental level, we try to identify the characteristics of problems which may be efficiently exploited by a given heuristic. This is based on the belief that a heuristic has a certain capacity to exploit special structures or characteristics of a problem. Thus, the heuristic may have thus some "favorite" problems. A long-term goal of the work is to look for a better understanding of the behavior of LS heuristics for solving problems and answering such fundamental questions as when and why a heuristic works or does not.

Acknowledgments

We would like to thank A. Caminada from the CNET for his assistance, P. Galinier for useful discussions and the referees for their comments on the paper.

References

1. D.H. Ackley, A connectionist machine for genetic hillclimbing. Kluwer Academic Publishers, 1987.
2. R. Dorne and J.K. Hao, "Constraint handling in evolutionary search: a case study about the frequency assignment", Submitted to the 4th Intl. Conf. on Parallel Problem Solving from Nature (PPSN'96), Berlin, Germany, Sept. 1996.
3. C. Fleurent and J.A. Ferland, "Object-oriented implementation of heuristic search methods for graph coloring, maximum clique, and satisfiability". in D.S. Johnson and M.A. Trick (eds.) "2nd DIMACS Implementation Challenge" (to appear).
4. C. Fleurent and J.A. Ferland, "Genetic and hybrid algorithms for graph coloring", to appear in G. Laporte, I. H. Osman, and P. L. Hammer (eds.), Special Issue Annals of Operations Research on Meta-heuristics in Combinatorial Optimization.
5. E.C. Freuder and R.J. Wallace, "Partial constraint satisfaction", Artificial Intelligence, Vol.58(1-3), pp21-70, 1992.
6. F. Glover and M. Laguna, "Tabu Search", in C. R. Reeves (eds.) Modern Heuristics for Combinatorial Problems, Blackwell Scientific Publishing, Oxford, Great Britain.
7. P. Hensen and B. Jaumard, Algorithms for the maximum satisfiability problem, Computing Vol.44, pp279-303, 1990.
8. A. Hertz and D. de Werra, "Using Tabu search techniques for graph coloring". Computing Vol.39, pp345-351, 1987.
9. D.S. Johnson, C.R. Aragon L.A. McGeoch and C. Schevon, "Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning". Operations Research, Vol.39(3), 378-406, 1991.
10. B.W. Kernighan and S. Lin, "An efficient heuristic for partitioning graphs", Bell System Technology Journal, Vol.49, 1970.
11. S. Kirkpatrick, C.D. Gelatt Jr. and M.P. Vecchi, "Optimization by simulated annealing", Science No.220, 671-680, 1983.
12. S. Lin and B.W. Kernighan, "An efficient heuristic for the traveling-salesman problem", Operations Research, Vol.21, pp498-516, 1973.
13. S. Minton, M.D. Johnston and P. Laird, "Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems", Artificial Intelligence, Vol.58(1-3), pp161-206, 1992.
14. C.H. Papadimitriou and K. Steiglitz, "Combinatorial optimization - algorithms and complexity". Prentice Hall, 1982.
15. A. Ortega, J.M. Raibaud, M.Karray, M.Marzoug, and A.Caminada, "Algorithmes de coloration des graphes et d'affectation des fréquences". TR CNET, N-T/PAB/SRM/RRM/4353, August 1995.
16. B. Selman, H.J. Levesque and M. Mitchell, "A new method for solving hard satisfiability problems". Proc. of AAAI-92, San Jose, CA, pp.440-446, 1992.
17. B. Selman and H.Kautz, "Domain-independent extensions to GSAT: solving large structured satisfiability problems". Proc. of IJCAI-93, Chambery, France, 1993.
18. E. Tsang, "Foundations of constraint satisfaction", Academic Press, 1993.

Defeasibility in CLP($\mathcal{Q}$) through Generalized Slack Variables

Christian Holzbaur[†], Francisco Menezes[‡] and Pedro Barahona[‡]

[†]Austrian Research Institute for Artificial Intelligence, and
Department of Medical Cybernetics and Artificial Intelligence
University of Vienna
Freyung 6, A-1010 Vienna, Austria
email: christian@ai.univie.ac.at

[‡]Departamento de Informática
Faculdade de Ciências e Técnologia
Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal
email: {fm,pb}@fct.unl.pt

Abstract. This paper presents a defeasible constraint solver for the domain of linear equations, disequations and inequalities over the body of rational/real numbers. As extra requirements resulting from the incorporation of the solver into an Incremental Hierarchical Constraint Solver (IHCS) scenario we identified: a)the ability to refer to individual constraints by a label, b) the ability to report the (minimal) cause for the unsatisfiability of a set of constraints, and c) the ability to undo the effects of a formerly activated constraint.
We develop the new functionalities after starting the presentation with a general architecture for defeasible constraint solving, through a solved form algorithm that utilizes a generalized, incremental variant of the Simplex algorithm, where the domain of a variable can be restricted to an arbitrary interval. We demonstrate how generalized slacks form the basis for the computation of explanations regarding the cause of unsatisfiability and/or entailment in terms of the constraints told, and the possible deactivation of constraints as demanded by the hierarchy handler.

Keywords: Constraint Logic Programming, Linear Programming, Defeasible Constraint Solving

1 Introduction

Although Constraint Logic Programming enhances the limited expressive power and execution efficiency of Logic Programming, it is insufficient to cope with problems for which many solutions might satisfy a set of mandatory (or hard) constraints of the problem, but where some solutions are preferred to others. In this case, the user should somehow select from the set of all solutions found

by a CLP program those that (s)he prefers, and this is not practical when the solution set is large.

An alternative approach is to use an overconstrained specification, including both hard constraints and soft constraints (that merely specify preferences), and have a system to compute the solutions that satisfy in the best possible way a subset of these preference constraints. This was the approach taken by [2, 16], that proposed an HCLP scheme that allows non-required (or soft) constraints to be specified with some preference level and rely on a constraint solver that explores this hierarchy of constraints to detect the best solutions.

Although the scheme is quite general, little details were published on the implementation of this scheme. In [13, 14] we presented, IHCS, an efficient and incremental defeasible constraint solver that is used as the kernel of an HCLP instance for finite domains. The key points of our implementation were a) early detection of failures through the use of the usual node- and arc-consistency techniques for these domains; b) detection of conflict sets, i.e. the sets of constraints responsible for the failures (this is done by keeping dependencies between constraints through shared variables by adaptation of the AC-5 algorithm [12]); c) selection from these conflict sets of constraints that should be relaxed, together with the selection of constraints (currently relaxed because of conflicts with the former) that can now be safely reactivated; and d) defeating the constraints, i.e. remove the effects of relaxed constraints avoiding reevaluation from scratch.

Although our scheme could in principle be applied to any other domain, early implementation of IHCS did not separate the constraint solver (responsible for the detection of failures and their causes) from the hierarchy manager (responsible for choosing, given some preference criterion, which constraints to relax and which to reactivate).

Moreover, the constraint solver detected unsatisfiable sets of constraints by means of constraint propagation on a constraint network, and the method is not applicable to domains that do not use this representation of constraints. This is the case with (linear) constraint solvers over the reals/rationals which rely on algebraic methods (e.g. some variant of the Simplex algorithm). To effectively apply our scheme to other domains it was thus necessary a) to clearly separate the constraint solver component from the hierarchy handler, and b) to enhance constraint solvers of these domains to cope with the new demands of defeasibility.

These requirements are twofold. On the one hand, the constraint solver must be able to explain the cause of unsatisfiability and/or entailment in terms of the constraints told. On the other hand, it must be able to cope with the incremental activation and deactivation of constraints as demanded by the hierarchy handler.

In this paper we propose a solution to these extensions for CLP($\mathcal{Q}$), a linear constraint solver over the body of rational numbers. Interestingly, both extensions can be realized with the single, simple idea of generalized slack variables.

The following sections will describe a general architecture for defeasible constraint solving, recapture the working of the traditional Simplex algorithm, introduce a variant through the generalization of slack variables, cover the identification of minimal conflict sets, and derive defeasibility.

2 An Architecture for Defeasible Constraint Solving

In [13, 14] an Incremental Hierarchical Constraint Solver (IHCS(X, $\preceq$)) is presented as a general framework to handle, incrementally, hierarchies of constraints in some domain X using some comparator $\preceq$. In these papers, only the instantiation of X to Finite Domains is addressed, and there is no clear division between the component that handles the constraint hierarchies, the comparators used to rank solutions, and the constraint solvers for specific domains.

This section presents the new architecture of IHCS that takes into account such separation, and makes it truly general and able to include different constraint solvers (and thus different domains). Figure 1 depicts this general architecture and the interface among the separate components.

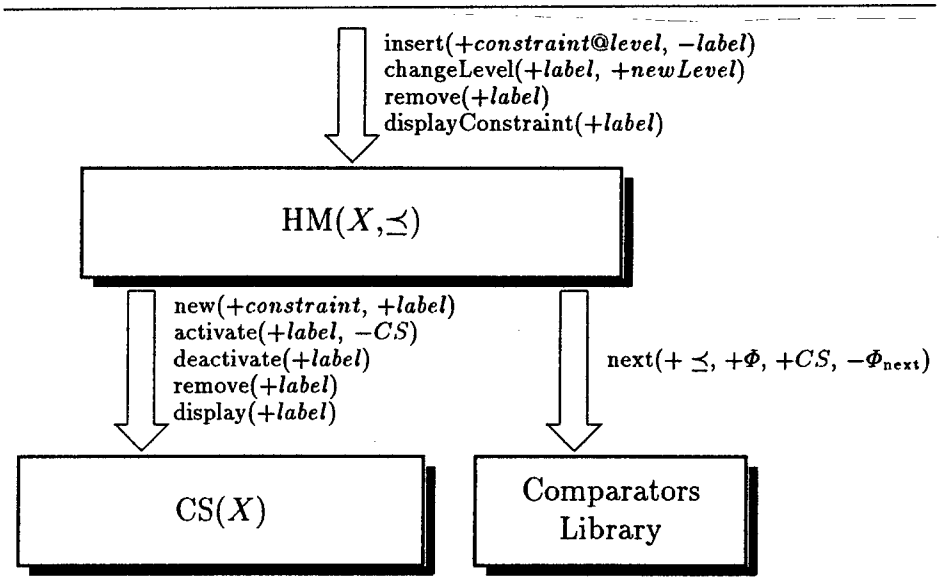


Fig. 1. The Architecture of IHCS(X,$\preceq$)

The functionality IHCS offers to any system requiring defeasible constraint handling (in our case, IHCS is embedded in a Prolog like engine to yield a HCLP(X, $\preceq$) language) is displayed at the top of the Figure. This interface highlights the defeasible nature of IHCS, by including primitives to add or remove a constraint and to promote or demote some existing constraint (by changing its strength or hierarchical level).

The Hierarchy Manager (HM) is responsible for demanding the activation and relaxation of constraints, so as to maintain the best solution. Since it has no specific knowledge of the domain theory X, it must rely on some specialized constraint solver CS(X) to check satisfiability of constraints on domain X. The

insertion of a new constraint into CS(X) is made with predicate new(*constraint*, *label*) which simply creates the constraint with a given label but does not activate it. Constraint labels are required for future reference to the corresponding constraints. This is the case of their activation, via predicate activate(*label*, CS), where CS (the *conflict set*) returns the set of constraints responsible for the possible unsatisfiability of the active constraints (CS is of course empty if these constraints are satisfiable). The reason why constraints are created and activated with different interface entries is that IHCS may require several activations or deactivations (via predicate deactivate(*label*)) of a certain constraint during the search for optimal solutions. A deactivated constraint may be removed from CS(X) with predicate remove(*label*).

A library of comparators includes a set of procedures to compute the next best configuration according to a diversity of criteria. Given the sets of constraints currently active and relaxed (the current configuration Φ), and a conflict set CS (returned by the CS(X) component), predicate next($\preceq$, Φ, CS, Φ_{next}) computes the next configuration Φ_{next} to be tried according to comparator $\preceq$.

To summarize, and to comply with IHCS requirements, a CS(X) must be:

1. **Incremental** - upon the activation of a constraint (demanded through predicate activate(*label*, CS)), the constraint solver must check the satisfiability of the active set of constraints together with the new constraint;
2. **Explanatory** - once unsatisfiability is detected, its causes should be reported to the HM (as a conflict set CS);
3. **Defeasible** - upon the deactivation of a constraint (demanded through predicate deactivate(*label*)), the effects of this formerly activated constraint should be removed avoiding reevaluation from scratch.

Of course all the above requirements impose that, the labels used by the hierarchical manager to refer to individual constraints are shared by the constraint solver.

This requirements are met by our finite domain constraint solver described in [14]. The rest of this paper explains the changes made to a Constraint Solver for linear constraints over rational/real numbers, namely its explanatory and defeasibility enhancements.

3 Simplex with Generalized Slack Variables

Classical Simplex [5] deals with a single sort of slack variables: $S_i \geq 0$. Free variables, negative variables and strict inequalities create minor problems, some of which are addressed by using pairs of slacks.

We will now generalize the concept of slack variables by allowing for arbitrary intervals as the domains of variables. This covers of course the classical Simplex slacks with a non-strict lower bound of zero.

Example 1.

Input constraints	Equations with classical slacks
$X \leq 10$	$X + S_1 = 10$
$X \leq 8$	$X + S_2 = 8$
$X \geq 2$	$-X + S_3 = -2$

Instead of introducing three slack variables with their corresponding rows in the Simplex tableaux, the bounds are represented as attributes of the affected variable directly.

Example 2.

Input constraints	Representation with generalized slacks
$X \leq 10$	
$X \leq 8$	$X_{[2,8]}$
$X \geq 2$	

The next section deals with the interaction of generalized slacks with higher dimensional constraints.

3.1 Solved Form for Inequalities over Bounded Variables

The idea proper has been realized a long time ago in the area of linear programming under the name of *bounded variable linear programs* [15]. In bounded variable linear programs, some or all variables are restricted to lie within individual lower and upper bounds. Such problems can of course be solved by including all bound restrictions as constraints, i.e. rows in the simplex tableau. The advantage of keeping them out of the tableau is that the size of the working basis is smaller. Trivial non-satisfiability, redundancy and implicit equalities are detected by trivial tests of $O(1)$ complexity. Obviously the thread matches and advances current activities in the CLP area that try to restrict the use of general decision methods to the cases where they are unavoidable [10].

We formalize bounded variable linear programs as:

$$\begin{array}{rl} & Minimize\ cx \\ subject\ to & (1.a)\ Ax = b \\ & (1.b)\ l_j \leq x_j \leq u_j \quad for\ j \epsilon J \\ & (1.c)\ x_i\ unrestricted\ for\ i \notin J \end{array} \tag{1}$$

Where $Ax = b$ denotes the subset of the constraints $\{c_i | dim(c_i) > 1\}$, and inequalities have been transformed into equations through the introduction of generalized slack variables. A feasible solution $\mathbf{x}$ of (1) is a *Basic Feasible Solution (BFS)* iff the set

$$\{A_{,j} : j \epsilon J, l_j \leq x_j \leq u_j\} \cup \{A_{,j} : j \notin J\} \tag{2}$$

where $A_{,j}$ is the j-th column vector of A, is linearly independent. A working basis for (1) is a square, nonsingular sub matrix of A of order m. Variables associated

with column vectors of the working basis will be called *basic* variables. All other variables will be called *non-basic* variables. It is clear that a feasible solution $\mathbf{x}$ is an extreme point in the solution space, iff there exists a corresponding working basis with the property that:

1. all non-basic variables are either *at* their lower or upper bound
2. the basic variables are *within* their bounds

The working basis, together with conditions 1 and 2 from above, constitutes our proposed *solved form* for linear inequalities over bounded variables.

Example 3.

Input constraints	Solved form
$x1 + x2 + 2x3 \leq 4,$ $3x2 + 4x3 \leq 6,$	$x1_{[0,2]} = 1 + \frac{1}{2}x2 + \frac{1}{2}s2 - s1$ $x3_{[0,\_]} = \frac{3}{2} - \frac{3}{4}x2 - \frac{1}{4}s2$ } *basis*
$0 \leq x1, x1 \leq 2,$ $0 \leq x2, x2 \leq 9,$ $0 \leq x3$	$s1_{[\overline{0},\_]}$ $s2_{[\overline{0},\_]}$ $x2_{[\overline{0},9]}$

Notational conventions: $x1_{[0,2]}$ means that $x1$ has a (non-strict) lower bound of zero and a (non-strict) upper bound of two. An unspecified bound is denoted as in $x3_{[0,\_]}$, where we have no finite upper bound. The active bound of non-basic variables is denoted by overlining as in $x2_{[\overline{0},9]}$. If you insert the values for the active bounds into the right hand sides (rhs) of the equations defining the basic variables $x1, x3$, you will find that the resulting values for $x1, x3$ are *within* the respective bounds. Note that only the two higher dimensional inequalities led to the introduction of slack variables $s1, s2$.

Algorithmic Details: Finding the solved form of a bounded variable linear program can be rephrased as a search problem, where we have:

1. A given initial state, consisting of a system where the solved form invariants may be violated.
2. The specification of a solution state through the solved form invariants.
3. The operators:
   (a) $pivot(x_i, x_j)$
   (b) $toggle\_active\_bound(x_i)$ for non-basic variables

The non-determinism in the selection of the the operators and their arguments can be removed by the same rules that are employed in the original Simplex algorithm:

- In order to enter the basis, the type of the non-basic variable must be compatibe with the sign of the coefficient of the variable
  in the objective function

- The leaving variable corresponds to the row in the working basis that imposes the tightest constraint on the entering variable
- The active bound of a variable may be toggled if the tightest constraint imposed on the variable through the working basis accomodates the change

A violation of the solved form is always detected by locating a basic variable that is out of its bounds. As the solved form will be computed incrementally, there will always be at most one such row and it will correspond to the $m+1$-st source constraint.

Theorem 1.

1. *The incremental solved form algorithm constitutes a decision algorithm for the satisfiability problem of a polyheral set*
2. *The incremental solved form algorithm detects implicitly fixed values*

Proof. The solved form obviously satisfies

$$\forall_{k \epsilon 1 \ldots m}\ l_k \leq inf(x_k) \leq \phi(x_k) \leq sup(x_k) \leq u_k \tag{3}$$

To establish the corresponding relation for the challenging row $i = m+1$, we interpret the linear combination of non-basic variables that defines the basic variable as artificial objective function.

$$x_{i_{[l,u]}} = \sum_{j | x_j \notin basis} k_{ij} x_j + b_i \tag{4}$$

The evaluation $\phi(x_i)$ of x_i with respect to the BFS may give rise to a repair action consisting in the iterated application of the operators *pivot* and *toggle_active_bound* in order to decrement (increment) $\phi(x_i)$ until

1. $l_i < \phi(x_i) < u_i$: solved form established. Satisfiable.
2. None of the operators is applicable, optimality, thus

$$\phi(x_i) = inf(x_i) \ \ or \ \ \phi(x_i) = sup(x_i) \tag{5}$$

   (a) $\phi(x_i) = inf(x_i) > u_i$: unsatisfiable
   (b) $\phi(x_i) = inf(x_i) = u_i \rightarrow x_i = u_i$: fixed value
   (c) $\phi(x_i) = sup(x_i) < l_i$: unsatisfiable
   (d) $\phi(x_i) = sup(x_i) = l_i \rightarrow x_i = l_i$: fixed value

□

Practical Details: The incremental solved form for bounded variable linear programs forms the basis for the implementation of the CLP($\mathcal{Q}$) and CLP($\mathcal{R}$) systems distributed with the SICStus and Eclipse Prolog. The coverage is at least as complete as that of earlier CLP($\mathcal{R}$) implementations: The system incrementally solves linear equations over rational or real valued variables, covers the lazy treatment of nonlinear equations, features a decision algorithm for linear inequalities that detects fixed values, removes redundancies, performs projections (quantifier elimination), allows for linear dis-equations, and provides for linear optimization.

It is coded in Prolog using *Attributed Variables* [8] which serve as direct access storage locations for properties associated with variables. At the same time, attributed variables make the unification part of a unification based language, Prolog in our particular case, user-definable within the language under extension [7, 9].

Empirics: In the following table we list the execution time ratio b/s between the solved form algorithm using bounded variables and a 'crippled' version which works like the original Simplex with one slack and a row for each inequality. The first two examples are from [4] computing the first and all solutions to the geometric covering problem where nine squares of unknown and different sizes are required fill an unspecified rectangle, the remaining ones are executions of a very simple minded branch and bound (BB) code on top of our solved form for some of the smaller examples from the MIPLIB mixed integer linear programming examples. Branch and bound is expected to benefit from generalized slacks because BB strengthens the original problem relaxation with simple inequalities like $x \leq \lfloor \phi(x) \rfloor$ and $x \geq \lceil \phi(x) \rceil$ when branching.

example	b/s
9 squares 1st	0.78
9 squares all	0.76
flugpl	0.48
stein15	0.68
sample2	0.70
bm23	0.73
egout	0.11
enigma	0.59
mod013	0.42
pipex	0.57
sentoy	0.77

On this collection, the solver with generalized slacks is roughly twice as fast, everything else held constant: same solver data structures, same base numeric (rationals), same machine. Basically the same ratios are obtained when computing with floating point numbers.

4 Determining Minimal Conflict Sets

In the section we will see how generalized slacks can be used to determine the reasons for the unsatisfiability of a set of constraints. We distinguish between equations, inequalities and disequalities, i.e. constraints of the form

$$\sum_{j=1}^{n} a_{ij} x_j \bowtie b_i \quad where \quad \bowtie \epsilon\{=, \neq, \leq, <, \geq, >\} \tag{6}$$

Interface: In the sequel we need the ability to refer to individual constraints by a symbolic name which we call a label. Figure 1 depicts the interface between the CLP($\mathcal{Q}$) solver kernel and the hierarchy manager part of IHCS(X,$\preceq$). The activation operation supplies the solver with the label of the constraint to be activated and the solver returns a conflict set (CS) where the label is an abstract data type not looked at by the solver, and the CS is a union of labels, possibly empty.

4.1 Failure Analysis for Equations

We solve systems of equations by Gaussian elimination. At any time, the set of variables is partitioned into basic and non-basic variables. The basic variables are expressed in terms of the non-basics. Upon the addition of a constraint it is *dereferenced* against the solved form. That is, references to the basic variables are replaced by their definitions. If the resulting expression is $0 = 0$, the constraint is entailed. The constraint is in conflict with the solved form if $0 = k, k \neq 0$. Otherwise, we solve for an arbitrary variable in the dereferenced expression and add this definition to the solved form.

We extend this scheme through the addition of a unique slack variable with bounds $[0, 0]$ to each equation. The basis for the validity of this operation is that one may substitute zero at any time for all such variables without changing the original problem statement. We call this special sort of slack variables *witness variables* after [6] where the very same trick was applied for a completely different purpose. The initial coefficients for the witness variables is immaterial, but 1 is convenient. As the solved form is manipulated, the coefficients change, and the witness variables track the dependencies between the constraints which originate from dereferencing and from pivot operations.

Example 4.

input constraint(s)	solved form
$a + b = 10$	$a = 10 - b - w_1$
$a + b = 10$	$a = 5 - \frac{1}{2}w_1 + \frac{1}{2}w_2$
$a = b$	$b = 5 - \frac{1}{2}w_1 - \frac{1}{2}w_2$

(7)

The two equations determine the values for a and b, as can be seen by substituting zero for w_i. Adding a third, incompatible constraint $a = 4$ dereferences into $-1 = -\frac{1}{2}w_1 + \frac{1}{2}w_2 + w_3$. That is $-1 = 0$ and the culprits are identified by the

witness variables $w_{1...3}$: removing any of the corresponding equations restores satisfiability.

Proposition 2. *Witness variables with nonzero coefficients in a dereferenced equation identify the original constraints which are responsible for the entailment or unsatisfiability of the equation.*

4.2 Failure Analysis for Disequalities

By the use of a unique slack variables we turn each

$$\sum_{j=1}^{n} a_{ij}x_j \neq b_i \quad into \quad \sum_{j=1}^{n} a_{ij}x_j - b_i = s_{nz} \tag{8}$$

where s_{nz} may assume any value but zero. The resulting equation is dealt with as outlined in the previous section. In the implementation, an obvious optimization is to combine the slack s_{nz} with the witness variable for the equation.

4.3 Failure Analysis for Inequalities

After the solved form algorithm fails to inc/decrease the row that violates the solved form, as described in section (3.1), the following holds:

Proposition 3. *The basic variables with non-zero coefficients in this row identify the constraints that are in conflict with the constraint the row represents itself.*

This is because the termination condition of the solved form algorithm, i.e. the non-applicability of the operators *pivot* and *toggle_active_bound* is, like in the original Simplex algorithm based on the signs, and in our case types, of the variables in the objective function. Upon termination, the value of the objective function is known *and* a corresponding set of non-basic variables is identified. A stronger result is:

Theorem 4. *Once the solved form algorithm detects unsatisfiability in (5), the constraints identified by the non-basic variables with nonzero coefficients constitute a* **minimal** *inconsistent subset of the set of constraints.*

We draw upon a result by deBacker [1], which extended Lassez's Quasi-Dual results [11] based on Fourier's theorem.

Theorem 5 Fourier 1827. *A set S of inequalities is inconsistent iff there exists a positive linear combination of inequalities which give* $0 \leq k$, *where* $k < 0$.

$$S : \left\{ \sum_{j=1}^{n} a_{ij}x_j \leq b_i \right\}_{i \epsilon 1..m} \tag{9}$$

Theorem 6 Lassez 1990. *For S as defined above its quasi dual is:*

$$Q : \left\{ \begin{array}{l} \sum_{i=1}^{m} \lambda_i a_{ij} = 0 \\ \sum_{i=1}^{m} \lambda_i = 1 \\ \forall i \epsilon 1..m, \lambda_i \geq 0 \end{array} \right\}_{j \epsilon 1..n} \tag{10}$$

- *If Q is empty then S is solvable*
- *Otherwise let $M = min \sum \lambda_i b_i$*
  - *If $M \geq 0$, S is solvable*
  - *If $M < 0$, S is unsolvable*

Theorem 7 deBacker 1991. *When S is unsolvable, we have a* witness *vertex of Q corresponding to M. A subset of S given by the indices $\lambda_i \neq 0$ is a minimal inconsistent subset of S.*

Proof of theorem 4. We exhibit the correspondence between the dual problem that arises from the repair action in the solved form algorithm for an unsatisfiable constraint and the quasi dual formulation for the whole system of constraints. The *dual* problem [15] for an optimization problem in standard form

$$\begin{array}{l} \left\{ \sum_{j=1}^{n} a_{ij} x_j \leq b_i \right\}_{i \epsilon 1..m} \\ maximize \sum_{j=1}^{n} c_j x_j \end{array} \tag{11}$$

is

$$\begin{array}{l} \left\{ \sum_{j=1}^{m} a_{ij} \lambda_j^{'} \geq c_i \right\}_{i \epsilon 1..n} \\ \forall j \epsilon 1..m, \lambda_j^{'} \geq 0 \\ minimize \sum_{j=1}^{m} b_j \lambda_j^{'} \end{array} \tag{12}$$

The quasi dual for the total system including the $m+1$-th row is

$$Q_{total} : \left\{ \begin{array}{l} \sum_{i=1}^{m+1} \lambda_i a_{ij} = 0 \\ \sum_{i=1}^{m+1} \lambda_i = 1 \\ \forall i \epsilon 1..m+1, \lambda_i \geq 0 \end{array} \right\}_{j \epsilon 1...n} \tag{13}$$

The dual and the quasi dual are related by:

$$\left\{ \underbrace{\sum_{i=1}^{m} \lambda_i^{'} a_{ij}}_{\geq c_j} + \lambda_{m+1} c_j = 0 \right\}_{j \epsilon 1...n} \tag{14}$$

Thus, except for $\sum_i \lambda_i = 1$, the dual and the quasi dual correspond. The presence of this sum in the quasi dual is just a technical trick to force a unique solution to the minimization problem in (10). Therefore, without it, and because in (12) the non-basic λ's are zero at the optimum, the λ's correspond under the scaling:

$$\lambda_{m+1}^{'} = 1 \; and \; \lambda_i^{'} = \lambda_i / \lambda_{m+1} \tag{15}$$

which yields of course the same incidence relation regarding minimal inconsistent subsets of S. □

Example 5. The first six of the following constraints are satisfiable. The addition of the seventh results in unsatisfiability.

source constraint	*label*
$x+3y+2z \geq 5,$	1
$2x+2y+z \geq 2,$	2
$4x-2y+3z \geq -1,$	3
$x \geq 0,$	4
$y \geq 0,$	5
$z \geq 0,$	6
$6x+5y+2z \leq 4$	7

(16)

Put into standard form, the quasi dual reads:

$$\begin{pmatrix} -1 & -2 & -4 & -1 & 0 & 0 & 6 \\ -3 & -2 & 2 & 0 & -1 & 0 & 5 \\ -2 & -1 & -3 & 0 & 0 & -1 & 2 \end{pmatrix} \bar{\lambda} = \bar{0}$$
$$\sum_{i=1}^{7} \lambda_i = 1, \forall i \epsilon 1..7, \lambda_i \geq 0 \qquad (17)$$
$$minimize(-5\lambda_1 - 2\lambda_2 + \lambda_3 + 4\lambda_7)$$
$$gives : \bar{\lambda} = (\tfrac{1}{9}, 0, 0, \tfrac{5}{9}, \tfrac{2}{9}, 0, \tfrac{1}{9})$$

Taking $\frac{1}{9}sc_1 + \frac{5}{9}sc_4 + \frac{2}{9}sc_5 + \frac{1}{9}sc_7$ results in $0 \leq -1$, where sc_i is the i-th source constraint.
In our solved for we have the following: after the addition of sc_7, the solved form is violated at the corresponding slack variable $s_{7[\_,0]}$, where the upper bound is 0 but $\phi(s_7) = \frac{47}{13}$:

$$\begin{aligned} (*)\ s_{7[\_,0]} &= \tfrac{47}{13} + \tfrac{75}{13}x - \tfrac{19}{13}s_1 + \tfrac{4}{13}s_3 \\ s_{2[\_,0]} &= \tfrac{-15}{13} - \tfrac{22}{13}x + \tfrac{8}{13}s_1 - \tfrac{1}{13}s_3 \\ y_{[0,\_]} &= \tfrac{17}{13} + \tfrac{5}{13}x - \tfrac{3}{13}s_1 + \tfrac{2}{13}s_3 \\ z_{[0,\_]} &= \tfrac{7}{13} - \tfrac{14}{13}x - \tfrac{2}{13}s_1 - \tfrac{3}{13}s_3 \\ s_{3[\_,\bar{0}]} & \\ x_{[\bar{0},\_]} & \\ s_{1[\_,\bar{0}]} & \end{aligned} \qquad (18)$$

The application of $pivot(y, s_3)$ reduces $\phi(s_7)$ to 1, but the solved form is still violated. None of the variables x, y, s_1 can enter the basis, thus $\phi(s_7) = 1 = inf(s_7)$.

$$\begin{aligned} (*)\ s_{7[\_,0]} &= 1 + 5x + 2y - s_1 \\ s_{2[\_,0]} &= \tfrac{-1}{2} - \tfrac{3}{2}x 1\tfrac{1}{2}y + \tfrac{1}{2}s_1 \\ s_{3[\_,0]} &= \tfrac{-17}{2} - \tfrac{5}{2}x - \tfrac{13}{2}y + \tfrac{1}{2}s_1 \\ z_{[0,\_]} &= \tfrac{5}{2} - \tfrac{1}{2}x - \tfrac{3}{2}y - \tfrac{1}{2}s_1 \\ y_{[\bar{0},\_]} & \\ x_{[\bar{0},\_]} & \\ s_{1[\_,\bar{0}]} & \end{aligned} \qquad (19)$$

Reading off the coefficients from (*), we get: $\lambda' = (-1, 0, 0, 5, 2, 0, 1)$ Note that the first component of this vector is negative because we assumed normalized

inequalities, i.e. every inequality expressed as $\sum k_{ij}x_i \leq b$, in the proof only. Application gives:

$$\begin{array}{rcl} -x-3y-2z \leq -5 & & -x-3y-2z \leq -5 \\ 5x \geq 0 & \overset{normalize}{\longrightarrow} & -5x \leq 0 \\ 2y \geq 0 & & -2y \leq 0 \\ 6x+5y+2z \leq 4 & & 6x+5y+2z \leq 4 \end{array} \rightarrow 0 \leq -1 \qquad (20)$$

5 Deactivation of Constraints

Again we organize this section after the classes of constraints we deal with. The basis for the correctness of the operations performed is the invertability of linear transformations. Space not permitting for more details, we only mention a) that entailed constraints have to be reactivated if no longer entailed because of the deactivation of an entailing constraint, and b) that deactivation has of course to deal with the transitive closure of consequences of fixed value detection/propagation.

5.1 Deactivating Equations

An equation is deactivated by solving for the corresponding witness variable. Between the introduction of the witness variable and the time we are about to relax the equation, the solved form was changed by linear transformations only, which is the guarantee that we can solve for the witness variable. Solving for a variable removes it from all the right hand sides of all basic variables where it occurs, and then we may abandon the row for the witness variable. Consider our example again:

Example 6.

input constraint(s)	solved form
$a+b=10$	$a = 10-b-w_1$
$a+b=10$	$a = 5-\frac{1}{2}w_1+\frac{1}{2}w_2$
$a=b$	$b = 5-\frac{1}{2}w_1-\frac{1}{2}w_2$

(21)

To deactivate the second equation $a = b$, we solve for $w_2 = -10 + 2a + w_1$, substitute and drop the row for w_2:

Example 7.

input constraint(s)	solved form
$a+b=10$	$b = 10-a-w_1$

(22)

Which is equivalent to the solved form for $a + b = 10$. If we deactivate the first equation instead, we have $w_1 = 10 - 2a + w_2$ and:

input constraint(s)	solved form
$a=b$	$b = a-w_2$

(23)

5.2 Deactivating Disequations

Recall that disequations are turned into equations via slacks. Deactivating a disequation is by solving for the witness variable for the corresponding equation, and dropping the row afterwards.

5.3 Deactivating Inequations

An inequation is deactivated by bringing the associated slack variable into the basis and by removing the corresponding bound. A variable is brought into the basis by pivoting it with the most constraining row in the basis. If there is no constraining row for a non-basic variable, we may simply drop the bound to be deactivated.

6 Conclusion

It turned out to be remarkable simple to meet the explanatory and defeasibility requirements of the IHCS scenario for the instantiation of the constraint solver component to CLP($\mathcal{Q}$). One concept, generalized slacks, provides both mechanisms. With regard to computational complexity, explanations are for free if we have to deal with inequalities free of fixed values only. If there are (implicitly) fixed values and/or additional equations, the extra cost for carrying along the witness variables is rewarded by the possibilities a) to deactivate the constraints later, and b) although not elaborated on here, to have backtracking without trailing in the constraint solver [6]. Our work shares objectives with [3]. Our improvement is in the addition of defeasibility to the thread and that we don't need an explicit inverse of the basis for CS computations.

With respect to applications we naturally envision classical dependency directed backtracking applications, but expect more rewarding results from the extra expressiveness and flexibility through the IHCS architecture.

Acknowledgments

This joint work was made possible by project SOL from the Human Capital and Mobility Programme of the European Union. Financial support for the Austrian Research Institute for Artificial Intelligence is provided by the Austrian Federal Ministry for Science, Research, and the Arts. The work at Faculdade de Ciências e Técnologia of the Universidade Nova de Lisboa was supported by Junta Nacional de Investigação Científica e Tecnológica (grant PBIC/C/TIT/1242/92).

References

1. Backer B.de, Beringer H.: Intelligent Backtracking for CLP Languages: An Application to CLP(R), in Saraswat V. & Ueda K.(eds.), *Symposion on Logic Programming*, MIT Press, Cambridge, MA, pp.405-419, 1991.

2. A. Borning, M. Maher, A. Martingale, and M. Wilson. Constraints hierarchies and logic programming. In Levi and Martelli, editors, *Logic Programming: Proceedings of the 6th International Conference*, pages 149–164, Lisbon, Portugal, June 1989. The MIT Press.
3. Burg J., Lang S.-D., Hughes C.E.: Finding Conflict Sets and Backtrack Points in CLP(R), in Hentenryck P.van(ed.), *Proceedings of the Eleventh International Conference on Logic Programming (ICLP94)*, MIT Press, Cambridge, MA, 1994.
4. Colmerauer A.: An Introduction to Prolog III, *Communications of the ACM*, 33(7), 69-90, 1990.
5. Dantzig G.B.: *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.
6. Hentenryck P.van, Ranachandran V.: *Backtracking without Trailing in CLP(R)*, Dept.of Computer Science, Brown University, CS-93-51, 1993.
7. Holzbaur C.: *Specification of Constraint Based Inference Mechanisms through Extended Unification*, Department of Medical Cybernetics and Artificial Intelligence, University of Vienna, Dissertation, 1990.
8. Holzbaur C.: Metastructures vs. Attributed Variables in the Context of Extensible Unification, in Bruynooghe M. & Wirsing M.(eds.), *Programming Language Implementation and Logic Programming*, Springer, LNCS 631, pp.260-268, 1992.
9. Holzbaur C.: Extensible Unification as Basis for the Implementation of CLP Languages, in Baader F., et al., *Proceedings of the Sixth International Workshop on Unification*, Boston University, MA, TR-93-004, pp.56-60, 1993.
10. Imbert J.-L., Cohen J., Weeger M.-D.: An Algorithm for Linear Constraint Solving: Its Incorporation in a Prolog Meta-Interpreter for CLP, in *Special Issue: Constraint Logic Programming, Journal of Logic Programming*, 16(3&4), 235-253, 1993.
11. Lassez J.L.: Parametric Queries, Linear Constraints and Variable Elimination, *in Proceedings of the Conference on Design and Implementation of Symbolic Computation Systems*, Capri, pp.164-173, 1990.
12. Menezes F., Barahona P.: Preliminary Formalization of an Incremental Hierarchical Constraint Solver. In L. Damas L and M. Filgueiras (eds.), In *Proceedings of EPIA '93*, Springer-Verlag, Porto, October 1993.
13. Menezes F., Barahona P.: An Incremental Hierarchical Constraint Solver. In V. Saraswat and P. Van Hentenryck, editors, *Principles and Practice of Constraint Programming*, MIT Press, 1995.
14. Menezes F., Barahona P.: Defeasible Constraint Solving. In *Proceedings of Iberamia 94*, McGraw-Hill Interamericana de Venezuela, Caracas, October 1994.
15. Murty K.G.: *Linear and Combinatorial Programming*, Wiley, New York, 1976.
16. Wilson M., Borning A.: Hierarchical Constraint Logic Programming. *J. Logic Programming*, 1993:16.

Inference Duality as a Basis for Sensitivity Analysis *

J. N. Hooker

Graduate School of Industrial Administration, Carnegie Mellon University,
Pittsburgh, PA 15213 USA,

Abstract. The constraint programming community has recently begun to address certain types of optimization problems. These problems tend to be discrete or to have discrete elements. Although sensitivity analysis is well developed for continuous problems, progress in this area for discrete problems has been limited. This paper proposes a general approach to sensitivity analysis that applies to both continuous and discrete problems. In the continuous case, particularly in linear programming, sensitivity analysis can be obtained by solving a dual problem. One way to broaden this result is to generalize the classical idea of a dual to that of an "inference dual," which can be defined for any optimization problem. To solve the inference dual is to obtain a proof of the optimal value of the problem. Sensitivity analysis can be interpreted as an analysis of the role of each constraint in this proof. This paper shows that traditional sensitivity analysis for linear programming is a special case of this approach. It also illustrates how the approach can work out in a discrete problem by applying it to 0-1 linear programming (linear pseudo-boolean optimization).

1 Introduction

Sensitivity analysis addresses the issue of how much the solution of an optimization problem responds to perturbations in the problem data. It is an indispensable element of applied modeling, perhaps as important as obtaining the solution itself. It is needed not only to anticipate the effect of changes in the problem, but to deal with the fact that in applied work, obtaining the information necessary to formulate an accurate model is often the hardest part of the task. Sensitivity analysis typically reveals that the solution depends primarily on a few key data, whereas the rest of the problem can be altered somewhat without appreciable effect. This allows one to focus time and resources on collecting and verifying the information that really matters. More generally, it directs the decision maker's attention to those aspects of the problem that should be closely watched.

By far the most widely used optimization tool is linear programming, for which sensitivity analysis is highly developed. One of the basic results is that the solution of the linear programming *dual* indicates the sensitivity of the optimal

* Partially supported by U.S. Office of Naval Research grant N00014-95-1-0517.

value to perturbations in the right-hand sides of the inequality constraints. These results have been extended to certain discrete optimization problems. There are duality theories for integer programming, for instance, that can serve as a basis for sensitivity analysis; a brief survey may be found in Section 23.7 of [13]. Integer dual solutions become very complex as the problem grows, however, and are rarely used in practice. These and related approaches (e.g., [12, 14]) are based on an investigation of how the optimal value depends on the right-hand sides of inequality and equality constraints, and it is unclear how the ideas would generalize to problems with other types of constraints.

The approach taken here is to define the *inference dual* of an optimization problem and use it as the basis for sensitivity analysis. The inference dual is the problem of inferring from the constraints a best possible bound on the optimal value. The solution of the dual is a proof, using an inference method that is appropriate to the problem. Sensitivity analysis can be viewed as an analysis of the role of each constraint in this proof. For example, a constraint may not even appear as a premise in the proof, in which case it is redundant, or if it does, the proof may yet go through if the constraint is weakened by a determinable amount. This type of analysis can in principle be carried out for any type of constraint.

There may be several proofs of an optimal bound, and if so sensitivity analysis differs somewhat in each. This phenomenon is known as "degeneracy" in classical mathematical programming, where it tends to be regarded as a technical nuisance. Here it is seen to be a natural outcome of the fact that more than one rationale can be given for an optimal solution.

To solve the inference dual, one must

a) identify inference rules that are complete for the type of constraints in the problem;
b) use the rules to prove optimality.

In the linear programming dual, for example, one infers inequalities from other inequalities. The inference rule is simple: all inequalities implied by a constraint set can be obtained by taking nonnegative linear combinations of the constraints. This is essentially the content of the classical "separation lemma" for linear programming, which is therefore a completeness theorem for linear inference. To find the particular linear combination that solves the dual, one can use information obtained in solving the original problem (the "primal"). The multipliers in this combination indicate the sensitivity of the optimal value to perturbations in the right-hand sides of the corresponding constraints.

The question here is how to address points (a) and (b) for discrete optimization problems. One answer lies in the idea of deriving a dual solution from information gathered while solving the primal problem. A discrete problem can be solved enumeratively by building a search tree that branches on values of the variables. The structure of this tree reflects the structure of a proof of optimality, in the following way. First, a constraint is added to require the objective function value to be less than the true minimum, so that the tree now establishes

infeasibility of the augmented constraint set. At each leaf node of the tree, a certain type of proposition (a "multivalent clause") that is violated at that node is inferred from one of the constraints. The tree now indicates the structure of a resolution proof that the multivalent clauses are unsatisfiable, using a "multivalent" resolution method that generalizes classical resolution. The inference rules required in (a) are therefore those of multivalent resolution, plus those needed to infer multivalent clauses from constraints. The proof required in (b) is given by the structure of the search tree. Sensitivity analysis consists generally in noting the role played by each constraint in the proof, and specifically in checking how much the constraints can be changed so that they still imply the multivalent clauses that are used as premises in the proof.

Inference duality also permits a generalization of Benders decomposition to any optimization problem. This technique allows one to generate "Benders cuts" that are analogous to nogoods but that exploit problem structure in a way that nogoods do not. This idea is developed in [7]. Other connections between logical inference and optimization are surveyed in [4, 6].

The paper begins with an elementary treatment of the inference dual and its role in linear programming sensitivity analysis. It then describes multivalent resolution and shows how an optimality proof of this kind can be recovered from an enumeration tree that solves the primal problem. It concludes with an application to linear 0-1 programming and some practical observations.

2 The Inference Dual

Consider a general optimization problem,

$$\begin{aligned} &\min f(x) \\ &\text{s.t. } x \in S \\ &\quad\ x \in D. \end{aligned} \tag{1}$$

The *domain* D is distinguished from the *feasible set* S. In most applications x is a vector $(x_1, \ldots, x_n)$, in which case D_{x_j} denotes the domain of x_j.

To state the inference dual it is necessary to define the notion of implication with respect to a domain D. Let P and Q be two propositions about x; that is, their truth or falsehood is determined by the value of x. *P implies Q with respect to D* (notated $P \xrightarrow{D} Q$) if Q is true for any $x \in D$ for which P is true.

The *inference dual* of (1) is

$$\begin{aligned} &\max z \\ &\text{s.t. } x \in S \xrightarrow{D} f(x) \geq z \end{aligned} \tag{2}$$

So the dual seeks the largest z for which $f(x) \geq z$ can be inferred from the constraint set. A strong duality theorem is true almost by definition. To state it, it is convenient to say that the optimal value of a minimization problem is respectively ∞ or $-\infty$ when the problem is infeasible or unbounded, and vice-versa for a maximization problem.

Theorem 1 Strong Inference Duality. *The optimization problem (1) has the same optimal value as its inference dual (2).*

Proof. If z^* is the optimal value of (1), then clearly $x \in S$ implies $f(x) \geq z^*$, which shows that the optimal value of the dual is at least z^*. The dual cannot have an optimal value larger than z^*, because this would mean that $f(x) = z^*$ cannot be achieved in (1) for any feasible x. If (1) is infeasible, then any z is feasible in (2), which therefore has optimal value ∞. If (1) is unbounded, then (2) is infeasible with optimal value $-\infty$. □

Because strong duality is a trivial affair for inference duality, interesting duality theorems must deal with some other aspect. A natural task for a duality theorem is to provide a complete method for deriving inferences in the dual, as explained earlier.

A dual solution provides sensitivity analysis because *it specifies the role played by each constraint in a deduction of the optimal value.* This is illustrated below in the cases of linear and 0-1 programming.

3 Linear Programming Duality and Sensitivity Analysis

A linear programming problem

$$\begin{aligned} &\min cx \\ &\text{s.t. } Ax \geq a \\ &\quad x \geq 0, \end{aligned} \tag{3}$$

where matrix A is $m \times n$, has the following inference dual.

$$\begin{aligned} &\max z \\ &\text{s.t. } (Ax \geq a, x \geq 0) \xrightarrow{R^n} cx \geq z. \end{aligned} \tag{4}$$

The dual looks for a *linear implication* $cx \geq z$ of the constraints that maximizes z. Linear implication is characterized by the following, which is equivalent to a classical separation lemma for linear programming.

Theorem 2 Linear implication.. *$Ax \geq a$ linearly implies $cx \geq z$ (i.e., $Ax \geq a \xrightarrow{R^n} cx \geq z$) if and only if $Ax \geq a$ is infeasible or there is a real vector $u \geq 0$ for which $uA \leq c$ and $ua \geq z$.*

This means that the dual (4) seeks a nonnegative linear combination $uAx \geq ua$ of $Ax \geq a$ that dominates $cx \geq z$ (i.e., $uA \leq c$ and $ua \geq z$) and that maximizes z. So the dual can be written in the classical way,

$$\begin{aligned} &\max ua \\ &\text{s.t. } uA \leq c \\ &\quad u \geq 0 \end{aligned} \tag{5}$$

The vector $u \in R^m$ in effect encodes a proof, because it gives instructions for deducing the optimal value z. The duality theorem for linear programming follows immediately from Theorem 2.

Corollary 3. *The optimal value of a linear programming problem (3) is the same as that of its classical dual (5), except when both are infeasible.*

Note that unlike inference duality, the classical theorem requires a regularity condition to the effect that either a problem or its dual must be feasible.

The optimal dual solution u^* provides sensitivity analysis because *it indicates the role of each constraint in a proof of optimality.* For instance, if $u_i^* = 0$, then constraint i has no role in the proof, and the constraint can be omitted without invalidating the proof and therefore without changing the optimal value.

More generally, one can reason as follows. If the vector a of right-hand sides is changed to $a + \Delta a$, then u^* remains a feasible solution of the resulting dual problem (5) with value $u^*(a + \Delta a)$. So the optimal dual value is at least $u^*(a + \Delta a)$, and by Corollary 3 the same is true of the optimal solution of the primal problem.

This means that if constraint i is strengthened by raising its right-hand side a_i by $\Delta a_i \geq 0$, the optimal value will rise at least $u_i^* \Delta a_i$. (In particular, it will rise to ∞ if the problem becomes infeasible.) If a_i is reduced by Δa_i, the optimal value can fall no more than $u_i^* \Delta a_i$. The increase or decrease is exactly $u_i^* \Delta a_i$ if Δa_i lies within easily computable bounds.

The dual problem can have several optimal extreme point solutions (i.e., optimal solutions that are not convex combinations of each other). This can occur when the primal solution is "degenerate." Each solution gives rise to a different sensitivity analysis.

A more complete exposition of linear programming sensitivity analysis may be found in [2].

4 A Multivalent Resolution Method

It is well known that the resolution method originally developed by Quine [9, 10] (and later extended to first order logic by Robinson [11]) provides a complete refutation method for propositional logic in conjunctive normal form.[2]

The resolution method is readily generalized to problems in which the variable domains contain more than two discrete values. The method that results is related to Cooper's algorithm for achieving k-consistency [3], but it is convenient here to cast it as an inference method. Consider a set S of *multivalent clauses* of the form,

$$(x_1 \in T_{i1}) \vee \ldots \vee (x_n \in T_{in}),$$

where each $x_j \in T_{ij}$ is a *literal*, and each $T_{ij} \subset D_{x_j}$. If $T_{ij} = \emptyset$, then the literal $x_j \in T_{ij}$ can be omitted from the disjunction, and it is convenient to say that the clause does not contain x_j. The *resolvent on* x_j of the clauses in S is

$$(x_j \in \bigcap_i T_{ij}) \vee \bigvee_{k \neq j} (x_k \in \bigcup_i T_{ik}).$$

[2] Quine's method, sometimes called consensus, was actually for formulas in disjunctive normal form, but it is easily dualized to treat conjunctive normal form.

For example, the first three clauses below resolve on x_1 to produce the fourth. Here each x_j has domain $\{1,2,3,4\}$.

$$(x_1 \in \{1,4\}) \vee (x_2 \in \{1\})$$
$$(x_1 \in \{2,4\}) \vee (x_2 \in \{2,3\})$$
$$(x_1 \in \{3,4\}) \vee (x_2 \in \{1\})$$
$$(x_1 \in \{4\}) \vee (x_2 \in \{1,2,3\})$$

To check a multivalent clause set S for satisfiability, identify a subset of clauses whose resolvent does not already belong to S. If there is no such subset, stop and conclude that S is satisfiable. If the resolvent is the empty clause (i.e, each $T_j = \emptyset$), stop and conclude that S is unsatisfiable. Otherwise add the resolvent to S and repeat. The proof that multivalent resolution is a sound and complete refutation method is parallel to that for ordinary resolution. A weaker result (Theorem 4, below) will suffice for present purposes.

5 Solving the Dual via the Primal

It is possible in general to characterize a *primal method* for solving a problem as one that examines possible values of the variables, and a *dual method* a one that examines possible proofs of optimality without interpreting the variables. In classical optimization, a branch-and-bound method is a primal method, whereas a pure cutting plane method is essentially a dual method (see [8] for background). Primal methods have generally proved more effective for optimization, although primal and dual methods are often combined, as in branch-and-cut and dual ascent algorithms.

Fortunately, the inference dual of a discrete optimization problem can be solved by examining the results of a primal method, in particular an enumeration tree that is generated to solve the primal problem. In fact, this approach yields both a complete refutation method for the type of inference used in the dual and an algorithm for constructing a proof of optimality. It will be seen that a refutation method, as opposed to a full inference method, suffices for sensitivity analysis.

The most straightforward way to solve (1) by enumeration is to branch on values of the variables until all feasible solutions have been found. The search backtracks whenever a feasible solution is found or some constraint is violated. The best feasible solution is optimal. A generic algorithm for generating the search tree appears in Fig. 1. The development below is readily modified to accommodate search algorithms that use bounding and other devices for pruning the tree.

Let z^* be the optimal value found by enumeration. To solve the dual, first modify the enumeration tree so that it refutes the claim that $f(x) < z^*$. This is done simply by adding the constraint $f(x) < z_t$ to the constraint set $\mathcal{C}$ for each node t at which a feasible solution is found, where z_t is the value of that solution. Then $f(x) < z_t$ can be regarded as the constraint violated at node t. There is now at least one violated constraint at every leaf node.

```
Let C be the set of constraints that define the feasible set S in (1).
Let L be a list of active nodes, initially containing the root node,
   which is associated with an empty set of assignments (labels).
Let z̄ be an upper bound on the optimal value; initially z̄ = ∞.
While L is nonempty:
   Remove a node from L, and let A be the set of assignments
      associated with the node.
   If the assignments in A violate no constraint in C then
      If A assigns values v1,...,vn to every variable x1,...,xn so as
      to satisfy every contraint in C (or the assignments in A
      can be readily extended to all variables so as to satisfy
      every constraint) then
         Let z̄ = min{z̄, f(v1,...,vn)}.
      Else
         Choose a variable xj that A does not assign a value.
         For each v ∈ Dxj:
            Add a node to L and associate it with A ∪ {xj = v}.
If z̄ < ∞ then z̄ is the optimal value of (1).
Else (1) is infeasible.
```

Fig. 1. A generic enumeration algorithm for solving the primal problem.

For any leaf node t let $(x_{j_1}, \ldots, x_{j_d}) = (v_1, \ldots, v_d)$ be the assignments made along the path from node t to the root. As just noted, these assignments violate some $C_t \in \mathcal{C}$. Because C_t is equivalent to the conjunction of all multivalent clauses it implies, $(x_{j_1}, \ldots, x_{j_d}) = (v_1, \ldots, v_d)$ violates some multivalent clause M_t implied by C_t. Without loss of generality M_t can be assumed to contain only variables in $\{x_{j_1}, \ldots, x_{j_d}\}$. The enumeration tree is now a refutation of $\bigwedge_t M_t$. Due to the following theorem, the tree's structure indicates how to construct a resolution proof of $\neg \bigwedge_t M_t$.

Theorem 4. *Consider an enumeration tree in which a multivalent clause M_t is associated with each node t. Let the clause associated with any nonleaf node be the resolvent on x_j of those associated with the node's children, where x_j is the variable on which the tree branches at node i. Then if M_t is falsified at each leaf node t, the clause associated with the root node is empty.*

Proof. It suffices to show that the clause associated with the root node is falsified, because only the empty clause is falsified when no variables are fixed. This can be proved by induction. It is given that M_t is falsified at any leaf node t. Now consider any nonleaf node k, and let nodes $1, \ldots, p$ be its children. By the induction hypothesis, $M_1, \ldots, M_p$ are falsified. Suppose that the search branches on variable x_j at node k and that $x_j = v_s$ is the assignment that generates child node s. Then M_s cannot contain a literal $x_j \in T_{sj}$ with $v_s \in T_{sj}$. If it did, M_s would be true at node s. Therefore $\bigcap_{s=1}^{p} T_{sj} = \emptyset$, which means that the resolvent M_k of $\{M_1, \ldots, M_p\}$ does not contain x_j. So M_k is falsified at node k. It follows by induction that the root clause is falsified. □

A refutation of $f(x) < z^*$ can therefore be constructed from two ingredients: multivalent resolution, and an algorithm for checking whether a constraint implies a multivalent clause that is violated by a given variable assignment $(x_{j_1}, \ldots, x_{j_m}) = (v_1, \ldots v_m)$. The construction proceeds as follows. For each leaf node t of the enumeration tree, select any constraint C_t violated at that node; if the node represents a feasible solution with value z_t, let C_t be $f(x) < z_t$. For each t identify a multivalent clause M_t that a) is implied by C_t, b) contains only variables among those that are fixed along the path from node t to the root, and c) is violated at node t. Then the desired refutation infers each M_t from C_t and then refutes $\bigwedge_t M_t$ using multivalent resolution, as indicated in Theorem 4.

Sensitivity analysis now consists of observing the role of the constraints in the refutation just constructed. Some specific results can be obtained by applying the three principles below. Let a node be *feasible* if no constraints in $\mathcal{C}$ are violated at the node, and infeasible otherwise. The principles do not require that the resolution refutation actually be constructed; only that the clauses M_t at infeasible nodes and the values z_t at feasible nodes be saved. Because the search tree may have a very large number of leaf nodes, it is practical to save only the *distinct* M_t's associated with a given constraint. So for any constraint $C \in \mathcal{C}$ let $\{M_t \mid t \in M(C)\}$ be the set of distinct clauses M_t for which $C = C_t$. The number of feasible nodes is likely to be small, however, and it is practical to keep a list of z_t for each feasible node t as well as which variables are fixed (and to what values they are fixed) at t.

(S1) If a constraint in $\mathcal{C}$ is not associated with any leaf nodes, then it is redundant and can be dropped without affecting the optimal value.

(S2) More generally, if a constraint $C \in \mathcal{C}$ is replaced with a constraint C' that still implies M_t for all $t \in M(C)$, then:

- **i)** the optimal value will not decrease;
- **ii)** the optimal value will be at least $\min_{t \in I}\{z_t\}$, where I is the set of feasible nodes at which C' is not violated.

(S3) Still more generally, if a constraint $C \in \mathcal{C}$ is replaced with C', then the optimal value will be at least

$$\min\left\{\min_{t \in I} z_t, \min_{t \in I'} \underline{z}_t\right\},$$

where I is as above, and I' is the set of nodes $t \in M(C)$ at which C' does not imply M_t. Also $\underline{z}_t$ is the minimum value of the objective function $f(x)$ subject only to $\neg M_t$. That is,

$$\underline{z}_t = \min\{f(x) \mid x_j \in D_{x_j} \setminus T_{tj} \text{ for all } x_j \text{ in } M_t\}.$$

These principles can clearly be adapted to analyze the effect of changing two or more constraints simultaneously.

6 0-1 Duality and Sensitivity Analysis

A 0-1 linear programming problem may be stated,

$$\begin{aligned} &\min\ cx \\ &\text{s.t. } Ax \geq a \\ &\quad x \in \{0,1\}^n. \end{aligned} \tag{6}$$

The inference dual is,

$$\begin{aligned} &\max\ z \\ &\text{s.t. } Ax \geq a \overset{\{0,1\}^n}{\longrightarrow} cx \geq z. \end{aligned} \tag{7}$$

A separation lemma for this dual would consist of a complete inference method for linear 0-1 inequalities. Such a method was presented in [5]. Only a complete refutation method is required for present purposes, however, and it can obtained as indicated in the previous section.

A refutation method is obtained by combining multivalent resolution with a method for checking whether a 0-1 inequality implies a multivalent clause that is falsified by a given variable assignment. Because the variables in this case are bivalent, multivalent clauses become ordinary clauses. Multivalent resolution therefore reduces to ordinary resolution. It is straightforward to check whether $ax \geq \alpha$ implies a clause that is falsified by $(x_{j_1}, \ldots, x_{j_d}) = (v_{j_1}, \ldots, v_{j_d})$. Let J be the set of indices j in $\{j_1, \ldots, j_d\}$ for which $a_j > 0$ if $v_j = 1$ and $a_j < 0$ if $v_j = 0$. Let $x_j(v)$ be x_j if $v = 1$ and $\neg x_j$ if $v = 0$. Then $\bigvee_{j \in J} x_j(1 - v_j)$ is the desired clause if $\sum_{j \in J} a_j v_j + \sum_{j \notin J} \max\{0, a_j\} < \alpha$, and otherwise there is no such clause.

Consider for example the problem,

$$\begin{aligned} \min\ & 7x_1 + 5x_2 + 3x_3 \\ \text{s.t. }\ & 2x_1 + 5x_2 - x_3 \geq 3\ (a) \\ & -x_1 + x_2 + 4x_3 \geq 4\ (b) \\ & x_1 + x_2 + x_3 \geq 2\ (c) \\ & x \in \{0,1\}^3 \end{aligned} \tag{8}$$

The enumeration tree of Fig. 2 solves the problem. The optimal solution value is 8.

A resolution proof that $z \geq 8$ can be reconstructed in a way very similar to that presented above for satisfiability problems. At each leaf node, one of the violated inequalities is chosen as a premise; in this case, constraints (a) and (b) suffice. If the leaf node represents a feasible solution, the premise consists of the constraint $cx \leq z - 1$. These combined premises must lead to a contradiction, thus proving that $z \geq 8$.

The contradiction can be demonstrated by resolution, as depicted in Fig. 3. The inequalities at nodes 8 and 9, for instance, respectively imply clauses that resolve to obtain $\neg x_1 \vee \neg x_2$ at node 4. The latter resolves with x_2, implied by the inequality at node 5, to obtain $\neg x_1$ at node 2.

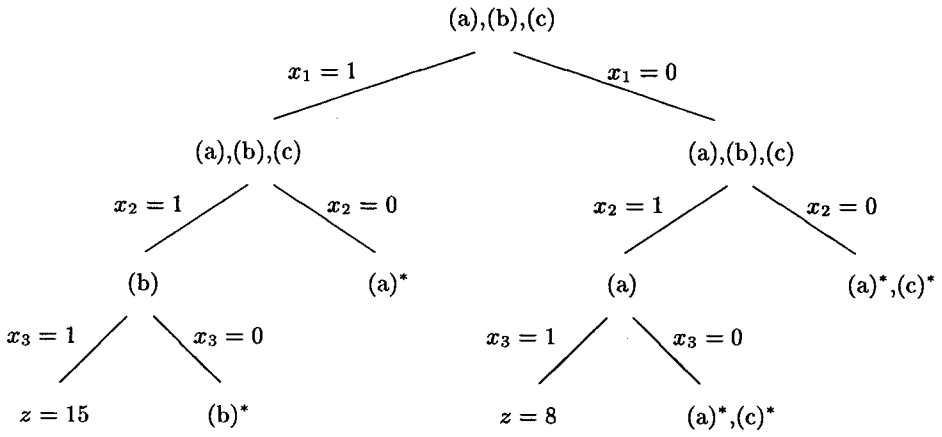


Fig. 2. *A solution of the subproblem by branching. The constraints remaining (i.e., not yet satisfied) at each nonleaf node are indicated. The constraints violated at each leaf node, if any, are indicated with an asterisk; if no constraints are violated, the objective function value z is shown.*

The inequality at node 10 implies a second clause $\neg x_1 \vee \neg x_3$ (not shown in Fig. 3) that resolves with x_3 at node 11. But this clause can be neglected because it is not falsified by variables fixed between nodes 10 and the root.

Because the clauses inferred at the leaf nodes are falsified by the fixed variables, the enumeration tree proves they are unsatisfiable. So there is a resolution proof of unsatisfiability. In this case, the empty clause is generated below the root, at node 3.

The three principles (S1)-(S3) cited earlier for carrying out sensitivity analysis are readily applied to this 0-1 programming example.

(S1) Because constraint (c) is associated with no leaf node in Fig. 3, it is redundant and can be dropped without changing the optimal solution.

(S2) Constraint (a) is associated with leaf nodes 5 and 7. Both M_5 and M_7 are the singleton clause x_2, and so one can set $M(C) = \{5\}$.

i) Constraint (a) can be altered in any fashion such that it continues to imply $x_2 = 1$, without reducing the optimal value. For instance, the right-hand side can be reduced to any number greater than 2 and increased arbitrarily. The coefficient of x_2 can changed arbitrarily (if the inequality becomes infeasible, it implies x_2 as well as any other clause). The coefficient of x_1 can be increased to any number less than 3 and reduced arbitrarily, and so forth. Constraint (b) can be similarly analyzed. It is not hard to write general procedures for this.

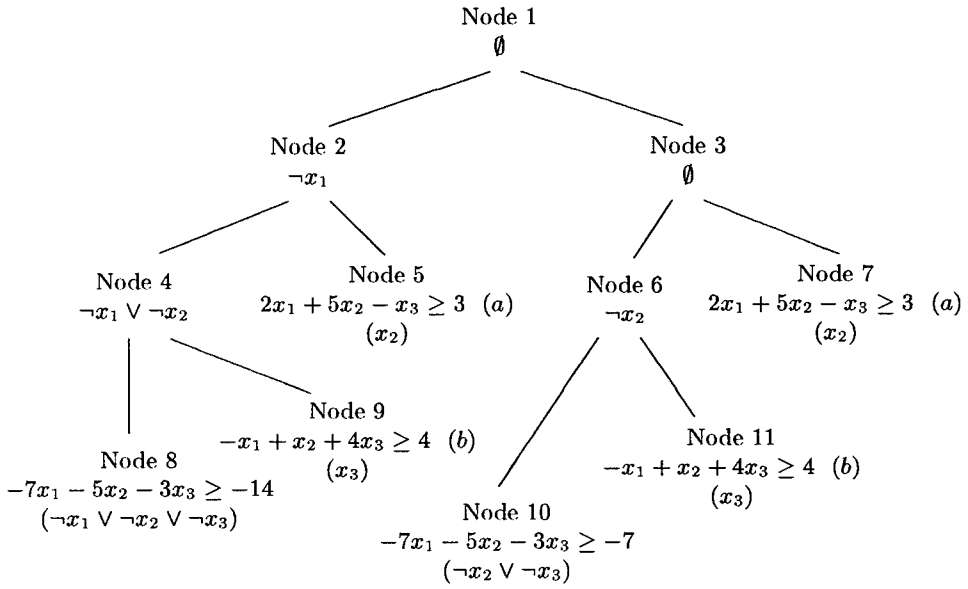


Fig. 3. *Construction of a proof of $z \geq 8$. The violated constraint at each leaf node is shown, along with a falsified clause it implies. Resolvents are shown at the nonleaf nodes.*

ii) Suppose the right-hand side of constraint (a) is raised to 5. Of the two feasible nodes 8 and 10, (a) is now violated at 10. The new optimal value is therefore at least $z_8 = 15$.

(S3) Suppose constraint (a) is weakened by changing it to $2x_1 + 5x_2 - x_3 \geq 2$, so that it no longer implies x_2 ($I' = \{5\}$). Constraint (a) of course remains unviolated at the feasible nodes 8 and 10 ($I = \{8, 10\}$). So the optimal value is at least

$$\min\{\min\{z_8, z_{10}\}, \underline{z}_5\} = 0,$$

where $\underline{z}_5 = \min\{7x_1 + 5x_2 + 3x_3 \mid \neg x_2\} = 0$. In this case no useful bound is obtained. Constraint (b) can be similarly analyzed.

7 Practical Considerations

The style of sensitivity analysis proposed here must be adapted to the problem context. The interests of practitioners should guide which questions are asked, and these questions should guide the type of perturbations that are studied. Even in classical linear programming, sensitivity analysis can provide information that is too complex and voluminous to be assimilated. One must take care to highlight

the results that are intelligible and relevant. At this point it is impossible to predict the problem classes in which inference duality can yield useful sensitivity results. Only practical trials can resolve this issue.

One possible impediment to the interpretation of sensitivity analysis is "massive degeneracy." There may be a large number of dual solutions, each giving rise to a different sensitivity analysis. Fortunately, a single dual solution can show a constraint to be *unimportant*. For instance, if a constraint is redundant in one dual solution, then it is redundant *simpliciter*, in the sense that it can be dropped without changing the solution. Or if one dual solution yields an upper bound on the effect of a problem alteration, this bound is valid in general. But to establish categorically that a constraint is *important*, one must show that it is important in all dual solutions.

The effects of degeneracy can be ameliorated somewhat. Degeneracy has two sources: many different search trees arrive at the same optimal value, and a given search tree can be analyzed in many different ways. The first source must be accepted, because it is usually impractical to solve a problem more than once. However, if the role of a few particular constraints are of interest, the search tree can be analyzed with this in mind. These constraints should be avoided, whenever possible, when associating violated constraints with leaf nodes. More generally, the multivalent clauses derived from violated constraints should be as weak as possible. Ultimately, however, degeneracy must be viewed as inherent in the nature of things rather than an artifact of the analysis. It is often an unavoidable fact that different rationales can be given for an optimal solution, each drawing on different information in the constraint set.

Although the results presented here apply only to problems that are solved by tree search, it is rare that a (verified) optimal solution is found by other means in discrete problems. As noted earlier, the analysis can be modified to accommodate tree searches that use such devices as bounding and relaxations to prune the tree.

It is also common for applications to require both discrete and continuous variables. It is an interesting research issue as to how the classical methods for continuous sensitivity analysis may be combined with the discrete methods presented here.

References

1. Barth, P., *Logic-Based 0-1 Constraint Solving in Constraint Logic Programming*, Kluwer (Dordrecht, 1995).
2. Chvátal, V., *Linear Programming*, W. H. Freeman (New York, 1983).
3. Cooper, M. C., An optimal k-consistency algorithm, *Artificial Intelligence* **41** (1989) 89-95.
4. Hooker, J. N., A quantitative approach to logical inference, *Decision Support Systems* **4** (1988) 45-69.
5. Hooker, J. N., Generalized resolution for 0-1 linear inequalities, *Annals of Mathematics and Artificial Intelligence* **6** (1992) 271-286.

6. Hooker, J. N., Logic-based methods for optimization, in A. Borning, ed., *Principles and Practice of Constraint Programming, Lecture Notes in Computer Science* **874** (1994) 336-349.
7. Hooker, J. N., Logic-based Benders decomposition (1996). Available on http://www.gsia.cmu.edu/afs/andrew/afs/jh38/jnh.html.
8. Nemhauser, G. L., and L. A. Wolsey, *Integer and Combinatorial Optimization*, Wiley (New York, 1988).
9. Quine, W. V., The problem of simplifying truth functions, *American Mathematical Monthly* **59** (1952) 521-531.
10. Quine, W. V., A way to simplify truth functions, *American Mathematical Monthly* **62** (1955) 627-631.
11. Robinson, J. A., A machine-oriented logic based on the resolution principle, *Journal of the ACM* **12** (1965) 23-41.
12. Schrage, L., and L. Wolsey, Sensitivity analysis for branch and bound integer programming, *Operations Research* **33** (1985) 1008-1023.
13. Schrijver, A., *Theory of Linear and Integer Programming*, Wiley (Chichester, 1986).
14. Skorin-Kapov, J., and F. Granot, Nonlinear integer programming: Sensitivity analysis for branch and bound, *Operations Research Letters* **6** (1987) 269-274.

Generalized Local Propagation: A Framework for Solving Constraint Hierarchies

Hiroshi Hosobe,[1] Satoshi Matsuoka,[2] and Akinori Yonezawa[1]

[1] Department of Information Science
[2] Department of Mathematical Engineering
University of Tokyo
3-1, Hongo 7-chome, Bunkyo-ku, Tokyo 113, JAPAN
{detail@is.s, matsu@ipl.t, yonezawa@is.s}.u-tokyo.ac.jp

Abstract. 'Constraint hierarchy' is a nonmonotonic system that allows programmers to describe over-constrained real-world problems by specifying constraints with hierarchical preferences, and has been applied to various areas. An important aspect of constraint hierarchies is the existence of efficient satisfaction algorithms based on local propagation. However, past local-propagation algorithms have been limited to multi-way equality constraints. We overcome this by reformulating constraint hierarchies with a more strict definition, and proposing *generalized local propagation* as a theoretical framework for studying constraint hierarchies and local propagation. Then, we show that *global semi-monotonicity* in satisfying hierarchies turns out to be a practically useful property in generalized local propagation. Finally, we discuss the relevance of generalized local propagation with our previous *DETAIL* algorithm for solving hierarchies of multi-way equality constraints.

Keywords: constraint hierarchies, nonmonotonicity, local propagation, multi-way constraints.

1 Introduction

Constraint hierarchies allow programmers to describe over-constrained real-world problems by specifying constraints with hierarchical *strengths* or preferences [1, 2], and have been applied to various research areas such as constraint logic programming [11, 13], constraint imperative programming [3], and graphical user interfaces [8, 9]. Intuitively, in a constraint hierarchy, the stronger a constraint is, the more it influences the solutions of the hierarchy. For example, the hierarchy of the constraints strong $x = 0$ and weak $x = 1$ yields the solution $x \leftarrow 0$. This property enables programmers to specify preferential or default constraints that may be used in case the set of required or non-default constraints are under-constrained. Moreover, constraint hierarchies are general enough to handle powerful constraint systems such as arithmetic equations and inequalities over reals. Additionally, they allow 'relaxing' of constraints with the same strength by applying, e.g., the least-squares method.

Theoretically, the key property of constraint hierarchies is *nonmonotonicity*. That is, addition of a new constraint to an existing hierarchy may change the set of solutions completely,[3] while in ordinary monotonic constraint systems, it would either preserve or reduce the solution set. For instance, if we add the constraint strong $x = 0$ to the hierarchy of weak $x = 1$, the solution will change from $x \leftarrow 1$ into $x \leftarrow 0$. Clearly, this nonmonotonic property gives us the power to specify default constraints.

An important aspect of constraint hierarchies as a nonmonotonic system is that there are efficient satisfaction algorithms proposed. We can categorize them into the following two approaches:

The refining method first satisfies the strongest level, and then, weaker levels successively. It is employed in the DeltaStar algorithm [11] and a hierarchical constraint logic programming language CHAL [10].

Local propagation gradually solves hierarchies by repeatedly selecting uniquely satisfiable constraints. It is mainly used in constraint solvers for graphical user interfaces such as DeltaBlue [4], SkyBlue [9], and *DETAIL* [6].

First, to illustrate the refining method, suppose we have a hierarchy consisting of required $x = y$, strong $y = z + 1$, medium $z = 0$, and weak $z = 1$. This is solved as follows: first, by satisfying the strongest constraint required $x = y$, the method reduces the set Θ of all variable assignments (mappings from variables to their values) to $\{\theta \in \Theta \mid \theta(x) = \theta(y)\}$; second, by fulfilling the next strongest one strong $y = z + 1$, we obtain $\{\theta \in \Theta \mid \theta(x) = \theta(y) \wedge \theta(y) = \theta(z) + 1\}$; third, evaluating medium $z = 0$ yields $\{\theta \in \Theta \mid \theta(x) = 1 \wedge \theta(y) = 1 \wedge \theta(z) = 0\}$; now, the weakest constraint weak $z = 1$ conflicts with the assignments that have been generated from the stronger constraints, and therefore, remains unsatisfied. As shown in this example, the refining method is a 'straightforward' algorithm for solving constraint hierarchies.

Next, to demonstrate local propagation, reconsider the hierarchy in the last example. Local propagation handles it as follows: first, since medium $z = 0$ can be uniquely solved, it acquires $\{\theta \in \Theta \mid \theta(z) = 0\}$; next, since the instantiation of z makes strong $y = z + 1$ uniquely satisfiable, it produces $\{\theta \in \Theta \mid \theta(y) = 1 \wedge \theta(z) = 0\}$; finally, computing required $x = y$, it outputs $\{\theta \in \Theta \mid \theta(x) = 1 \wedge \theta(y) = 1 \wedge \theta(z) = 0\}$. Note it must reject the weakest constraint weak $z = 1$ at the beginning; otherwise, it would yield an incorrect or empty solution. As suggested with this example, local-propagation algorithms must *plan* in what order they will choose and solve constraints, discarding the ones that lead to incorrect solutions.

Local propagation takes advantage of the potential locality of typical (possibly, non-hierarchical) constraint networks in graphical user interfaces. Basically, it is efficient because it uniquely solves a single constraint in each step. In addition, when a variable value is repeatedly updated by an operation such

[3] Wilson and Borning refer to the property, in a less familiar word, as 'disorderly' [12]. Instead, they use nonmonotonicity for another concept in hierarchical constraint logic programming.

as dragging in interactive interfaces, it can easily re-evaluate only the necessary constraints. However, local propagation has been restricted to *multi-way equality constraints* which can be uniquely solved for each variable, e.g. linear equations over reals. Also, it cannot find multiple solutions for a given constraint hierarchy due to the uniqueness.

Naturally, a question arises whether we can 'generalize' local propagation to solve hierarchies of more powerful constraints without losing its efficiency. In this research, we first reformulate the constraint hierarchy theory, and then introduce a property of constraint systems called *global semi-monotonicity*, which is weaker than monotonicity but not disordered nonmonotonicity. Next, we propose *generalized local propagation*, a theoretical framework for investigating local propagation on constraint hierarchies, and show that global semi-monotonicity exhibits a practically useful property in generalized local propagation. Finally, to illustrate the utilization of GLP, we relate it with our previous *DETAIL* algorithm for multi-way equality constraints that can be simultaneously solved or properly relaxed [6].

2 Related Work

This section briefly overviews previous researches on nonmonotonic constraint systems from the viewpoint of local propagation.

Borning et al., the originators of constraint hierarchies [1], studied properties of hierarchies [2, 12], and also developed local-propagation algorithms called DeltaBlue [4] and SkyBlue [9]. However, their research on theoretical properties did not cover local propagation on constraint hierarchies, but rather mainly focused on hierarchical constraint logic programming (HCLP) [11, 12, 13].

Jampel constructed a certain HCLP instance that separates the HCLP scheme into compositional and non-compositional parts [7]. The method is expected to improve the efficiency of interpreters and compilers since the compositional part is efficiently implementable. However, it is unclear whether such a method is applicable to local propagation.

Freuder and Wallace proposed partial constraint satisfaction for handling constraint satisfaction problems that are impossible or impractical to solve [5]. Theoretically, it is general enough to simulate constraint hierarchies. However, the presented algorithms were ones searching for approximate solutions by 'weakening' problems over finite domains.

3 Generalized Local Propagation

In this section, we first reformulate constraint hierarchies, and then introduce global semi-monotonicity of constraint hierarchies. Next, we generalize local propagation on constraint hierarchies, and study its properties for obtaining correct solutions.

3.1 A Reformulation of Constraint Hierarchies

Before generalizing local propagation, we modify the original formulation of constraint hierarchies in [2] so that it will allow us to better investigate local propagation. Intuitively, the main changes are to explicitly parameterize target hierarchies, and to replace concrete embedded functions/relations with abstract ones satisfying reasonable conditions. First, we define basic terms and symbols. Let $\boldsymbol{X}$ be the set of all variables, $\boldsymbol{D}$ the domain of the variables, and $\boldsymbol{C}$ the set of all constraints.[4] Given a constraint c, $\boldsymbol{X}(c)$ denotes the set of all the variables constrained by c, and given a set C of constraints, we define $\boldsymbol{X}(C) = \{x \in \boldsymbol{X} \mid \exists c \in C. x \in \boldsymbol{X}(c)\}$. A strength of a constraint is an integer l such that $0 \leq l \leq w$, where w is some positive integer. Intuitively, the larger the integer is, the weaker the strength is. Let $\boldsymbol{L}$ be the set of all the strengths. A constraint c with a strength l is represented by c/l. A constraint hierarchy is a finite set H of constraints with strengths, and $\boldsymbol{H}$ expresses the set of all constraint hierarchies. For brevity, we write a variable as x, a constraint as c, a strength as l, and a constraint hierarchy as H, possibly with primes or subscripts.

To represent solutions to constraint hierarchies, we use variable assignments. A variable assignment, denoted as θ, is a mapping from $\boldsymbol{X}$ to $\boldsymbol{D}$, and $\boldsymbol{\Theta}$ indicates the set of all variable assignments. Given a set X of variables, we define $\theta(X) = \theta'(X)$ as $\forall x \in X. \theta(x) = \theta'(x)$.

To assign semantics to constraints, we first introduce *error functions* in the same manner as the original formalization of constraint hierarchies [2]:

Definition 1 (error function). An error function for l is a mapping $e_l : \boldsymbol{C} \times \boldsymbol{\Theta} \to \{0\} \cup \boldsymbol{R}^+$ such that for any c, θ, and θ', $\theta(\boldsymbol{X}(c)) = \theta'(\boldsymbol{X}(c)) \Rightarrow e_l(c,\theta) = e_l(c,\theta')$.

Intuitively, $e_l(c,\theta)$ indicates the error of c/l under θ, which is zero if c/l is exactly satisfied, and positive otherwise. The condition requires that errors of a constraint under two variable assignments are equal if the assignments have equal values for each constrained variable.

Next, we introduce *level comparators*:

Definition 2 (level comparator). A level comparator for l is a ternary relation $\overset{\cdot/l}{\leq}: \boldsymbol{H} \times \boldsymbol{\Theta} \times \boldsymbol{\Theta}$ such that for any H, H', θ, θ', and θ'',[5]

$$\forall c \in \boldsymbol{C}. (c/l \in H \Leftrightarrow c/l \in H') \Rightarrow (\theta \overset{H/l}{\leq} \theta' \Leftrightarrow \theta \overset{H'/l}{\leq} \theta') \quad (1)$$

$$\forall c/l \in H. e_l(c,\theta) = e_l(c,\theta'') \Rightarrow (\theta \overset{H/l}{\leq} \theta' \Leftrightarrow \theta'' \overset{H/l}{\leq} \theta') \quad (2)$$

$$\forall c/l \in H. e_l(c,\theta') = e_l(c,\theta'') \Rightarrow (\theta \overset{H/l}{\leq} \theta' \Leftrightarrow \theta \overset{H/l}{\leq} \theta'') \quad (3)$$

$$\forall c/l \in H. e_l(c,\theta) \leq e_l(c,\theta') \Rightarrow \theta \overset{H/l}{\leq} \theta' \quad (4)$$

[4] We simply define variables and constraints as elements in the corresponding sets, and separately provide their semantics using certain functions and relations.

[5] When we write $\forall c/l \in H$, we mean that the universal quantifier $\forall$ is associated only with c. In other words, l is either free or quantified by another preceding one.

$$\theta \overset{H/l}{\leq} \theta' \wedge \theta' \overset{H/l}{\leq} \theta'' \Rightarrow \theta \overset{H/l}{\leq} \theta'' \tag{5}$$

$$\theta \overset{H/l}{\leq} \theta' \wedge \theta \overset{H'/l}{\leq} \theta' \Rightarrow \theta \overset{H \cup H'/l}{\leq} \theta' . \tag{6}$$

Intuitively, $\theta \overset{H/l}{\leq} \theta'$ means "θ is better than or similar to θ' according to l of H." Conditions (1)–(3) say that the scope of a level comparator is restricted to be inside a designated level. Condition (4) indicates that if errors of all constraints at a level under an assignment are smaller than or equal to those under another assignment, then the former assignment is better than or similar to the latter according to the level. Condition (5) is 'transitivity' of a level comparator. Condition (6) means that if, in two hierarchies, an assignment is better than or similar to another according to the level, then the relation holds in the combination of the hierarchies.

For convenience, we define $\overset{\cdot/l}{\geq}$ (worse than or similar to), $\overset{\cdot/l}{\sim}$ (similar to), $\overset{\cdot/l}{<}$ (better than), $\overset{\cdot/l}{>}$ (worse than), and $\overset{\cdot/l}{\not\sim}$ (incomparable with) as follows: $\theta \overset{H/l}{\geq} \theta' \Leftrightarrow \theta' \overset{H/l}{\leq} \theta$; $\theta \overset{H/l}{\sim} \theta' \Leftrightarrow \theta \overset{H/l}{\leq} \theta' \wedge \theta \overset{H/l}{\geq} \theta'$; $\theta \overset{H/l}{<} \theta' \Leftrightarrow \theta \overset{H/l}{\leq} \theta' \wedge \neg\theta \overset{H/l}{\sim} \theta'$; $\theta \overset{H/l}{>} \theta' \Leftrightarrow \theta \overset{H/l}{\geq} \theta' \wedge \neg\theta \overset{H/l}{\sim} \theta'$; $\theta \overset{H/l}{\not\sim} \theta' \Leftrightarrow \neg\theta \overset{H/l}{\leq} \theta' \wedge \neg\theta \overset{H/l}{\geq} \theta'$.

The original definition of level comparators is quite different from Definition 2 in the following ways: it separates $\overset{\cdot/l}{\leq}$ into $\overset{\cdot/l}{<}$ and $\overset{\cdot/l}{\sim}$, and defines them constructively; it includes (1)–(3) operationally; it seems to implicitly assume (4); it does not require the transitivity of $\overset{\cdot/l}{\sim}$ unlike (5); it presents no condition like (6). Theoretically, the greatest difference is the lack of the transitivity of $\overset{\cdot/l}{\sim}$, which we will discuss later in Subsect. 3.4.

A useful example of a level comparator is the *least-squares level comparator*, defined as $\theta \overset{H/l}{\leq} \theta' \Leftrightarrow \sum_{c/l \in H} e_l(c,\theta)^2 \leq \sum_{c/l \in H} e_l(c,\theta')^2$. Here, two variable assignments are compared by summing squares of errors of constraints at the level. It is easy to prove that the definition fulfills the conditions for level comparators. Used in satisfaction of constraint hierarchies, it works as the least-squares method within level l.

Next, we define *constraint-hierarcy comparators* that totally compare hierarchies by combining level comparators:

Definition 3 (constraint-hierarcy comparator). A constraint-hierarcy comparator is a ternary relation $\dot{<}$: $\boldsymbol{H} \times \boldsymbol{\Theta} \times \boldsymbol{\Theta}$ such that for any H, θ, and θ', $\theta \overset{H}{\dot{<}} \theta' \Leftrightarrow \exists l \in \boldsymbol{L}.(\forall l' \in \boldsymbol{L}.l' < l \Rightarrow \theta \overset{H/l'}{\sim} \theta') \wedge \theta \overset{H/l}{<} \theta'$.

Intuitively, $\theta \overset{H}{\dot{<}} \theta'$ means "θ is better than θ' according to H." It is defined as lexicographic ordering with level comparators as its components. Consequently, the result of a level comparator has absolute priority over those of weaker ones.

For convenience, we define $\dot{>}$ (worse than), $\dot{\sim}$ (similar to), $\dot{\leq}$ (better than or similar to), $\dot{\geq}$ (worse than or similar to), and $\dot{\not\sim}$ (incomparable with) as follows:

$\theta \overset{H}{>} \theta' \Leftrightarrow \theta' \overset{H}{<} \theta$; $\theta \overset{H}{\sim} \theta' \Leftrightarrow \forall l \in \boldsymbol{L}.\theta \overset{H/l}{\sim} \theta'$; $\theta \overset{H}{\leq} \theta' \Leftrightarrow \theta \overset{H}{<} \theta' \vee \theta \overset{H}{\sim} \theta'$; $\theta \overset{H}{\geq} \theta' \Leftrightarrow \theta \overset{H}{>} \theta' \vee \theta \overset{H}{\sim} \theta'$; $\theta \overset{H}{\not\sim} \theta' \Leftrightarrow \neg\theta \overset{H}{\leq} \theta' \wedge \neg\theta \overset{H}{\geq} \theta'$.

The following definition describes the satisfaction of constraint hierarchies using a constraint-hierarcy comparator:

Definition 4 (constraint-hierarcy satisfier). A constraint-hierarcy satisfier is a mapping $S : 2^{\boldsymbol{\Theta}} \times \boldsymbol{H} \rightarrow 2^{\boldsymbol{\Theta}}$ defined as $S(\Theta, H) = \{\theta \in \Theta \mid \neg\exists\theta' \in \Theta.\theta' \overset{H}{<} \theta\}$.

As a shorthand, we write $S(H)$ instead of $S(\boldsymbol{\Theta}, H)$. Intuitively, $S(\Theta, H)$ is the set of assignments obtained by nonmonotonically satisfying H in Θ. By definition, an assignment in $S(\Theta, H)$ is an element in Θ such that there is no better assignment in Θ when compared according to H.

Finally, we define solutions of constraint hierarchies:

Definition 5 (solution). A solution to H is a variable assignment in $S(H)$.

In other words, a solution to H is an assignment found by nonmonotonically satisfying H in the set of all assignments.

One difference between the original and our formulations in defining constraint-hierarcy comparators is that the original restricts top-level constraints to be required, whereas ours allows conflicting constraints at the top level. This is because our definition of hierarchy satisfiers excludes the special treatment of the top level. However, the resulting solutions are the same so far as the top level is not over-constrained. Also, even if we add the condition for the top level to be required, we can easily accommodate it in our following proofs.

3.2 Global Semi-Monotonicity

We define a useful property called *global semi-monotonicity* (GSM) in satisfying constraint hierarchies as follows:

Definition 6 (global semi-monotonicity). S is globally semi-monotonic iff for any H and H', $S(H) \cap S(H') \subseteq S(H \cup H')$.

GSM requires that any common solution to two constraint hierarchies is also a solution to their combination. It is not only natural but also weak (or general) in a sense that the condition is true for any two hierarchies sharing no solutions.

GSM, by definition, is not limited to constraint hierarchies. In a similar style, we can express basic properties of constraint systems. For example, we can represent ordinary monotonicity as $S(H) \cap S(H') = S(H \cup H')$, where the difference from GSM is that it has $S(H) \cap S(H') \supseteq S(H \cup H')$. Thus, we can see that GSM lacks the familiar style of the monotonic property, $S(H) \supseteq S(H \cup H')$. (Such a universal style of formal properties is helpful in comparing different nonmonotonic systems.)

We present a useful class of GSM constraint-hierarcy satisfiers called *global constraint-hierarcy satisfiers*, using *global level comparators* and *global constraint-hierarcy comparators*:

Definition 7 (global level comparator). A level comparator $\overset{\cdot/l}{\leq}$ is global iff for any H, H', θ, and θ',

$$\theta \overset{H/l}{<} \theta' \wedge \theta \overset{H'/l}{\sim} \theta' \Rightarrow \theta \overset{H\cup H'/l}{<} \theta' \tag{7}$$

$$\theta \overset{H/l}{\not\sim} \theta' \Rightarrow \neg\theta \overset{H\cup H'/l}{\sim} \theta' \tag{8}$$

$$\neg\theta \overset{H/l}{<} \theta' \wedge \neg\theta \overset{H'/l}{<} \theta' \Rightarrow \neg\theta \overset{H\cup H'/l}{<} \theta' \ . \tag{9}$$

Definition 8 (global constraint-hierarcy comparator). A constraint-hierarcy comparator is global iff each level comparator is global.

Definition 9 (global constraint-hierarcy satisfier). A constraint-hierarcy satisfier is global iff its constraint-hierarcy comparator is global.

An example of global level comparators is the least-squares level comparator. Most level comparators presented in the original formulation are also global. The following theorem proves that global satisfiers are GSM:

Theorem 10. *S is GSM if S is global.*

Proof. By contradiction: Assume that there exists a θ that is in $S(H)$ and $S(H')$, but not in $S(H \cup H')$. Then, for some θ', $\theta' \overset{H\cup H'}{<} \theta$ holds, that is, for some l, $(\forall l' \in \boldsymbol{L}.\, l' < l \Rightarrow \theta' \overset{H\cup H'/l'}{\sim} \theta) \wedge \theta' \overset{H\cup H'/l}{<} \theta$. By (7) and (8), $\theta' \overset{H\cup H'/l'}{\sim} \theta$ implies $(\theta' \overset{H/l'}{<} \theta \wedge \theta' \overset{H'/l'}{>} \theta) \vee (\theta' \overset{H/l'}{\sim} \theta \wedge \theta' \overset{H'/l'}{\sim} \theta) \vee (\theta' \overset{H/l'}{>} \theta \wedge \theta' \overset{H'/l'}{<} \theta)$, and by (9), $\theta' \overset{H\cup H'/l}{<} \theta$ implies $\theta' \overset{H/l}{<} \theta \vee \theta' \overset{H'/l}{<} \theta$. Hence, it must be either of the following two cases:

Case $\exists l' \in \boldsymbol{L}.\, l' \leq l \wedge (\forall l'' \in \boldsymbol{L}.\, l'' < l' \Rightarrow \theta' \overset{H/l''}{\sim} \theta \wedge \theta' \overset{H'/l''}{\sim} \theta) \wedge \theta' \overset{H/l'}{<} \theta$. Then, $\theta' \overset{H}{<} \theta$ holds, which is a contradiction to $\theta \in S(H)$.

Case $\exists l' \in \boldsymbol{L}.\, l' \leq l \wedge (\forall l'' \in \boldsymbol{L}.\, l'' < l' \Rightarrow \theta' \overset{H/l''}{\sim} \theta \wedge \theta' \overset{H'/l''}{\sim} \theta) \wedge \theta' \overset{H'/l'}{<} \theta$. Then, $\theta' \overset{H'}{<} \theta$ holds, which is a contradiction to $\theta \in S(H')$. □

The converse, that GSM satisfiers are global, is not true; in fact, we have not found weaker conditions for level comparators that yield a set equivalent to GSM. However, we believe that most useful GSM satisfiers are global.[6]

[6] Actually, we could make the converse true if we strengthened the formulation of constraint hierarchies by allowing only 'modular' hierarchy comparators as follows: let level comparators be in a certain set including the least-squares level comparator, and also let hierarchy comparators need to be arbitrarily composed of level comparators in the set. For modular hierarchy comparators, the truth of the converse is easily provable since we can create a non-GSM satisfier by combining any non-global and the least-squares level comparators. Another set of level comparators without the least-squares level comparator may exist, but is unlikely to be more useful.

Global hierarchy comparators might seem strongly related to globally-better comparators in the original formulation, but in fact, they are different. A globally-better comparator is a hierarchy comparator composed of level comparators that compare reals generated by combining errors of constraints. One instance, least-squares-better, is composed of the least-squares level comparators, and therefore, is global. However, worst-case-better, composed of the worst-case level comparators defined as $\theta \overset{H/l}{\leq} \theta' \Leftrightarrow \max_{c/l \in H} e_l(c,\theta) \leq \max_{c/l \in H} e_l(c,\theta')$, is not global because (7) does not hold. Generally, for level comparators of globally-better comparators, (8) is true since they compare reals, i.e. $\neg\, \theta \overset{H/l}{\not\sim} \theta'$. However, it depends on actual instances of level comparators whether both (7) and (9) hold.

3.3 Generalized Local Propagation

Classical local propagation satisfies a constraint network by successively solving individual constraints in an order closely associated with the network topology. Here we generalize local propagation so that it can solve a set of constraints in one step and can also introduce an arbitrary order among such constraint sets. For this purpose, we introduce *ordered partitions* as follows: a partition of a constraint hierarchy is a set generated by decomposing the hierarchy into disjoint subsets called *blocks*; given a partition P, an ordered partition of P is a pair $\langle P, \leq_P \rangle$, where $\leq_P$ is an arbitrary partial order among blocks in P. For brevity, we write $B <_P B'$ instead of $B \leq_P B' \wedge B \neq B'$.

Using ordered partitions into blocks, we define *generalized local propagation* (GLP) in the following way:

Definition 11 (generalized local propagation). Generalized local propagation with S is a mapping $\pi_S(\langle P, \leq_P \rangle)$ defined as follows:

$$\pi_S(\langle P, \leq_P \rangle) = \begin{cases} \boldsymbol{\Theta} & \text{if } |P| = 0 \\ \bigcap_{B \in terminals(\langle P, \leq_P \rangle)} S(\pi_S(before(\langle P, \leq_P \rangle, B)), B) & \text{otherwise} , \end{cases}$$

where *terminals* and *before* are as follows:

$$terminals(\langle P, \leq_P \rangle) = \{B' \in P \mid \neg\, \exists\, B'' \in P.\, B' <_P B''\}$$

$$before(\langle P, \leq_P \rangle, B) = \langle P', \leq_{P'} \rangle \begin{cases} P' = \{B' \in P \mid B' <_P B\} \\ \leq_{P'} = \{\langle B', B'' \rangle \in P' \times P' \mid B' \leq_P B''\} \end{cases} .$$

Intuitively, $terminals(\langle P, \leq_P \rangle)$ is the set of all blocks at terminal positions, and $before(\langle P, \leq_P \rangle, B)$ is the 'ordered sub-partition' of $\langle P, \leq_P \rangle$, where all blocks are before B. For example, consider the ordered partition $\langle P, \leq_P \rangle$ of the blocks B_1, B_2, ..., B_9, as illustrated in Fig. 1. The partial order $\leq_P$ is defined as the reflexive transitive closure of all the arrows in Fig. 1. Then, $terminals(\langle P, \leq_P \rangle)$ is the set $\{B_8, B_9\}$. Also, $before(\langle P, \leq_P \rangle, B_9)$ is the pair consisting of the set

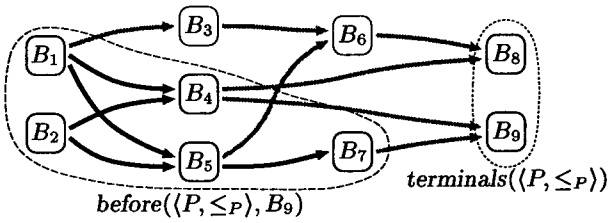


Fig. 1. An ordered partition

$\{B_1, B_2, B_4, B_5, B_7\}$ and the partial order defined as the reflexive transitive closure of the black arrows. Thus, B_9 is satisfied in the set of assignments obtained by applying GLP to blocks before B_9. Accordingly, we can view GLP as a process that successively solves each blocks in some order respecting $\leq_P$. This is always possible because $\leq_P$ is a partial order.

The next lemma shows that by using a global satisfier, GLP respects the similarity of variable assignments for ordered partitions that satisfy the conditions below:

Lemma 12. *Let S be global. Given an arbitrary H, $\langle P, \leq_P \rangle$ of H, and θ in $\pi_S(\langle P, \leq_P \rangle)$, then any θ' is in $\pi_S(\langle P, \leq_P \rangle)$ if $\theta' \overset{H}{\sim} \theta$ and*

$$\forall B \in P. \forall c/l \in B. e_l(c, \theta) > 0 \Rightarrow \forall B' \in P. B' <_P B \Rightarrow \forall c'/l' \in B'. l' < l \ . \quad (10)$$

Proof. By contradiction: Assume that there exists some θ' which is not in $\pi_S(\langle P, \leq_P \rangle)$. Then, it is necessary that for some B_1 in P, θ' is in $\pi_S(before(\langle P, \leq_P \rangle, B_1))$, but not in $S(\pi_S(before(\langle P, \leq_P \rangle, B_1)), B_1)$. Because $\theta' \overset{H}{\sim} \theta$ holds and S is global, $\theta \overset{B_1}{\not\sim} \theta'$ does not hold. Therefore, $\theta \overset{B_1}{<} \theta'$ must hold, that is, there exists some l_1 such that $(\forall l \in \boldsymbol{L}. l < l_1 \Rightarrow \theta \overset{B_1/l}{\sim} \theta') \wedge \theta \overset{B_1/l_1}{<} \theta'$. This implies that for some B in P, $\theta \overset{B/l_1}{>} \theta'$ holds. Since θ must be in $S(\pi_S(before(\langle P, \leq_P \rangle, B)), B)$, it must be either of the following two cases:

Case $\theta' \in \pi_S(before(\langle P, \leq_P \rangle, B)) \wedge \theta' \notin S(\pi_S(before(\langle P, \leq_P \rangle, B)), B)$. Since $\theta \overset{B/l_1}{>} \theta'$ holds, there must exist some l_2 such that $l_2 < l_1$ and $\theta \overset{B/l_2}{<} \theta'$.

Case $\theta' \notin \pi_S(before(\langle P, \leq_P \rangle, B))$. Then, for some B' in P such that $B' <_P B$, θ' is in $\pi_S(before(\langle P, \leq_P \rangle, B'))$, but not in $S(\pi_S(before(\langle P, \leq_P \rangle, B')), B')$. Since $\theta \overset{B/l_1}{>} \theta'$ implies $\exists c/l_1 \in B. e_{l_1}(c, \theta) > 0$, and also since (10) holds, B' contains only stronger constraints than l_1. Therefore, there exists some l_2 such that $l_2 < l_1$ and $\theta \overset{B'/l_2}{<} \theta'$.

Beginning with $\theta \overset{B_1/l_1}{<} \theta'$, both of the two cases resulted in that there exist some l_2 and B_2 such that $l_2 < l_1$ and $\theta \overset{B_2/l_2}{<} \theta'$. Clearly, it causes an infinite

sequence $l_1, l_2, \ldots$ such that $l_i > l_{i+1}$. However, since each l_i is a non-negative integer, it is a contradiction. □

Intuitively, Lemma 12 says that if GLP using a global satisfier generates a variable assignment under which constraints with errors have only stronger constraints before them, then it yields all similar (i.e. $\overset{H}{\sim}$) assignments. Note that the sufficient condition (10) allows constraints without errors to be placed after weaker ones.

In the following theorem, we prove that such variable assignments are solutions to the constraint hierarchy:

Theorem 13. *Let S be global. Given an arbitrary H, $\langle P, \leq_P \rangle$ of H, and θ in $\pi_S(\langle P, \leq_P \rangle)$, then θ is a solution to H if* (10) *holds.*

Proof. By induction on the size of P:

Induction base. If $|P| = 0$, the proposition holds.

Induction step. Assume that if $|P| < n$, the proposition holds. Now, let $|P| = n$. For any B in $terminals(\langle P, \leq_P \rangle)$, θ must be in $S(\pi_S(before(\langle P, \leq_P \rangle, B)), B)$. Therefore, by the induction hypothesis, θ is in $S(H_B)$, where H_B is the union of blocks of $before(\langle P, \leq_P \rangle, B)$. Now, we assume (for contradiction) that there exists some θ' such that $\theta' \overset{H_B \cup B}{<} \theta$, that is, for some l, $(\forall l' \in \boldsymbol{L}.\, l' < l \Rightarrow \theta' \overset{H_B \cup B/l'}{\sim} \theta) \wedge \theta' \overset{H_B \cup B/l}{<} \theta$. It must be either of the following two cases:

Case $\exists l' \in \boldsymbol{L}.\, l' \leq l \wedge (\forall l'' \in \boldsymbol{L}.\, l'' < l' \Rightarrow \theta' \overset{H_B/l''}{\sim} \theta \wedge \theta' \overset{B/l''}{\sim} \theta) \wedge \theta' \overset{H_B/l'}{<} \theta$. Then, $\theta' \overset{H_B}{<} \theta$ holds. Therefore, $\theta \notin S(H_B)$, which is a contradiction.

Case $\exists l' \in \boldsymbol{L}.\, l' \leq l \wedge (\forall l'' \in \boldsymbol{L}.\, l'' < l' \Rightarrow \theta' \overset{H_B/l''}{\sim} \theta \wedge \theta' \overset{B/l''}{\sim} \theta) \wedge \theta' \overset{B/l'}{<} \theta$. Then, for some c/l' in B, $e_{l'}(c, \theta) > 0$ must hold. By (10), H_B contains only stronger constraints than l'. Therefore, $\theta' \overset{H_B}{\sim} \theta$ holds. By Lemma 12, θ' is also in $\pi_S(before(\langle P, \leq_P \rangle, B))$. However, since $\theta' \overset{B}{<} \theta$ holds, it implies $\theta \notin S(\pi_S(before(\langle P, \leq_P \rangle, B)), B)$, which is a contradiction.

Both cases caused contradiction. Therefore, there never exists such θ', i.e. θ is in $S(H_B \cup B)$. Since S is global, θ is also in $S(H)$ by Theorem 10. □

The theorem presents a strategy to design algorithms for solving constraint hierarchies. As noted, the sufficient condition permits constraints without errors to be located after weaker ones. In other words, we can delay the satisfaction of a strong constraint with no error until some appropriate time, for example, "when the constraint becomes uniquely satisfiable." Actually, Theorem 13 gracefully explains why the *DETAIL* algorithm obtains solutions by using local propagation, which we will describe in Sect. 4.

An important instance of such GLP is the refining method. Since constraints have no weaker constraints before them in the method, it can be easily understood by Theorem 13 that it generates only correct solutions, i.e. is sound. In addition, using a certain kind of global satisfiers, the refining method yields all solutions, i.e. is complete:

Proposition 14. *Let S be global such that for any H, θ, and θ', $\neg\, \theta \overset{H}{\not<} \theta'$. For any H and $\langle P, \leq_P \rangle$ of H, $\pi_S(\langle P, \leq_P \rangle) = S(H)$ if $P = \{B_l \mid l \in \boldsymbol{L}\}$ and $B_l \leq_P B_{l'} \Leftrightarrow l \leq l'$, where B_l is the level l of H.*

By this proposition, a refining-method algorithm using a global hierarchy and globally-better comparator, e.g. least-squares-better, is sound and complete, because any globally-better comparator satisfies $\neg\, \theta \overset{H}{\not<} \theta'$.[7]

Next, we define *local level comparators*, *local constraint-hierarcy comparators*, and *local constraint-hierarcy satisfiers*:

Definition 15 (local level comparator). A level comparator $\overset{\cdot/l}{\leq}$ is local iff for any H, θ, and θ', (11) $\theta \overset{H/l}{\leq} \theta' \Rightarrow \forall c/l \in H.\, e_l(c, \theta) \leq e_l(c, \theta')$.

Definition 16 (local constraint-hierarcy comparator). A constraint-hierarcy comparator is local iff each level comparator is local.

Definition 17 (local constraint-hierarcy satisfier). A constraint-hierarcy satisfier is local iff its constraint-hierarcy comparator is local.

By (4) and (11), a local level comparator results in $\theta \overset{H/l}{\leq} \theta' \Leftrightarrow \forall c/l \in H.\, e_l(c, \theta) \leq e_l(c, \theta')$, which is equivalent to level comparators of locally-better comparators in the original formalization. With additional restrictions on multi-way equality constraints, we can regard our formulation as a theoretical basis of efficient constraint-hierarcy satisfaction algorithms such as DeltaBlue.

The following proposition indicates a critical difference between global hierarchy and globally-better comparators:

Proposition 18. *Any local constraint-hierarcy comparator is global.*

The original formulation presented locally-better and globally-better as separate concepts. However, we successfully integrated locally-better and an important class of globally-better into global hierarchy comparators via GSM.

Using a local satisfier, we can obtain a theorem with a weaker sufficient condition than that of Theorem 13:

Theorem 19. *Let S be local. Given an arbitrary H, $\langle P, \leq_P \rangle$ of H, and θ in $\pi_S(\langle P, \leq_P \rangle)$, then θ is a solution to H if*

$$\forall B \in P.\, \forall c/l \in B.\, e_l(c, \theta) > 0 \Rightarrow \forall B' \in P.\, B' <_P B \Rightarrow \forall c'/l' \in B'.\, l' \leq l \ . \quad (12)$$

The difference of (12) from (10) is the existence of equality in $l' \leq l$, which indicates that (12) is weaker than (10). Since it will provide more freedom to organize ordered partitions, we can expect to develop more efficient constraint solving algorithms using local satisfiers. For example, we can regard the blocked constraint lemma presented in the DeltaBlue paper [4] as a specialization of Theorem 19.

[7] It is probably possible to weaken the sufficient conditions for level comparators since the condition for ordered partitions is too strong in the refining method.

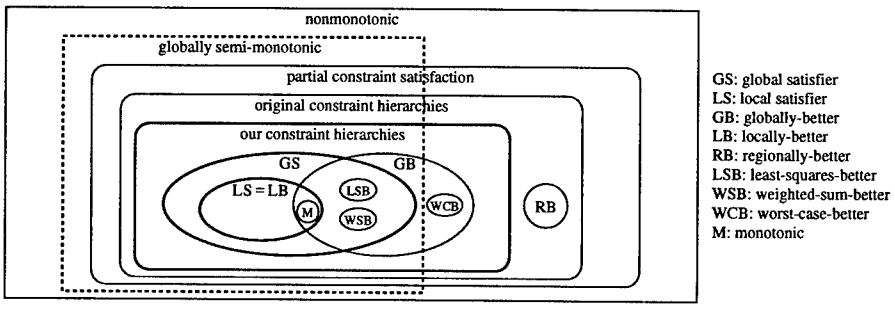


Fig. 2. Relationship of nonmonotonic constraint systems

3.4 Discussion

In this subsection, we review the relationship among nonmonotonic constraint systems, which is roughly illustrated in Fig. 2.

Partial constraint satisfaction [5] is a considerably general theory. Therefore, it will include various nonmonotonic systems, which are not necessarily efficiently solvable.

Our reformulation of constraint hierarchies has become narrower than the original one,[8] because we necessitated $\overset{\cdot/l}{\sim}$ to be transitive by (5). For example, we exclude regionally-better in [11] since its level comparator is defined as $\theta \overset{H/l}{<} \theta' \Leftrightarrow \forall c/l \in H. e_l(c,\theta) \leq e_l(c,\theta') \wedge \exists c/l \in H. e_l(c,\theta) < e_l(c,\theta')$ and $\theta \overset{H/l}{\sim} \theta' \Leftrightarrow \neg\theta \overset{H/l}{<} \theta' \wedge \neg\theta \overset{H/l}{>} \theta'$, where $\overset{H/l}{\sim}$ is not transitive. However, excluding such level comparators contributed to theoretical cleanness and development of generalized local propagation.

It is important to find an expressive and efficiently solvable class of nonmonotonic constraint systems. Except regionally-better and worst-case-better, all the hierarchy comparators presented in the original formulation are global by our formulation. We believe that this fact supports the expressiveness of our global satisfiers with respect to constraint hierarchies. Also, we claim that Theorem 13 for global satisfiers and Theorem 19 for local satisfiers are useful in designing efficient constraint satisfaction algorithms.

4 The *DETAIL* Algorithm

To show how to employ the results in the last section, we relate them with the *DETAIL* algorithm, which we proposed in [6]. *DETAIL* is an incremental algorithm for solving constraint hierarchies based on local propagation. It always

[8] Strictly speaking, as noted earlier, our theory allows conflicting constraints at the top level, while the original theory restricts top-level constraints to be required.

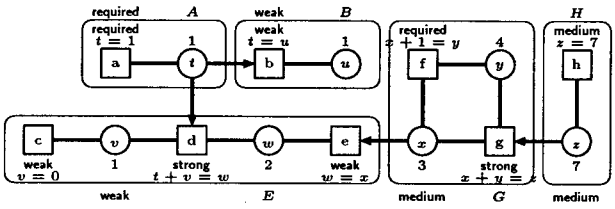


Fig. 3. A configuration of constraint cells

stores planning data instead of an appropriate ordered partition of the current hierarchy, and modifies the plan if a constraint is added to or removed from the hierarchy.

DETAIL handles multi-way equality constraints extended so that it can simultaneously satisfy or properly relax them, in addition to solving them individually as is with classical local propagation. To process such constraints, *DETAIL* maintains a set of *constraint cells* instead of an ordered partition into blocks. A constraint cell can be regarded as a block including output variables, where the constraints in the block are uniquely solved for the output variables. Also, it never shares variables with any other cells. For example, to solve the constraint strong $x + y = 3$ for variable x, *DETAIL* yields a cell of strong $x + y = 3$ and x. By contrast, to simultaneously solve strong $x + y = 3$ and weak $x - y = 1$, it generates a cell of the two constraints and the variables x and y. Similarly, to relax strong $x = 0$ and strong $x = 1$, it produces a cell consisting of the two constraints and x.[9] *DETAIL* solves such constraint cells with pluggable numerical modules called *subsolvers* using e.g. Gaussian elimination.

By the definition of constraint cells, we can determine dependency among cells. Additionally, if we prohibit cyclic dependency, we can naturally identify the overall dependency among cells with a partial order among blocks. Then, we can perform GLP in a 'unique' manner as is with conventional local propagation. For example, consider the hierarchy with the constraints [a], [b], ..., [h] in Fig. 3, where the squares and circles represent constraints and variables respectively, and the boxes with round corners indicate cells. Clearly, in the order respecting the cell dependencies, such as A, B, H, G, and E, we can uniquely solve constraints in each cell.

The other issue is how to determine configurations of cells that obtain correct solutions. To guarantee the sufficient condition (10) for Theorem 13, we employed *walkabout strengths*, which had been first introduced in DeltaBlue [4]. In *DETAIL*, walkabout strengths, associated with constraint cells, are defined to propagate strengths of the weakest constraints. For example, in Fig. 3, the walkabout strength medium of cell G is inherited from the weakest constraint [h]

[9] SkyBlue also realizes simultaneous satisfaction by calling 'cycle solvers,' but provides no features for relaxing constraints [9].

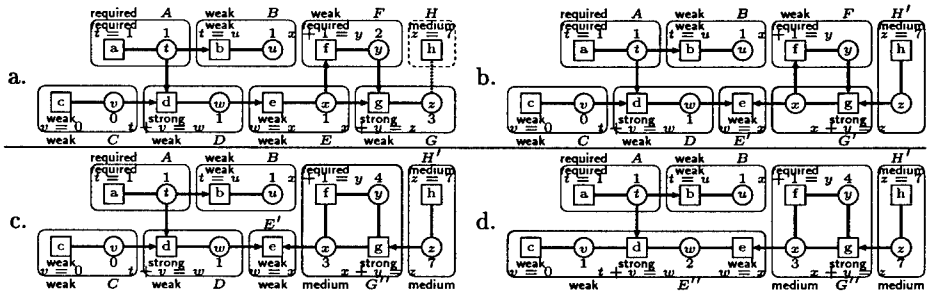


Fig. 4. Adding a constraint to a hierarchy of equalities

among all the constraints that the variables in G depend on, i.e. [f], [g], and [h]. Therefore, the walkabout strength of a cell indicates that there are only constraints with equal or stronger strengths before/in the cell. Thus, it can be easily verified whether a configuration of cells satisfies the sufficient condition for Theorem 13. For example, in Fig. 3, although the weak constraints [c] and [e] in E have positive errors, the walkabout strengths required and medium of the preceding cells A and G indicate that all the forward constraints are stronger than weak.

Now, we demonstrate the *DETAIL* algorithm by example. Figure 4a illustrates the initial configuration of cells, and suppose that we add a new constraint [h] medium $z = 7$ to it. The current solution $z = 3$ conflicts with [h], and the walkabout strength weak of G shows that there is one or more weak constraints in or before G. Therefore, we must change the configuration in the following steps:

1. First, move along the path from the new cell to the nearest source of the walkabout strength, i.e. from H to E, reversing the dependency between them, as shown in Fig. 4b. Note the multi-way equality property of constraints always enables us to perform the reversing operation [6].
2. Next, merge cyclic dependencies generated from the previous step if any. In the example, we collapse the cycle of G' and F as illustrated in Fig. 4c.
3. Third, check whether the victimized cell E' has any preceding cells with the same walkabout strength weak. Figure 4c shows that D is such a cell. Since it violates the sufficient condition for generating solutions, merge all the transitively adjacent cells with the same walkabout strength, i.e. E', D, and C (but not B). Then, we obtain the final configuration in Fig. 4d.

In step 3, we merged all the transitively adjacent cells with the same walkabout strength to ensure the sufficient condition (10) for global satisfiers. However, if we use a local satisfier, we only need to guarantee the weaker condition (12), and therefore, we can omit step 3 (the final configuration would have been Fig. 4c). *DETAIL* also provides the support for local satisfiers, which usually results in smaller constraint cells that can be solved more efficiently.

5 Conclusions and Status

We reformulated the definition of constraint hierarchies, and proposed generalized local propagation to theoretically study local propagation therein. We showed that globally semi-monotonic satisfaction of hierarchies exhibits a practically useful property for generalized local propagation.

By applying the results, we are extending the *DETAIL* algorithm to handle 'multi-way inequality constraints.' We already established its basis, and actually implemented a prototype constraint solver. Due to the existence of inequalities, the new algorithm is exponential in time complexity unlike the original *DETAIL*, which is polynomial. Therefore, we are mainly exploring performance techniques such as efficient scheduling and pruning of constraints.

References

1. Borning, A., R. Duisberg, B. Freeman-Benson, A. Kramer, and M. Woolf, "Constraint Hierarchies," in *OOPSLA'87*, ACM, Oct. 1987, pp. 48–60.
2. Borning, A., B. Freeman-Benson, and M. Wilson, "Constraint Hierarchies," *Lisp and Symbolic Computation*, vol. 5, 1992, pp. 221–268.
3. Freeman-Benson, B. N. and A. Borning, "Integrating Constraints with an Object-Oriented Language," in *ECOOP'92*, no. 615 in LNCS, Springer-Verlag, June/July 1992, pp. 268–286.
4. Freeman-Benson, B. N., J. Maloney, and A. Borning, "An Incremental Constraint Solver," *Comm. ACM*, vol. 33, no. 1, Jan. 1990, pp. 54–63.
5. Freuder, E. C. and R. J. Wallace, "Partial Constraint Satisfaction," *Artificial Intelligence*, vol. 58, 1992, pp. 21–70.
6. Hosobe, H., K. Miyashita, S. Takahashi, S. Matsuoka, and A. Yonezawa, "Locally Simultaneous Constraint Satisfaction," in *PPCP'94*, no. 874 in LNCS, Springer-Verlag, Oct. 1994, pp. 51–62.
7. Jampel, M., "A Compositional Theory of Constraint Hierarchies (Operational Semantics)," in *Proc. Workshop on Over-Constrained Systems at CP'95*, Sept. 1995.
8. Maloney, J. H., A. Borning, and B. N. Freeman-Benson, "Constraint Technology for User-Interface Construction in ThingLab II," in *OOPSLA'89*, ACM, Oct. 1989, pp. 381–388.
9. Sannella, M., "SkyBlue: A Multi-Way Local Propagation Constraint Solver for User Interface Construction," in *UIST'94*, ACM, Nov. 1994, pp. 137–146.
10. Satoh, K. and A. Aiba, "Computing Soft Constraints by Hierarchical Constraint Logic Programming," Tech. Rep. TR-610, ICOT, Japan, Jan. 1991.
11. Wilson, M., "Hierarchical Constraint Logic Programming (Ph.D. Dissertation)," Tech. Rep. 93-05-01, Dept. of Computer Science and Engineering, University of Washington, May 1993.
12. Wilson, M. and A. Borning, "Extending Hierarchical Constraint Logic Programming: Nonmonotonicity and Inter-Hierarchy Comparison," in *Proc. North American Conference on Logic Programming*, 1989.
13. Wilson, M. and A. Borning, "Hierarchical Constraint Logic Programming," *J. Logic Programming*, vol. 16, no. 3/4, July/Aug. 1993, pp. 277–319.

Transformations Between HCLP and PCSP

Michael Jampel*[1], Jean-Marie Jacquet**[1], David Gilbert[2] and Sebastian Hunt[2]

[1] mja,jmj@info.fundp.ac.be, Institut d'Informatique, F.U.N.D.P., B-5000 Namur, Belgium.

[2] drg,seb@cs.city.ac.uk, Dept. of Computer Science, City University, Northampton Square, London EC1V 0HB, U.K.

Abstract. We present a general methodology for transforming between HCLP and PCSP in both directions. HCLP and PCSP each have advantages when modelling problems, and each have advantages when implementing models and solving them. Using the work presented in this paper, the appropriate paradigm can be used for each of these steps, with a meaning-preserving transformation in between if necessary.

1 Introduction

The Hierarchical Constraint Logic Programming (HCLP) scheme of Borning, Wilson, and others [3, 10, 12] greatly extends the expressibility of the general CLP scheme [7]. A semantics has been defined for HCLP [10, 11] and some instances of it have been implemented [8, 10].

The Partial Constraint Satisfaction (PCSP) scheme of Freuder and Wallace [4, 6] is an interesting extension of CSP, which allows the relaxation and optimisation of problems. Extensive empirical studies have been made of some of its instances [6].

There is a widespread view that some link exists between particular HCLP problems and particular PCSP problems, but no general method of transforming one into the other is present in the literature. General frameworks have been developed of which HCLP and PCSP are particular instances [1, 9], but they do not provide a method for transforming between them. In this paper we present a completely general method for finding the HCLP equivalent of any PCSP problem, and vice versa.

Our motivation is mainly methodological, to allow the use of whichever paradigm is appropriate for specification, even if the other one is more appropriate for execution. But we also have a more theoretical motivation, namely to show the relationship between the two formalism's orthogonal approaches to over-constrained systems (OCSs). The two orthogonal approaches are as follows:

* Michael Jampel has been funded by City University during his PhD, and is currently funded by the European Community under the TMR scheme.

** Jean-Marie Jacquet is supported by the Belgian National Fund for Scientific Research as a Research Associate. Part of this work was carried out in the context of the INTAS project 93-1702 "Efficient Symbolic Computing."

HCLP reorganises the *structure* of an over-constrained problem, by specifying relationships between constraints; PCSP keeps a flat structure to the problem, but changes the *meaning* of the individual constraints (by adding elements to the domain).

The structure of this paper is as follows. In Sect.2 we introduce HCLP and PCSP. We then make some preliminary remarks in Sect.3. We discuss transforming HCLP into PCSP in Sect.4, and the transformation from PCSP to HCLP in Sect.5. Both these sections contain illustrative examples and pseudo-code. Finally in Sect.6 we present some conclusions and also mention further work.

2 Background

2.1 Hierarchical Constraint Logic Programming

A good introduction to HCLP can be found in Molly Wilson's PhD thesis [10, chapter 4] or in the early reference [3]; here is a brief overview. CLP can be extended to a Hierarchical CLP scheme including both 'hard' and 'soft' constraints. The HCLP scheme is parameterised not only by the constraint domain $\mathcal{D}$ but also by the 'comparator' $\mathcal{C}$, which is used to compare and select from the different ways of satisfying the soft constraints.

The strengths of the different constraints are indicated by a non-negative integer label. Constraints labelled with a zero are *required* (hard), while constraints labelled j for some $j > 0$ are optional (soft), and are preferred over those labelled k, where $k > j$. (A program can include a list of symbolic names, such as *required*, *strongly-preferred*, etc., for the strength labels, which will be mapped to the natural numbers by the interpreter. If the strength label on a constraint is omitted, it is assumed to be *required*.)

The constraint store σ (a set) is partitioned into the set of required constraints S_0 and the set of optional ones S_i. The solution set for the whole hierarchy is a subset of the solution set of S_0, such that no other solution could be 'better', i.e. for all levels up to k, S_k is completely satisfied, and for level S_{k+1} this solution is better than all others, in terms of some comparator. Backtracking and incomparable hierarchies give rise to multiple possible solution sets, each a subset of the solution to S_0.

'Better' is defined with respect to some comparator [12]. The key notion is that a comparator is a function from a solution of a set of constraints to a sequence of numbers, which are then ordered lexicographically; the first element of the sequence measures how well the solution satisfies the required constraints, the second how well the strongest optional constraints are satisfied, etc.; the earlier in the order, the better that solution is.

See Sect.3.3 for more detail on the particular aspects of HCLP involved in the transformations.

2.2 Partial Constraint Satisfaction Problems

Freuder has developed a theory of Partial Constraint Satisfaction Problems (PCSPs) to weaken systems of constraints which have no solutions, or for which

finding a solution would take too long [4, 6]. PCSP is formalised as containing three components

$$\langle (P, U), (PS, \leq), (M, (N, S)) \rangle$$

where P is a constraint satisfaction problem (CSP), U is a set of 'universes' i.e. a set of potential values for each of the variables in P, $(PS, \leq)$ is a problem space with PS a set of problems and $\leq$ a partial order over problems, M is a distance function over the problem space, and (N, S) are necessary and sufficient bounds on the distance between the given problem P and some solvable member of the problem space PS.

A solution to a PCSP is a problem P' from the problem space and its solution, where the distance between P and P' is less than N. If the distance between P and P' is minimal, then this solution is optimal.

Constraint Satisfaction Problems. The definition of a constraint satisfaction problem is well known: it consists of a pair $\langle V, C \rangle$ where V is a set of variables, each with a domain (extension), and C is a set of constraints[1]. Solving a CSP involves finding one value from the domain of each variable such that all the constraints are satisfied simultaneously. Generally the CSP world restricts itself to considering binary constraints over variables with finite domains. A constraint c between two variables x and y can be denoted c_{xy}.

(The domains of the variables in V are usually considered as unary constraints, but in order to simplify the presentation in [4] they are represented as binary constraints between a variable and itself. The value v is in the domain of a variable x if c_{xx} contains (v, v). In fact, unless there are elements in the domain of a variable which do not appear in any constraint, it is redundant to state individual variable domains explicitly: we can always reconstruct them by saying that $U :: \{i \mid (i, j) \in C_{UV} \text{ or } (k, i) \in C_{WU}, \text{ for all } j, k, V, W\}$.)

The Problem Space. A problem space PS is a partially-ordered set of CSPs where the order $\leq$ is defined as follows (*sols(P)* denotes the set of solutions to a CSP called P):

$$P_1 \leq P_2 \text{ iff } sols(P_1) \supseteq sols(P_2)$$

Note that the ordering is over *problems*, but defined in terms of *solutions*. The problem space for a PCSP must contain the original problem P, which can provide the maximal element in the order, for standard problem spaces.(In the most general case, PS can in fact contain Q such that $P \leq Q$ or such that P and Q are incomparable. But if we take the conjunction of all the constraints in all the problems in PS and create a single problem R, then R will definitely be the greatest element in the order.) If P has no solutions, then *sols(P)* $= \{\}$, which is a subset of all other sets.

[1] Constraints are relations over the variables in V. In CSPs, they are usually treated extensionally, i.e. a binary constraint is just considered as a set of pairs.

The obvious problem space to explore when trying to weaken a problem is the collection of all problems Q such that $Q \leq P$, but it may also be useful to consider only some of these Qs, i.e. those problems which have been weakened in a particular way which makes sense in the context of the system that we are trying to model.

Weakening a Problem. There are four ways to weaken a CSP: (a) enlarging the domain of a variable, (b) enlarging the domain of a constraint, (c) removing a variable, and (d) removing a constraint. Consider example Z above: if none of your shirts match your shoes, you could buy new shoes (variable domain enlargement / augmentation), you could decide that certain shoes do, after all, go with a certain shirt (constraint augmentation), you could decide not to wear shoes at all (variable removal), or you could ignore clashes between shoes and shirts (constraint removal). (As a comparison with these four methods, in HCLP we could decide that the constraint that shirts match shoes is simply not very important.)

Freuder shows in [4] that these can all be considered in terms of (b) above i.e. enlarging constraint domains (adding extra pairs to the relation which defines the constraint). (a) As we have already decided to consider the domains of variables as binary constraints c_{xx}, domain enlargement can clearly be achieved by constraint augmentation. (d) Enlarging a constraint c_{xy} until it equals $x \times y$ (the cartesian product of the domains) has the same effect as removing it altogether. (c) Removing all the constraints on a variable achieves the aim of removing the variable itself.

The Distance Function. Different distance functions are possible, but one obvious one is derived from the partial order on the problem space. If $M(P, P')$ equals the number of solutions not shared by P and P', then when $P' \leq P$ the distance function measures how many solutions have been added by the relaxation of P. Another distance function is a count of the number of constraint values not shared by P and P', and yet others could be based on HCLP-like strength labels. Freuder suggests that a distance function may be used which will tend to find weakened problems with certain properties, for example one whose constraint graph has certain structural properties (for example, see [5]).

3 Transformation: Preliminary Remarks

HCLP and PCSP are not identical in scope, therefore it is impossible to transform all of HCLP into PCSP. However, the work presented in the rest of this paper is complete in the sense we present transformations for every single aspect which can be transformed. First of all, however, we discuss those parts of HCLP which are outside the scope of PCSP, and make other preliminary remarks.

3.1 Differences Which Will Not Be Transformed Away

Firstly[2], CLP in general defines a class of programming languages, which place constraint solving in a logic programming framework, whereas CSP defines a set of problems, techniques, and algorithms. We could embed PCSP in a logic programming framework, and then a comparison with HCLP would make sense, or we can ignore the programming language aspects of HCLP, and compare the resulting theory of 'constraint hierarchies' with PCSP. In this section we will consider the latter approach, i.e. when we say 'HCLP' we really mean 'constraint hierarchies'.

Secondly, CSP techniques are always defined with finite domains whereas the CLP framework extends to continuous domains such as the real numbers. We will only attempt to transform HCLP(FD); however, we *will* transform metric comparators as well as predicate ones. (Metric comparators required a notion of 'distance' between points in the domain, but there is no reason why this distance cannot be discrete.)

Finally, in HCLP the required constraints are special; the difference between required and strong constraints is richer than the difference between, say, strong and weak. PCSP does not have this special class of required constraints. This is discussed further in the next section.

3.2 PCSP with Distinguished Required Constraints

In Sect.2.2, we presented the standard formalisation of PCSPs as $\langle (P, U), (PS, \leq), (M, (N, S))\rangle$. We can modify this to allow us to denote a subset of the constraints in P as 'required', giving a theory which can be called P_{R}CSP (our additions in italics):

$$\langle (P, R, U), (PS, \leq), (M, (N, S))\rangle$$

where P is a constraint satisfaction problem, $R \subseteq P$ *is a set of constraints*, U is a set of 'universes' i.e. a set of potential values for each of the variables in P, $(PS, \leq)$ is a problem space with PS a set of problems *each of which contains all the constraints in* R, and $\leq$ a partial order over problems, M is a 'distance function' on the problem space, and (N, S) are necessary and sufficient bounds on the distance between the given problem P and some solvable member of the problem space PS. A solution to a PCSP is a problem P' from the problem space and its solution, where the distance between P and P' is less than N, and *where all the constraints in* R *are satisfied*. If the distance between P and P' is minimal, then this solution is optimal.

In Sect.2.2 we noted that Freuder states that the obvious problem space to explore when trying to weaken a problem is the collection of all problems Q

[2] The three points mentioned in this section are reasonably straightforward, but have not been explicitly made in any publication. They were mentioned to one of the authors by Borning [Private Communication], but we were already aware of them independently.

such that $Q \leq P$, but we also noted that it may be useful to consider only some of these Qs, i.e. those problems which have been weakened in a particular way which makes sense in the context of the system that we are trying to model [5]. Therefore we note that $\mathrm{P_R}$CSP can be considered simply as selecting those Qs which satisfy all the constraints in R.

One way to select the appropriate part of the problem space is to choose a distance function which gives an infinitely large distance for all other parts. If distance functions are generally denoted by μ, from now on we will assume the existence of a particular function μ_∞, usually parameterised by a set of required constraints σ, which defines a distance of zero to any problem which satisfies all the constraints in σ, and a distance of infinity to all other problems. If T is some arbitrary problem drawn from the problem space, then

$$\mu_{\infty(\sigma)} = \begin{cases} 0, & \text{if } \mathit{sols}(T) \subseteq \mathit{sols}(\sigma) \\ \infty, & \text{otherwise} \end{cases}$$

$\mu_{\infty(\sigma_r)}$ will be the first element of the sequence of functions $\mu = [\mu_r, \mu_s, \mu_w, \ldots]$ parameterised by the constraints at each level of the hierarchy. For example, if the comparator used is UCB, then $\mu = [\mu_{\infty(\sigma_r)}, \mu_{UCB(\sigma_s)}, \mu_{UCB(\sigma_w)}, \ldots]$.

The main conclusion of this section is that we can deal with the issue of required constraints in a straightforward and localised manner. Therefore, perhaps surprisingly, in the rest of this paper we do not really need to emphasise the difference between PCSP and $\mathrm{P_R}$CSP.

3.3 Characterisation of HCLP and PCSP

In this section we present those aspects which are relevant for the transformation process. The relevant aspects for HCLP are

$$\langle \mathbf{H} = (\mathbf{H_0}, \mathbf{H_1}, \mathbf{H_2}, \ldots]), \mathcal{C} = (e, \mathbf{E}, g) \rangle$$

where $\mathbf{H}$ is a hierarchy of constraints, made up of all the required constraints $\mathbf{H_0}$, the strongly preferred constraints $\mathbf{H_1}$, weaker preferences $\mathbf{H_2}$ etc. The comparator $\mathcal{C}$ is used to compare different solutions; it is made up of an error function e which calculates the error of a possible solution with respect to one constraint, $\mathbf{E}$ which simply maps e pointwise over all the constraints in one level of the hierarchy $\mathbf{H_i}$, and a combining function g which combines the elements of the sequence produced by $\mathbf{E}$, resulting in a score for that solution with respect to all the constraints at that level of the hierarchy. For example g might be 'max', or 'sum', or 'least squares'. The resulting sequence of errors $[r, s, w, \ldots]$ giving the errors with respect to each level if the hierarchy, are used to order different possible solutions lexicographically. The lowest element in the order indicates the best solution.

PCSP is formalised as a triple $\langle (P, U), (PS, \leq), (M, (N, S)) \rangle$, but we need only consider certain elements of it as follows: P is a constraint satisfaction problem, and M is a distance function which selects the consistent problem 'nearest' to P.

When transforming HCLP into PCSP, we will take all the constraints in $\mathbf{H}$ without their strength labels as being P. We will use the strength label information and the comparator to construct the appropriate distance function.

When transforming PCSP into HCLP, the constraints in the hierarchy will just be the constraints in P, and the distance function will be used to define their strength labels (i.e. which of the $\mathbf{H_i}$ should contain each constraint) and the comparator $\mathcal{C}$.

In the case of the standard PCSP distance function, all the constraints from P must be placed in the same non-required level of the hierarchy, but it does not matter which one is used. Arbitrarily, we choose to label them 'strong' and so put them in $\mathbf{H_1}$.

4 Transforming HCLP into PCSP

4.1 Creating the Distance Function

The base problem (P) is *all* the constraints in the hierarchy, without their strength labels. U, PS, and (N, S) remain as they would for an original PCSP based on P. (By 'original PCSP' we mean one written down by a user, as opposed to one created by automatically transforming an HCLP problem.)

The distance function will be calculated from a combination of the HCLP comparator and the particular hierarchy of labelled constraints, and the hierarchy will lead to it being stratified into a lexicographic order. The distance function μ derived from a hierarchy with n levels will be stratified into n parts, whose results will be ordered lexicographically (i.e. it will not calculate a single distance of the relaxed CSP from P). Each relaxation (each problem drawn from the problem space PS) will be annotated with a sequence $[d_0, d_1, d_2, \ldots, d_{n-1}]$ each element of which is calculated by the respective distance function in $\mu = [\mu_0, \ldots, \mu_{n-1}]$. (The required level is formally called level 0, the strongest non-required level is 1, down to $n-1$ for the weakest level.) For example, in the case of a hierarchy containing only required, strong and weak constraints, each candidate problem will be annotated with a sequence $[r, s, w]$, where r is the distance according to μ_r, the part of the distance function derived from the required constraints, s is the distance according to μ_s, the part of the distance function derived from the strong constraints, and w is the weak distance, calculated by μ_w. We then order the various relaxations according to the lexicographical order of their sequences.

The distance function calculates the distance of one of the problems T in the problem space PS from the 'ideal' set of constraints which would have distance zero (i.e. completely satisfy all the constraints in the original problem). In fact, as the original constraints might be inconsistent, it is possible that no such ideal set exists.

Let us define *sols*(T) to be the set of solutions to T. We required T to be consistent, and so *sols*(T) will never be empty. Each member of *sols*(T) is a valuation, i.e. an assignment of a value from its domain to each variable in T.

We can calculate how well a particular valuation satisfies the constraints σ using the machinery developed by Borning and Wilson for HCLP.

T may have more than one solution, and hence may give rise to more than one valuation, therefore we define the distance of T from σ to be the *maximum* of distances of each of the valuations in T. This is necessary because HCLP's comparators take as input the set of original constraints and a single valuation / possible solution. The output is the score for that particular valuation, which can then be used to place that valuation in an order. In PCSP, however, distance functions create an order over sets of constraints; a set of constraints can have many solutions, and so we have to choose the score of one of them. We choose the worst (largest) score, i.e. this set of constraints can never give an answer with a score worse than x. For example, if T is said to be a distance of 2 from σ, that means that any solution of T is a distance of *at most* 2 from σ.

Therefore, using some HCLP terminology including denoting a general comparator by C (defined in terms of g, e, and $\mathbf{E}$), the PCSP distance function defined in terms of the set σ of constraints from one optional level of the hierarchy is:

$$\mu_{C(\sigma)}(T) = \max\{g(\mathbf{E}(\sigma\tau) \mid \tau \in sols(T)\}$$

In other words, we treat all the constraints in σ as a sequence, apply a particular valuation τ to each of them, calculate the error for each member of the sequence, combine the errors using g, and then take the maximum of the errors for all the τ and treat it as the error for T.

The various distance functions, each parameterised by the constraints from a different level of the hierarchy, will lead to results which are lexicographically ordered, just as in HCLP. The main difference between standard HCLP and our work is that we interpose the step of taking the maximum error for each of the valuations in T between the application of g and placing in an order.

In the case of UCB, $g(\mathbf{v}) = \sum_{i=1}^{|\mathbf{v}|} v_i$ and $e = e_p$ is the simple predicate error function which returns 0 for each constraint in σ which is consistent with τ, and 1 for each inconsistent constraint [2, 12]. $\mathbf{E}$ is e raised over sequences, i.e. its input is a sequence of constraints, and its output in this case is a sequence of 0's and 1's. g then adds all these individual errors. Least-squares-better (LSB) has a more complicated $e = e_d$, which measures the error as a 'distance' in a metric space. g then sums the squares of these errors:

$$\mu_{UCB(\sigma)}(T) = \max\left\{\sum_{c\in\sigma} e_p(c,\tau) \mid \tau \in sols(T)\right\}$$

$$\mu_{LSB(\sigma)}(T) = \max\left\{\sum_{c\in\sigma} e_d(c,\tau)^2 \mid \tau \in sols(T)\right\}$$

4.2 Pseudo-code

In Fig.1 we present logic-programming-style pseudo-code which transforms HCLP into PCSP in the manner described in this section. The procedure has a collec-

```
transform-HCLP-PCSP( (LabelledConstraints, Comparator),
                     (Constraints, DF) ) :-
    partition-constraints( LabelledConstraints,
                     (Required, Strong, Weak, ... ),
    remove-labels( (Required, Strong, Weak, ...),
                     (UL-Required, UL-Strong, UL-Weak, ...) ),

       % DF = Distance function.
       % The type of DF is determined by the e and g functions
       % (which are determined by the choice of comparator), and
       % and also parameterised by the different levels of constraints
    distance-function-0( (Required, Comparator), DF-R(UL-Required) ),
    distance-function-1( (Strong, Comparator), DF-S(UL-Strong) ),
    distance-function-2( (Weak, Comparator), DF-W(UL-Weak) ),
       ...
    collect-distance-functions( (DF-R(UL-Required), DF-S(UL-Strong),
                     DF-W(UL-Weak)...), DF ),
    collect-constraints( (UL-Required, UL-Strong, UL-Weak,...),
                     Constraints ).
```

Fig. 1. HCLP into PCSP

tion of labelled constraints and a comparator as input, and outputs a collection of unlabelled constraints and a distance function. Therefore if we do not have an implementation of HCLP, we can replace a call to it by the two calls `transform-HCLP-PCSP, PCSP`.

4.3 Example

In this section we present an example of an over-constrained system and its specification and solution in HCLP, and then show its transformation into PCSP.

Consider the problem of choosing matching clothes (example adapted from Freuder and Wallace [6]). A robot wishes to wear a shirt, some shoes, and some trousers, and wants them all to match each other. There are various choices for the different items and various constraints between them. We can easily model this using three finite domain variables with a number of binary constraints between them. If we use the letter S to denote the variable for shirts, then we can use F for shoes (footwear) and T for trousers. The domain of the shirt variable will be $S :: \{r, w\}$ for red and white respectively, and similarly shoes and trousers will have domains $F :: \{c, s\}$ for cordovans and sneakers, and $T :: \{b, d, g\}$, for blue, denim, and grey. A constraint that shirts must match footwear will be denoted C_{SF}, and so on. Then, using Freuder and Wallace's assumptions about which clothes go with which, the complete problem can be expressed formally

as follows (we will call this model Z):

$S :: \{r, w\},\ F :: \{c, s\},\ T :: \{b, d, g\}$
$C_{ST} :: \{(r, g), (w, b), (w, d)\},\ C_{FT} :: \{(s, d), (c, g)\},\ C_{SF} :: \{(w, c)\}$

This problem is over-constrained; it has no solutions. We can see this by choosing the red shirt, and tracing the implications of this choice. We must choose the grey trousers, which forces us to choose the cordovans as footwear. But according to C_{SF}, the cordovans only go with the white shirt. Contradiction. We can trace the effects of choosing the white shirt in the same way, also arriving at a contradiction. Therefore we need to consider some way of relaxing or weakening the problem until solutions can be found.

Example in HCLP. Let us use HCLP strength labels to indicate our assumption that, say, shirts and trousers are more important than footwear, and let us choose the *unsatisfied-count-better* (UCB) comparator:

strong C_{ST}, weak C_{FT}, weak C_{SF}

The solutions to this hierarchy will equal the solutions to the two equally acceptable relaxed problems (C_{ST}, C_{FT}) and (C_{ST}, C_{SF}) which are, in the variable order (S, F, T), $\{(r, c, g), (w, s, d)\}$ and $\{(w, c, b), (w, c, d)\}$ respectively.

HCLP Formulation Transformed into PCSP. The base set of constraints for the PCSP formulation will be all the constraints from the HCLP version, but without strength labels. The distance function will be in two parts $\mu = [\mu_{UCB(C_{ST})}, \mu_{UCB(C_{FT}, C_{SF})}]$, one of which measures the relaxation of the strong constraint, and another for the weak level of the hierarchy. The order will be the lexicographic order over the sequences of integers $[s, w]$ produced by μ.

UCB and $\mu_{UCB(\sigma)}$ are elsewhere in this paper, as is the notion of constraint augmentation. Here it suffices to say that the best solutions, i.e. those earliest in the order created by the distance function, will be the sets of constraints $\{C_{ST}, C_{FT}, C'_{SF}\}$, $\{C_{ST}, C_{FT}, C''_{SF}\}$, $\{C_{ST}, C'_{FT}, C_{SF}\}$, and $\{C_{ST}, C''_{FT}, C_{SF}\}$, where $C'_{SF} = \{(w, c), (\mathbf{r}, \mathbf{c})\}$, i.e. C_{SF} augmented with the extra tuple (r, c), and the other three solutions also contain one augmented constraint ($C''_{SF} = \{(w, c)$, $(\mathbf{w}, \mathbf{s})\}$, $C'_{FT} = \{(s, d), (c, g), (\mathbf{c}, \mathbf{b})\}$, $C''_{FT} = \{(s, d), (c, g), (\mathbf{c}, \mathbf{d})\}$). The solutions from these four sets of constraints, in variable order (S, F, T), are $\{(r, c, g)\}$, $\{(w, s, d)\}$, $\{(w, c, b)\}$, and $\{(w, c, d)\}$, identical to the HCLP solutions.

5 Transforming PCSP into HCLP

5.1 Transforming the Standard PCSP Distance Function

To transform PCSP with the standard distance function into HCLP, we take the constraints in P and give them all the same arbitrary non-required strength label, say 'strong'. Thus they will be placed in $\mathbf{H_1}$. Then we use the HCLP

comparator unsatisfied-count-better (UCB). We claim that this is the correct comparator to use, i.e. we claim that the solutions calculated by HCLP using UCB are the same as those in PCSP, and the particular solutions which are best according to PCSP will also be best according to UCB. (The intuition is as follows: the number of unsatisfied constraints counted by UCB is the same as the number of constraints which would need a single domain augmentation to create a consistent CSP, thus UCB measures an equivalent distance to that measured in PCSP.)

Certain combinations of augmented constraints in the PCSP formulation, which duplicate solutions found at a closer distance, will not appear in the HCLP answer, but all the solutions to these combinations *will* appear. (Here is an analogy: if the list of PCSP solutions, in order from best to worst, is $[a, b, c, a, d, a, e]$, the list of HCLP solutions may be $[a, b, c, d, e]$. So although the lists are not equal, the fact that a should be chosen before b or d is present in both representations.) See Fig.2 for pseudo-code.

Detailed Defence of Choice of UCB. This section contains a detailed defence of our choice of UCB as the comparator to use in HCLP when transforming from PCSP. It addresses one possible key objection, but does not affect the presentation in subsequent sections of the paper.

Consider those PCSP weakenings which involve more than one augmentation of a single constraint. We claim that the following complaint about our choice of UCB is unjustified: "UCB will just detect that a constraint had been violated by a valuation. It wouldn't detect that two different augmentations would be necessary for the constraint not to be violated." It is incoherent because two augmentations can never be necessary for a *single* constraint not to be violated. Two augmentations to a single constraint might, however, lead to an additional two or more solutions, but we can ignore this situation due to the following claim:

Claim: the additional solutions caused by $n \geq 2$ augmentations of a single constraint can be completely separated into n classes, each of which contains solutions caused by only one of the n augmentations. The CSPs represented by these singly-augmented constraints will all appear in the partial order induced by the distance function, and they will all appear earlier than the CSP containing the n-augmented constraint. Therefore, no solutions will be lost by ignoring all multiply-augmented constraints. Therefore, the fact that UCB only picks out those solutions which violate the smallest number of singly-augmented constraints, does not change the set of solutions computed. (All that would happen is that two solutions s_1 and s_2 will separately appear as, say, the equal-best solutions to the hierarchy, but their union will fail to appear as a second-best or third-best solution.)

Example: Let A' denote the constraint A with one extra tuple added to its domain, in the usual manner. Usually there will be more than one way to augment A; these alternatives may be indicated by A'_1, A'_2, etc. Let A'' generally denote two augmentations to A, and specifically $A''_{1,2}$ denote that the two

```
transform-PCSP-HCLP( (Constraints, DF(Type,SpecialCons)),
                     (LabelledConstraints, Comparator) ) :-
    % if Type = standard, then UCB comparator will be chosen, etc.
    % if no constraints are highlighted by the distance function,
    % then SpecialCons will be empty and all constraints are 'strong'

  partition-constraints( (Constraints, DF(Type,SpecialCons))
                     (UL-Strong, UL-Weak, ... ),
  add-labels( (UL-Strong, UL-Weak, ...), (Strong, Weak, ...) ),

  create-comparator(Type, Comparator)
  collect-constraints( (Strong, Weak, ...), LabelledConstraints ).
```

Fig. 2. PCSP into HCLP

augmentations are equivalent to $A'_1 \cup A'_2$. Then our claim is that all the solutions to the CSP $\{A''_{1,2}, B''_{3,4}, C''_{5,6}\}$ are present in the union of the solution sets $\{A'_1, B'_3, C'_5\} \cup \{A'_2, B'_3, C'_5\} \cup \{A'_1, B'_4, C'_5\} \cup \{A'_2, B'_4, C'_5\} \cup \ldots$. In other words, we can ignore multiple augmentations of a single constraint.

Intuition: Consider the CSP as a graph, with each variable represented by a node and each constraint represented by an edge. The tuples which make up the constraint are labels for the edges. A solution to the CSP is a path through every edge in the graph, consistent with the labels. If we add a label to an edge, we are increasing by one the number of paths between the two nodes connected by that edge[3]. If instead we added a different label, we would again increase the number of paths between these two nodes by one. It is intuitively clear that adding these two labels simultaneously will add precisely two paths between the two nodes: any path can only take account of one of the two labels on the edge. We could have arrived at the same set of total paths through the graph by taking two copies of the original graph, adding one new label to each of them, finding the new paths caused by this single extra label, and then eventually taking the union of the two sets of paths.

Proof: Consider various binary constraints over different pairs selected from n variables X_1, X_2, X_3, etc. We can define the *expansion* C^*_{ij} of each constraint C_{ij}, which originally related X_i and X_j, to a set of n-tuples by creating a tuple for each element of the cartesian product of the variables not originally involved in the constraint:

$$C^*_{ij} = \{(v_1, \ldots, v_i, v_j, \ldots, v_n) \mid (v_i, v_j) \in C_{ij}, (v_k, k \neq i, k \neq j) \in \mathrm{dom}(X_k)\}$$

[3] The number of paths through the entire graph may increase by more than one. If there are k paths leading into the start node of the edge under consideration, and l paths leading away from the end node, then adding a path between the two nodes may increase the number of paths through the entire graph by up to kl.

Example: if X has domain $\{a, b\}$, Y has domain $\{c, d\}$, and Z has domain $\{e, f\}$, and if $A_{XY} = \{(a, c), (b, d)\}$, then $A^*_{XY} = \{(a, c, e), (a, c, f), (b, d, e), (b, d, f)\}$.

It is clear that the solution to a CSP is precisely the intersection of the expanded versions of each of its constraints. Thus instead of considering the solution of the set of constraints $\{A_{XY}, B_{YZ}, C_{XZ}\}$, we can just consider $A^*_{XY} \cap B^*_{YZ} \cap C^*_{XZ}$. (This is the similar to the relational database idea of a 'join' between two relations.)

If we add one pair to the domain of one of the constraints in a CSP, it is equivalent to adding a set of n-tuples to the domain of that constraint's expanded version, where the other places in the tuple are filled with all possible combinations of elements from the domains of all the other variables. Continuing with the example, let us assume, without loss of generality, that we have augmented constraint B. This leads to adding a set of n-tuples to B^*; let us call this set of additional tuples R. We can imagine adding a different pair to B which would lead to adding a different set to B^*, say R'. If we add both pairs to B at the same time, it is clear that we must add $R \cup R'$ to B^*.

Our claim is that we can ignore CSPs where one constraint has been multiply augmented; all their solutions will be present in the union of the solutions to CSPs with singly-augmented constraints. This is equivalent to claiming

$$A^* \cap (B^* \cup \underline{(R \cup R')}) \cap C^* = (A^* \cap (B^* \cup \underline{R}) \cap C^*) \underline{\cup} (A^* \cap (B^* \cup \underline{R'}) \cap C^*)$$

The proof is a straightforward exercise in the use of the distributivity laws of set theory ($J \cup (K \cap L) = (J \cup K) \cap (J \cup L)$ and its dual), with one use of the idempotence of set union ($K \cup K = K$).

Therefore, using UCB as our comparator in the automatically generated HCLP version of a PCSP is acceptable. So our transformation from PCSP to HCLP holds.

5.2 Transforming Non-Standard Distance Functions

We have shown above how to transform problems using the standard PCSP distance function into HCLP. We now consider three other possibilities, firstly where all the variables and constraints are treated equally by the distance function but the distance is not defined as minimum augmentation, secondly where some of the *variables* in the problem are highlighted, and finally where some of the *constraints* are highlighted.

Non-Specific (Homogeneous) Distance Functions. All the constraints are put at the 'strong' level of the hierarchy resulting from the transformation. The combining function embodied by the distance function must be transformed into an HCLP-like comparator, specifically into an error function for each constraint and a combining function which combines the errors at each level.

```
?-  HCLP( (LabelledConstraints, Comparator), HCLP-Solutions ),
    transform-HP( (LabelledConstraints, Comparator),
                          (Constraints, DF) )
    PCSP( (Constraints, DF), PCSP-Solutions ),
    equiv( HCLP-Solutions, PCSP-Solutions ).
```

Fig. 3. Equivalence

Distance Functions which Prefer a Subset of the Variables. In general CSPs are considered in terms of binary constraints. The theory can be extended, but complications are introduced. CLP, on the other hand, is indifferent to the arity of constraints. Therefore, if a PCSP problem has some kind of cost function which selects solutions which minimise the value of some function of (some of) the variables, we can simply treat it as another constraint. If the use of the cost function is expressed in the usual way ("Do not violate any constraints in order to minimise the function") then it can be labelled 'weak', while all the constraints in the original PCSP are labelled 'strong.' If it is acceptable to violate constraints in order to minimise the function, then the inverse strength labelling can be used.

Distance Functions which Prefer a Subset of the Constraints. This possibility can be transformed into HCLP in a very straightforward manner: the preferred constraints are labelled 'strong', while the others are labelled 'weak'. If there are multiple subsets with some order over them, then clearly more HCLP strength levels can be used.

6 Conclusions and Further Work

6.1 Conclusions

We have developed a general methodology for transforming between HCLP and PCSP. We have clarified various issues, and provided a proof of correctness. We have shown that strength labels, associated with constraints in HCLP, contain information which is necessary to define the global distance function in PCSP.

The main claim of this paper can be stated as a query in logic programming terms (Fig.3). We assume the existence of two procedures, each of which interfaces to a standard implementation of HCLP and PCSP respectively. An equivalent claim to that in Fig. 3 is that if we do not have an implementation of HCLP, we can replace a call to it by the two calls `transform-HP`, `PCSP`, and vice-versa.

HCLP and PCSP each have advantages when modelling problems, and each have advantages when implementing models and solving them. Using the work

presented in this paper, the appropriate paradigm can be used for each of these steps, with a meaning-preserving transformation in between if necessary.

6.2 Further Work

We would like to investigate issues of algorithmic complexity within the two paradigms.

Acknowledgements. Thanks to Alan Borning, Thomas Schiex, Rob Scott, and Roland Yap for many helpful discussions about CLP.

References

1. Stefano Bistarelli, Ugo Montanari, and Francesca Rossi. Constraint Solving over Semirings. In *IJCAI'95*, Montreal, August 1995.
2. Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint Hierarchies. *Lisp and Symbolic Computation*, 5:223–270, 1992.
3. Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint Hierarchies and Logic Programming. In *ICLP'89* Lisbon, Portugal, June 1989.
4. Eugene Freuder. Partial Constraint Satisfaction. In *IJCAI'89*, August 1989.
5. Eugene Freuder. Exploiting Structure in Constraint Satisfaction Problems. In *Constraint Programming: Proceedings 1993 NATO ASI Parnu, Estonia*, pages 54–79. Springer, 1994.
6. Eugene Freuder and Richard Wallace. Partial Constraint Satisfaction. *Artificial Intelligence*, 58:21–70, 1992.
7. Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *POPL'87* Munich, 1987.
8. Francisco Menezes, Pedro Barahona, and Philippe Codognet. An Incremental Hierarchical Constraint Solver. In Paris Kanellakis, Jean-Louis Lassez, and Vijay Saraswat, editors, *PPCP'93: First Workshop on Principles and Practice of Constraint Programming*, Providence RI, 1993.
9. Thomas Schiex, Hélène Fargier, and Gerard Verfaillie. Valued Constraint Satisfaction Problems: Hard and Easy Problems. In *IJCAI'95*, Montreal, August 1995.
10. Molly Wilson. *Hierarchical Constraint Logic Programming.* PhD thesis, University of Washington, Seattle, May 1993. (Also available as Technical Report 93-05-01).
11. Molly Wilson and Alan Borning. Extending Hierarchical Constraint Logic Programming: Nonmonotonicity and Inter-Hierarchy Comparison. In *NACLP'89* Cleveland, Ohio, 1989.
12. Molly Wilson and Alan Borning. Hierarchical Constraint Logic Programming. *Journal of Logic Programming*, 16(3):277–318, July 1993.

A Test for Tractability

Peter Jeavons[1], David Cohen[1] and Marc Gyssens[2]

[1] Department of Computer Science, Royal Holloway, University of London, UK
e-mail: p.jeavons@dcs.rhbnc.ac.uk
[2] Department WNI, University of Limburg, B-3590 Diepenbeek, Belgium
e-mail: gyssens@charlie.luc.ac.be

Abstract. Many combinatorial search problems can be expressed as 'constraint satisfaction problems', and this class of problems is known to be NP-complete in general. In this paper we investigate restricted classes of constraints which give rise to tractable problems. We show that any set of constraints must satisfy a certain type of algebraic closure condition in order to avoid NP-completeness. We also describe a simple test which can be applied to establish whether a given set of constraints satisfies a condition of this kind. The test involves solving a particular constraint satisfaction problem, which we call an 'indicator problem'.

Keywords: Constraint satisfaction problem, complexity, NP-completeness, indicator problem

1 Introduction

Solving a constraint satisfaction problem is known to be an NP-complete problem in general [13] even when the constraints are restricted to binary constraints. However, many of the problems which arise in practice have special properties which allow them to be solved efficiently. The question of identifying restrictions to the general problem which are sufficient to ensure tractability is important from both a practical and a theoretical viewpoint, and has been extensively studied.

Such restrictions may either involve the structure of the constraints, in other words, which variables may be constrained by which other variables, or they may involve the nature of the constraints, in other words, which combinations of values may be allowed for variables which are mutually constrained. Examples of the first approach may be found in [4, 5, 7, 14, 15] and examples of the second approach may be found in [2, 9, 10, 11, 14, 18, 19].

In this paper we take the second approach, and investigate those classes of constraints which ensure tractability in whatever way they are combined. A number of distinct classes of constraints with this property have previously been identified and shown to be maximal [2, 10, 9]. It is currently unknown whether there are any further tractable constraint classes still to be identified.

In [9] we showed that all known examples of such classes are characterized by a simple algebraic closure condition. This naturally raised the question of whether *all* possible tractable classes satisfy such a closure condition. In this paper we answer this question by proving that any class of constraints which does

not give rise to NP-complete problems must indeed satisfy such a closure condition, and hence this property is a *necessary* condition for a class of constraints to be tractable (assuming that P is not equal to NP).

Furthermore, we describe a simple test to establish whether a given set of constraints satisfies this necessary algebraic condition. The test involves calculating the solutions to a fixed constraint satisfaction problem involving constraints from the given set.

The paper is organised as follows. In Section 2 we give the basic definitions, and describe a general form of algebraic closure condition for a set of relations. In Section 3 we show that this condition is necessary for tractability, and in Section 4 we describe a test for this condition. Finally we summarise the results presented and draw some conclusions.

2 Definitions

2.1 The constraint satisfaction problem

Notation 1 *For any set D, and any natural number n, we denote the set of all n-tuples of elements of D by D^n. For any tuple $t \in D^n$, and any i in the range 1 to n, we denote the value in the ith coordinate position of t by $t[i]$. The tuple t will be written in the form $\langle t[1], t[2], \ldots, t[n] \rangle$.*

A subset of D^n is called an *n-ary relation* over D.

We now define the (finite) constraint satisfaction problem which has been widely studied in the Artificial Intelligence community [13, 14, 12]

Definition 2. An instance of a *constraint satisfaction problem* consists of

- a finite set of variables, V;
- a finite domain of values, D;
- a set of constraints $\{C_1, C_2, \ldots, C_q\}$.
  Each constraint C_i is a pair (S_i, R_i), where S_i is a list of variables of length m_i, called the *constraint scope*, and R_i is an m_i-ary relation over D, called the *constraint relation*. The tuples of R_i indicate the allowed combinations of simultaneous values for the variables in S_i.

The length of the tuples in the constraint relation of a given constraint will be called the *arity* of that constraint. In particular, unary constraints specify the allowed values for a single variable, and binary constraints specify the allowed combinations of values for a pair of variables. A *solution* to a constraint satisfaction problem is a function from the variables to the domain such that the image of each constraint scope is an element of the corresponding constraint relation.

Deciding whether or not a given problem instance has a solution is NP-complete in general [13] even when the constraints are restricted to binary constraints. In this paper we shall consider how restricting the allowed constraint relations to some fixed subset of all the possible relations affects the complexity of this decision problem. We therefore make the following definition.

Definition 3. For any set of relations, Γ, $\mathbf{CSP}(\Gamma)$ is defined to be the class of decision problems with

INSTANCE: A constraint satisfaction problem P in which all constraint relations are elements of Γ.
QUESTION: Does P have a solution?

If there exists an algorithm which solves every problem in CSP(Γ) in polynomial time, then we shall say that Γ is a *tractable* set of relations.

Example 1. The binary inequality relation over a set D, denoted $\neq_D$, is defined as

$$\neq_D = \{\langle d_1, d_2\rangle \in D^2 \mid d_1 \neq d_2\}.$$

Note that CSP($\{\neq_D\}$) corresponds precisely to the GRAPH $|D|$-COLORABILITY problem [6]. This problem is tractable when $|D| \leq 2$ and NP-complete when $|D| \geq 3$.

Example 2. We now describe four relations which will be used as examples of constraint relations throughout the paper.

Each of these relations is a set of tuples of elements from the domain $D = \{0, 1, 2\}$, as defined below:

$$R_1 = \{\ \langle 0,0\rangle,\ \langle 1,2\rangle,\ \langle 0,1\rangle,\ \langle 2,1\rangle\ \}$$

$$R_2 = \{\ \langle 0,1,2\rangle,\ \langle 1,2,0\rangle,\ \langle 2,0,1\rangle\ \}$$

$$R_3 = \{\ \langle 0,1\rangle,\ \langle 0,2\rangle,\ \langle 1,0\rangle,\ \langle 1,1\rangle,\ \langle 1,2\rangle,\ \langle 2,0\rangle,\ \langle 2,1\rangle,\ \langle 2,2\rangle\ \}$$

$$R_4 = \{\ \langle 0,1\rangle,\ \langle 0,2\rangle,\ \langle 1,0\rangle,\ \langle 1,2\rangle,\ \langle 2,0\rangle,\ \langle 2,1\rangle\ \}$$

CSP($\{R_1, R_2, R_3, R_4\}$) contains all constraint satisfaction problems in which the constraint relations are all equal to one of R_1, R_2, R_3 or R_4.

Note that $R_4 = \neq_D$, so by Example 1, CSP(Γ) is NP-complete for any subset, Γ, of the set $\{R_1, R_2, R_3, R_4\}$ containing the relation R_4.

The complexity of CSP(Γ) for subsets of $\{R_1, R_2, R_3, R_4\}$ which do *not* contain R_4 will be determined using the techniques developed later in this paper.

2.2 Operations on relations

In Section 3 we shall examine conditions on a set of relations Γ which allow known NP-complete problems to be reduced to CSP(Γ). The reductions will be described using standard operations from relational algebra [1], which are described in this section.

Definition 4. We define the following operations on relations.

- Let R_1 be an n-ary relation over a domain D and let R_2 be an m-ary relation over D. The *Cartesian product* $R_1 \times R_2$ is defined to be the $(n+m)$-ary relation

$$R_1 \times R_2 = \{\langle t[1], t[2], \ldots, t[n+m]\rangle \mid (\langle t[1], t[2], \ldots, t[n]\rangle \in R_1) \wedge (\langle t[n+1], t[n+2], \ldots, t[n+m]\rangle \in R_2)\}.$$

- Let R be an n-ary relation over a domain D. Let $1 \leq i, j \leq n$. The *equality selection* $\sigma_{i=j}(R)$ is defined to be the n-ary relation

$$\sigma_{i=j}(R) = \{t \in R \mid t[i] = t[j]\}.$$

- Let R be an n-ary relation over a domain D. Let $i_1, \ldots, i_k$ be a subsequence of $1, \ldots, n$. The *projection* $\pi_{i_1,\ldots,i_k}(R)$ is defined to be the k-ary relation

$$\pi_{i_1,\ldots,i_k}(R) = \{\langle t[i_1], \ldots, t[i_k]\rangle \mid t \in R\}.$$

It is well-known that the combined effect of two constraints in a constraint satisfaction problem may be obtained by performing a relational *join* operation [1] on the two constraint relations [7]. The next result is a simple consequence of the definition of the relational join operation.

Lemma 5. *The join of relations R_1 and R_2 can be calculated by performing a sequence of Cartesian product, equality selection and projection operations on R_1 and R_2.*

In view of this result, it will be convenient to use the following notation.

Notation 6 *The set of all relations which may be obtained from a given set of relations, Γ, using some sequence of Cartesian product, equality selection, and projection operations will be denoted Γ^+.*

2.3 Operations on tuples

We will show in Section 3 that any tractable set of relations over a set D must satisfy certain algebraic conditions. In order to describe these conditions we need to consider arbitrary operations on D, in other words, arbitrary functions from D^k to D, for arbitrary values of k.

Any such operation on D may be extended to an operation on tuples over D by applying the operation in each coordinate position separately (i.e., pointwise). Hence, any operation defined on the domain of a relation may be used to define an operation on the tuples in that relation, as follows:

Definition 7. Let R be an n-ary relation over a domain D, and let $\otimes : D^k \to D$ be a k-ary operation on D.

For any collection of k tuples, $t_1, t_2, \ldots, t_k \in R$, (not necessarily all distinct) the n-tuple $\otimes(t_1, t_2, \ldots, t_k)$ is defined as follows:

$$\otimes(t_1, t_2, \ldots, t_k) =$$
$$\langle \otimes(t_1[1], t_2[1], \ldots, t_k[1]), \otimes(t_1[2], t_2[2], \ldots, t_k[2]), \ldots, \otimes(t_1[n], t_2[n], \ldots, t_k[n]) \rangle.$$

Using this definition, we now define the following closure property of relations.

Definition 8. Let R be a relation over a domain D, and let $\otimes : D^k \to D$ be a k-ary operation on D.

R is said to be *closed under* $\otimes$ if, for all $t_1, t_2, \ldots, t_k \in R$ (not necessarily all distinct),

$$\otimes(t_1, t_2, \ldots, t_k) \in R.$$

Example 3. Let $\triangle$ denote the ternary operation which returns the first repeated value of its three arguments, or the first value if they are all distinct.

The relation R_2 defined in Example 2 is closed under $\triangle$, since applying the $\triangle$ operation to any 3 elements of R_2 yields an element of R_2. For example,

$$\triangle(\langle 0,1,0 \rangle, \langle 1,2,0 \rangle, \langle 1,2,0 \rangle) = \langle 1,2,0 \rangle \in R_2.$$

The relation R_1 defined in Example 2 is *not* closed under $\triangle$, since applying the $\triangle$ operation to the last 3 elements of R_1 yields a tuple which is not an element of R_1:

$$\triangle(\langle 1,2 \rangle, \langle 0,1 \rangle, \langle 2,1 \rangle) = \langle 1,1 \rangle \notin R_1.$$

For any set of relations Γ, and any operation $\otimes$, if every $R \in \Gamma$ is closed under $\otimes$, then we shall say that Γ is closed under $\otimes$. The next lemma indicates that the property of being closed under some operation is preserved by each of the operations on relations described in Section 2.2.

Lemma 9. *Let R_1 and R_2 be relations which are closed under $\otimes$, for some operation $\otimes$.*

The following relations are also closed under $\otimes$:

1. *the Cartesian product, $R_1 \times R_2$;*
2. *any projection of R_1 or R_2;*
3. *any equality selection from R_1 or R_2.*

Proof. Follows immediately from the definitions.

We shall be particularly interested in operations that depend on more than one argument. We therefore make the following definition:

Definition 10. An operation $\otimes : D^k \to D$ is called *essentially unary* if there exists some non-constant unary operation $f : D \to D$ and some i in the range 1 to k, such that $\otimes(d_1, d_2, \ldots, d_k) = f(d_i)$ for all $d_1, d_2, \ldots, d_k$.

Note that constant functions are excluded from this definition, and so are *not* essentially unary.

3 Closure and Tractability

In this Section we will show that any set of relations which is only closed under essentially unary operations will give rise to a class of problems which is NP-complete. Assuming that P is not equal to NP, this implies that any tractable set of relations must be closed under some operation which is not essentially unary. In other words, this algebraic property is a necessary condition for tractability.

Theorem 11. *For any finite set of relations, Γ, over a finite set D, either*

1. $\mathrm{CSP}(\Gamma)$ *is NP-complete; or*
2. Γ *is closed under some operation $\otimes$ which is not essentially unary.*

Proof. When $|D| \leq 2$, then we may assume without loss of generality that $D \subseteq \{0, 1\}$, where 0 corresponds to the Boolean value False and 1 corresponds the Boolean value True. It follows that the class of problems $\mathrm{CSP}(\Gamma)$ corresponds to the GENERALISED SATISFIABILITY problem over the set of logical relations Γ, as defined in [16] (see also [6]). It was established in [16] that this problem is NP-complete unless one of the following conditions holds:

1. Every relation in Γ contains the tuple $(0, 0, \ldots, 0)$;
2. Every relation in Γ contains the tuple $(1, 1, \ldots, 1)$;
3. Every relation in Γ is definable by a formula in conjunctive normal form in which each conjunct has at most one negated variable.
4. Every relation in Γ is definable by a formula in conjunctive normal form in which each conjunct has at most one unnegated variable.
5. Every relation in Γ is definable by a formula in conjunctive normal form in which each conjunct contains at most 2 literals.
6. Every relation in Γ is the set of solutions of a system of linear equations over the finite field GF(2).

It is straightforward to show that in each of these cases Γ is closed under some operation which is not essentially unary (see [8] for details). Hence the result holds when $|D| = 2$.

For larger values of $|D|$ we proceed by induction. Assume that $|D| \geq 3$ and the result holds for all smaller values of $|D|$. Let $m = |D|(|D| - 1)$ and let $n = |D|^m$. Let M be an m by n matrix over D in which the columns consist of all possible m-tuples over D (in some order). Let R_0 be the relation consisting of all the tuples occuring as rows of M. The only condition we place on the choice of order for the columns of M is that $\pi_{1,2}(R_0) = \neq_D$, where $\neq_D$ is the binary inequality relation over D, as defined in Example 1.

We now construct a relation $\hat{R}_0$ which is the 'closest approximation' to R_0 that we can obtain from the relations in Γ using the Cartesian product, equality selection and projection operations:

$$\hat{R}_0 = \bigcap \{R \in (\Gamma \cup D^1)^+ \mid R_0 \subseteq R\}.$$

Since this is a finite intersection, and intersection is a special case of join we have from Lemma 5 that $\hat{R}_0 \in (\Gamma \cup D^1)^+$. In other words, the relation $\hat{R}_0$ may be obtained as a derived constraint relation in some problem belonging to CSP(Γ).

There are now two cases to consider:

1. If there exists some tuple $t_0 \in \hat{R}_0$ with $t_0[1] = t_0[2]$, then we will construct, using t_0, an appropriate operation under which Γ is closed.
   Define the function $\otimes : D^m \to D$ by setting $\otimes(d_1, d_2, \ldots, d_m) = t_0[j]$ where j is the unique column of M corresponding to the m-tuple $\langle d_1, d_2, \ldots, d_m \rangle$. We will show that Γ is closed under $\otimes$.
   Choose any $R \in \Gamma$, and let p be the arity of R. We are required to show that R is closed under $\otimes$. Consider any sequence $t_1, t_2 \ldots, t_m$ of tuples of R (not necessarily distinct), and for $i = 1, 2, \ldots, p$, let c_i be the m-tuple $\langle t_1[i], t_2[i], \ldots, t_m[i] \rangle$. For each pair of indices, i, j, such that $c_i = c_j$, apply the equality selection $\sigma_{i=j}$ to R, to obtain a new relation R'.
   Now choose a maximal set of indices, $I = \{i_1, i_2, \ldots, i_s\}$, such that the corresponding c_i are all distinct, and construct the relation $R'' = \pi_I(R') \times D^{n-|I|}$. Finally, permute the coordinate positions of R'' (by a sequence of Cartesian product, equality selection, and projection operations), such that $R'' \supseteq R_0$ (this is always possible, by the construction of R_0 and R''). Since $R'' \in (\Gamma \cup D^1)^+$, we know that t_0 is a tuple of R'', by the definition of $\hat{R}_0$. Hence the appropriate projection of t_0 is an element of R, and R is closed under $\otimes$.
   If $\otimes$ is essentially unary, then let $f : D \to D$ be the corresponding unary operation, and set

   $$f(D) = \{f(d) \mid d \in D\};$$
   $$f(\Gamma) = \{\{\langle f(d_1), f(d_2), \ldots, f(d_r) \rangle \mid \langle d_1, d_2, \ldots, d_r \rangle \in C\} \mid C \in \Gamma\}.$$

   By the choice of t_0, f cannot be injective, so $|f(D)| < |D|$. By the inductive hypothesis, we know that either CSP($f(\Gamma)$) is NP-complete (in which case CSP(Γ) must also be NP-complete) or else $f(\Gamma)$ is closed under some operation $\otimes$ which is not essentially unary (in which case Γ is closed under the operation $f\otimes$, which is also not essentially unary). Hence, the result follows by induction in this case.
2. Alternatively, if $\hat{R}_0$ contains no tuple t such that $t[1] = t[2]$, then $\pi_{1,2}(\hat{R}_0) = \neq_D$, so $\neq_D \in (\Gamma \cup D^1)^+$. But this implies that CSP($\{\neq_D\}$) is reducible to CSP(Γ), since every occurence of the constraint relation $\neq_D$ may be replaced with an equivalent collection of constraints with relations chosen from Γ. However, it was pointed out in Example 1 that CSP($\{\neq_D\}$) corresponds to the GRAPH $|D|$-COLORABILITY problem [6], which is NP-complete when $|D| \geq 3$. Hence, this implies that CSP(Γ) is NP-complete, and the result holds in this case also.

We may sharpen this result a little further by bounding the possible arity of the closure operations on Γ which we need to consider, as the next result shows.

Theorem 12. *For any set of relations Γ over a finite set D, if Γ is closed under some operation $\otimes$ which is not essentially unary, then it is also closed under some operation $\hat{\otimes}$ of arity at most* $\max\{3, |D|\}$, *which is not essentially unary.*

Proof. (This proof is adapted from the proof of Lemma 1.14 in [17]). Let Γ be a set of relations which is closed under some operation $\otimes$ which is not essentially unary. Let $\hat{\otimes}$ be an operation of the smallest possible arity, such that Γ is closed under $\hat{\otimes}$ and $\hat{\otimes}$ is not essentially unary, and let k be the arity of $\hat{\otimes}$. If $k \leq 3$, then the result holds, so we only need to consider the case when $k \geq 4$.

Now consider the operations which are obtained from $\hat{\otimes}$ by identifying two arguments. Since Γ is closed under these operations, and they have a lower arity than $\hat{\otimes}$, they must all be essentially unary, by the choice of $\hat{\otimes}$.

Hence, if we identify the first two arguments we have

$$\hat{\otimes}(\widehat{x_1, x_1}, x_3, x_4, \ldots, x_k) = f_{12}(x_i)$$

for some non-constant unary operation f_{12} and some $i \in \{1, 2, \ldots, k\}$. Similarly, if we identify the third and fourth arguments we have

$$\hat{\otimes}(x_1, x_2, \widehat{x_3, x_3}, x_5, \ldots, x_k) = f_{34}(x_j)$$

for some non-constant unary operation f_{34} and some $j \in \{1, 2, \ldots, k\}$. This means that

$$\hat{\otimes}(\widehat{x_1, x_1}, \widehat{x_3, x_3}, x_5, \ldots, x_k) = f_{12}(x_i) = f_{34}(x_j)$$

for all possible choices of $x_1, x_3, x_5, x_6, \ldots, x_k$, which implies that either $i \notin \{1, 2\}$ or $j \notin \{3, 4\}$. It follows from this that we can permute the order of the arguments of $\hat{\otimes}$ to obtain a function $\oplus$ which satisfies the identity

$$\oplus(x_1, \widehat{x, x}, x_4, \ldots, x_k) = f(x_1)$$

for some non-constant unary operation f. In particular, we have $\oplus(x_1, y, y, \ldots, y) = f(x_1)$. Now, for all distinct pairs of indices i, j in $\{2, 3 \ldots, k\}$, we know that the $k-1$-ary function

$$\oplus(x_1, x_2, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_{j-1}, x, x_{j+1}, \ldots, x_k)$$

is an essentially unary function, and since we have just shown that for one particular choice of $x_2, x_3, \ldots, x_k$ it equals $f(x_1)$, we know that it must equal $f(x_1)$ in all cases.

Similarly, (using the fact that $k \geq 4$), for all indices $i \in \{2, 3, \ldots, k\}$, we can establish that

$$\oplus(x, x_2, \ldots, x_{i-1}, x, x_{i+1}, \ldots, x_k) = f(x).$$

Now, if $k > |D|$ then there are more arguments than values, so at least one argument value must be repeated somewhere, and in that case we would have $\oplus(x_1, x_2, \ldots, x_k) = f(x_1)$. Since $\oplus$ is just $\hat{\otimes}$ with the order of the arguments permuted, we have contradicted the fact that $\hat{\otimes}$ is not essentially unary. This means that we must have $k \leq |D|$, and the result follows.

4 A Test for Tractability

Let Γ be a set of relations over a set D. Note that the operations under which Γ is closed are simply mappings from D^k to D, for some k, which satisfy certain constraints, as described in Definition 8. In this Section we show that it is possible to identify these operations by solving a single constraint satisfaction problem. In fact, we shall show that these closure operations are precisely the solutions to a constraint satisfaction problem of the following form.

Definition 13. Let Γ be a set of relations over a finite domain D.

For any natural number $m > 0$, the *indicator problem* for Γ of order m is defined to be the constraint satisfaction problem $\mathcal{IP}(\Gamma, m)$ with

- Set of variables D^m;
- Domain of values D;
- Set of constraints $\{C_1, C_2, \ldots, C_q\}$, such that for each $R \in \Gamma$, and for each sequence $t_1, t_2, \ldots, t_m$ of tuples from R, there is a constraint $C_i = (S_i, R)$ with $S_i = (v_1, v_2, \ldots, v_n)$ where n is the arity of R and $v_j = \langle t_1[j], t_2[j], \ldots, t_m[j] \rangle$.

Example 4. Consider the relation R_1 over $D = \{0, 1, 2\}$, defined in Example 2.

The indicator problem for $\{R_1\}$ of order 1, $\mathcal{IP}(\{R_1\}, 1)$, has 3 variables and 4 constraints. The set of variables is

$$\{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle\},$$

and the set of constraints is

$$\begin{aligned}\{\ &((\langle 0\rangle, \langle 0\rangle), R_1),\\ &((\langle 0\rangle, \langle 1\rangle), R_1),\\ &((\langle 1\rangle, \langle 2\rangle), R_1),\\ &((\langle 2\rangle, \langle 1\rangle), R_1)\ \}.\end{aligned}$$

The indicator problem for $\{R_1\}$ of order 2, $\mathcal{IP}(\{R_1\}, 2)$, has 9 variables and 16 constraints. The set of variables is

$$\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 0,2\rangle, \langle 1,0\rangle, \langle 1,1\rangle, \langle 1,2\rangle, \langle 2,0\rangle, \langle 2,1\rangle, \langle 2,2\rangle\},$$

and the set of constraints is

$$\begin{aligned}\{\ &((\langle 0,0\rangle, \langle 0,0\rangle), R_1), ((\langle 0,0\rangle, \langle 0,1\rangle), R_1),\\ &((\langle 0,0\rangle, \langle 1,0\rangle), R_1), ((\langle 0,0\rangle, \langle 1,1\rangle), R_1),\\ &((\langle 0,1\rangle, \langle 0,2\rangle), R_1), ((\langle 0,1\rangle, \langle 1,2\rangle), R_1),\\ &((\langle 0,2\rangle, \langle 0,1\rangle), R_1), ((\langle 0,2\rangle, \langle 1,1\rangle), R_1),\\ &((\langle 1,0\rangle, \langle 2,0\rangle), R_1), ((\langle 1,0\rangle, \langle 2,1\rangle), R_1),\\ &((\langle 1,1\rangle, \langle 2,2\rangle), R_1), ((\langle 1,2\rangle, \langle 2,1\rangle), R_1),\\ &((\langle 2,0\rangle, \langle 1,0\rangle), R_1), ((\langle 2,0\rangle, \langle 1,1\rangle), R_1),\\ &((\langle 2,1\rangle, \langle 1,2\rangle), R_1),\ ((\langle 2,2\rangle, \langle 1,1\rangle), R_1)\ \}.\end{aligned}$$

Example 5. Consider the relation R_2 over $D = \{0, 1, 2\}$, defined in Example 2.

The indicator problem for $\{R_2\}$ of order 1, $\mathcal{IP}(\{R_2\}, 1)$, has 3 variables and 3 constraints. The set of variables is

$$\{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle\},$$

and the set of constraints is

$$\begin{aligned}\{\ &((\langle 0\rangle, \langle 1\rangle, \langle 2\rangle), R_2),\\ &((\langle 1\rangle, \langle 2\rangle, \langle 0\rangle), R_2),\\ &((\langle 2\rangle, \langle 0\rangle, \langle 1\rangle), R_2)\ \}.\end{aligned}$$

The indicator problem for $\{R_2\}$ of order 2, $\mathcal{IP}(\{R_2\}, 2)$, has 9 variables and 9 constraints. The set of variables is

$$\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 0,2\rangle, \langle 1,0\rangle, \langle 1,1\rangle, \langle 1,2\rangle, \langle 2,0\rangle, \langle 2,1\rangle, \langle 2,2\rangle\},$$

and the set of constraints is

$$\begin{aligned}\{\ &((\langle 0,0\rangle, \langle 1,1\rangle, \langle 2,2\rangle), R_2),\\ &((\langle 0,1\rangle, \langle 1,2\rangle, \langle 2,0\rangle), R_2),\\ &((\langle 0,2\rangle, \langle 1,0\rangle, \langle 2,1\rangle), R_2),\\ &((\langle 1,0\rangle, \langle 2,1\rangle, \langle 0,2\rangle), R_2),\\ &((\langle 1,1\rangle, \langle 2,2\rangle, \langle 0,0\rangle), R_2),\\ &((\langle 1,2\rangle, \langle 2,0\rangle, \langle 0,1\rangle), R_2),\\ &((\langle 2,0\rangle, \langle 0,1\rangle, \langle 1,2\rangle), R_2),\\ &((\langle 2,1\rangle, \langle 0,2\rangle, \langle 1,0\rangle), R_2),\\ &((\langle 2,2\rangle, \langle 0,0\rangle, \langle 1,1\rangle), R_2)\ \}.\end{aligned}$$

Example 6. Consider the relations R_1 and R_2 over $D = \{0, 1, 2\}$, defined in Example 2.

The indicator problem for $\{R_1, R_2\}$ of order 1, $\mathcal{IP}(\{R_1, R_2\}, 1)$, has 3 variables and 7 constraints. The set of variables is

$$\{\langle 0\rangle, \langle 1\rangle, \langle 2\rangle\},$$

and the set of constraints is equal to the union of the set of constraints of $\mathcal{IP}(\{R_1\}, 1)$, as defined in Example 4, and the set of constraints of $\mathcal{IP}(\{R_2\}, 1)$, as defined in Example 5.

The indicator problem for $\{R_1, R_2\}$ of order 2, $\mathcal{IP}(\{R_1, R_2\}, 2)$, has 9 variables and 25 constraints. The set of variables is

$$\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 0,2\rangle, \langle 1,0\rangle, \langle 1,1\rangle, \langle 1,2\rangle, \langle 2,0\rangle, \langle 2,1\rangle, \langle 2,2\rangle\},$$

and the set of constraints is equal to the union of the set of constraints of $\mathcal{IP}(\{R_1\}, 2)$, as defined in Example 4, and the set of constraints of $\mathcal{IP}(\{R_2\}, 2)$, as defined in Example 5.

Solutions to the indicator problem for Γ of order m are functions from D^m to D, or in other words, m-ary operations on D. We now show that they are precisely the m-ary operations under which Γ is closed.

Theorem 14. *For any set of relations Γ over domain D, the set of solutions to $\mathcal{IP}(\Gamma, m)$ is equal to the set of m-ary operations under which Γ is closed.*

Proof. By Definition 8, we know that Γ is closed under the m-ary operation $\otimes$ if and only if $\otimes$ satisfies the condition $\otimes(t_1, t_2, \ldots, t_m) \in R$ for each possible choice of $R \in \Gamma$ and $t_1, t_2, \ldots, t_m \in R$ (not necessarily all distinct). But this is equivalent to saying that $\otimes$ satisfies all the constraints in $\mathcal{IP}(\Gamma, m)$, so the result follows.

Example 7. Consider the relation R_1 over $D = \{0, 1, 2\}$, defined in Example 2.

The indicator problem for $\{R_1\}$ of order 1, defined in Example 4, has 2 solutions, which may be expressed in tabular form as follows:

	Variables		
	$\langle 0 \rangle$	$\langle 1 \rangle$	$\langle 2 \rangle$
Solution 1	0	0	0
Solution 2	0	1	2

One of these solutions is a constant operation, so CSP($\{R_1\}$) is tractable, by Proposition 9 of [9]. In fact, any problem in CSP($\{R_1\}$) has the solution which assigns the value 0 to each variable, so this class of problems is trivial.

The indicator problem for $\{R_1\}$ of order 2, defined in Example 4, has 4 solutions, which may be expressed in tabular form as follows:

	Variables								
	$\langle 0,0 \rangle$	$\langle 0,1 \rangle$	$\langle 0,2 \rangle$	$\langle 1,0 \rangle$	$\langle 1,1 \rangle$	$\langle 1,2 \rangle$	$\langle 2,0 \rangle$	$\langle 2,1 \rangle$	$\langle 2,2 \rangle$
Solution 1	0	0	0	0	0	0	0	0	0
Solution 2	0	1	2	0	1	2	0	1	2
Solution 3	0	0	0	1	1	1	2	2	2
Solution 4	0	0	0	0	1	0	0	0	2

The first of these solutions is a constant operation, and the second and third are essentially unary operations. However, the fourth solution shown in the table is more interesting. It is easily checked that this operation is an associative, commutative, idempotent (ACI) binary operation, so we have a second proof that CSP($\{R_1\}$) is tractable, by Theorem 16 of [9]. Furthermore, this result shows that R_1 may be combined with any other relations (of any arity) which are also closed under this ACI operation to obtain larger tractable problem classes.

Example 8. Consider the relation R_2 over $D = \{0, 1, 2\}$, defined in Example 2.

The indicator problem for $\{R_2\}$ of order 1, defined in Example 5, has 3 solutions, which may be expressed in tabular form as follows:

	Variables		
	$\langle 0 \rangle$	$\langle 1 \rangle$	$\langle 2 \rangle$
Solution 1	0	1	2
Solution 2	1	2	0
Solution 3	2	0	1

The indicator problem for $\{R_2\}$ of order 3, has a very large number of solutions, including the operation $\triangle$, defined in Example 3. Hence, we know that CSP(Γ) is tractable by Theorem 13 of [9]. Furthermore, R_2 may be combined with any other relations which are also closed under this operation to obtain larger tractable problem classes.

In order to draw conclusions about the tractability of a set of constraints we need to be able to distinguish closure operations which are essentially unary from those which are not. By Definition 10, this means checking whether there exists some coordinate position which has the property that when the input value in that position is constant, then the value of the operation remains constant. This checking may be carried out in $m|D|^m$ steps.

Alternatively, the next result shows that the *number of solutions* to certain indicator problems provides a sufficient condition for establishing NP-completeness, without needing to examine the individual solutions in detail.

Corollary 15. *Let Γ be a set of relations over domain D, let S_1 be the set of non-constant solutions to $\mathcal{IP}(\Gamma, 1)$, and let S_m be the set of solutions to $\mathcal{IP}(\Gamma, m)$, where $m = \max\{3, |D|\}$.*

If $|S_m| \leq m|S_1|$, then CSP(Γ) *is NP-complete.*

Proof. By Theorem 11 and Theorem 12, we know that either CSP(Γ) is NP-complete, or Γ is closed under some operation with arity at most m which is not essentially unary.

By Theorem 14, S_m is the set of all m-ary operations under which Γ is closed. By Definition 10, the number of these which are essentially unary is equal to the number of non-constant unary operations under which Γ is closed, multiplied by m. By Theorem 14, this number is $m|S_1|$. Hence, $|S_m| \geq m|S_1|$ in all cases.

In the limiting case, when $|S_m| = m|S_1|$, we know that every element of S_m is essentially unary, so Γ is not closed under any m-ary operation which is *not* essentially unary. It follows that Γ is not closed under any operation of arity lower than m which is not essentially unary, so CSP(Γ) must be NP-complete.

When $|D| = 2$, the converse result also holds [8]. Hence solving the indicator problems of order 1 and order 3 provides a simple and complete test for tractability of any set of relations over a domain with 2 elements. This answers a question posed by Schaefer in 1978 [16] concerning the existence of an efficient test for tractability in the GENERALISED SATISFIABILITY problem. Note that carrying out the test requires finding the number of solutions to a constraint satisfaction problem with just 8 Boolean variables.

For larger domains, Corollary 15 establishes NP-completeness for many sets of constraints without the need for individually constructed reduction arguments, as the following examples illustrate.

Example 9. Consider the relations R_1 and R_2 over $D = \{0, 1, 2\}$, defined in Example 2.

The indicator problem for $\{R_1, R_2\}$ of order 1, defined in Example 6, has 1 solution, corresponding to the identity operation.

The indicator problem for $\{R_1, R_2\}$ of order 3, has 3 solutions (which are all essentially unary).

Hence CSP($\{R_1, R_2\}$) is NP-complete, by Corollary 15.

Example 10. Consider the relations R_1, R_2, R_3 and R_4 over $D = \{0, 1, 2\}$, defined in Example 2.

By counting the solutions to the indicator problems of order 1 and order 3 for each relation and each pair of distinct relations in this set, we are able to complete the analysis of the complexity of CSP(Γ) for each possible subset Γ of these relations.

Relations Γ	# Solutions $\mathcal{IP}(\Gamma, 1)$ (non-constant)	# Solutions $\mathcal{IP}(\Gamma, 3)$	Complexity of CSP(Γ)
$\{R_1\}$	1	12	Trivial (Example 7)
$\{R_2\}$	3	$\gg 9$	Polynomial (Example 8)
$\{R_3\}$	10	$\gg 30$	Trivial
$\{R_4\}$	6	18	NP-complete (Example 1)
$\{R_1, R_2\}$	1	3	NP-complete (Corollary 15)
$\{R_1, R_3\}$	1	3	NP-complete (Corollary 15)
$\{R_1, R_4\}$	1	3	NP-complete (Example 1)
$\{R_2, R_3\}$	1	3	NP-complete (Corollary 15)
$\{R_2, R_4\}$	1	3	NP-complete (Example 1)
$\{R_3, R_4\}$	2	6	NP-complete (Example 1)

For all larger sets of relations $\Gamma \subseteq \{R_1, R_2, R_3, R_4\}$, we have that Γ contains at least one of the pairs of relations shown in the table, and so CSP(Γ) is NP-complete.

How practical is the test proposed here in general? For any set of relations Γ over domain D, the indicator problem of order m has $|D|^m$ variables and $\sum_{R \in \Gamma} |R|^m$ constraints. Hence, for small values of $|D|$ the size of the relevant indicator problems is very small, and the solutions may be found easily. This remains true even when the arity of the relations in Γ is large.

As the domain size increases, the size of the indicator problems increases rapidly, and it becomes impractical to compute all solutions. On the other hand, for cases of interest, it may be possible to establish from known properties of

the constraints that the relevant indicator problems will have particular types of solution, without carrying out a complete solution algorithm. This question is currently being investigated.

5 Conclusion

In this paper we have shown how the algebraic properties of relations may be used to distinguish between sets of relations which give rise to tractable constraint satisfaction problems and those which give rise to NP-complete classes of problems.

In particular, we have established that any set of relations which does not give rise to an NP-complete class of constraint satisfaction problems must be closed under some operation which is not essentially unary.

Furthermore, we have proposed a method for determining the operations under which a set of relations is closed by solving a particular form of constraint satisfaction problem, which we have called an indicator problem.

For problems where the domain contains just two elements these results provide a necessary and sufficient condition for tractability (assuming that P is not equal to NP), and an efficient test to distinguish the tractable sets of relations.

For problems with larger domains the closure condition we have described is a necessary condition for tractability. The converse of this condition provides a sufficient condition for NP-completeness, which we have shown is widely applicable and easy to test.

We are now investigating the application of these results to particular problem types, such as temporal problems involving subsets of the interval algebra. We are also attempting to determine how the presence of particular algebraic closure properties in the constraints may be used to derive appropriate efficient algorithms for tractable problem classes.

References

1. Codd, E.F., "A Relational Model of Data for Large Shared Databanks", *Communications of the ACM 13* (1970), pp. 377–387.
2. Cooper, M.C., Cohen, D.A., Jeavons, P.G., "Characterizing tractable constraints", *Artificial Intelligence 65*, (1994), pp. 347–361.
3. Dechter, R., & Pearl, J., "Structure identification in relational data", *Artificial Intelligence 58* (1992) pp. 237–270.
4. Dechter, R. & Pearl J. "Network-based heuristics for constraint-satisfaction problems", *Artificial Intelligence 34* (1988), pp. 1–38.
5. Freuder, E.C., "A sufficient condition for backtrack-bounded search", *Journal of the ACM 32* (1985) pp. 755–761.
6. Garey, M.R., & Johnson, D.S., *Computers and intractability: a guide to NP-completeness*, Freeman, San Francisco, California, (1979).
7. Gyssens, M., Jeavons, P., Cohen, D., "Decomposing constraint satisfaction problems using database techniques", *Artificial Intelligence 66*, (1994), pp. 57–89.

8. Jeavons, P.G., "An algebraic characterization of tractable constraints", Technical Report CSD-TR-95-05, Royal Holloway, University of London (1995).
9. Jeavons, P., Cohen D., Gyssens, M., "A unifying framework for tractable constraints", In *Proceedings 1st International Conference on Principles and Practice of Constraint Programming—CP '95* (Cassis, France, September 1995), *Lecture Notes in Computer Science*, 976, Springer-Verlag, Berlin/New York, 1995, pp. 276–291.
10. Jeavons, P.G., & Cooper, M.C., "Tractable constraints on ordered domains", *Artificial Intelligence 79* (1996), pp. 327–339.
11. Kirousis, L., "Fast parallel constraint satisfaction", *Artificial Intelligence 64*, (1993), pp. 147–160..
12. Ladkin, P.B., & Maddux, R.D., "On binary constraint problems", *Journal of the ACM 41* (1994), pp. 435–469.
13. Mackworth, A.K. "Consistency in networks of relations", *Artificial Intelligence 8* (1977) pp. 99–118.
14. Montanari, U., "Networks of constraints: fundamental properties and applications to picture processing", *Information Sciences 7* (1974), pp. 95–132.
15. Montanari, U., & Rossi, F., "Constraint relaxation may be perfect", *Artificial Intelligence 48* (1991), pp. 143–170.
16. Schaefer, T.J., "The complexity of satisfiability problems", *Proc 10th ACM Symposium on Theory of Computing (STOC)* , (1978) pp. 216–226.
17. Szendrei, A., *Clones in Universal Algebra*, Seminaires de Mathematiques Superieures 99, University of Montreal, (1986).
18. van Beek, P., "On the Minimality and Decomposability of Row-Convex Constraint Networks", *Proceedings of the Tenth National Conference on Artificial Intelligence, AAAI-92*, MIT Press, (1992) pp. 447–452.
19. Van Hentenryck, P., Deville, Y., Teng, C-M., "A generic arc-consistency algorithm and its specializations", *Artificial Intelligence 57* (1992), pp. 291–321.

Combination of Constraint Systems II: Rational Amalgamation*

Stephan Kepser, Klaus U. Schulz

CIS, Universität München
Oettingenstraße 67, 80538 München, Germany
e-mail: kepser/schulz@cis.uni-muenchen.de

Abstract. In a recent paper[2], the concept of "free amalgamation" has been introduced as a general methodology for interweaving solution structures for symbolic constraints, and it was shown how constraint solvers for two components can be lifted to a constraint solver for the free amalgam. Here we discuss a second general way for combining solution domains, called *rational amalgamation*. In praxis, rational amalgamation seems to be the preferred combination principle if the two solution structures to be combined are "rational" or "non-wellfounded" domains. It represents, e.g., the way how rational trees and rational lists are interwoven in the solution domain of Prolog III, and a variant has been used by W. Rounds for combining feature structures and hereditarily finite non-wellfounded sets. We show that rational amalgamation is a general combination principle, applicable to a large class of structures. As in the case of free amalgamation, constraint solvers for two component structures can be combined to a constraint solver for their rational amalgam. From this algorithmic point of view, rational amalgamation seems to be interesting since the combination technique for rational amalgamation avoids one source of non-determinism that is needed in the corresponding scheme for free amalgamation.

1 Introduction

One idea behind constraint solving is to use specialized formalisms and inference mechanisms to solve domain-specific tasks. In many applications, however, one is faced with a complex combination of different problems, which means that a system tailored to solving a single problem can only be applied if it is possible to combine it with other specialized systems. The present paper, as its predecessor [BS95], marks one step in a program where we try to characterize the most important *general constructions* for combining solution domains and constraint solvers for symbolic constraints. A general combination method, in our sense, has to give answers to two problems. First, it must offer a general construction for combining two solution domains. Second, a combination algorithm has

* This work was supported by a DFG grant (SSP "Deduktion") and by the EC Working Group CCL, EP6028.

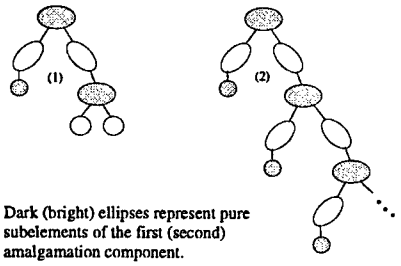
[2] see [BS95].

to be given that reduces the problem of solving "mixed" constraints over the combined solution domain to the problem of solving "pure" constraints over the two component structures. We think that it is in fact possible to characterize a small set of fundamental combination methods that describe—modulo minor deviations—all known instances of combined symbolic constraint systems in this sense.

In [BS95] the notion of the *free amalgamated product* of two component structures was introduced. This product is characterized by a universality-property: it represents a most general object among all structures that can be considered as a reasonable combination of the two components. For SC-structures over disjoint signatures an explicit construction of the free amalgamated product of two components was given and it was shown how given constraint solvers for the component structures can be combined to a constraint solver for the free amalgam.

In the present paper we introduce a second systematic way to combine constraint systems over SC-structures, called *rational amalgamation.* Free and rational amalgamation both yield a combined structure with "mixed" elements that interweave a finite number of "pure" elements of the two components in a particular way. The difference between both constructions becomes transparent when we ignore the interior structure of these pure subelements and consider them as construction units with a fixed arity, similar to "complex function symbols". Under this perspective, and ignoring details, mixed elements of the free amalgam can be considered as finite trees, whereas mixed elements of the rational amalgam are like rational trees.

Mixed element of free amalgam (1) and of rational amalgam (2).



Dark (bright) ellipses represent pure subelements of the first (second) amalgamation component.

On this background it should not be surprising that in praxis rational amalgamation appears to be the preferred combination principle in situations where the two solution structures to be combined are themselves "rational" or "cyclic" domains: for example, it represents the way how rational trees and rational lists are interwoven in the solution domain of Prolog III ([Co90]), and a variant of rational amalgamation has been used to combine feature structures with non-wellfounded sets in a system introduced by W. Rounds [Ro88].

We introduce rational amalgamation as a general construction that can be used to combine so-called non-collapsing SC-structures over disjoint signatures. It is then shown how constraint solving in the rational amalgam can be reduced

to constraint solving in the components. The decomposition scheme that is used is closely related to the decomposition algorithm for free amalgamation, but it avoids one highly non-deterministic step that is needed in the latter scheme. Hence, when matters of efficiency become important, rational amalgamation might be the better choice.

Let us now briefly indicate which insights could be gained from a classification of the basic methodologies for combining constraints systems. Below we shall summarize what has been obtained so far.

1. It helps to understand the scale of possibilities and the general limitations for combining constraints systems.
2. It might facilitate the design of new combined constraint systems, and it helps to understand existing instances of combination from a general point of view.
3. It establishes new and interesting connections between the theory of constraint solving and other areas such as, e.g., universal algebra and logic.
4. The relationship between different methodologies for combining constraint systems is interesting per se, we hope to verify.

1. From our present perspective, which is explained in the conclusion, free and rational amalgamation, and a related construction called "infinite amalgamation" seem to be the most important combination principles in a spectrum of related methods. Furthermore, we are confident that the abstract definition of an SC-structure, as introduced in [BS95] and used here, captures a maximal class of (unsorted!) structures where these combination principles can be applied in a uniform way. This class covers most of the non-numerical and non-finite solution domains that are used in constraint solving. All the solution domains that are considered in the area of unification modulo equational theories are SC-structures. Furthermore, the algebra of rational trees, feature structures, and structures with finite or rational nested sets, lists and multisets are SC-structures.
2. The techniques that are described in this paper show, e.g., that there is a common and general methodology behind Colmerauer's combination of rational trees and rational lists in the solution domain of Prolog III ([Co90]) and Rounds' [Ro88] combination of feature structures with non-wellfounded sets. The abstract techniques described in the present paper can be used, e.g., to obtain similar combinations that mix rational trees, feature structures, rational lists, nested multi-sets, or quotient term algebras for collapse-free equational theories over disjoint signatures in arbitrary manner.
3. The definition of an SC-structure properly generalizes the notion of a free structure. Still, SC-structures have what is sometimes called the "unique mapping property" of free structures, and a major part of the theory of free structures as developed in universal algebra can be lifted to the case of SC-structures. A detailled (mathematical) investigation of this point is in progress. Furthermore, it has turned out that the methods for combining solution domains developed in [BS95] and here, and the general methods for combining logics described by

Gabbay [Ga96] and Pfalzgraf [Pf91, PS94] follow the very same abstract idea. See [Ga96] for a first discussion of this issue.

The conclusion will be used to comment on item 4. For lack of space we had to omit proofs. All proofs can be found in the long version [KS96] of the paper.

2 Preliminaries

A signature Σ consists of a set Σ_F of function symbols and a disjoint set Σ_P of predicate symbols (not containing "="), each of fixed arity. The signatures considered in this paper may be finite or countably infinite. Expressions $\mathcal{A}^\Sigma$ denote Σ-structures over the carrier set A, and f_A (p_A) stands for the interpretation of $f \in \Sigma_F$ ($p \in \Sigma_P$) in $\mathcal{A}^\Sigma$. Σ-terms $(t, t_1, \ldots)$ and atomic Σ-formulas (of the form $t_1 = t_2$, or of the form $p(t_1, \ldots, t_n)$) are built as usual. A Σ-formula φ is written in the form $\varphi(v_1, \ldots, v_n)$ in order to indicate that the set $Var(\varphi)$ of free variables of φ is a subset of $\{v_1, \ldots, v_n\}$. We write $\mathcal{A}^\Sigma \models \varphi(v_1/a_1, \ldots, v_n/a_n)$ if φ becomes true in $\mathcal{A}^\Sigma$ under all assignments that map v_i to $a_i \in A$, for $1 \leq i \leq n$.

Σ-homomorphisms, Σ-isomorphisms, and Σ-endomorphisms are defined as usual, see e.g. [Ma71, BS95]. With End_A^Σ we denote the monoid of all endomorphisms of $\mathcal{A}^\Sigma$, with composition as operation. If $g : A \to B$ and $h : B \to C$ are mappings, then $g \circ h : A \to C$ denotes their composition. Expressions like $\vec{v}, \vec{a}$ are used to denote finite sequences. If $\vec{a} = a_1, \ldots, a_n$ is a sequence of elements of A and if m is a mapping with domain A, then $m(\vec{a})$ denotes the sequence $m(a_1), \ldots, m(a_n)$. If $\vec{v} = v_1, \ldots, v_n$, then $\mathcal{A}^\Sigma \models \varphi(\vec{v}/\vec{a})$ is shorthand for $\mathcal{A}^\Sigma \models \varphi(v_1/a_1, \ldots, v_n/a_n)$. The symbol "$\uplus$" denotes disjoint set union.

3 Non-collapsing SC-structures

In this section we introduce the class of structures for which we can use the rational amalgamation construction (Definition 7). First we recall the definition of SC-structures given in [BS95]. We consider a fixed Σ-structure $\mathcal{A}^\Sigma$, and $\mathcal{M}$ always denotes a submonoid of End_A^Σ.

Definition 1. Let A_0, A_1 be subsets of $\mathcal{A}^\Sigma$. Then A_0 *stabilizes* A_1 *with respect to* $\mathcal{M}$ iff all elements m_1 and m_2 of $\mathcal{M}$ that coincide on A_0 also coincide on A_1. For $A_0 \subseteq A$ the *stable hull of* A_0 *with respect to* $\mathcal{M}$ is the set

$$SH_{\mathcal{M}}^{\mathcal{A}}(A_0) := \{a \in A \mid A_0 \text{ stabilizes } \{a\} \text{ with respect to } \mathcal{M}\}.$$

$SH_{\mathcal{M}}^{\mathcal{A}}(A_0)$ is always a Σ-substructure of $\mathcal{A}^\Sigma$, and $A_0 \subseteq SH_{\mathcal{M}}^{\mathcal{A}}(A_0)$. The stable hull of A_0 can be larger than the Σ-subalgebra generated by A_0.

Definition 2. The set $X \subseteq A$ is an $\mathcal{M}$-*atom set for* $\mathcal{A}^\Sigma$ if every mapping $X \to A$ can be extended to an endomorphism in $\mathcal{M}$.

Definition 3. A countably infiniteΣ-structure $\mathcal{A}^\Sigma$ is an *SC-structure* (simply combinable structure) iff there exists a submonoid $\mathcal{M}$ of $End_{\mathcal{A}}^{\Sigma}$ such that $\mathcal{A}^\Sigma$ has an infinite $\mathcal{M}$-atom set X where every $a \in A$ is stabilized by a finite subset of X with respect to $\mathcal{M}$. We denote this SC-structure by $(\mathcal{A}^\Sigma, \mathcal{M}, X)$. If $\mathcal{M} = End_{\mathcal{A}}^{\Sigma}$, then $(\mathcal{A}^\Sigma, End_{\mathcal{A}}^{\Sigma}, X)$ is called a *strong SC-structure*.

Example 1. The class of SC-structures contains, e.g., all free structures (see, e.g., [Ma71]), rational tree algebras ([Co84, Ma88]), feature structures (for specificity, we refer to [AP94]), feature structures with arity ([ST94, BT94]), domains with nested, finite or rational lists (rational lists are used in Prolog III, see [Co90]), and domains with nested, finite or rational sets (as introduced in [Ac88] and used in [Ro88]).[3] In each case, we have to take the non-ground variant since we assume the existence of a countably infinite set of atoms. With the exception of feature structures, all these structures are strong SC-structures. For details we refer to [BS95].

In the rest of this section, $(\mathcal{A}^\Sigma, \mathcal{M}, X)$ denotes a fixed SC-structure.

Lemma 4. *Let $\varphi(v_1, \ldots, v_k)$ be a positive Σ-formula, $m \in \mathcal{M}$, let $a_1, \ldots, a_k \in A$. Then $\mathcal{A}^\Sigma \models \varphi(v_1/a_1, \ldots, v_k/a_k)$ implies $\mathcal{A}^\Sigma \models \varphi(v_1/m(a_1), \ldots, v_k/m(a_k))$.*

A fundamental property of SC-structures is the following ([BS95], Lemma 13): for each $a \in A$ there exists a *unique minimal* finite set $Y \subseteq X$ such that $a \in SH_{\mathcal{M}}^{\mathcal{A}}(Y)$.

Definition 5. The *stabilizer of $a \in A$ with respect to $\mathcal{M}$*, $Stab_{\mathcal{M}}^{\mathcal{A}}(a)$, is the unique minimal finite subset Y of X such that $a \in SH_{\mathcal{M}}^{\mathcal{A}}(Y)$. The stabilizer of $A' \subseteq A$ is the set $Stab_{\mathcal{M}}^{\mathcal{A}}(A') := \bigcup_{a \in A'} Stab_{\mathcal{M}}^{\mathcal{A}}(a)$.

The next lemma plays a crucial role in the rational amalgamation construction. It is used in many proofs.

Lemma 6. *Let $m \in \mathcal{M}$ be an endomorphism of the SC-structure $(\mathcal{A}^\Sigma, \mathcal{M}, X)$ such that the restriction of m on X is a mapping $X \to X$. If $Stab_{\mathcal{M}}^{\mathcal{A}}(a) = \{x_1, \ldots, x_k\}$, then $Stab_{\mathcal{M}}^{\mathcal{A}}(m(a)) \subseteq \{m(x_1), \ldots, m(x_k)\}$. If m is an automorphism, then $Stab_{\mathcal{M}}^{\mathcal{A}}(m(a)) = \{m(x_1), \ldots, m(x_k)\}$.*

We can now characterize the subclass of SC-structures for which we can use the rational amalgamation construction.

Definition 7. An SC-structure $(\mathcal{A}^\Sigma, \mathcal{M}, X)$ is *non-collapsing* if every endomorphism $m \in \mathcal{M}$ maps non-atoms to non-atoms (i.e., $m(a) \in A \setminus X$ for all $a \in A \setminus X$ and all $m \in \mathcal{M}$).

E.g., quotient term algebras for collapse-free equational theories (see [BS94]), rational tree algebras, feature structures, feature structures with arity, and the domains with nested, finite or rational lists (as mentioned in 1) are always non-collapsing.

[3] The signatures of these structures may be finite or countably infinite, because each element is a finite or rational "tree" and hence composed using a finite part of the signature only. This guarantees that the complete structure is countably infinite, as demanded in Definition 3.

4 The Domain of the Rational Amalgam

The definition of the underlying domain is the most complicated step of the rational amalgamation construction. The description would be much simpler if we would restrict the construction to components where the elements have a particular form (e.g., the form of trees). But such a restriction would contradict our motivation to describe a general construction. We shall first introduce the notion of a "braid" and its standard normal form. The set of braids in standard normal form will represent the carrier of the rational amalgam. We shall describe the rational amalgamation of two component structures. There are, however, no difficulties to interweave any finite number of components in the same way.

Throughout this section $(\mathcal{A}^\Sigma, \mathcal{M}, X)$ and $(\mathcal{B}^\Delta, \mathcal{N}, Y)$ denote two fixed non-collapsing SC-structures over disjoint signatures. The atom sets X and Y have the form $X = Z \uplus \mathcal{O}_A$ and $Y = Z \uplus \mathcal{O}_B$, where the sets $Z, \mathcal{O}_A$, and $\mathcal{O}_B$ are all infinite, and where $\mathcal{O}_A \cap \mathcal{O}_B = \emptyset$. The atoms in Z will be called *bottom atoms*, the atoms in $\mathcal{O}_A$ ($\mathcal{O}_B$) will be called *open atoms*. In the braid construction, the bottom atoms will play the role of ordinary atoms, or leaves. Open atoms, in contrast, are only used to connect elements of both structures. With $\mathcal{O}_A(a)$ and $\mathcal{O}_A(A')$ we denote the set of open atoms occurring in the stabilizer of $a \in A$ ($A' \subseteq A$) with respect to $\mathcal{M}$. Similarly expressions $\mathcal{O}_B(b)$ ($\mathcal{O}_B(B')$) are used to denote the set of open atoms occurring in the stabilizer of $b \in B$ ($B' \subseteq B$) with respect to $\mathcal{N}$.

Definition 8. Let $\mathcal{O}'_A \subseteq \mathcal{O}_A$, $\mathcal{O}'_B \subseteq \mathcal{O}_B$, let $\pi_A : \mathcal{O}'_A \to B$, $\pi_B : \mathcal{O}'_B \to A$, let $\pi := \pi_A \cup \pi_B$. An element $a \in A$ is *directly linked to* $b \in B$ via π if there is an $o \in \mathcal{O}_B(b)$ such that $a = \pi_B(o)$. Analogously $b \in B$ is directly linked to $a \in A$ via π if there exists an $o \in \mathcal{O}_A(a)$ such that $b = \pi_A(o)$. An element $a \in A \cup B$ is a *π-descendant of* $b \in A \cup B$ if there exists a sequence $a = a_0, a_1, \ldots, a_n = b$ ($n \geq 0$) such that each a_i is directly linked to a_{i+1} via π, for $0 \leq i \leq n-1$.

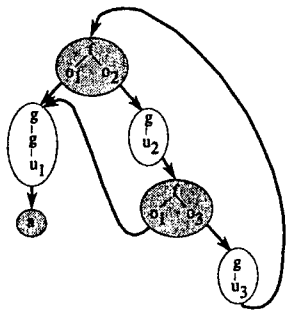
Definition 9. A *braid* of type A over $\mathcal{A}^\Sigma, \mathcal{B}^\Delta$ is a quintuple $\mathcal{K} = \langle a, C, D, \pi_A, \pi_B \rangle$, where

1. $a \in A \setminus \mathcal{O}_A$,
2. C is a finite subset of A containing a. All elements of $C \setminus \{a\}$ are *non-atomic*. D is a finite set of *non-atomic* elements of B,
3. $\pi_A : \mathcal{O}_A(C) \to D$ and $\pi_B : \mathcal{O}_B(D) \to C$ are mappings. For $\langle o, e \rangle \in \pi_A \cup \pi_B$, e is always a *non-atomic* element,
4. each element in $C \cup D$ is a π-descendant of a, for $\pi := \pi_A \cup \pi_B$.

The element a is called the *root* of $\mathcal{K}$. The elements in the sets C and D are called the *elements of $\mathcal{K}$ of type A and B* respectively. The functions π_A and π_B are called the *linking functions of $\mathcal{K}$ of type A and B* respectively.

Braids of type B, with root in $B \setminus \mathcal{O}_B$, are defined symmetrically. A braid $\mathcal{K}$ is called *trivial* if the root of $\mathcal{K}$ is a bottom atom $z \in Z$. In this case, z is the only element of the braid. It does not make sense to distinguish between the trivial braid $\langle z, \{z\}, \emptyset, \emptyset, \emptyset \rangle$ of type A and the trivial braid $\langle z, \emptyset, \{z\}, \emptyset, \emptyset \rangle$ of type B. We identify both braids. Hence, trivial braids have mixed type.

Example 2. The following figure represents a braid over two termalgebras, for signatures $\Sigma = \{f, a\}$ and $\Delta = \{g\}$ respectively.



We sometimes write $\mathcal{O}_A(\mathcal{K})$ and $\mathcal{O}_B(\mathcal{K})$ for $\mathcal{O}_A(C)$ and $\mathcal{O}_B(D)$ respectively, and $\mathcal{O}(\mathcal{K})$ denotes the union $\mathcal{O}_A(\mathcal{K}) \cup \mathcal{O}_B(\mathcal{K})$. A quintuple $\mathcal{K} = \langle a, C, D, \pi_A, \pi_B \rangle$ that satisfies Conditions 1-3 of Definition 9 will be called a *prebraid.*

Definition 10. Let $\mathcal{K} = \langle a, C, D, \pi_A, \pi_B \rangle$ be a braid. Then the braid $\mathcal{K}' := \langle a', C', B', \pi'_A, \pi'_B \rangle$ (of type A or B) is a *subbraid* of $\mathcal{K}$ if $a' \in C \cup D$, $C' \subseteq C$, $D' \subseteq D$, $\pi'_A \subseteq \pi_A$, and $\pi'_B \subseteq \pi_B$.

Sub(pre)braids of prebraids are defined in the same way.

Lemma 11. *For each element e of a prebraid $\mathcal{K}$ there exists a unique subbraid of $\mathcal{K}$ with root e.*

The concrete open atoms that are used to organize links between elements of distinct type in a given braid should be regarded as irrelevant. The following, purely algebraic notions are used to formalize this idea. An endomorphism $m \in \mathcal{M}$ ($n \in \mathcal{N}$) is called *admissible* if m (n) leaves all bottom atoms $z \in Z$ fixed and if $m(o) \in \mathcal{O}_A$ ($n(o) \in \mathcal{O}_B$) for all $o \in \mathcal{O}_A$ ($o \in \mathcal{O}_B$).[4] Automorphisms are called admissible if they define a permutation of the set of open atoms while leaving bottom atoms fixed. A pair $(m, n) \in \mathcal{M} \times \mathcal{N}$ is called admissible if both m and n are admissible.

Definition 12. Let $\mathcal{K} = \langle a, C, D, \pi_A, \pi_B \rangle$ and $\mathcal{K}' = \langle a', C', D', \pi'_A, \pi'_B \rangle$ be two prebraids, say, of type A. $\mathcal{K}'$ is called a *variant* of $\mathcal{K}$ if there exists an admissible pair of automorphisms (m, n) such that $a' = m(a)$, $C' = \{m(c) \mid c \in C\}$, $D' = \{n(d) \mid d \in D\}$, $\pi'_A := \{\langle m(o), n(d) \rangle \mid \langle o, d \rangle \in \pi_A\}$, and $\pi'_B := \{\langle n(o), m(c) \rangle \mid \langle o, c \rangle \in \pi_B\}$.

Two (pre)braids that are variants of each other are meant to denote the same object. But then we should not distinguish between two subbraids of one and the same (pre)braid if they are variants. In order to identify such subbraids, we use admissible pairs of endomorphisms of a particular type.

[4] Intuitively, admissible endomorphisms cause a "renaming" of open atoms, compare Lemma 6. They may identify distinct open atoms.

Definition 13. The admissible pair (m, n) is a *simplifier* for the prebraid $\mathcal{K} = \langle a, C, D, \pi_A, \pi_B \rangle$ if the following conditions hold:

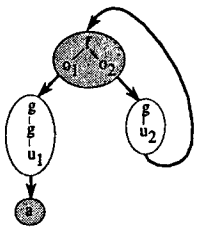
- $\forall o_1, o_2 \in \mathcal{O}_A(C)$: $m(o_1) = m(o_2)$ implies $n(\pi_A(o_1)) = n(\pi_A(o_2))$,
- $\forall o_1, o_2 \in \mathcal{O}_B(D)$: $n(o_1) = n(o_2)$ implies $m(\pi_B(o_1)) = m(\pi_B(o_2))$.

Definition 14. Let (m, n) be a simplifier for the prebraid $\mathcal{K} = \langle a, C, D, \pi_A, \pi_B \rangle$. The *image of* $\mathcal{K}$ *with respect to* (m, n) is the prebraid $\mathcal{K}^{(m,n)} := \langle a', C', D', \pi'_A, \pi'_B \rangle$ with the following components:[5]

1. $a' := m(a)$,
2. $C' := \{m(c) \mid c \in C\}$ and
   $D' := \{n(d) \mid d \in D\}$,
3. $\pi'_A := \{\langle m(o), n(d) \rangle \mid \langle o, d \rangle \in \pi_A, m(o) \in \mathcal{O}_A(C')\}$, and
   $\pi'_B := \{\langle n(o), m(c) \rangle \mid \langle o, c \rangle \in \pi_B, n(o) \in \mathcal{O}_B(D')\}$.

Now assume that $\mathcal{K}$ is a braid. The *braid-image of* $\mathcal{K}$ *with respect to* (m, n), $\mathcal{K}^{\langle m,n \rangle}$, is the unique subbraid of $\mathcal{K}^{(m,n)}$ with root a'.[6]

Example 3. Here is the braid-image of the braid from Example 2 under the simplification (m, n) where m maps o_3 to o_2 and n maps u_3 to u_2:



Lemma 15. *Let (m, n) be a simplifier for the prebraid $\mathcal{K}$. If the restrictions of m and n on $\mathcal{O}_A(\mathcal{K})$ and $\mathcal{O}_B(\mathcal{K})$ respectively are injective, then $\mathcal{K}^{(m,n)}$ is a variant of $\mathcal{K}$.*

Call a simplifier (m, n) for $\mathcal{K}$ *strict* if the restriction of m on $\mathcal{O}_A(\mathcal{K})$ or the restriction of n on $\mathcal{O}_B(\mathcal{K})$ is *not* injective. A prebraid $\mathcal{K}'$ is called *irreducible* if $\mathcal{K}'$ does not have a strict simplifier.

[5] Using Lemma 6 and the fact that both $\mathcal{A}^{\Sigma}$ and $\mathcal{B}^{\Delta}$ are non-collapsing it is trivial to verify that $\mathcal{K}^{(m,n)}$ is a prebraid.

[6] We would like to mention here one technical point behind the definition of a simplifier. The set
$(\{m(o) \mid o \in \mathcal{O}_A(C)\} \setminus \mathcal{O}_A(C')) \cup (\{n(o) \mid o \in \mathcal{O}_B(D)\} \setminus \mathcal{O}_B(D'))$ is called the set of *pending atoms* of the simplification step leading from $\mathcal{K}$ to $\mathcal{K}^{(m,n)}$. Pending atoms may in fact occur (compare the inclusion mentioned in Lemma 6). This phenomenon can be used to show that image and braid-image may really be different, and it is the source of many technical problems for the mathematical treatment of simplification.

Lemma 16. *(a) If the prebraid $\mathcal{K} = \langle a, C, D, \pi_A, \pi_B \rangle$ is irreducible, then π_A and π_B are injective.*
(b) If $\mathcal{K}'$ is a subbraid of the irreducible prebraid $\mathcal{K}$, then $\mathcal{K}'$ is irreducible.
(c) If $\mathcal{K}_1$ and $\mathcal{K}_2$ are subbraids of the irreducible prebraid $\mathcal{K}$, and if $\mathcal{K}_1$ and $\mathcal{K}_2$ are variants, then $\mathcal{K}_1 = \mathcal{K}_2$.

The following theorem represents the main result about simplification of braids.[7]

Theorem 17. *Let $\mathcal{K} = \mathcal{K}_0, \mathcal{K}_1, \ldots, \mathcal{K}_k$ be a sequence of braids such that each braid $\mathcal{K}_{i+1}$ is the braid-image of $\mathcal{K}_i$ under a strict simplification, for $i = 0, \ldots, k-1$. Then $k \leq |\mathcal{O}(\mathcal{K})|$. If $\mathcal{K}'$ is an irreducible braid that is reached from $\mathcal{K}$ by a sequence of consecutive simplification steps (always taking braid images), then there exists a simplifier (m, n) for $\mathcal{K}$ such that $\mathcal{K}^{\langle m,n \rangle} = \mathcal{K}'$. If two irreducible braids $\mathcal{K}_1$ and $\mathcal{K}_2$ can be reached from $\mathcal{K}$ by sequences of consecutive simplification steps (always taking braid-images), then $\mathcal{K}_1$ and $\mathcal{K}_2$ are variants.*

On the basis of Theorem 17 it is simple to see that we may introduce the following equivalence relation on the set of all braids.

Definition 18. Two braids are called *equivalent* if they can be simplified to the same irreducible braid image. If $\mathcal{K}$ is a braid, $[\mathcal{K}]$ denotes the set of all braids that are equivalent to $\mathcal{K}$.

Standard normalization

In order to define the underlying domain of the rational amalgam we shall now introduce a standard normal form for each braid. Let $\mathcal{O}_A^*$ be a subset of the set $\mathcal{O}_A$ of open atoms of A^Σ that has the same cardinality as the set of all equivalence classes of non-trivial[8] braids of type B. Similarly, let $\mathcal{O}_B^*$ be a subset of the set $\mathcal{O}_B$ of open atoms of B^Δ that has the same cardinality as the set of all equivalence classes of non-trivial braids of type A. Let $\mathcal{A}_*^\Sigma := SH_{\mathcal{M}}^{\mathcal{A}}(Z \cup \mathcal{O}_A^*)$, and let $\mathcal{B}_*^\Delta := SH_{\mathcal{N}}^{\mathcal{B}}(Z \cup \mathcal{O}_B^*)$. Lemma 10 of [BS95] shows

Lemma 19. *Every bijection between $Z \cup \mathcal{O}_A^*$ and $Z \cup \mathcal{O}_A$ extends to a Σ-isomorphism between $\mathcal{A}_*^\Sigma$ and $\mathcal{A}^\Sigma$. Similarly every bijection between $Z \cup \mathcal{O}_B^*$ and $Z \cup \mathcal{O}_B$ extends to a Δ-isomorphism between $\mathcal{B}_*^\Delta$ and $\mathcal{B}^\Delta$.*

We may now enumerate the elements of $\mathcal{O}_A^*$ and of $\mathcal{O}_B^*$ in the form

$$\mathcal{O}_A^* = \{o_{[\mathcal{K}]} \mid \mathcal{K} \text{ is a nontrivial braid of type } B\},$$
$$\mathcal{O}_B^* = \{o_{[\mathcal{K}]} \mid \mathcal{K} \text{ is a nontrivial braid of type } A\}.$$

This means that $[\mathcal{K}] \mapsto o_{[\mathcal{K}]}$ establishes a bijection between the set of all equivalence classes of non-trivial braids of type A (B) and $\mathcal{O}_B^*$ ($\mathcal{O}_A^*$). Let $\mathcal{K} =$

[7] The proof given in [KS96] is very technical, a corresponding result for simplification of prebraids is proved first.
[8] compare Definition 9.

$\langle a, C, D, \pi_A, \pi_B \rangle$ be a prebraid. For each open atom $o \in \mathcal{O}_A(C)$ ($o \in \mathcal{O}_B(D)$) we say that o *points in* $\mathcal{K}$ *to* $\mathcal{K}'$ iff $\mathcal{K}'$ is the unique subbraid of $\mathcal{K}$ with root $\pi_A(o)$ ($\pi_B(o)$)[9].

Definition 20. An irreducible prebraid $\mathcal{K}$ is in *standard normal form* if $\mathcal{O}_A(\mathcal{K}) \cup \mathcal{O}_B(\mathcal{K}) \subseteq \mathcal{O}_A^* \cup \mathcal{O}_B^*$ and if every open atom $o \in \mathcal{O}_A(\mathcal{K}) \cup \mathcal{O}_B(\mathcal{K})$ points in $\mathcal{K}$ to a subbraid $\mathcal{K}'$ such that $o = o_{[\mathcal{K}']}$.

With $A \odot B$ we denote the set of all braids over $\mathcal{A}^\Sigma$ and $\mathcal{B}^\Delta$ in standard normal form. Note that trivial braids are always in standard normal form.

Lemma 21. *Let $\mathcal{K}$ be a prebraid. Let (m, n) denote the admissible pair of endomorphisms that maps each $o \in \mathcal{O}_A(\mathcal{K}) \cup \mathcal{O}_B(\mathcal{K})$ to $o_{[\mathcal{K}']}$ where $\mathcal{K}'$ is the unique braid such that o points in $\mathcal{K}$ to $\mathcal{K}'$. Then (m, n) is a simplifier for $\mathcal{K}$ and $\mathcal{K}^{(m,n)}$ is in standard normal form.*

Definition 22. The process where we apply to a given (pre)braid $\mathcal{K}$ the simplifier (m, n) that maps each open atom $o \in \mathcal{O}(\mathcal{K})$, pointing in $\mathcal{K}$ to the subbraid $\mathcal{K}'$, to the open atom $o_{[\mathcal{K}']} \in \mathcal{O}_A^* \cup \mathcal{O}_B^*$ will be called *standard simplification* of $\mathcal{K}$. The prebraid $\mathcal{K}^{(m,n)}$ (the braid $\mathcal{K}^{(m,n)}$) will be called *the standard (braid) normal form* of $\mathcal{K}$.

Obviously all subbraids of a prebraid in standard normal form are again in standard normal form.

Lemma 23. *For each (pre)braid $\mathcal{K}$ there exists exactly one (pre)braid $\mathcal{K}'$ in standard normal form such that $\mathcal{K}$ and $\mathcal{K}'$ are equivalent.*

Lemma 24. *Given $e \in (A_* \cup B_*) \setminus (\mathcal{O}_A^* \cup \mathcal{O}_B^*)$ there exists a unique braid $\mathcal{K} \in A \odot B$ such that e is the root of $\mathcal{K}$.*

5 The rational amalgamated product

Given the underlying domain of the rational amalgam of $\mathcal{A}^\Sigma$ and $\mathcal{B}^\Delta$ as constructed above, there is now a perfectly natural way to introduce functions and relations that interpret the symbols of the mixed signature $\Sigma \cup \Delta$. Consider the two functions $root_A : A \odot B \to A_*$ and $root_B : A \odot B \to B_*$:

$$root_A(\mathcal{K}) := \begin{cases} \text{the root of } \mathcal{K} & \text{if } \mathcal{K} \text{ is trivial or has type } A \\ o_{[\mathcal{K}]} \in \mathcal{O}_A^* & \text{if } \mathcal{K} \text{ is non-trivial and has type } B. \end{cases}$$

$$root_B(\mathcal{K}) := \begin{cases} \text{the root of } \mathcal{K} & \text{if } \mathcal{K} \text{ is trivial or has type } B \\ o_{[\mathcal{K}]} \in \mathcal{O}_B^* & \text{if } \mathcal{K} \text{ is non-trivial and has type } A. \end{cases}$$

As a direct consequence of Lemma 24 we obtain

Lemma 25. *The functions $root_A$ and $root_B$ are bijections.*

[9] compare Lemma 11.

Here is now the definition of the rational amalgamated product.

Definition 26. The *rational amalgamated product* $\mathcal{A}^\Sigma \odot \mathcal{B}^\Delta$ *of* $\mathcal{A}^\Sigma$ *and* $\mathcal{B}^\Delta$ is the following $(\Sigma \cup \Delta)$-structure with carrier $A \odot B$:

1. Let $f \in \Sigma$ be an n-ary function symbol, let $\mathcal{K}_1, \ldots, \mathcal{K}_n \in A \odot B$. We define $f_{\mathcal{A}\odot\mathcal{B}}(\mathcal{K}_1, \ldots, \mathcal{K}_n) = root_A^{-1}(f_{\mathcal{A}_*}(root_A(\mathcal{K}_1), \ldots, root_A(\mathcal{K}_n)))$.
2. Let $p \in \Sigma$ be an n-ary predicate symbol, let $\mathcal{K}_1, \ldots, \mathcal{K}_n \in A \odot B$. We define $\mathcal{A}^\Sigma \odot \mathcal{B}^\Delta \models p(\mathcal{K}_1, \ldots, \mathcal{K}_n)$ iff $\mathcal{A}_*^\Sigma \models p(root_A(\mathcal{K}_1), \ldots, root_A(\mathcal{K}_n))$.

The interpretation of the function symbols $g \in \Delta$ and the predicate symbols $q \in \Delta$ in $\mathcal{A}^\Sigma \odot \mathcal{B}^\Delta$ is defined symmetrically, using $root_B$.

Theorem 27. *As a* Σ*-structure,* $\mathcal{A}^\Sigma \odot \mathcal{B}^\Delta, \mathcal{A}^\Sigma$ *and* $\mathcal{A}_*^\Sigma$ *are isomorphic, and* $root_A : \mathcal{A}^\Sigma \odot \mathcal{B}^\Delta \to \mathcal{A}_*^\Sigma$ *is a* Σ*-isomorphism. As a* Δ*-structure,* $\mathcal{A}^\Sigma \odot \mathcal{B}^\Delta, \mathcal{B}^\Delta$, *and* $\mathcal{B}_*^\Delta$ *are isomorphic, and* $root_B : \mathcal{A}^\Sigma \odot \mathcal{B}^\Delta \to \mathcal{B}_*^\Delta$ *is a* Δ*-isomorphism.*

Let us add some evidence for the naturalness of rational amalgamation. First we consider the case where the two components are strong non-collapsing SC-structures over disjoint signatures. This is the situation where we can build both the free amalgam and the rational amalgam with our actual methods.

Theorem 28. *The free amalgamated product is–modulo isomorphism—a substructure of the rational amalgamated product.*

The result shows that there are interesting relationships between distinct amalgamation constructions.

Theorem 29. *The rational amalgamated product of two algebras of rational trees over disjoint signatures is isomorphic to the algebra of rational trees over the combined signature.*

This shows that our general construction, complicated as it might appear, yields the expected result when we consider more concrete situations.

6 Combination of Constraint Solvers

Our last aim is to show how constraint solvers for two component structures can be combined to a constraint solver for their rational amalgamated product. Constraint solvers, as considered here, are essentially algorithms that decide solvability of quantifier-free positive formulae in a given solution domain. We (mostly) disregard disjunction since its integration is a triviality.

Definition 30. Let Γ be a signature. A Γ*-constraint* is a conjunction of atomic Γ-formulae.

In order to decide solvability of a "mixed" $(\Sigma \cup \Delta)$-constraint in a rational amalgamated product $\mathcal{A}^\Sigma \odot \mathcal{B}^\Delta$ we shall decompose it into two pure constraints over the signatures Σ and Δ respectively. These output constraints are equipped with additional restrictions of a particular type:

Definition 31. An *A/N (atom/non-atom) declaration* for a constraint γ is a pair (U, W) such that $U \uplus W \subseteq Var(\gamma)$ is a disjoint union. Both U and W may be empty. A solution ν_A of a constraint γ in an SC-structure $(\mathcal{A}^\Sigma, \mathcal{M}, X)$ is called a *solution* of $\langle \gamma, U, W \rangle$ if ν_A assigns distinct atoms to the variables in U, and arbitrary non-atomic elements of A to the variables in W.

In order to avoid some ballast in proofs we shall assume that at least one of the two components is a *non-trivial* SC-structure, which means that it has at least one non-atomic element. We may now formulate our main result concerning combination of constraint solvers in the case of rational amalgamation.

Theorem 32. *Let $\mathcal{A}^\Sigma$ and $\mathcal{B}^\Delta$ be two non-collapsing SC-structures over disjoint signatures, let $\mathcal{A}^\Sigma \odot \mathcal{B}^\Delta$ denote their rational amalgam. Assume that at least one of the two components is a non-trivial SC-structure. Then solvability of $(\Sigma \cup \Delta)$-constraints in $\mathcal{A}^\Sigma \odot \mathcal{B}^\Delta$ is decidable if solvability of (Σ- resp. Δ-) constraints with A/N declarations is decidable for $\mathcal{A}^\Sigma$ and $\mathcal{B}^\Delta$.*

There seems to be no general way to characterize solvability of Γ-constraints with A/N declarations in purely logical terms. But for a restricted class of component structures—a class which is of particular interest in the context of rational amalgamation—a logical characterization of the problems that we have to solve in the two component structures can be given.

Definition 33. A non-collapsing SC-structure $(\mathcal{A}^\Sigma, \mathcal{M}, X)$ is called *rational* if for every atom $x \in X$ and every element $a \in A$ there exists an endomorphism $m \in \mathcal{M}$ that leaves all atoms $x' \neq x$ fixed such that $m(x) = m(a)$.[10]

The algebra of rational trees over a given signature is always a rational SC-structure. The same holds for feature structures ([AP94]), feature structures with arity, and domains with nested, rational lists. For rational SC-structures we obtain the following extension of Theorem 32.

Theorem 34. *Let $\mathcal{A}^\Sigma$ and $\mathcal{B}^\Delta$ be two non-trivial rational SC-structures over disjoint signatures, let $\mathcal{A}^\Sigma \odot \mathcal{B}^\Delta$ denote their rational amalgam. Then the positive existential theory of $\mathcal{A}^\Sigma \odot \mathcal{B}^\Delta$ is decidable if the positive universal-existential theory is decidable for both components $\mathcal{A}^\Sigma$ and $\mathcal{B}^\Delta$.*

It is interesting to contrast this formulation with the corresponding combination result for free amalgamation (Theorem 22 of [BS95]) which needs *stronger* assumptions on the components: *Let $\mathcal{A}^\Sigma$ and $\mathcal{B}^\Delta$ be two strong SC-structures over disjoint signatures, let $\mathcal{A}^\Sigma \otimes \mathcal{B}^\Delta$ denote their free amalgam. Then the positive existential theory of $\mathcal{A}^\Sigma \otimes \mathcal{B}^\Delta$ is decidable if the full positive theory is decidable for both components $\mathcal{A}^\Sigma$ and $\mathcal{B}^\Delta$.*

[10] The existence of such an endomorphism is trivial if $x \notin Stab^{\mathcal{A}}_{\mathcal{M}}(a)$. In this case we may always take the endomorphism $m = m_{x-a}$ of $\mathcal{M}$ that maps x to a and leaves all other atoms fixed. The situation of interest is the case where $x \in Stab^{\mathcal{A}}_{\mathcal{M}}(a)$ and $x \neq a$.

Corollary 35. *Rational amalgamated products* $\mathcal{A}_1^{\Sigma_1} \odot \cdots \odot \mathcal{A}_k^{\Sigma_k}$ *have decidable positive existential theory if the nontrivial components* $\mathcal{A}_i^{\Sigma_i}$ *are rational tree algebras, or nested, rational lists, or feature structures*[11]*, or feature-structures with arity, for* $i = 1, \ldots, k$*, and if the signatures of the components are pairwise disjoint.*

In the rest of this section we describe the underlying algorithm that yields the basis for Theorem 32 and is used to combine given constraint solvers for two components to a constraint solver for the rational amalgam. The algorithm reduces a mixed constraint γ in the signature $(\Sigma \cup \Delta)$ non-deterministically to a pair of constraints with A/N declarations over the "pure" signatures Σ and Δ respectively. For simplicity we assume that the input formula γ has the form $\gamma = \gamma_0^\Sigma \wedge \gamma_0^\Delta$ where γ_0^Σ is a conjunction of atomic Σ-formulae, and γ_0^Δ is a conjunction of atomic Δ-formulae. Moreover we assume that γ does not contain any equation between variables. These assumptions do not really restrict the generality of the approach: simple techniques like "variable abstraction", now standard in this area, may be used to transform an arbitrary $(\Sigma \cup \Delta)$-constraint φ into a constraint γ of the form given above, preserving solvability in both directions.

Algorithm 1 The *input* is a mixed constraint $\gamma = \gamma_0^\Sigma \wedge \gamma_0^\Delta$ of the form described above. Let $V_0 = Var(\gamma_0^\Sigma) \cap Var(\gamma_0^\Delta)$ denote the set of *shared* variables of γ. The algorithm has two steps, both are nondeterministic.

Step 1: Variable identification. *Consider all possible partitions of the set of all shared variables,* V_0*. Each of these partitions yields one of the new constraints as follows. The variables in each class of the partition are "identified" with each other by choosing an element of the class as representative, and replacing in the input formula all occurrences of variables of the class by this representative.*

Step 2: Choose signature labels. *Let* $\gamma_1^\Sigma \wedge \gamma_1^\Delta$ *denote one of the formulae obtained by Step 1, let* V_1 *denote the set of representants of shared variables. The set* V_1 *is partitioned in two subsets* U *and* W *in some arbitrary way.*

Let $\sigma = \gamma_1^\Sigma$, let $\delta = \gamma_1^\Delta$. For each of the choices made in Step 1 and 2, the algorithm yields an *output pair* $(\langle \sigma, U, W \rangle, \langle \delta, W, U \rangle)$, each component representing a constraint with A/N declaration. □

Ignoring several details, Algorithm 1 can be obtained from the corresponding algorithm for free amalgamation by omitting one non-deterministic step (namely the choice of a linear ordering on the set of shared variables). This shows that Algorithm 1 is more efficient than the combination scheme for free amalgamation. The proof of Theorem 32 is based on the following proposition which is verified in [KS96].

Proposition 36. *The input formula* γ *has a solution in* $\mathcal{A}^\Sigma \odot \mathcal{B}^\Delta$ *if and only if there exists an output pair* $(\langle \sigma, U, W \rangle, \langle \delta, W, U \rangle)$ *of Algorithm 1 such that* $\langle \sigma, U, W \rangle$ *has a solution in* $\mathcal{A}^\Sigma$ *and* $\langle \delta, W, U \rangle$ *has a solution in* $\mathcal{B}^\Delta$*.*

[11] As in Examples 1 we refer to [AP94], for specificity.

7 Conclusion

In this paper we have introduced rational amalgamation, a general methodology for combining constraint systems. The present work, in connection with the discussion of free amalgamation in [BS95], seems to suggest a new view of the problem of combining solution domains and constraint solvers. There is now strong evidence that the situation considered in [BS95] and in this paper—the construction of "mixed" elements of a combined domain, given the "pure" elements of two component structures as construction units—is quite similar to the process of building the elements of a single structure, given the symbols of a fixed signature as construction units. We are confident that this analogy will help to isolate the most important methods for combining structures, and to understand the relationship and the differences between different amalgamation constructions.

When we compose elements, given the symbols of a fixed signature, three different structures may be obtained in a direct way, depending on the composition principle, namely the free term algebra, the algebra of rational trees, and the algebra of infinite trees. The privileged role of these three algebras, and the rich amount of interesting relationships between them, are now well-understood (e.g., [Co83, Ma88]). We believe that free amalgamation, rational amalgamation and a further construction called "infinite amalgamation" (still to be investigated) reflect this role on the higher level of amalgamation constructions. Many of the results that we have obtained for free and rational amalgamation can be interpreted in this sense:

- The universality-property of the free amalgamated product (see [BS95]) reflects the status of the free term algebra as the absolutely free Σ-algebra.
- We have seen that the free amalgamated product is always a substructure of the rational amalgamated product. This reflects the fact that the free term algebra is always a substructure of the algebra of rational trees.
- It is well-known that the unification algorithm for the algebra of rational trees can be considered as the variant of the unification algorithm for the free term algebra where we omit the occur-check. Similarly, the decomposition scheme for rational amalgamation as given here (i.e., Algorithm 1) is essentially the decomposition scheme for free amalgamation where we omit the "interstructural" occur-check that is provided by the choice of a linear ordering in the latter scheme.

We would not be surprised if much more principles, techniques and theorems, well-known on the level of tree constructions, could be lifted to the level of combining structures. Our experience with rational amalgamation seems to indicate that this is a difficult, but promising line of research if we want to understand the scale of possibilities, and the limitations for combining solution domains and constraint solvers.

References

[Ac88] P. Aczel, "Non-well-founded Sets," *CSLI* Lecture Notes **14**, Stanford Univ., 1988.

[AP94] H. Ait-Kaci, A. Podelski, and G. Smolka, "A feature-based constraint system for logic programming with entailment," *Theoretical Comp. Science* **122**, 1994, pp.263–283.

[BS95] F. Baader and K.U. Schulz, "On the Combination of Symbolic Constraints, Solution Domains, and Constraint Solvers," in: Proceedings CP'95, U.Montanari, F.Rossi (Eds.), Springer LNCS 976, pp. 380-397.

[BS94] F. Baader, J. Siekmann, "Unification Theory," in D.M. Gabbay, C. Hogger, and J. Robinson, Editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, Oxford University Press, Oxford, UK, 1994, pp. 41–125.

[BT94] R. Backofen and R. Treinen, "How to Win a Game with Features," in *Constraints in Computational Logics, Proc. CCL'94*, J.-P. Jouannaud (Ed.), Springer LNCS **845**, 1994, pp. 320-335.

[Co84] A. Colmerauer, "Equations and inequations on finite and infinite trees," in: *Proc. 2nd Int. Conf. on Fifth Generation Computer Systems*, 1984, pp.85–99.

[Co90] A. Colmerauer, "An introduction to PROLOG III," *C. ACM* **33**, 1990, pp. 69–90.

[Co83] B. Courcelle, "Fundamental Properties of Infinite Trees," *Theoretical Computer Science* **25**, 1983, pp. 95–169.

[Ga96] D.M. Gabbay, "An Overview of Fibred Semantics and the Combination of Logics," to appear in *Proceedings of the First International Workshop "Frontiers of Combining Systems" FroCoS'96*, F. Baader and K.U. Schulz, Eds., Kluwer Academic Publishers, 1996.

[KS96] S. Kepser and K.U. Schulz, "Combination of Constraint Systems II: Rational Amalgamation," long version of this paper, available via anonymous ftp from ftp.cis.uni-muenchen.de (directory "schulz", file name "RationalAmalgamation.ps.Z").

[Ma88] M.J. Maher, "Complete axiomatizations of the algebras of finite, rational and infinite trees," in: *Proceedings of Third Annual Symposium on Logic in Computer Science, LICS'88*, pp. 348–357, Edinburgh, Scotland, 1988.

[Ma71] A.I. Mal'cev, "The Metamathematics of Algebraic Systems," volume 66 of *Studies in Logic and the Foundation of Mathematics*, North Holland, Amsterdam, London, 1971.

[Pf91] J. Pfalzgraf, "Logical Fiberings and Polycontextural Systems," In *Fundamentals of Artificial Intelligence Research*, Ph. Jorrand and J. Kelemen, Eds., Springer LNCS **535** (1991).

[PS94] J. Pfalzgraf and K. Stokkerman, "On Robotics Scenarios and Modeling with Fibered Structures," In *Springer series Texts and Monographs in Symbolic Computation, Automated Practical Reasoning: Algebraic Approaches*, J. Pfalzgraf and D. Wang, Eds., Springer Verlag, 1994.

[Ro88] W.C. Rounds, "Set values for unification based grammar formalisms and logic programming," *Research Report* CSLI-88-129, Stanford, 1988.

[ST94] G. Smolka, R. Treinen, "Records for Logic Programming," *J. of Logic Programming* **18**(3) (1994), pp. 229-258.

Tractable Disjunctions of Linear Constraints

Manolis Koubarakis

Dept. of Computation
UMIST
P.O. Box 88
Manchester M60 1QD
United Kingdom
manolis@sna.co.umist.ac.uk

Abstract. We study the problems of deciding consistency and performing variable elimination for disjunctions of linear inequalities and inequations with *at most one* inequality per disjunction. This new class of constraints extends the class of generalized linear constraints originally studied by Lassez and McAloon. We show that deciding consistency of a set of constraints in this class can be done in polynomial time. We also present a variable elimination algorithm which is similar to Fourier's algorithm for linear inequalities.

1 Introduction

Linear constraints over the reals have recently been studied in depth by researchers in constraint logic programming (CLP) and constraint databases (CDB) [JM94, KKR95, Kou94]. The most important operations in CLP and CDB systems is deciding consistency of a set of constraints, and performing variable elimination.

Disjunctions of linear constraints over the reals are important in many applications [JM94]. The problem of deciding consistency for an arbitrary set of disjunctions of linear constraints is NP-complete [Son85]. It is therefore interesting to discover classes of disjunctions of linear constraints for which consistency can be decided in PTIME. In [LM89a], Lassez and McAloon have studied the class of *generalized linear constraints* which includes linear inequalities (e.g., $2x_1 + 3x_2 - 5x_3 \leq 6$) and disjunctions of linear inequations[1] (e.g., $2x_1 + 3x_2 - 4x_3 \neq 4 \vee x_2 + x_3 + x_5 \neq 7$). Among other things, they have shown that the consistency problem for this class can be solved in PTIME.

[Kou92, IvH93, Imb93, Imb94]) have studied the problem of variable elimination for generalized linear constraints. The basic algorithm for variable elimination has been discovered independently in [Kou92] and [Imb93], but [Kou92] has used the result only in the context of temporal constraints. The basic algorithm is essentially an extension of Fourier's elimination algorithm [Sch86] to deal with disjunctions of inequations. If S is a set of constraints, let $|S|$ denote its cardinality. Let $C = I \cup D_n$ be a set of generalized linear constraints, where I

[1] Some people prefer the term disequations [Imb94].

is a set of inequalities and D_n is a set of disjunctions of inequations. If we eliminate m variables from C using the basic algorithm proposed by Koubarakis and Imbert then the resulting set contains $O(|I|^{2^m})$ inequalities and $O(|D_n||I|^{2^{m+1}})$ disjunctions of inequations. A lot of these constraints are redundant. Imbert has studied this problem in more detail and presented more advanced algorithms that eliminate redundant constraints [Imb93, Imb94].

The contributions of this paper are the following:

- We extend the class of generalized linear constraints to include disjunctions with an unlimited number of inequations and *at most one* inequality per disjunction. For example:

$$3x_1 + x5 - 4x_3 \leq 7 \ \vee \ 2x_1 + 3x_2 - 4x_3 \neq 4 \ \vee \ x_2 + x_3 + x_5 \neq 7$$

  The resulting class will be called the class of *Horn* constraints since there seems to be some analogy with Horn clauses. We show that deciding consistency can still be done in PTIME for this class (Theorem 9).
- We extend the basic variable elimination algorithm of [Kou92, Imb93] for the new class of Horn constraints.

The paper is organized as follows. Section 2 introduces the basic concepts needed for the developments of this paper. Section 3 presents the algorithm for deciding consistency. Section 4 presents the algorithm for variable elimination. Section 5 discusses future work.

2 Preliminaries

We consider the n-dimensional Euclidean space $\mathcal{R}^n$. We assume that the readers are familiar with linear constraints over $\mathcal{R}^n$. We will consider linear inequalities (e.g. $2x_1 + 3x_2 - 5x_3 \leq 6$), equations (e.g., $2x_1 + 3x_2 - 5x_3 = 6$) and inequations (e.g., $2x_1 + 3x_2 - 5x_3 \neq 6$). If S is a set of constraints then the *solution set* of S will be denoted by $Sol(S)$. We will use the same notation for the solution set of a single constraint.

Let us now present some concepts of convex geometry [Sch86, Gru67]. We will take the definitions of these concepts from [LM89a]. If V is a subspace of the n-dimensional Euclidean space $\mathcal{R}^n$ and p a vector in $\mathcal{R}^n$ then the translation $p + V$ is called an *affine space*. The intersection of all affine spaces that contain a set S is an affine space, called the *affine closure* of S and denoted by $Aff(S)$. If e is a linear equation then the solutions set of e is called a *hyperplane*. In $\mathcal{R}^3$ the hyperplanes are the planes. In $\mathcal{R}^2$ the hyperplanes are the straight lines. A hyperplane is an affine space and every affine space is the intersection of a finite number of hyperplanes. If E is a set of equalities then $Sol(E)$ is an affine space. If i is a linear inequality then the solution set of i is called a *half-space*. If I is a set of inequalities then $Sol(I)$ is the intersection of a finite number of half-spaces, and is called a *polyhedral* set.

A set $S \subseteq \mathcal{R}^n$ is called *convex* if the line segment joining any pair of points in S is included in S. Affine subspaces of $\mathcal{R}^n$ are convex. Half-spaces are convex. Also, polyhedral sets are convex.

Let us now define the class of constraints that we will consider.

Definition 1. A *Horn constraint* is a disjunction $d_1 \vee d_2 \vee \cdots \vee d_n$ where each d_i, $i = 1, \ldots, n$ is a weak linear inequality or a linear inequation, and the number of inequalities among $d_1, \ldots, d_n$ does not exceed one. If there are no inequalities then a Horn constraint will be called *negative*. Otherwise it will be called *positive*.

Example 1. The following are examples of Horn constraints:

$$3x_1 + x_5 - 4x_3 \leq 7 \vee 2x_1 + 3x_2 - 4x_3 \neq 4 \vee x_2 + x_3 + x_5 \neq 7$$
$$4x_1 + x_3 \neq 3 \vee 5x_2 - 3x_5 + x_4 \neq 6$$

The first constraint is positive while the second is negative.

According to the above definition weak inequalities can also be considered as positive Horn constraints. However, with the exception of Section 4, we will usually find it more convenient to consider inequalities separately.

Negative Horn constraints have been considered before in [LM89a, LM89b, Kou92, IvH93, Imb93, Imb94, Kou95]. To the best of our knowledge positive Horn constraints have not been considered before. If d is a positive Horn constraint then $d \equiv \neg(E \wedge i)$ where E is a conjunction of equations and i is an inequality. We will often use this notation for positive Horn constraints.

We do not need to introduce strict inequalities in the above definition. A strict inequality like $x_1 + x_2 + x_3 < 5$ can be equivalently written as follows:

$$x_1 + x_2 + x_3 \leq 5,\ x_1 + x_2 + x_3 \neq 5$$

Similarly, a constraint $x_1+x_2+x_3 < 5 \vee \phi$ where ϕ is a disjunction of inequations is equivalent to the conjunction of the following constraints:

$$x_1 + x_2 + x_3 \leq 5 \vee \phi,\ x_1 + x_2 + x_3 \neq 5 \vee \phi$$

A similar observation is made in [BB95] in the context of the ORD-Horn class of temporal constraints.

If d is a negative Horn constraint then the solution set of d is $Sol(d) = \mathcal{R}^n \setminus Sol(\neg d)$. The constraint $\neg d$ is a conjunction of equations thus $Sol(\neg d)$ is an affine space. If $\neg d$ is inconsistent then d is equivalent to *true* (e.g., $x \neq 2 \vee x \neq 3$). In the rest of the paper we will ignore negative Horn constraints that are equivalent to *true*.

If d is a positive Horn constraint of the form $\neg(E \wedge i)$ then $Sol(d) = \mathcal{R}^n \setminus Sol(\neg d)$. The constraint $\neg d$ is a conjunction $E \wedge i$ where E is a conjunction of equations and i is a strict inequality. If $E \wedge i$ is inconsistent then its corresponding Horn constraint d is equivalent to *true* (e.g., $d \equiv x \neq 2 \vee x \leq 3$). If $E \wedge i$ is consistent and $Sol(i) \subseteq Sol(E)$ then $d \equiv \neg E$, so d is actually a

negative Horn constraint (e.g., $x \neq 2 \vee y \neq 2 \vee x \geq 3 \equiv x \neq 2 \vee y \neq 2$). If $E \wedge i$ is consistent and $Sol(i) \not\subseteq Sol(E)$ then its solution set will be called a *half affine space*. In $\mathcal{R}^3$ the half affine spaces are half-lines or half-planes. For example, $z = 2 \wedge x > 0$ is a half plane. In the rest of the paper we will ignore positive Horn constraints equivalent to a negative Horn constraint or *true*.

3 Deciding Consistency

[LM89a] showed that negative Horn constraints can be treated independently of one another for the purposes of deciding consistency. The following is one of their basic results.

Theorem 2. Let $C = I \cup D_n$ be a set of constraints where I is a set of linear inequalities and D_n is a set of negative Horn constraints. Then C is consistent if and only if I is consistent, and for each $d \in D_n$ the set $I \cup \{d\}$ is consistent.

Whether a set of inequalities is consistent or not, can be decided in PTIME using Kachian's linear programming algorithm [Sch86]. We can also detect in PTIME whether $I \cup \{d\}$ is consistent by simply running Kachian's algorithm $2n$ times to decide whether I implies every equality e in the conjunction of n equalities $\neg d$. In other words, deciding consistency in the presence of negative Horn contraints can be done in PTIME.[2]

Is it possible to extend this result to the case of positive Horn constraints? In what follows, we will answer this question affirmatively. Let us start by pointing out that the independence property of negative Horn constraints does *not* carry over to positive ones.

Example 2. Let $I = \{x \geq 1, x \leq 5, y = 3\}$. The constraint sets

$$I \cup \{\neg(y = 3 \wedge x > 1)\} \text{ and } I \cup \{\neg(y = 3 \wedge x = 1)\}$$

are consistent. But the set $I \cup \{\neg(y = 3 \wedge x > 1), \neg(y = 3 \wedge x = 1)\}$ is inconsistent.

Fortunately, there is still enough structure available in our problem which we can exploit to come up with a PTIME consistecy checking algorithm. Let $C = I \cup D_p \cup D_n$ be a set of constraints where I is a set of inequalities, D_p is set of positive Horn constraints, and D_n is a set of negative Horn constraints. Intuitively, the solution set of C is empty only if the polyhedral set defined by I is *covered* by the affine spaces and half affine spaces defined by the Horn constraints.

The algorithm CONSISTENCY shown in Figure 1 proceeds as follows. Initially, we check whether I is consistent. If this is the case, then we proceed to examine whether $Sol(I)$ can be covered by $Sol(\{\neg d : d \in D_p \cup D_n\})$. To verify this, we

[2] The exact algorithm that Lassez and McAloon give in [LM89a] is different but this is not significant for the purposes of this paper.

Algorithm CONSISTENCY
Input: A set of constraints $C = I \cup D_p \cup D_n$.
Output: "consistent" if C is consistent. Otherwise "inconsistent".
Method:

If I is inconsistent **then return** "inconsistent"

Repeat
  $Done \leftarrow true$
  For each $d \in D_p \cup D_n$ **do**
    If $d \equiv \neg(E \wedge i) \in D_p$ and $Sol(I) \subseteq Sol(E)$ **then**
      $I \leftarrow I \wedge \neg i$
      If I is inconsistent **then return** "inconsistent"
      $Done \leftarrow false$
      Remove d from D_p
    Else If $d \in D_n$ and $Sol(I) \subseteq Sol(\neg d)$ **then**
      Return "inconsistent"
    End If
  End For
Until *Done*
Return "consistent"

Fig. 1. Deciding consistency of a set of Horn constraints

make successive passes over $D_p \cup D_n$. In each pass, we carry out two checks. The *first check* discovers whether there is any positive Horn constraint $d \equiv \neg(E \wedge i)$ such that $Sol(I)$ is included in the affine space defined by E. If this is the case then d is discarded and I is updated to reflect the part possibly "cut off" by d. The resulting solution set $Sol(I)$ is still a polyhedral set. An inconsistency can arise if $Sol(I)$ is reduced to $\emptyset$ by successive "cuts". In each pass we also check whether there is an affine space (represented by the negation of a negative Horn constraint) which covers $Sol(I)$. In this case there is an inconsistency as well. The algorithm stops when there are no more affine spaces or half affine spaces that pass the two checks. In this case C is consistent.

Let us now prove the correctness of algorithm CONSISTENCY. First, we will need a few technical lemmas. The first two lemmas show that the sets resulting from successive "cuts" inflicted on $Sol(I)$ by positive Horn constraints passing the first check of the algorithm are indeed polyhedral. The lemmas also give a way to compute the inequalities defining these sets.

Lemma 3. Let I be a set of inequalities and $\neg(E \wedge i)$ be a Horn constraint such that $Sol(I) \subseteq Sol(E)$. Then $Sol(I \wedge \neg(E \wedge i)) = Sol(I \wedge \neg i)$.

Proof. Let $\overline{x} \in Sol(I \wedge \neg(E \wedge i))$. Then $\overline{x} \in Sol(I)$ and $\overline{x} \in Sol(\neg(E \wedge i))$. If $\overline{x} \in Sol(\neg(E \wedge i))$ then $\overline{x} \in Sol(\neg E)$ or $\overline{x} \in Sol(\neg i)$. But $Sol(I) \cap Sol(\neg E) = \emptyset$ because $Sol(I) \subseteq Sol(E)$. Therefore, $\overline{x} \in Sol(I)$ and $\overline{x} \in Sol(\neg i)$. Equivalently, $\overline{x} \in Sol(I \wedge \neg i)$. The other direction of the proof is trivial.

Lemma 4. Let I be a set of inequalities and $d_k \equiv \neg(E_k \wedge i_k)$, $k = 1, \ldots, m$ be a set of Horn constraints such that $Sol(I) \subseteq Sol(E_1)$ and

$$Sol(I \wedge \bigwedge_{k=1}^{l} \neg i_k) \subseteq Sol(E_{l+1}) \text{ for } l = 1, \ldots, m-1.$$

Then

$$Sol(I \wedge \bigwedge_{k=1}^{m} d_k) = Sol(I \wedge \bigwedge_{k=1}^{m} \neg i_k).$$

Proof. The proof is by induction on m. The base case $m = 1$ follows from Lemma 3. For the inductive step, let us assume that the lemma holds for $m-1$ Horn constraints. Then

$$Sol(I \wedge \bigwedge_{k=1}^{m} d_k) = Sol(I \wedge \bigwedge_{k=1}^{m-1} d_k) \cap Sol(d_m) =$$

$$Sol(I \wedge \bigwedge_{k=1}^{m-1} \neg i_k) \cap Sol(d_m) = Sol((I \wedge \bigwedge_{k=1}^{m-1} \neg i_k) \wedge d_m)$$

using the inductive hypothesis.
The assumptions of this lemma and Lemma 3 imply that

$$Sol((I \wedge \bigwedge_{k=1}^{m-1} \neg i_k) \wedge d_m) = Sol(I \wedge \bigwedge_{k=1}^{m} \neg i_k).$$

Thus

$$Sol(I \wedge \bigwedge_{k=1}^{m} d_k) = Sol(I \wedge \bigwedge_{k=1}^{m} \neg i_k).$$

The following lemmas show that if there are Horn constraints that do not pass the two checks of algorithm CONSISTENCY then the affine spaces or half affine spaces corresponding to their negations cannot cover the polyhedral set defined by the inequalities.

Lemma 5. Let S be a convex set of dimension d and suppose that $S_1, \ldots S_n$ are convex sets of dimension $d_i < d$, $i = 1, \ldots, n$. Then $S \not\subseteq \bigcup_{i=1}^{n} S_i$.

Proof. See Lemma 2 of [LM89a].

Lemma 6. Let I be a consistent set of inequalities and $d_k \equiv \neg(E_k \wedge i_k)$, $k = 1, \ldots, m$ be a set of Horn constraints such that $Sol(I) \not\subseteq Sol(E_k)$ and $Sol(I) \cap Sol(E_k) \neq \emptyset$ for all $k = 1, \ldots, m$. Then $Sol(I) \not\subseteq \bigcup_{k=1}^{m} Sol(\neg d_k)$.

Proof. The proof is very similar to the proof of Theorem 1 of [LM89a].

Since $Sol(I) \not\subseteq Sol(E_k)$ then $Aff(Sol(I)) \not\subseteq Sol(E_k)$. This means that $Sol(E_k) \cap Aff(Sol(I))$ is an affine space of strictly lower dimension than $Aff(Sol(I))$. Then $Sol(E_k) \cap Sol(I)$ is of strictly lower dimension than $Sol(I)$ since the dimension of $Sol(I)$ is equal to that of $Aff(Sol(I))$. Thus from Lemma 5, $Sol(I) \not\subseteq \bigcup_{k=1}^{m} Sol(E_k)$. Notice now that $Sol(\neg d_k) \subseteq Sol(E_k)$ for all $k = 1, \ldots, m$. Therefore $\bigcup_{k=1}^{m} Sol(\neg d_k) \subseteq \bigcup_{k=1}^{m} Sol(E_k)$. We can now conclude that $Sol(I) \not\subseteq \bigcup_{k=1}^{m} Sol(\neg d_k)$.

The following theorems demonstrate that the algorithm CONSISTENCY is correct and can be implemented in PTIME.

Theorem 7. If algorithm CONSISTENCY returns "inconsistent" then its input C is inconsistent.

Proof. If the algorithm returns "inconsistent" in its first line then I, and therefore C, is inconsistent.

If the algorithm returns "inconsistent" in the third if-statement then there are positive Horn constraints $d_k \equiv \neg(E_k \wedge i_k)$, $k = 1, \ldots, m \leq |D_p|$ such that the assumptions of Lemma 4 hold for I and $d_1, \ldots, d_m$. Therefore

$$Sol(I \wedge \bigwedge_{k=1}^{m} d_k) = Sol(I \wedge \bigwedge_{k=1}^{m} \neg i_k) = \emptyset.$$

Consequently, $Sol(C) = \emptyset$ because $Sol(C) \subseteq Sol(I \wedge \bigwedge_{k=1}^{m} d_k)$.

If the algorithm returns "inconsistent" in the fourth if-statement then there are positive Horn constraints $d_1, \ldots, d_n \in D_p$ and negative constraint $d_{m+1} \in D_n$ such that the assumptions of Lemma 2 hold for I and $d_1, \ldots, d_m$, and

$$Sol(I \wedge \bigwedge_{k=1}^{m} d_k) \subseteq Sol(\neg d_{m+1}).$$

But then

$$Sol(C) \subseteq Sol(I \wedge \bigwedge_{k=1}^{m+1} d_m) = Sol(I \wedge \bigwedge_{k=1}^{m} d_m) \cap Sol(d_{m+1}) = \emptyset.$$

Theorem 8. If algorithm CONSISTENCY returns "inconsistent" then its input C is inconsistent.

Proof. If the algorithm returns "consistent" then I is consistent. Let $d_1, \ldots, d_m$ be the positive Horn constraints removed from $D_p \cup D_n$ by the algorithm, and $d_{m+1}, \ldots, d_n$ be the remaining Horn constraints. Then

$$Sol(C) = Sol(I \wedge \bigwedge_{k=1}^{n} d_k) = Sol(I \wedge \bigwedge_{k=1}^{m} d_k) \setminus \bigcup_{k=m+1}^{n} Sol(\neg d_k) =$$

$$Sol(I \wedge \bigwedge_{k=1}^{m} \neg i_k) \setminus \bigcup_{k=m+1}^{n} Sol(\neg d_k).$$

Notice that $Sol(I \wedge \bigwedge_{k=1}^{m} \neg i_k) \neq \emptyset$ otherwise the algorithm outputs "inconsistent" in Step 2. Also, $Sol(I \wedge \bigwedge_{k=1}^{m} \neg i_k) \not\subseteq Sol(E_k)$ for all $k = m+1, \ldots, n$ otherwise the algorithm would have removed d_k from $D_p \cup D_n$.

Without any loss of generality we can also assume that

$$Sol(I \wedge \bigwedge_{k=1}^{m} \neg i_k) \cap Sol(E_k) \neq \emptyset$$

for all $k = m+1, \ldots, n$ (if this does not hold for constraint d_k, this constraint can be discarded without changing $Sol(C)$). From Lemma 6 we can now conclude that $Sol(I \wedge \bigwedge_{k=1}^{m} \neg i_k) \not\subseteq \bigcup_{k=m+1}^{n} Sol(\neg d_k)$. Therefore $Sol(C) \neq \emptyset$.

Theorem 9. The algorithm CONSISTENCY can be implemented in PTIME.

Proof. It is not difficult to see that the algorithm can be implemented in PTIME. The consistency of I can be checked in PTIME using Kachian's algorithm for linear programming [Sch86]. The test $Sol(I) \subseteq Sol(E)$ can be verified by checking whether every equation e in the conjunction E is implied by I. This can be done in PTIME using Kachian's algorithm $2n$ times where n is the number of equations in E. In a similar way one can implement the test $Sol(I) \subseteq Sol(\neg d)$ in PTIME when d is a negative Horn constraint.

4 Variable Elimination

In this section we study the problem of variable elimination for sets of Horn constraints. The algorithm VARELIMINATION, shown in Figure 2, eliminates a given variable x from a set of Horn constraints C. More variables can be eliminated by successive applications of VARELIMINATION. For the purposes of this algorithm we consider inequalities as positive Horn constraints.

The algorithm VARELIMINATION is similar to the one studied in [Kou92, Imb93] for the case of negative Horn constraints.

Theorem 10. The algorithm VARELIMINATION is correct.

Proof. Let the variables of C be $X = \{x, x_2, \ldots, x_n\}$. If $(x^0, x_2^0, \ldots, x_n^0) \in \mathcal{R}^n$ is an element of $Sol(C)$ then it can be easily seen that it is also an element of $Sol(C')$.

Conversely, take $(x_2^0, \ldots, x_n^0) \in \mathcal{R}^{n-1} \cap Sol(C')$ and consider the set $C(x, x_2^0, \ldots, x_n^0)$. If this set is simplified by removing constraints equivalent to true, disjunctions equivalent to false, and redundant constraints then

$$C(x, x_2^0, \ldots, x_n^0) = \{l^0 \leq x, x \leq u^0\} \cup \{x \neq a_k^0,\ k = 1, \ldots, \lambda\}.$$

Let us now assume (by contradiction) that there is no value $x^0 \in \mathcal{R}^n$ such that $(x^0, x_2^0, \ldots, x_n^0) \in Sol(C)$. This can happen only under the following cases:

Algorithm VARELIMINATION
Input: A set of Horn constraints C in variables X, and a variable $x \in X$ to be eliminated from C.
Output: A set of Horn constraints C' in variables $X \setminus \{x\}$ such that $Sol(C') = Projection_{X\setminus\{x\}}(Sol(C))$.
Method:

Rewrite each constraint containing x as $x \leq U \vee \phi$ or $L \leq x \vee \phi$ or $x \neq A \vee \phi$ where ϕ is a disjunction of inequations.

For each pair of positive Horn constraints $x \leq U \vee \phi_1$ and $L \leq x \vee \phi_2$ **do**
    Add $L \leq U \vee \phi_1 \vee \phi_2$ to C'
End For

For each pair of positive Horn constraints $x \leq U \vee \phi_1$ and $L \leq x \vee \phi_2$ **do**
    For each negative Horn constraint $x \neq A \vee \phi$ **do**
        Add $A \neq L \vee A \neq U \vee \phi$ to C'
    End For
End For

Add each constraint not containing x to C'
Return C'

Fig. 2. A variable elimination algorithm

1. $u^0 < l^0$. If inequalities $x \leq u^0$ and $l^0 \leq x$ come from positive Horn constraints $x \leq u \vee \phi_1$ and $l \leq x \vee \phi_2$ then

$$\phi_1(x_2^0, \ldots, x_n^0) \equiv \phi_1(x_2^0, \ldots, x_n^0) \equiv false$$

   otherwise these constraints would have been discarded from $C(x, x_2^0, \ldots, x_n^0)$ during its simplification. But because $l \leq u \vee \phi_1 \vee \phi_2 \in C'$ and $(x_2^0, \ldots, x_n^0) \in Sol(C')$ then $l^0 \leq u^0$. Contradiction!
2. $l^0 = u^0 = a_j^0,\ 1 \leq j \leq \lambda$. With reasoning similar to the above, we can show that this case is also impossible.

Finally, we can conclude that there exists a value $x^0 \in \mathcal{R}$ such that

$$(x, x_2^0, \ldots, x_n^0) \in Sol(C).$$

Let $C = I \cup D_p \cup D_n$ be a set of constraints. Eliminating m variables from C with repeated applications of the above algorithm will result in a set with $O((|I|+|D_p|)^{2^m})$ positive Horn constraints and $O(|D_n|\,(|I|+|D_p|)^{2^{m+1}})$ negative Horn constraints. Many of these contraints will be redundant; it is therefore important to extend this work with efficient redundancy elimination algorithms that can be used together with VARELIMINATION.

5 Conclusions and Future Work

In future work we would like to follow the steps of [Kou92, Kou95] and consider the applicability of the results of this paper to temporal reasoning. It is clear that the class of constraints covered here is more expressive than the quantitative temporal constraints of [Kou92, Kou95] and the ORD-Horn class of [BB95]. An open question is whether one can use the results of this paper to find more efficient algorithms for the ORD-Horn class of qualitative temporal constraints[3]. In [Kou95] we carried out successfully a similar investigation for the class of PA networks [VKvB89] using the results of [Kou92].

Another interesting problem would be to study more advanced variable elimination algorithms for Horn constraints. The results of [Imb93, Imb94] that apply to negative Horn constraints only, should be a good starting point in this direction.

References

[BB95] Nebel Bernhard and Hans-Jürgen Bürckert. Reasoning about temporal relations: A maximal tractable subclass of Allen's interval algebra. *Journal of the ACM*, 42(1):43–66, January 1995.

[Gru67] B. Grunbaum. *Convex Polytopes.* John Wiley and Sons, 1967.

[Imb93] J.-L. Imbert. Variable Elimination for Generalized Linear Constraints. In *Proceedings of the 10th International Conference on Logic Programming*, 1993.

[Imb94] J.-L. Imbert. Redundancy, Variable Elimination and Linear Disequations. In *Proceedings of the International Symposium on Logic Programming*, pages 139–153, 1994.

[IvH93] J.-L. Imbert and P. van Hentenryck. On the Handling of Disequations in CLP over Linear Rational Arithmetic. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, Logic Programming Series, pages 49–71. MIT Press, 1993.

[JM94] J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19-20:503–581, 1994.

[KKR95] P.C. Kanellakis, G.M. Kuper, and P.Z. Revesz. Constraint Query Languages. *Journal of Computer and System Sciences*, 51:26–52, 1995.

[Kou92] M. Koubarakis. Dense Time and Temporal Constraints with $\neq$. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference (KR'92)*, pages 24–35. Morgan Kaufmann, San Mateo, CA, October 1992.

[Kou94] M. Koubarakis. Foundations of Indefinite Constraint Databases. In A. Borning, editor, *Proceedings of the 2nd International Workshop on the Principles and Practice of Constraint Programming (PPCP'94)*, volume 874 of *Lecture Notes in Computer Science*, pages 266–280. Springer Verlag, 1994.

[3] Nebel and Bürckert mention this as an open problem, but it is not clear how one can get more efficient algorithms using their techniques.

[Kou95] M. Koubarakis. From Local to Global Consistency in Temporal Constraint Networks. In *Proceedings of the 1st International Conference on Principles and Practice of Constraint Programming (CP'95)*, volume 976 of *LNCS*, pages 53–69, Cassis, France, September 1995.

[LM89a] Jean-Louis Lassez and Ken McAloon. A Canonical Form for Generalized Linear Costraints. Technical Report RC15004 (#67009), IBM Research Division, T.J. Watson Research Center, 1989.

[LM89b] Jean-Louis Lassez and Ken McAloon. A Canonical Form for Generalized Linear Costraints. In *TAPSOFT '89, Advanced Seminar on Foundations of Innovative Software Development, Lecture Notes in Computer Science 351*, pages 19–27. Springer Verlag, 1989.

[Sch86] A. Schrijver, editor. *Theory of Integer and Linear Programming.* Wiley, 1986.

[Son85] E. Sontag. Real Addition and the Polynomial Time Hierarchy. *Information Processing Letters*, 20:115–120, 1985.

[VKvB89] Marc Vilain, Henry Kautz, and Peter van Beek. Constraint Propagation Algorithms for Temporal Reasoning: A Revised Report. In D.S. Weld and J. de Kleer, editors, *Readings in Qualitative Reasoning about Physical Systems*, pages 373–381. Morgan Kaufmann, 1989.

Exploiting the Use of DAC in MAX-CSP*

Javier Larrosa[1] and **Pedro Meseguer**[2]
[1]Universitat Politècnica de Catalunya, Dep. Llenguatges i Sistemes Informàtics
Pau Gargallo 5, 08028 Barcelona, SPAIN
E-mail: larrosa@lsi.upc.es
[2]Institut d'Investigació en Intel.ligència Artificial, CSIC
Campus UAB, 08193 Bellaterra, SPAIN.
E-mail: pedro@iiia.csic.es

Abstract. Following the work of Wallace, who introduced the use of directed arc-consistency in MAX-CSP algorithms using DAC counts, we present a number of improvements of DAC usage for the P-EFC3 algorithm. These improvements include: (i) a better detection of dead-ends, (ii) a more effective form for value pruning, and (iii) a different heuristic criterion for value ordering. Considering the new DAC usage, we have analyzed some static variable ordering heuristics previously suggested, and we propose new ones which have been shown effective. The benefits of our proposal has been assessed empirically solving random CSP instances, showing a clear performance gain with respect to previous approaches.

1 Introduction

Constraint satisfaction problems (CSP) consider the assignment of values to variables under a set of constraints. A solution is a total assignment satisfying every constraint. If such assignment does not exist, the problem is overconstrained, and it may be of interest to find a total assignment satisfying as many constraints as possible. This problem is called the maximal constraint satisfaction problem (MAX-CSP), and a solution is a total assignment satisfying the maximum number of constraints. MAX-CSP is of interest in several areas of application [Fox, 87; Feldman and Golumbic, 90; Bakker *et al.*, 93].

P-EFC3 is one of the best algorithms for MAX-CSP [Freuder and Wallace, 92]. It is a branch and bound algorithm including forward checking in order to anticipate dead-ends before they actually occur. To detect dead-ends, P-EFC3 computes a lower bound of the number of unsatisfied constraints from (i) the set of past (assigned) variables, and (ii) the effect of past variables on future (unassigned) ones, by using *inconsistency counts* (IC). Recently, this lower bound has been improved by including unsatisfied constraints from the set of future variables. This has been implemented using *directed arc-consistency counts* (DAC) [Wallace, 94], and the reported empirical results show a clear improvement in performance with respect to pure P-EFC3. DAC are computed in a preprocessing step following a variable ordering, which is later followed by P-EFC3 for variable instantiation.

In this paper, we present further improvements on the DAC usage inside the P-

* This research has been supported by the Spanish CICYT under the project #TAP93-0451.

EFC3 algorithm. Our proposal considers three points: (i) a better dead-end detection, by increasing the lower bound associated to the current node, (ii) a more effective criterion for value pruning, and (iii) a different heuristic criterion for value ordering. All these modifications are simple consequences of the following fact: for each feasible value of a future variable, its associated IC and DAC can be added producing a lower bound of the number of inconsistencies involving that value if it is eventually assigned to the variable. This produces a better lower bound for branch and bound, which causes a significant improvement in the algorithm efficiency compared with the previous version presented in [Wallace, 94]. In addition, given that P-EFC3 with DAC follows a *static variable ordering* (SVO), we have analyzed some SVO heuristics previously reported, as well as new ones. We have assessed their relative efficiency solving random MAX-CSP instances.

It is worth noting that DAC use is not only a technical improvement on the P-EFC3 algorithm. In addition, the use of DAC allowed us to discover an easy-hard-easy pattern in the search effort of solving MAX-CSP instances with constraint tightness as varying parameter [Larrosa and Meseguer, 96], analogous to the pattern observed when solving CSP instances [Prosser, 94; Smith, 94]. This phenomenon does not appear when solving MAX-CSP using branch and bound based algorithms without considering some degree of consistency among future variables.

This paper is organized as follows. In Section 2 we present related algorithms for MAX-CSP. In Section 3 we provide some preliminaries and definitions required in the rest of the paper. In Section 4, we analyze the use of DAC in [Wallace, 94], and present our improvements. In Section 5, we discuss several SVO heuristics. Both DAC improvements and heuristics are evaluated in Section 6, where we provide empirical results solving random CSP instances. Finally, Section 7 contains the conclusions of this work.

2 Related Work

The simplest algorithm for MAX-CSP follows a *branch and bound* scheme. This algorithm performs a systematic traversal on the search tree generated by the problem where a node has associated a set of assigned variables (called *past variables*) and a set of uninstantiated (or *future*) variables. At each node one future variable is selected (*current* variable) and all its feasible values are considered for instantiation. Associated with each node there is a cost function, defined as the number of constraints violated by the assigned variables. This cost function is called the *distance* from a total assignment satisfying every constraint. Branch and bound keeps track of the best solution obtained so far, which is the total assignment with minimum distance in the explored part of the search tree. When a partial assignment has a distance greater than or equal to the distance of the current best solution, this line of search is abandoned because it cannot lead to a better solution than the current one and it is said that a *dead end* has been found. The distance of the current best solution is used as an *upper bound* of the allowable cost, while the distance of the current partial assignment is a *lower bound* of the cost for any assignment including this partial assignment.

The basic branch and bound can be enhanced with more sophisticated strategies

based on previous work on solvable CSP. *Prospective* algorithms look ahead to compute some form of local consistency between past and future variables. The most common prospective algorithm is *forward checking*. It evaluates the impact of the current partial assignment on future variables, which allows the algorithm to improve the lower bound. *Retrospective* algorithms remember previous actions in order to avoid repeating them in the future and, in this way, they can save redundant constraint checks. Retrospective algorithms are *backjumping* and *backmarking*. All these algorithms are complete. For a detailed description, see [Freuder and Wallace, 92].

Several heuristics for variable and value selection have been developed. Regarding SVO heuristics, variables can be ordered by decreasing width. This heuristic is enhanced when combined with a second heuristic to break ties, such as minimum domain size, maximum degree, or largest mean ACC (arc consistency counts) in its domain. These combinations are called conjunctive width heuristics [Wallace and Freuder, 93]. For static value ordering, values are ordered by increasing ACC. Regarding *dynamic variable ordering* (DVO) in forward checking, variables are ordered either by the largest mean of inconsistency counts in their domains or by minimum domain size, and dynamic value ordering considers values by increasing IC [Freuder and Wallace, 92]. Two heuristics for dynamic variable and value ordering are given in [Larrosa and Meseguer, 95], based on gradients of a local consistency function.

Finally, the notion of *directed arc-consistency* was first introduced by [Dechter and Pearl, 88] in the context of CSP. The use of DAC in MAX-CSP as a method to improve the computation of the lower bound for the current partial assignment is proposed in [Wallace, 94]. In a pre-processing step, DAC are computed for each value following a fixed variable order. This order must remain as SVO for variable instantiation in the forward checking algorithm. DAC counts are added to other counts (distance and IC) to compute a better lower bound for the current partial assignment.

3 Preliminaries

A discrete binary CSP is defined by a finite set of variables $\{X_i\}$ taking values on discrete and finite domains $\{D_i\}$ under a set of binary constraints $\{R_{ij}\}$. A constraint R_{ij} is a subset of $D_i \times D_j$, containing the permitted values for X_i and X_j. The number of variables is n and, without loss of generality, we will assume a common domain D for all the variables, m being its cardinality. A global solution of the CSP is an assignment of values to variables satisfying every constraint. If no solution exists the CSP is overconstrained; in this case we are interested in finding solutions satisfying a maximum number of constraints. This problem is usually referred as MAX-CSP.

P-EFC3 is an efficient algorithm to solve MAX-CSP [Freuder and Wallace, 92] and its pseudo-code appears in Figure 1. P-EFC3 follows the branch and bound schema enhanced with forward checking. It keeps for all feasible values of future variables the number of inconsistencies with previous assignments. The IC associated with value l of variable X_i, ic_{il}, is the number of inconsistencies that value l of X_i has with the assignments of past variables. Denoting by $\boldsymbol{F}$ the set of indexes of future variables, the sum $\sum_{j \in \boldsymbol{F}} \min_k \{ic_{jk}\}$ is a lower bound of the number of inconsistencies

that will occur in future variables if the current partial assignment is extended into a total one. Therefore, this term can be included to compute the lower bound of the current partial assignment (lines 12 and 14). In addition, a value l of a future variable X_i can be pruned if the distance of the current partial assignment plus ic_{il} is not lower than the best distance (lines 23 and 26). Given a variable ordering, the DAC associated to a value l of a variable X_i, dac_{il}, is the number of variables which are arc-inconsistent with value l for X_i and appear after X_i in the ordering [Wallace, 94]. dac_{il} is a lower bound of the number of inconsistencies that X_i will have with variables after X_i in the ordering if l is assigned to X_i.

```
procedure P-EFC3(current_solution,distance,
                 remaining_variables,remaining_domains)
1  Xi := select-next-variable(remaining_variables);
2  values := sort-values(Xi,remaining_domains);
3  while values ≠ ∅ do
4     l := first(values);
5     new_distance := distance + ic_il;
6     if (remaining_variables - Xi = ∅) then
7       if (new_distance < best_distance) then
8          best_distance := new_distance;
9          best_solution := current_solution + (Xi,l);
10      endif
11    else
12      if (new_distance + Σ_{j∈F} min_k{ic_jk} < best_distance) then
13         new_remaining_domains := look_ahead(domains,Xi,l);
14        if (not empty_domain and
              (new_distance + Σ_{j∈F} min_k{ic_jk} < best_distance)) then
15           P-EFC3(current_solution+(Xi,l),new_distance,
                    remaining_variables-Xi,new_remaining_domains);
16        endif
17      endif
18    endif
19    values := values - l;
20 endwhile
endprocedure

function look_ahead (domains,Xi,l)
21 forall j∈F do
22   forall k∈Feasibles do
23     if (new_distance + ic_jk ≥ best_distance) then prune(Xj,k);
24     elsif (inconsistent(Xi,l,Xj,k)) then
25       ic_jk := ic_jk + 1;
26       if (new_distance+ic_jk ≥ best_distance) then prune(Xj,k);
27       endif
28     endif
29   endforall
30 endforall
31 return updated_domains;
endfunction
```

Fig. 1. The extended forward checking algorithm.

4 Using DAC with Forward Checking

The usefulness of using DAC with forward checking was introduced in [Wallace, 94]. It was proposed to take advantage of DAC usage inside the P-EFC3 algorithm in three ways:

1. *Anticipating the detection of dead-ends.* A dead end occurs when it is known that a partial assignment cannot be extended into a total assignment better than the best solution found so far. This is done by comparing the lower bound of the partial assignment with the distance of the best solution found. Wallace proposed the inclusion of inconsistencies among future variables in the lower bound computation [Wallace, 94], using the following expression,

$$\sum_{j \in F} \min_k \{dac_{jk}\}$$

to estimate the minimum number of inconsistencies that will occur among future variables. This term can be added to the expression (lines 12 and 14),

$$new\_distance + \sum_{j \in F} \min_k \{ic_{jk}\}$$

increasing the lower bound of the current node. The three summands can be added because they refer to inconsistencies produced by different constraints: *new_distance* refers to violated constraints among past variables, the sum of minimum IC refers to constraints between past and future variables, and the sum of minimum DAC refers to constraints among future variables. Therefore no constraint is considered more that once. Note that in line 12, it is also possible to add the term dac_{il} to the lower bound, but not in line 14. This is because dac_{il} refers to constraints between X_i and future variables, and line 12 —before *look_ahead*— does not consider that information in its sum of minimum IC. However, in line 14 —after *look_ahead*— this information has become apparent in the IC of arc-inconsistent variables. Therefore, if dac_{il} had been included in line 14, it would have been counted twice. Lines 12 and 14 are replaced by,

12 **if** (new_distance+dac_{il}+$\sum_{j \in F} \min_k\{ic_{jk}\}$+$\sum_{j \in F} \min_k\{dac_{jk}\}$<best_distance)

14 **if** (**and** new_distance+$\sum_{j \in F} \min_k\{ic_{jk}\}$+$\sum_{j \in F} \min_k\{dac_{jk}\}$<best_distance)

Note that the new lower bound is greater than or equal to the previous, so this version of the algorithm cannot visit more nodes than pure P-EFC3.

2. *Increasing the number of unfeasible values.* A feasible value k of a future variable X_j is pruned when it is known that it cannot be in the best solution (lines 23 and 26). The sum of its IC plus DAC counts, $ic_{jk} + dac_{jk}$, is the number of inconsistencies that *at least* will occur if value k is eventually assigned to X_j. Therefore, lines 23 and 26 are replaced by,

23 **if** (new_distance+dac_{il}+ic_{jk}+dac_{jk}≥best_distance) **then** prune(X_j,k)

26 **if** (new_distance+ic_{jk}+dac_{jk} ≥ best_distance) **then** prune(X_j,k)

Again, dac_{il} cannot be used in line 26 because inconsistencies produced by the constraint between X_i and X_j have already been propagated as inconsistency counts by *look_ahead*.

3. *Value ordering*. The order in which feasible values of a variable are considered depends on a heuristic criterion. The information contained in DAC can be used to order values. In [Wallace, 94] values are ordered by increasing DAC (or ACC, when they are computed).

This version of P-EFC3 using DAC was proposed in [Wallace, 94], and we will refer to it as P-EFC3+DAC1. In the following we present some improvements in the use of DAC, which consider again the three points addressed by Wallace (we will refer to this version as P-EFC3+DAC2):

1. *Anticipating the detection of dead-ends*. Since ic_{il} and dac_{il} refer to different sets of constraints, we can take their addition as the minimum number of inconsistencies that the assignment of value l to variable X_i will produce if that assignment is eventually done. Therefore we can replace the two sums of minimum IC and DAC,

$$\sum_{j \in F} \min_k \{ic_{jk}\} + \sum_{j \in F} \min_k \{dac_{jk}\}$$

by the sum of minimum (IC+DAC),

$$\sum_{j \in F} \min_k \{ic_{jk} + dac_{jk}\}$$

as a lower bound of the effect of assigning future variables. Note that,

$$\sum_{j \in F} \min_k \{ic_{jk}\} + \sum_{j \in F} \min_k \{dac_{jk}\} \leq \sum_{j \in F} \min_k \{ic_{jk} + dac_{jk}\}$$

so the use of this expression will be, at least, as effective as the previous one. In the P-EFC3 algorithm, this means to substitute lines 12 and 14 by,

12 **if** (new_distance+dac_{il} + $\sum_{j \in F} \min_k\{ic_{jk}+dac_{jk}\}$ < best_distance)

14 **if** (**and** new_distance + $\sum_{j \in F} \min_k\{ic_{jk}+dac_{jk}\}$ < best_distance)

2. *Increasing the number of unfeasible values*. So far, a value k of a future variable X_j is pruned when the sum of the distance of the current partial assignment plus the number of inconsistencies caused by the assignment of k to X_j is no lower than the best distance. In addition, we can include the minimum number of inconsistencies which will occur among the rest of future variables, no matter which value will be finally assigned to X_j. This minimum number is as follows,

$$\sum_{p \in F\text{-}j} \min_q \{ic_{pq} + dac_{pq}\}$$

Therefore, lines 23 and 26 can be replaced by,

23 **if** (new_distance + dac_{i1} + ic_{jk} + dac_{jk} + $\sum_{p \in F-j} \min_q\{ic_{pq}+dac_{pq}\}$ ≥ best_distance) **then** prune(X_j,k)

26 **if** (new_distance + ic_{jk} + dac_{jk} + $\sum_{p \in F-j} \min_q\{ic_{pq}+dac_{pq}\}$ ≥ best_distance) **then** prune(X_j,k)

Inside the *look_ahead* procedure some IC may be incremented, causing some increase in $\sum_{p \in F-j} \min_k\{ic_{jk}+dac_{jk}\}$. Nevertheless, in our implementation this expression is not updated during the *look_ahead* call; it is computed only once at the beginning of *look_ahead*, and its value is maintained although it may become not updated. However, the algorithm is still correct because IC propagation can only increase this expression, and we use it as a lower bound. It is an open question whether the continuous updating of this sum of minima is cost effective with respect to our implementation. Note that the difference consists in whether this expression is a local variable or a function call inside of *look_ahead*.

3. *Value ordering*. We consider that the information contained in DAC is complementary with the contained in IC, and they can be added to estimate the goodness of a value. In our implementation we order values by increasing IC+DAC as a combination of the heuristics presented in [Freuder and Wallace, 92] where values are ordered by increasing IC, and [Wallace, 94] where values are ordered by increasing DAC or ACC (if it is computed). Note that value ordering is done under a heuristic criterion and its benefit cannot be guaranteed in general.

The three aspects where DAC can improve the performance of P-EFC3 have been introduced separately, but they are closely related. If pruning is more effective, the sum of minimum $ic_{jk}+dac_{jk}$ is higher and dead-ends are detected earlier. If values are considered in a better order, better upper bounds are sooner established, pruning is more effective and dead-ends are found at higher levels of the tree. Therefore, it is expected that the addition of these features will magnify its effect in the performance of the algorithm.

5 Static Variable Ordering Heuristics

It is well known that the use of heuristics for variable and value ordering is of great importance in MAX-CSP algorithms. A right choice of heuristics can produce large savings in the search effort. The use of DAC requires a static ordering of variables because DAC can only be used if variables are considered for instantiation in the same order than DAC were computed. For this reason, we analyze different SVO heuristics to be used with P-EFC3+DAC2. First, let us consider the following observation,

Observation 1: Given a SVO $\{X_1, X_2, ..., X_n\}$ used by P-EFC3 (either using DAC or not) with values ordered by increasing IC (or IC+DAC), P-EFC3 visits at most $\Theta(m^k)$ nodes, where k is the index of the first variable in the ordering from which no constraint exists between variables which are posterior to X_k in the ordering.

Proof (sketched): At level k there is no constraint among future variables, so $dac_{jl} = 0$ $\forall j > k$ and ic_{jl} is exactly the number of inconsistencies that value l will have if it is assigned to X_j. $\sum_{j \in F} \min_q \{ic_{jq} + dac_{jq}\}$ is exactly the number of inconsistencies that future variables will produce if they are instantiated with the values that minimize this expression, which is the best that can be done extending the current partial assignment. When visiting a node at level k, every constraint has become explicit and every potential inconsistency has been propagated into IC of future variables. Therefore, the algorithm will not proceed to deeper levels unless the best current solution can be improved. In this case, due to the value ordering criteria used, level n will be reached from level k without backtracking and a new upper bound will be established. The algorithm will return to level k without attempting new assignments because the upper bound becomes smaller or equal than the lower bound. Therefore, the only exponential growth of the search space is in the k first levels of the tree. □

This observation shows how to determine for P-EFC3 (with or without DAC) backtracking free areas of the search space. It can be extended to all P-EFC algorithms and the proof remains the same. In the following, we discuss two objectives that a SVO should pursue to take the maximum advantage of P-EFC3+DAC2.

1. Based on Observation 1, a first objective is selecting a SVO which *minimizes the* k in order to stop the possible exponential growth of the search space at higher levels of the tree.
2. The efficiency of branch and bound largely depends on the quality of its lower bound. At high levels of the tree it is desirable to have high lower bounds to perform an early detection of dead ends. In our algorithm the lower bound is computed using distance, IC and DAC. Trying to increase the lower bound at early levels of the tree, we can order variables according to three different criteria:
   2.1. Selecting first variables with a high expected contribution to distance.
   2.2. Selecting first variables with a high expected contribution to IC of futures.
   2.3. Selecting variables in such an order that DAC contribution is maximized.

With these objectives in mind we analyze several SVO heuristics, some of them based on the graph topology. The *degree* of a variable X_i is the number of variables constrained with it. Given an ordering, we define the *forward degree* of X_i as the number of variables constrained with X_i and appearing after it in the ordering. Conversely, the *backward degree* of X_i is the number of variables constrained with X_i and appearing before it in the ordering (this concept is also called the *width* of a variable). Obviously, the sum of forward and backward degrees of a variable is the variable degree. Variables can be ordered by:

1. *Decreasing backward degree* (BD). This heuristic was already proposed in [Wallace and Freuder, 93], and it considers first variables most constrained with past variables. When one of these variables is selected, it is expected that its values will have high IC (from the high level of connection with past variables), so after variable assignment the distance of the current partial assignment is likely to increase (objective 2.1). The main disadvantage of this heuristic is the lack of

```
procedure forward_degree_SVO
   V:={X1,...,Xn}
   while V≠0 do
        select as the next variable the one sharing most
        constraints with the rest of variables in V
        remove it from V
   endwhile
endprocedure
```

Fig. 2. Pseudo-code for computing FD SVO heuristic.

information in the first levels of the tree.

2. *Decreasing forward degree* (FD). It considers first variables most constrained with future variables. This heuristic approximates objective 1, because, if a variable X_i is constrained with p future variables and we want to have unconstrained future variables, we must either remove X_i or the other p variables. We will probably achieve sooner the goal if we remove X_i. This heuristic also approximates the objective 2.2, especially in the first levels of the search tree, where IC are low and it may be better to select variables highly constrained with future variables to maximize the propagation of IC towards future variables. Figure 2 shows the pseudocode for computing FD.
3. *Decreasing degree* (DG). This heuristic was already proposed in [Wallace and Freuder, 93], and it can be seen as a combination of the two previous, BD and FD. At first levels of the tree, DG selects variables highly constrained with future variables (FD dominates), but at deep levels DG selects variables highly connected with past variables (BD dominates). Regarding the above mentioned objectives, DG gradually combines the approximation to objectives 1 and 2.2 (FD component) with objective 2.1 (BD component).
4. *Decreasing ACC mean* (AC). ACC were introduced in [Freuder and Wallace, 92]. Here, we consider it as a way to maximize the DAC contribution to the lower bound (objective 2.3). Variables with high ACC will probably have also high DAC if they are considered first in the ordering, because only those arc-inconsistencies referring to prior variables will not be in their DAC.

All these heuristics can be combined, priorizing one and using another (or even a third) to break ties [Wallace and Freuder, 93]. In the following Section, we provide empirical results of several combinations of heuristics for random CSP classes.

6 Experimental Results

We have evaluated empirically the proposed algorithms and SVO heuristics with fixed and variable tightness random CSP. A *fixed tightness random problem* is characterized by $<n,m,p_1,p_2>$ where n is the number of variables, m is the number of values for each variable, p_1 is the graph *connectivity* as the proportion of existing constraints (the number of constrained variable pairs is exactly $p_1 n(n\text{-}1)/2$), and p_2 is the constraint *tightness* as the proportion of forbidden value pairs between two constrained variables (the number of forbidden value pairs is exactly $p_2 m^2$). The

constrained variables and their nogoods are randomly selected [Smith, 94; Prosser, 94]. A *variable tightness random problem* is characterized by $<n,m,p_1,p_2^{inf},p_2^{sup}>$ where the tightness is randomly chosen in the interval $[p_2^{inf},p_2^{sup}]$ for each pair of constrained variables [Larrosa and Meseguer, 95]. Four sets of problems were generated:

1. Fixed tightness $<15,5,p_1,p_2>$, with p_1 taking values 25/105, 50/105, 75/105 and 105/105, and p_2 taking values 1/25, 2/25, ..., 25/25.
2. Fixed tightness $<10,10,p_1,p_2>$,with p_1 taking values 15/45, 25/45, 35/45 and 45/45, and p_2 taking values 1/100, 2/100, ..., 100/100.
3. Variable tightness $<15,5,p_1,1/25,24/25>$, with p_1 taking values 25/105, 45/105, 65/105, 85/105 and 105/105
4. Variable tightness $<10,10,p_1,1/100,99/100>$, with p_1 taking values 15/45, 25/45, 35/45 and 45/45.

generating for each parameter setting 50 problems, forming four sets of 5000, 20000, 200 and 200 instances respectively. They have been solved with an upper bound in the search effort of 40,000,000 consistency checks. All algorithms were implemented in C and run on a SUN SparkStation 20.

The first experiment was devised to evaluate the effect of using DAC with P-EFC3 and the benefits of our implementation. We solved the class of $<n=15, m=5>$ fixed tightness problems using plain P-EFC3, P-EFC3+DAC1 and P-EFC3+DAC2, ordering values by increasing IC, DAC and IC+DAC respectively. Variables were ordered lexicographically. The average search effort, as the number of consistency checks, appears in Fig. 3. It can be observed that the most efficient algorithm is P-EFC3+DAC2, which saves up to 70% of consistency checks performed by P-EFC3+DAC1. The algorithms using DAC are clearly better than pure P-EFC3. In addition, there is a different pattern in the difficulty of problems depending on whether DAC are used or not. If DAC are not used, problems become harder in average when tightness is increased. If DAC are used, we observe an *easy-hard-easy* pattern in the search effort. The left easy part corresponds to problems with low or intermediate tightness. They have solutions satisfying every (or almost every) constraint, and P-EFC3 (either with or without DAC) does not have to invest much effort to find them. The right easy part corresponds to problems with very high tightness. They have many arc-inconsistencies, DAC take high values and have an important contribution to the lower bound, causing P-EFC3 with DAC to perform a more efficient pruning than pure P-EFC3. The hard part, where the peak in the search effort occurs, corresponds to problems with high tightness. For these problems, the DAC effect is not enough to prune at high levels of the tree. For a more comprehensive description of this phenomenon see [Larrosa and Meseguer, 96].

The second experiment was devoted to evaluate single SVO heuristics. Since all heuristics, except ACC mean, are based on graph topology, full connectivity problems were discarded in this experiment. We solved the class of $<n=15,m=5>$ fixed tightness problems using P-EFC3+DAC2 and the following SVO heuristics: BD, FD, DG and AC, breaking ties lexicographically. The results appear in Figure 4, from which we observe that BD, without a second criteria to break ties, is a bad heuristic (as was already affirmed in [Wallace and Freuder, 93]). AC gives a good advantage with respect to BD. FD and DG are the best heuristics, without a clear

difference between them. The dominance of FD and DG can be justified by the fact that they include several objectives for SVO heuristics (see Section 5). The savings caused by SVO heuristics are really important; this can be realized comparing Figures 3 and 4.

The third experiment was devoted to evaluate combinations of two SVO heuristics, where the second is used to break ties. We selected a second criterion complementary to the first with respect to the objectives discussed in Section 5. We solved the class of $<n=15,m=5>$ fixed tightness problems using P-EFC3+DAC2, testing the following combinations: FD/BD, FD/AC, BD/FD, BD/AC and DG/AC. The results appear in Figure 5, from which we observe that BD as first criterion, with a second heuristic to break ties (FD or AC), is a bad SVO heuristic. This observation does not match with the experiments presented in [Wallace and Freuder, 93], although this discrepancy may be explained by the differences in the problems to solve (in [Wallace and Freuder, 93] random problems had variable domain and tightness). With respect to the three other combinations, there is not a large difference among them; it seems that FD/BD slightly dominates in the peak of the search effort. Comparing Figures 4 and 5, we observe that the addition of a second criterion for FD and DG causes only small improvements in their performance.

The fourth experiment aimed at comparing static and dynamic ordering heuristics. We solved the class of $<n=10,m=10>$ fixed tightness problems with P-EFC3 using the *largest mean* DVO heuristic (LM, it selects the variable with the largest mean of IC among its feasible values [Freuder and Wallace, 92]), and with P-EFC3+DAC1 and P-EFC3+DAC2 using both the FD/BD heuristic. The results appear in Figure 6, from which we observe that DAC usage plus FD/BD dominates clearly pure P-EFC3 with LM. In addition, they confirm that P-EFC3+DAC2 dominates P-EFC3+DAC1 when the FD/BD heuristic is used.

The fifth experiment was devoted to check the performance of the algorithms on variable tightness problems. We solved the classes of $<n=10,m=10>$ and $<n=15,m=5>$ variable tightness problems with P-EFC3 using LM heuristic, and P-EFC3+DAC1 and P-EFC3+DAC2 using FD/BD. The results appear in Figure 7, where the horizontal axis represents varying connectivity. From these results we observe that P-EFC3+DAC2 surpasses clearly P-EFC3+DAC1 in the whole set of problems, which indicates that the improvement in performance showed for fixed tightness does not depend on constraint homogeneity. In addition, we see that P-EFC3 with LM surpasses P-EFC3+DAC1 in one problem class. Further experiments on smaller variable tightness intervals are required, to qualify properly this phenomenon.

In summary, experimental results confirm clearly the practical benefits of the proposed algorithm, P-EFC3+DAC2, for fixed and variable tightness problems. The benefits over a previous approach are maintained when using SVO heuristics, which in addition, can largely improve the algorithm performance. The analysis of heuristics based on SVO objectives (Section 5) provided some explanation on their relative efficiency. Results suggest that the combination FD/BD is the option of choice, because it is slightly superior to other combinations and it is quite easy to compute.

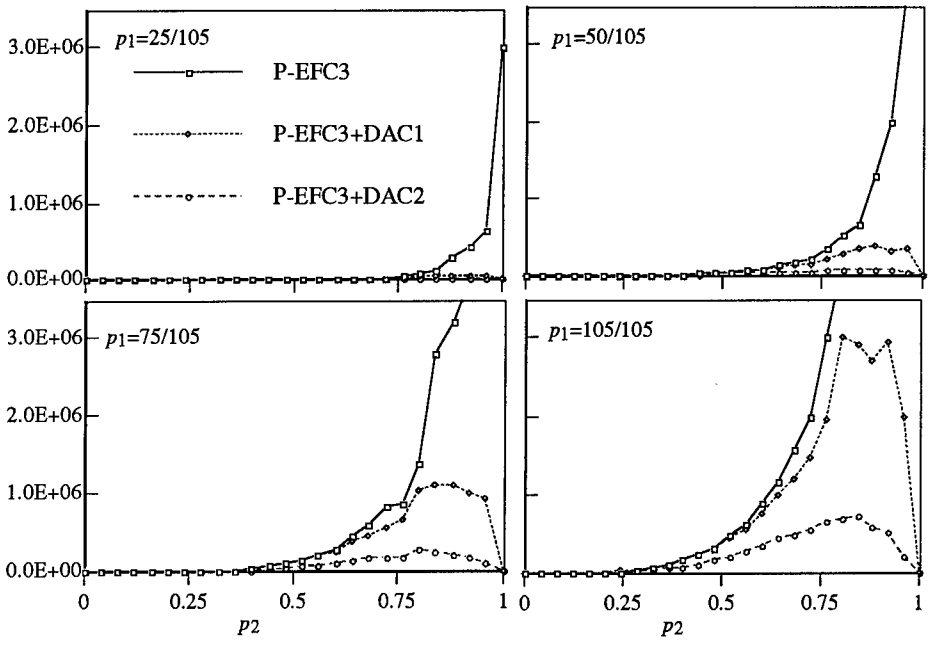


Fig. 3. Number of consistency checks for the class of <n=15, m=5> fixed tightness problems solved with different algorithms with lexicographical variable ordering.

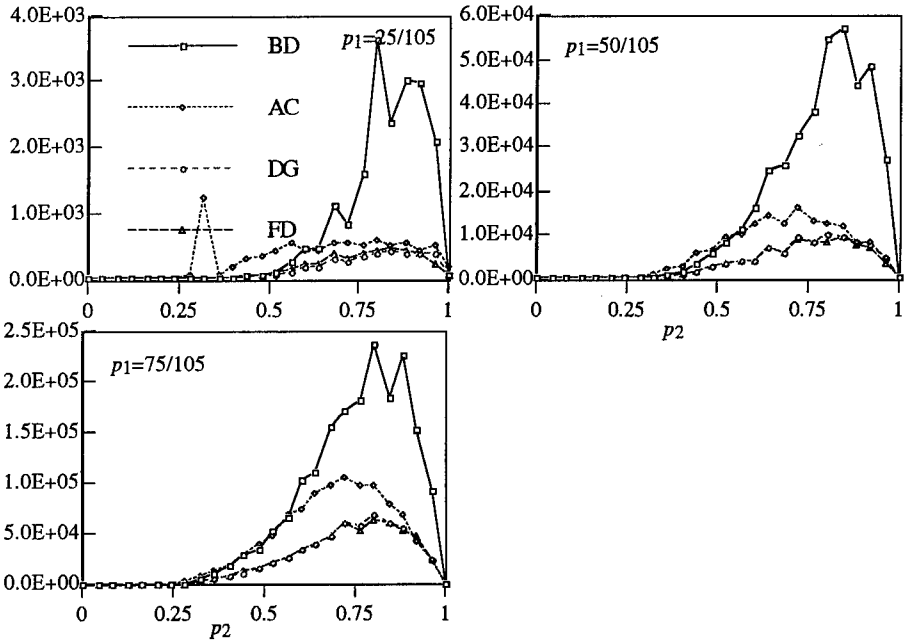


Fig. 4. Number of consistency checks for the class of <n=15, m=5> fixed tightness problems solved with P-EFC3+DAC2 with different SVO heuristics.

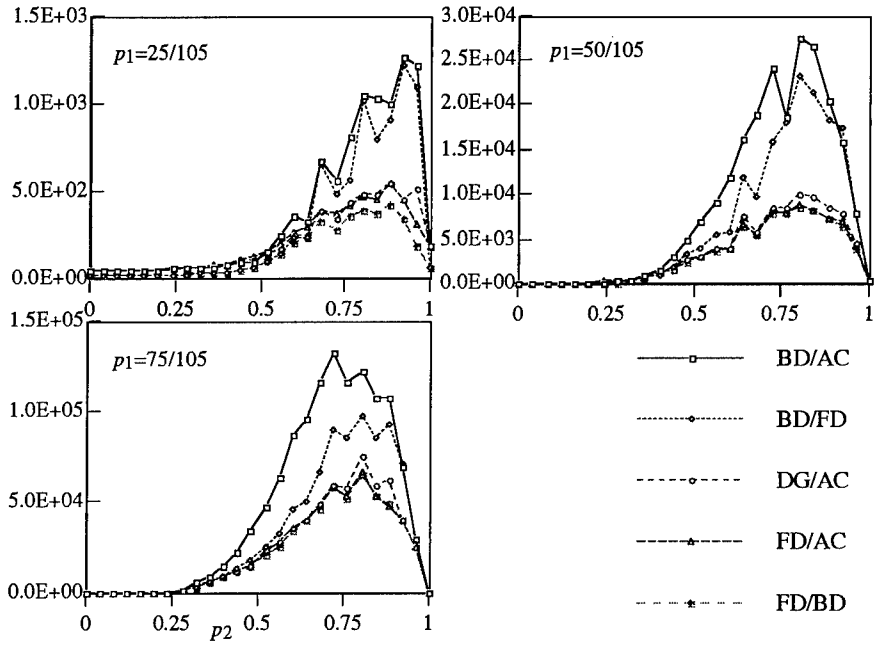


Fig. 5. Number of consistency checks for the class of <n=15, m=5> fixed tightness problems solved with P-EFC3+DAC2 with different SVO heuristic combinations.

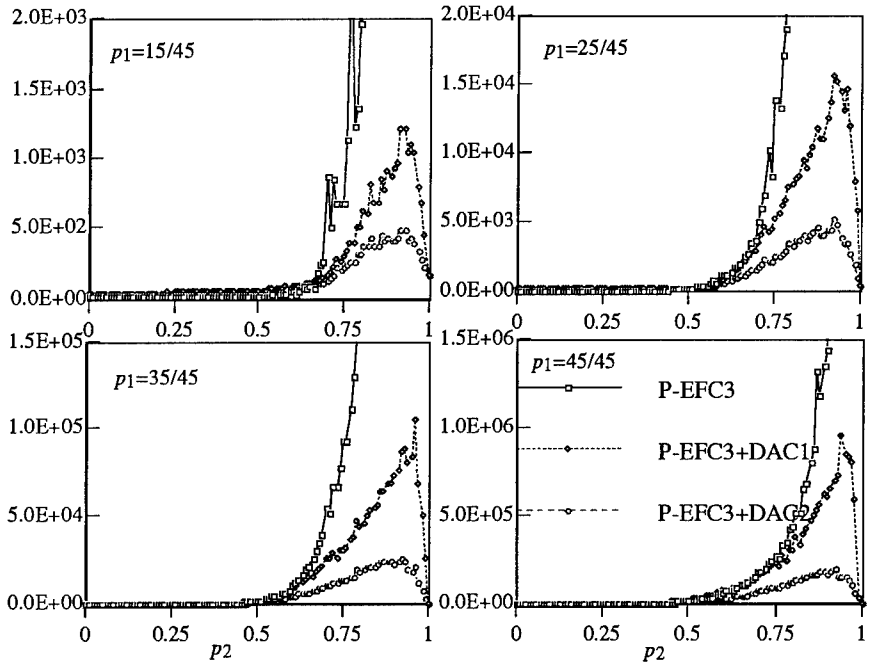


Fig. 6. Number of consistency checks for the class of <n=10, m=10> fixed tightness problems solved with different algorithms and heuristics.

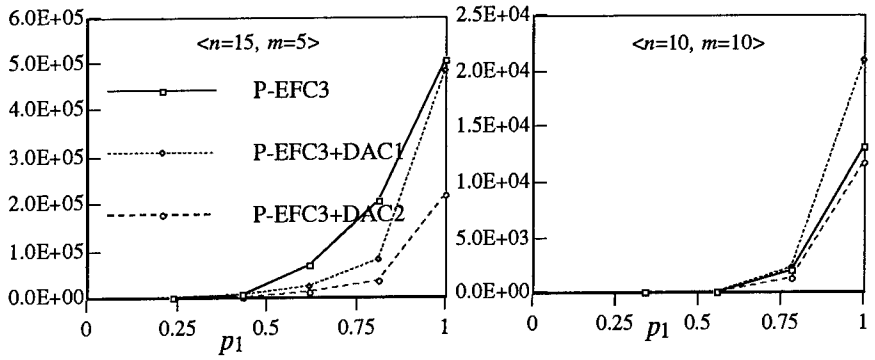


Fig. 7. Number of consistency checks for two classes of variable tightness problems solved with different algorithms.

7 Conclusions

From this work we extract the following conclusions. First, information coming exclusively from future variables is very valuable for improving the performance of MAX-CSP solving algorithms. This idea was first developed in the interesting work of [Wallace, 94] introducing the use of DAC counts, which has been enhanced in the present paper. Second, considering this information from future variables, we have analyzed some SVO heuristics already proposed and we have generated new heuristics which have been found effective. And third, although DAC use could be considered in principle as a technical refinement for existing algorithms without any other implication, its usage has allowed us to discover a region of very tight problems which are easy for MAX-CSP.

Acknowledgements

We thank Fahiem Bacchus for providing us his CSP random problem generator. We thank Maite López for her technical support on writing this paper, and Dolores Bondía and Romero Donlo for their collaboration on the preparation of this work.

References

Bakker R., Dikker F., Tempelman F. and Wognum P. (1993). Diagnosing and solving overdetermined constraint satisfaction problems, *Proceedings of IJCAI-93*, 276-281.

Dechter R. and Pearl J. (1988). Network-based heuristics for constraint satisfaction problems, *Artificial Intelligence*, 34, 1-38.

Feldman R. and Golumbic M. C. (1990). Optimization algorithms for student scheduling via constraint satisfiability, *Computer Journal*, vol. 33, 356-364.

Fox M. (1987). *Constraint-directed Search: A Case Study on Jop-Shop Scheduling*. Morgan-Kauffman.

Freuder E. C. and Wallace R. J. (1992). Partial constraint satisfaction, *Artificial Intelligence*, 58:21-70.

Larrosa J. and Meseguer P. (1995). Optimization-based Heuristics for Maximal Constraint Satisfaction, *Proceedings of CP-95*, 103-120.

Larrosa J. and Meseguer P. (1996). Phase Transition in MAX-CSP, *Proceedings of ECAI-96*, 190-194.

Prosser P. (1994). Binary constraint satisfaction problems: some are harder than others, *Proceedings of ECAI-94*, 95-99.

Smith B. M. (1994). Phase thansition and the mushy region in constraint satisfaction problems, *Proceedings of ECAI-94*, 100-104.

Wallace R. J. and Freuder E. C. (1993). Conjunctive width heuristics for maximal constraint satisfaction, *Proceedings of AAAI-93*, 762-778.

Wallace R. J. (1994).Directed Arc Consistency Preprocessing as a Strategy for Maximal Constraint Satisfaction, *ECAI94 Workshop on Constraint Processing*, M. Meyer editor, 69-77.

A New Approach for Weighted Constraint Satisfaction: Theoretical and Computational Results

Hoong Chuin LAU

Dept. of Computer Science, Tokyo Institute of Technology
2-12-1 Ookayama, Meguro-ku, Tokyo 152, Japan
hclau@cs.titech.ac.jp

Abstract. We consider the Weighted Constraint Satisfaction Problem which is a central problem in Artificial Intelligence. Given a set of variables, their domains and a set of constraints between variables, our goal is to obtain an assignment of the variables to domain values such that the weighted sum of satisfied constraints is maximized. In this paper, we present a new approach based on randomized rounding of semidefinite programming relaxation. Besides having provable worst-case bounds, our algorithm is simple and efficient in practice, and produces better solutions than other polynomial-time algorithms such as greedy and randomized local search.

1 Introduction

An instance of the binary Weighted Constraint Satisfaction Problem (W-CSP) is defined by a set of variables, their associated domains of values and a set of binary constraints governing the assignment of variables to values. Each constraint is associated with a positive integer weight. The output is an assignment which maximizes the weighted sum of satisfied constraints. W-CSP is a generalization of important combinatorial optimization problems such as the Maximum Cut Problem. Many real-world problems can be represented as W-CSP, among which are scheduling and timetabling problems. In scheduling for example, our task is to assign resources to jobs under a set of constraints, some of which are more important than others. Most often, instances are over-constrained and no solution exists that satisfies all constraints. Thus, our goal is to find an assignment which maximizes the weights of the satisfied constraints.

Finding optimal solutions of W-CSP is known to be computationally hard. In the CSP research community, work in W-CSP is not as abundant as work in the standard CSP. Freuder and Wallace [3, 4] gave the first formal definition of PCSP which is a special case of W-CSP having unit weights. For PCSP, the objective is to satisfy as many constraints as possible. Freuder and Wallace proposed a polynomial time algorithm based on reverse breadth-first search to solve PCSP whose underlying constraint network is a tree. For the general PCSP, they proposed a general framework based on branch-and-bound and its enhancements

in [17, 16]. Incomplete algorithms which yield near-optimal or *approximate* solutions have also been investigated, including heuristic repair methods [18], the connectionist architecture GENET [12] and guided local search [14].

Our work is motivated mainly by the potential application of W-CSP in scheduling. With the rapid increase in the speed of computing and the growing need for efficiency in scheduling, it becomes increasingly important to explore ways of obtaining better schedules at some extra computational cost, short of going all the way towards the usually futile attempt of finding a guaranteed optimal schedule. Our paper describes a new approach meant to achieve this goal.

Another highlight of our approach is that the solution obtained has a provable worst-case bound in terms of weight when compared with the optimal solution. This contrasts with conventional incomplete algorithms which are empirically good but are not guaranteed to perform well in the worst case. In [8], we performed a worst-case analysis of local search for PCSP. In [9], Lau and Watanabe obtained lower and upper bounds of various rounding methods for W-CSP. In this paper, our emphasis is to apply the theory proposed in [9] to practice: we will give a careful account of the mathematical modelling, the algorithm design, as well as experimental performance of our algorithm. The experimental results turn out, to our pleasant surprise, to be much stronger than the theoretical worst-case bound. Nevertheless, besides being theoretically interesting, the knowledge of the worst-case performance gives us some peace of mind: that our algorithm will never perform embarrassingly poorly. This is an important factor to consider in scheduling critical resources.

Randomized Rounding Our approach is heavily based on the notion of *randomized rounding*. Randomized algorithms have proved to be powerful in the design of approximate algorithms for combinatorial optimization problems. An interesting and efficient algorithmic paradigm is that of randomized rounding, due to Raghavan and Thompson [11]. The key idea is to formulate a given optimization problem as an integer program and then find an approximate solution by solving a polynomial-time solvable convex mathematical program such as a linear program. The linear program must constitute a "relaxation" of the problem under consideration, i.e. all integer solutions are feasible for the linear program and have the same value as they do in the integer program. One easy way to do this is to drop the integrality conditions on the variables. Given the optimal fractional solution of the linear program, the question is how to find a good integer solution. Traditionally, one rounds the variables to the nearest integers. Randomized rounding is a technique which treats the values of the fractional solution as a probability distribution and obtains an integer solution using this distribution. Raghavan and Thompson showed, using basic probability theory, that the values chosen under the distribution do in fact yield a solution near the expectation, thus giving good approximate solutions to the integer program.

Essentially, we will solve W-CSP using what is known as *randomized rounding of a semidefinite program relaxation*. A semidefinite program is the optimization

problem of a linear function of a symmetric matrix subject to linear equality constraints and the constraint that the matrix be positive semidefinite. Semidefinite programming is a generalization of linear programming and a special case of convex programming. The simplex method can be generalized to semidefinite programs. By interior point methods, one can show that semidefinite programming is solvable in polynomial time under some realistic assumptions. Semidefinite programming, like linear programming, has been an active research topic among the Operations Research community; for details, see a good survey written by Alizadeh [1]. In practice, there are solvers which yield solutions quickly for reasonably large instances, as our experiments would show. The idea is to represent W-CSP as a quadratic integer program and solve a corresponding instance of semidefinite programming. The solution returned by the semidefinite program is then rounded to a valid assignment by randomized rounding. This approach yields a randomized algorithm. To convert it into a deterministic algorithm, we apply the *method of conditional probabilities* which is a well-known probabilistic method in combinatorics. This method is clearly explained in the text of Alon and Spencer [2] and the idea will be adapted for our purpose in this paper.

Hence, our approach offers a *polynomial-time algorithm* which can be efficiently implemented. Our experiments illustrate that this approach can handle problems of sizes beyond what enumerative search algorithms can handle, and thus is a candidate for solving real-world large-scale problem instances. Recently, Goemans and Williamson applies a similar approach to find approximate solutions for the Maximum Cut Problem [6]. Besides having a strong provable worst-case bound, their computational experiments show that, on a number of different types of random graphs, their algorithm yields solutions which are usually within 4% from the optimal solution.

This paper is organized as follows. Section 2 gives the definitions and notations which will be used throughout the paper. In section 3, we introduce the method of conditional probabilities and derive a linear-time greedy algorithm which can be used to derandomize our randomized rounding algorithms. We prove that a naive application of the greedy algorithm always returns a solution whose weight is guaranteed to be a fraction of s times the total weight, where $0 \leq s \leq 1$ is the *strength*[1] of the constraints. In section 4, we show how to formulate W-CSP by quadratic integer programs. We discuss how randomized rounding can be used to yield solutions which have a constant worst-case bound for domain size $k \leq 3$. A more sophisticated rounding scheme due to Goemans and Williamson [6] will be discussed, which yields a better worst-case bound for domain size 2. However, their algorithm cannot be de-randomized in a practical sense to date. In section 5, we compare the performance of our algorithm with other approaches experimentally on random W-CSP instances. For all 300 instances, we are able to obtain solutions whose values are within 4% from the optimal. This is significantly better than the other polynomial-time algorithms under comparison.

[1] For standard unweighted CSP, this measure is often known as *looseness*.

2 Preliminaries

Let $V = \{1, \ldots, n\}$ be a set of variables. Each variable has a *domain* which contains the set of values that can be assigned. For simplicity, we assume that all domains have fixed size k and are equal to the set $K = \{1, \ldots, k\}$. A *constraint* between two variables i and l is a binary relation over $K \times K$ which defines the pairs of values that i and l can take simultaneously. Given an assignment $\sigma : V \longrightarrow K$, the constraint is said to be *satisfied* iff the pair (σ_i, σ_l) is an element of the relation. The Weighted Constraint Satisfaction Problem (W-CSP) is defined by a set V of variables, a collection M of constraints, integer k, and a weight function $w : M \longrightarrow Z^+$. The output is an assignment $\sigma : V \longrightarrow K$ such that weighted sum of satisfied constraints (or simply weight) is maximized.

Denote by W-CSP(k) the class of instances with domain size k. For each constraint $j \in M$, let w_j, R_j and $s_j = \|R_j\|/k^2$ denote its weight, relation and *strength* (looseness) respectively; let α_j and β_j denote the indices of the two variables incident on j; and let $c_j(u,v) = 1$ if $(u,v) \in R_j$ and 0 otherwise. That is, $c_j(u,v)$ indicates whether the value pair (u,v) is consistent in constraint j. Let $s = \sum_{j \in M} w_j s_j / \sum_{j \in M} w_j$ denote the strength of a W-CSP instance (i.e. the weighted average strengths of all its constraints). Note that $s \geq 1/k^2$ because a constraint relation contains at least 1 out of the k^2 possible pairs. A W-CSP instance is *satisfiable* iff there exists an assignment which satisfies all constraints simultaneously.

A quadratic integer program (QIP) has the form:

$$\begin{array}{lll} \text{maximize } \sum_{i,l} a_{il} x_i x_l + \sum_i b_i x_i & & \\ \text{subject to } \sum_i c_{ij} x_i = d_j & & \text{for all constraints } j \\ x_i \text{ integer}, & & \text{for all variables } i. \end{array}$$

We say that a maximization problem P *can be approximated within* $0 < \epsilon \leq 1$ iff there exists a polynomial-time algorithm A such that for all input instances y of P, the ratio $A(y)/OPT(y)$ is at least ϵ, where $A(y)$ and $OPT(y)$ denote the objective value of the solution returned by A and the optimal objective value of y respectively. The quantity ϵ is commonly known as the *performance guarantee* or *approximation ratio* for P. The ratio is *absolute* if the denominator is the maximum possible objective value instead of $OPT(y)$. In the case of W-CSP for example, the maximum possible objective value is the sum of edge weights, although the optimal value can be much smaller. Hence, the absolute ratio is always a lower bound of (and therefore better bound than) the performance guarantee. Observe that the ratio is as close to 1 as the solution is close to an optimum solution.

3 Method of Conditional Probabilities

In this section, we introduce the method of conditional probabilities. We derive an efficient linear-time greedy algorithm which can be used to derandomize the

randomized rounding algorithm presented later. We will also analyse the worst-case performance of a naive application of the greedy algorithm.

Consider an instance of W-CSP. Suppose we are given an n by k matrix $\Pi = (p_{iu})$ such that all $p_{i,u} \in [0..1]$ and $\sum_{u=1}^{k} p_{i,u} = 1$ for all $1 \leq i \leq n$. If we assign each variable i independently to value u with probability $p_{i,u}$, we obtain a probabilistic assignment whose expected weight is given by

$$\hat{W} = \sum_{j \in M} w_j \times \Pr[\text{constraint } j \text{ is satisfied}] = \sum_{j \in M} w_j \Big(\sum_{u,v \in K} p_{\alpha_j,u} \cdot p_{\beta_j,v} \cdot c_j(u,v) \Big).$$

The method of conditional probabilities specifies that there must exist an assignment whose weight is at least $\hat{W}$ and that such an assignment can be found deterministically in polynomial time provided that certain conditional probabilities can be computed efficiently.

We will show how to derive one such algorithm. Let σ be a partial assignment such that variables $1, \ldots, t-1$ have been assigned values, and variables $t, \ldots, n$ are unassigned. Define:

$$q_{i,u} = \begin{cases} p_{i,u}, & \text{if } i \geq t \\ 1, & \text{if } i \leq t-1 \text{ and } \sigma_i = u \\ 0, & \text{otherwise.} \end{cases}$$

That is, $q_{i,u}$ is the probability that variable i is assigned value u *with respect to the partial assignment* σ. Notice that if σ is completely unassigned, then $q_{i,u} = p_{i,u}$ for all i and u. The expected weight of σ is given by:

$$\tilde{W} = \sum_{j \in M} w_j \left(\sum_{u,v \in K} q_{\alpha_j,u} \cdot q_{\beta_j,v} \cdot c_j(u,v) \right).$$

This suggests that we can construct a complete assignment of expected weight at least $\hat{W}$ iteratively in a greedy fashion: each time, assign a variable to a value such that expected weight of the resulting partial assignment is the *maximum* over all partial assignments. The greedy (derandomization) algorithm is codified as follows:

```
procedure Greedy:
begin
  set W̃ = Ŵ;
  for all i = 1,...,n do
    for all u ∈ K compute W̃_u;
    assign i to v s.t. W̃_v is maximized;
    set W̃ = W̃_v;
  endfor
end.
```

At the beginning of iteration i, $\tilde{W}$ denotes the expected weight of the partial assignment where variables $1,\ldots,i-1$ are fixed and variables $i,\ldots,n$ are assigned according to distribution Π. $\tilde{W}_u$ denotes the expected weight of the same partial assignment, except that variable i is fixed to the value u. From the law of conditional probabilities:

$$\tilde{W} = \sum_{u=1}^{k} \tilde{W}_u \cdot p_{i,u}.$$

Since we always pick v such that $\tilde{W}_v$ is maximized, $\tilde{W}$ is non-decreasing in all iterations.

Therefore, to obtain assignments of large weights, the key factor is to obtain the probability distribution matrix Π such that the *expected weight* is as large as possible. In the following, we consider the most naive probability distribution – the *random* assignment, i.e. for all i and u, we have $p_{i,u} = 1/k$. By linearity of expectation (i.e. expected sum of random variables is equal to the sum of expected values of random variables), the expected weight of the random assignment is given by,

$$\hat{W} \;=\; \sum_{j \in M} w_j \cdot s_j \;=\; s \sum_{j \in M} w_j.$$

That is, the expected weight is s times the total edge weights, implying that W-CSP(k) can be approximated within absolute ratio s.

For this naive approach, $\tilde{W}_u$ can be derived from $\tilde{W}$ as follows. Maintain a vector r where r_j stores the probability that constraint j is satisfied given that variables $1,\ldots,i-1$ are fixed and the remaining variables assigned randomly. Then, $\tilde{W}_u$ is just $\tilde{W}$ offset by the change in probabilities of satisfiability of those constraints incident to variable i. More precisely,

$$\tilde{W}_u = \tilde{W} + \sum_{j \text{ incident to } i} w_j(r'_j - r_j)$$

where r'_j is the new probability of satisfiability of constraint j. Letting l be the second variable connected by j, r'_j is computed as follows:

if $l < i$ (i.e. l has been assigned)
then set r'_j **to 1 if** $(\sigma_l, u) \in R_j$ **and 0 otherwise**
else set r'_j **to the fraction** $\#\{v \in K \,:\, (u,v) \in R_j\}/k$.

Clearly, the computation of each $\tilde{W}_u$ takes $O(m_i k)$ time, where m_i is the degree of variable i. Hence, the total time needed is $O(\sum m_i k^2) = O(mk^2)$, which is linear in the size of the input.

4 Randomized Rounding of Semidefinite Program

In this section, we present our main algorithm. Essentially, the idea is to represent W-CSP as a QIP and apply randomized rounding to its semidefinite programming relaxation. The resulting randomized algorithm is then de-randomized into a deterministic algorithm by the method of conditional probabilities.

4.1 A Simple and Efficient Rounding Scheme

Consider an instance of W-CSP(k) and formulate a corresponding 0/1 QIP instance (Q) as follows. For every variable $i \in V$, define k 0/1 variables $x_{i,1}, \ldots, x_{i,k}$ in (Q) such that i is assigned to u in the W-CSP instance iff $x_{i,u}$ is assigned to 1 in (Q).

$$\begin{aligned}
&\text{Q: maximize } \sum_{j \in M} w_j f_j(x) \\
&\text{subject to } \sum_{u \in K} x_{i,u} = 1 \text{ for } i \in V \quad [1] \\
&\qquad x_{i,u} \in \{0,1\} \text{ for } i \in V \text{ and } u \in K
\end{aligned}$$

In the above formulation, $f_j(x) = \sum_{u,v} c_j(u,v) x_{\alpha_j,u} x_{\beta_j,v}$ encodes the satisfiability of constraint j and hence the objective function gives the weight of the assignment. Inequality [1] ensures that every W-CSP variable gets assigned to exactly one value.

Next, convert this 0/1 QIP into an equivalent QIP (Q′) whose variables takes values $\{-1,+1\}$:

$$\begin{aligned}
&\text{Q′: maximize } \sum_{j \in M} w_j f'_j(x) \\
&\text{subject to } \sum_{u \in K} x_0 x_{i,u} = -(k-2) \text{ for } i \in V \\
&\qquad x_{i,u} \in \{-1,+1\} \text{ for } i \in V \text{ and } u \in K \\
&\qquad x_0 = +1
\end{aligned}$$

Here, we have $f'_j(x) = \frac{1}{4} \sum_{u,v} c_j(u,v) \left(1 + x_0 x_{\alpha_j,u}\right) \left(1 + x_0 x_{\beta_j,v}\right)$. The reason for introducing a dummy variable x_0 is so that all terms occurring in the formulation are quadratic, which is necessary for the subsequent semidefinite programming relaxation.

Having formulated the W-CSP instance as a QIP, the next step is to find an appropriate relaxation which is polynomial-time solvable. One such candidate is the linear programming relaxation. It has been shown that linear programming relaxations do not yield a strong bound compared with semidefinite programming relaxations for small domain sizes [9]. In the following, we will only discuss semidefinite programming relaxations.

The essential idea is to coalesce a quadratic term $x_i x_j$ into a matrix variable $y_{i,j}$. Let Y denote the $(kn+1) \times (kn+1)$ matrix comprising these matrix variables. The resulting relaxation problem (P) is the following:

$$\begin{aligned}
&\text{P: maximize } \sum_{j \in M} w_j F_j(Y) \\
&\text{subject to } \sum_{u \in K} y_{0,iu} = -(k-2) \text{ for } i \in V \\
&\qquad y_{iu,iu} = 1 \text{ for } i \in V \text{ and } u \in K \quad [2] \\
&\qquad y_{0,0} = 1 \\
&\qquad Y \text{ symmetric positive semidefinite.}
\end{aligned}$$

Here, $F_j(X) = \frac{1}{4}\sum_{u,v} c_j(u,v)(1 + y_{\alpha_j u,\beta_j v} + y_{0,\alpha_j u} + y_{0,\beta_j v})$.

Hence, we have a semidefinite program, which can be solved in polynomial time within an additive factor (see [1]). By a well-known theorem in Linear Algebra, a $t \times t$ matrix Y is symmetric positive semidefinite iff there exists a full row-rank matrix $r \times t$ ($r \leq t$) B such that $Y = B^T B$ (see for example, [7]). One such matrix B can be obtained in $O(n^3)$ time by an incomplete Cholesky's decomposition. Since Y has all 1's on its diagonal (by inequality [2]), the decomposed matrix B corresponds precisely to a list of t unit-vectors $X_1, \ldots, X_t$ where column c of B gives the vector X_c. Furthermore, these vectors have the nice property that $X_c \cdot X_{c'} = y_{c,c'}$. The notation $X_1 \cdot X_2$ denote the inner product of the vectors X_1 and X_2.

Domain Size 2 and 3 We propose the following randomized algorithm:

1. (Relaxation) Solve the semidefinite program (P) to optimality (within an additive factor) and obtain an optimal set of vectors X^*.
2. (Randomized Rounding) Construct an assignment for the W-CSP instance as follows. For each i, assign variable i to value u with probability $1 - \frac{\arccos(X_0^* \cdot X_{i,u}^*)}{\pi}$.

The Rounding step has the following intuitive meaning: the smaller the angle between $X_{i,u}^*$ and X_0^*, the higher the probability that i would be assigned to u.

For the case of $k = 2$, the expected weight of the probabilistic assignment can be shown to be at least 0.408 times the weight of the optimal solution [9]. The randomized rounding step can be converted into a deterministic algorithm using the technique discussed in section 3. Hence, we can approximate W-CSP(2) within a worst-case bound of 0.408.

For the case of $k = 3$, the technical difficulty is in ensuring that the sum of probabilities of assigning a variable to the three values is exactly 1. Fortunately, by introducing additional valid inequalities, it is possible to enforce this condition, which we will now explain.

Call two vectors X_1 and X_2 *opposite* if $X_1 = -X_2$.

Lemma 1. *Given 4 unit vectors a, b, c, d, if*

$$a \cdot b + a \cdot c + a \cdot d = -1 \quad (1)$$
$$b \cdot a + b \cdot c + b \cdot d = -1 \quad (2)$$
$$c \cdot a + c \cdot b + c \cdot d = -1 \quad (3)$$
$$d \cdot a + d \cdot b + d \cdot c = -1 \quad (4)$$

then a, b, c and d must form two pairs of opposite vectors.

Proof. $\frac{1}{2}[(3) + (4) - (1) - (2)]$ gives:

$$a \cdot b = c \cdot d.$$

Similarly, one can show that $a \cdot c = b \cdot d$ and $a \cdot d = b \cdot c$. This means that they form either two pairs of opposite vectors or two pairs of equal vectors. Suppose we have the latter case, and w.l.o.g., suppose $a = b$ and $c = d$. Then, by (1), $a \cdot c = a \cdot d = -1$, implying that we still have two pairs of opposite vectors (a, c) and (b, d). □

Now add the following set of $4n$ valid equations into (Q'). For all i:

$$x_0(x_{i,1} + x_{i,2} + x_{i,3}) = -1$$
$$x_{i,1}(x_0 + x_{i,2} + x_{i,3}) = -1$$
$$x_{i,2}(x_0 + x_{i,2} + x_{i,3}) = -1$$
$$x_{i,3}(x_0 + x_{i,2} + x_{i,3}) = -1$$

By Lemma 1, the relaxation problem (P) will return a set of vectors with the property that for each i, there exists at least one vector $\tilde{X} \in \{X_{i,1}, X_{i,2}, X_{i,3}\}$ opposite to X_0 while the remaining two are opposite to each other. Noting that $1 - \frac{\arccos(X_0 \cdot \tilde{X})}{\pi} = 0$, the sum of probabilities of assigning i to the other two values is exactly 1. Thus, we have reduced the case of $k = 3$ to the case of $k = 2$.

Larger Domain Size Note that the above rounding works only for cases of $k \leq 3$. For domain size greater than 3, we use the following rounding strategy:

assign variable i to value u with probability $\frac{1+X_0^* \cdot X_{i,u}^*}{2}$.

This rounding scheme always works for all values of k since the sum of probabilities for each variable i is equal to

$$\frac{1}{2}\sum_{u=1}^{k}\left(1 + X_0^* \cdot X_{i,u}^*\right) = 1.$$

Unfortunately, we are unable to date to analyse the worst case performance of this rounding scheme.

4.2 Rounding Scheme of Goemans and Williamson

Recently, Goemans and Williamson [6] proposed a nice rounding scheme for approximating the Maximum Satisfiability Problem. This scheme can be adopted to give an improved bound for W-CSP(2).

Model a given instance of W-CSP by the following QIP. Each variable has domain $\{-1, +1\}$. In this way, we can directly use $x_i \in \{-1, +1\}$ to indicate the value assigned to variable i. Introduce an additional variable x_0. Again, the variable i is assigned to $+1$ iff $x_i = x_0$.

$$\text{Q: maximize } \sum_{i<l} w_j f_j(x)$$
$$\text{subject to } \quad x_i \in \{-1, +1\} \text{ for } i \in V \bigcup \{0\}$$

$(+1,+1)$	$(+1,-1)$	$(-1,+1)$	$(-1,-1)$	$f_j(x)$
$\surd$	$\surd$	$\surd$	$\surd$	not a constraint
$\surd$	$\surd$	$\surd$	$\times$	$\frac{1}{4}((1+x_0x_i)+(1+x_0x_l)+(1-x_ix_l))$
$\surd$	$\surd$	$\times$	$\surd$	$\frac{1}{4}((1+x_0x_i)+(1-x_0x_l)+(1+x_ix_l))$
$\surd$	$\times$	$\surd$	$\surd$	$\frac{1}{4}((1-x_0x_i)+(1+x_0x_l)+(1+x_ix_l))$
$\times$	$\surd$	$\surd$	$\surd$	$\frac{1}{4}((1-x_0x_i)+(1-x_0x_l)+(1-x_ix_l))$
$\surd$	$\surd$	$\times$	$\times$	$\frac{1}{2}(1+x_0x_i)$
$\surd$	$\times$	$\surd$	$\times$	$\frac{1}{2}(1-x_0x_i)$
$\surd$	$\times$	$\times$	$\surd$	$\frac{1}{2}(1+x_ix_l)$
$\times$	$\surd$	$\surd$	$\times$	$\frac{1}{2}(1-x_ix_l)$
$\times$	$\surd$	$\times$	$\surd$	$\frac{1}{2}(1-x_0x_l)$
$\times$	$\times$	$\surd$	$\surd$	$\frac{1}{2}(1+x_0x_l)$
$\surd$	$\times$	$\times$	$\times$	$\frac{1}{4}((1+x_0x_i)+(1+x_0x_l)+(1+x_ix_l)-2)$
$\times$	$\surd$	$\times$	$\times$	$\frac{1}{4}((1+x_0x_i)+(1-x_0x_l)+(1+x_ix_l)-2)$
$\times$	$\times$	$\surd$	$\times$	$\frac{1}{4}((1-x_0x_i)+(1+x_0x_l)+(1+x_ix_l)-2)$
$\times$	$\times$	$\times$	$\surd$	$\frac{1}{4}((1-x_0x_i)+(1-x_0x_l)+(1+x_ix_l)-2)$
$\times$	$\times$	$\times$	$\times$	not a constraint

Table 1. Table of functions associated with constraint relations. Assume that constraint j is incident to variables i and l. The symbols $\surd$ and $\times$ indicate whether each value pair is an element of the relation.

where $f_j(x)$ encodes the satisfiability of constraint j. Table 1 gives the function f_j associated with all 16 possible constraint relations.

Therefore, the problem (Q) can be expressed as:

$$\begin{aligned} &\text{Q}'\text{: maximize } \sum_{i<l} [a_{il}(1-x_ix_l)+b_{il}(1+x_ix_l)-c_{il}] \\ &\text{subject to} \quad x_i \in \{-1,+1\} \text{ for } i \in V \bigcup \{0\} \end{aligned}$$

where the coefficients a_{il}, b_{il} and c_{il} are *non-negative*.

The following hyperplane partitioning algorithm was proposed in [6]:

1. (Relaxation) Solve (P) optimally and obtain an optimal set of vectors X^*.
2. (Randomized Rounding) Let r be a unit-vector chosen uniformly at random. Construct an assignment x for (Q') as follows. For each $i = 0,\ldots,n$, if $r \cdot X_i^* \geq 0$, then set $x_i = +1$ else set $x_i = -1$.
3. (Normalizing) Construct an assignment for the given W-CSP instance as follows. If $x_0 = +1$ then return x as the assignment, else ($x_0 = -1$) return x with all values flipped as the assignment.

Basically, the Rounding step chooses a random hyperplane through the origin of the unit sphere (with r as its normal) and partitions the variables into those vectors that lie on the same side of the hyperplane. The Normalizing step is needed to undo the effect of the additional variable x_0 in case it is set to -1. Using this algorithm, W-CSP(2) can be approximated within a worst case bound of 0.634, which can be improved to 0.878 for satisfiable instances [9].

Unfortunately, this method cannot be easily derandomized to date. A derandomization method for the above algorithm was proposed in [6]. Unfortunately, the method was discovered to contain a fatal flaw by Mahajan and Ramesh [10]. The authors then presented a different derandomization approach in [10]. However, from the practical point of view, their method is inefficient. This rounding strategy will not be reported in our experiments.

5 Computational Experience

In this section, we report our computational experience. Our experiments are conducted on the SUN Sparc UNIX workstation. Random numbers are generated using the standard UNIX `long random()` function, initialized with a random seed which depends on the time of the day.

We naturally wanted to test our algorithms on hard W-CSP instances. However, we learnt that for PCSP, there is no localized region of hard problems – hard problems are located throughout the instance space, with difficulty increasing with increasing edge density of the constraint graph, increasing tightness of constraints, and naturally, increasing domain size [15]. Our main concern is the performance of our algorithm against other incomplete algorithms which run in polynomial time. The measure of performance is the approximation ratio. Since it is time-consuming to compute optimal solutions for reasonably large instances, we experiment on *satisfiable* instances whose optimal value is always the sum of all edge weights. This allows us to compute the approximation ratio without obtaining optimal solutions.

Generation of Random Instances With the above considerations, we generate W-CSP instances of n variables from a distribution parameterized by the *edge probability* $0 \leq q \leq 1$ and the *consistency probability* $0 \leq \lambda \leq 1$ according to the following rules:

1. Generate a random graph G of n nodes such that an edge (i.e. constraint) exists between any two variables with probability q.
2. To generate a satisfiable instance, we first generate a random assignment $\hat{\sigma}$. For each edge (i_1, i_2) in G, construct the constraint relation as follows. Insert the value pair $(\hat{\sigma}_{i_1}, \hat{\sigma}_{i_2})$ with probability 1 and all other $k^2 - 1$ pairs with probability λ. To generate a non-satisfiable instance, we simply insert the value pair $(\hat{\sigma}_{i_1}, \hat{\sigma}_{i_2})$ with probability λ.
3. Edge weights are randomly generated in the range [0..999].

This generation method has been used by others and an online (unweighted) implementation is in [13].

Algorithms Four algorithms are compared. **Greedy LS** refers to hill-climbing local search with an initial assignment generated greedily, i.e. arrange the variables in a linear order and assign them in sequence the value that maximizes the weighted sum of satisfied constraints. **Random LS** refers to hill-climbing local search with a

random initial assignment. Rand Round refers to our simple rounding algorithm, and RR LS refers to hill-climbing local search with an initial assignment generated by Rand Round. To solve a semidefinite program, we use the solver written by Fujisawa and Kojima [5].

Experiment 1 In the first set of experiments, we fix $n = 64$ and $k = 2$ and generate random satisfiable instances by the abovementioned method. We consider three edge densities (sparse, medium and dense; $m = \binom{n}{2}$ refers to the total number of edges) and for each density, we vary the consistency probability from 0.1 to 0.9. For each case, 10 random satisfiable instances are generated and solved respectively by the four algorithms The respective mean approximation ratios are obtained. Table 2 gives the outcome of the experiment. Figures are rounded to three decimal places and 1.000(-) is used to denote a value which is rounded to 1.000.

Experiment 2 In the second set of experiments, we fix $n = 20$ and $k = 5$. The same scenerio as Experiment 1 is repeated. Table 3 gives the outcome of the experiment.

Some Observations (Satisfiable instances)

1. Greedy LS performs well on dense instances, but not so well on sparse ones.
2. Random LS performs reasonably well on sparse instances but not so well on dense instances.
3. Rand Round performs consistently well on all instances, achieving at least 97% optimality for $k = 2$ and 86% optimality for $k = 5$. More importantly, Rand Round outperforms Greedy LS and Random LS for $k = 5$. It is worth noting that the average CPU time required by our implementation of Rand Round is 22.2 seconds for each instance in Experiment 1 and 10.5 seconds for each instance in Experiment 2.
4. RR LS outperforms all other approaches in all cases, achieving 99% optimality for $k = 2$ and 96% optimality for $k = 5$. It is worth noting that, in all cases, the standard deviation corresponding to each mean approximation ratio is smaller than those of Greedy LS and Random LS. This means that our algorithm gives consistently good approximation solutions for the instances tested.

Experiments 3 and 4 In the third and fourth sets of experiments, we generate non-satisfiable instances and measure the *absolute* ratios. We fix $n = 64$, $k = 2$ and $n = 20$, $k = 5$ respectively. The same scenerio as Experiment 1 is repeated. Table 4 and 5 give the outcome of the experiments.

Edge Prob (q)	Consistency Prob (λ)	Mean approximation ratios			
		Greedy LS	Random LS	Rand Round	RR LS
Sparse ($2n/m$)	0.10	0.935	0.998	0.991	1.000
	0.30	0.916	0.991	0.979	0.995
	0.50	0.930	0.984	0.970	0.994
	0.70	0.974	0.989	0.987	0.992
	0.90	0.996	0.993	0.999	0.999
Medium ($n \log n/m$)	0.10	1.000	1.000	1.000	1.000
	0.30	1.000	0.946	0.999	1.000
	0.50	0.999	1.000	0.991	1.000
	0.70	0.988	0.973	0.988	1.000(-)
	0.90	0.991	0.996	0.996	0.999
Dense ($n^2/3m$)	0.10	1.000	0.725	1.000	1.000
	0.30	1.000	0.789	1.000	1.000
	0.50	1.000	0.861	1.000	1.000
	0.70	1.000	0.886	0.999	1.000
	0.90	0.997	0.970	0.995	1.000(-)

Table 2. Experiment 1.

Edge Prob (q)	Consistency Prob (λ)	Mean approximation ratios			
		Greedy LS	Random LS	Rand Round	RR LS
Sparse ($2n/m$)	0.10	0.752	0.793	0.931	1.000
	0.30	0.753	0.892	0.915	0.960
	0.50	0.911	0.945	0.941	0.975
	0.70	0.981	0.905	0.992	0.999
	0.90	1.000	0.883	1.000	1.000
Medium ($n \log n/m$)	0.10	0.823	0.381	0.998	1.000
	0.30	0.751	0.790	0.867	1.000
	0.50	0.889	0.848	0.928	0.968
	0.70	0.961	0.972	0.962	0.982
	0.90	0.999	0.912	0.999	1.000
Dense ($n^2/3m$)	0.10	0.893	0.387	1.000	1.000
	0.30	0.919	0.537	0.997	1.000
	0.50	0.949	0.700	0.922	0.984
	0.70	0.927	0.953	0.938	0.966
	0.90	0.998	0.965	0.999	1.000(-)

Table 3. Experiment 2

Edge Density (q)	Consistency (p)	Mean absolute ratios			
		Greedy LS	Random LS	Rand. Round	RR + LS
Sparse ($2n/m$)	0.10	0.244	0.243	0.263	0.264
	0.30	0.555	0.567	0.564	0.579
	0.50	0.769	0.770	0.771	0.792
	0.70	0.904	0.926	0.920	0.933
	0.90	0.984	0.991	0.995	0.997
Medium ($n \log n/m$)	0.10	0.214	0.212	0.211	0.218
	0.30	0.472	0.479	0.468	0.479
	0.50	0.662	0.659	0.650	0.661
	0.70	0.846	0.852	0.841	0.856
	0.90	0.977	0.980	0.977	0.981
Dense ($n^2/3m$)	0.10	0.162	0.161	0.159	0.163
	0.30	0.387	0.389	0.382	0.390
	0.50	0.592	0.588	0.584	0.594
	0.70	0.784	0.784	0.777	0.786
	0.90	0.950	0.950	0.946	0.952

Table 4. Experiment 3

Edge Density (q)	Consistency Prob (λ)	Mean absolute ratios			
		Greedy LS	Random LS	Rand Round	RR LS
Sparse ($2n/m$)	0.10	0.455	0.489	0.472	0.503
	0.30	0.752	0.779	0.773	0.825
	0.50	0.909	0.915	0.946	0.963
	0.70	0.977	0.928	0.989	0.996
	0.90	0.984	0.871	1.000(-)	1.000
Medium ($n \log n/m$)	0.10	0.367	0.390	0.367	0.395
	0.30	0.659	0.643	0.623	0.662
	0.50	0.822	0.848	0.847	0.870
	0.70	0.956	0.964	0.956	0.977
	0.90	0.997	0.994	0.999	1.000
Dense ($n^2/3m$)	0.10	0.298	0.311	0.283	0.313
	0.30	0.597	0.605	0.583	0.612
	0.50	0.787	0.794	0.778	0.809
	0.70	0.933	0.941	0.925	0.943
	0.90	0.996	0.945	0.998	1.000

Table 5. Experiment 4

6 Conclusion

In this paper, we have proposed a new approach for finding good approximate solutions for the Weighted CSP (W-CSP). This method is based on a recent breakthrough in randomized algorithms among the theoretical Computer Science community, and a much-researched area of semidefinite programming among the Operations Research community. Our algorithm runs in polynomial time in the worst-case, and it is dependent heavily on the speed of solving a semidefinite program. The good news is that semidefinite programs are solvable quickly in practice, much like linear programs, and much research is going on in the Operations Research community to develop even faster algorithms based on interior point methods. Another advantage of our algorithm is that it has a provable worst-case bound to ensure that our algorithm will never perform embarrassingly poorly.

Experimentally, our algorithm works well for satisfiable W-CSP instances drawn random from a distribution parameterized by the edge probability and consistency. For non-satisfiable instances, the improvement is less dramatic. It remains to apply our algorithm to solve real-world instances.

We have proposed two rounding strategies to round fractional solutions to valid assignments. This opens up a new research avenue for considering other rounding strategies which exhibit both good worst-case bound as well as empirical performance.

Acknowledgements

I would like to thank Osamu Watanabe for discussions on theoretical results, Nobuhiro Yugami for discussions on computational results, and Richard Wallace for telling me about hard PCSPs and existing implementations.

References

1. F. Alizadeh. Interior point methods in semidefinite programming with applications to combinatorial optimization. *SIAM J. Optimiz.*, 5(1):13–51, 1995.
2. Noga Alon and Joe Spencer. *The Probabilistic Method.* Wiley Interscience Ser. Disc. Math. and Optimiz., 1992.
3. Eugene C Freuder. Partial constraint satisfaction. In *Proc. Int'l Joint Conf. Artif. Intell. (IJCAI-89)*, pages 278–283, Detriot, MI, 1989.
4. Eugene C. Freuder and Richard J. Wallace. Partial Constraint Satisfaction. *Artif. Intell.*, 58(1-3):21–70, 1992.
5. Katsuki Fujisawa and Masakazu Kojima. Sdpa (semidefinite programming algorithm) user's manual. Technical Report B-308, Dept. Information Science, Tokyo Inst. of Technology, 1995. Online implementation available at `ftp.is.titech.ac.jp` under directory `/pub/OpsRes/software`.
6. Michel X. Goemans and David P. Williamson. Approximation algorithms for MAX CUT and MAX 2SAT. In *Proc. 26th ACM Symp. on Theory of Computing*, pages 422–431, 1994. Full version to appear in *J. ACM.*
7. P. Lancaster and M Tismenetsky. *The Theory of Matrices.* Academic Press, Orlando, FL, 1985.
8. H. C. Lau. Approximation of constraint satisfaction via local search. In *Proc. 4th Wrksp. on Algorithms and Data Structures (WADS)*, pages 461–472. Springer Verlag Lect. Notes Comp. Sci. (955), 1995.
9. H. C. Lau and O. Watanabe. Randomized approximation of the constraint satisfaction problem. In *Proc. Fifth Scandinavian Wrksp. on Algorithm Theory (SWAT).* Springer Verlag Lect. Notes Comp. Sci., 1996. To appear.
10. S. Mahajan and H. Ramesh. Derandomizing semidefinite programming based approximation algorithms. In *Proc. 36th IEEE Symp. on Found. of Comp. Sci.*, pages 162–168, 1995.
11. P. Raghavan and C. D. Thompson. Randomized rounding: A technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.
12. Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.
13. Peter van Beek. On-line C-programs available at `ftp.cs.ualberta.ca` under directory `/pub/ai/csp`.
14. Chris Voudouris and Edward Tsang. Parital constraint satisfaction problems and guided local search. Technical Report No. CSM-250, Dept. Computer Science, University of Essex, 1995.
15. Richard J. Wallace, 1995. Personal communication.
16. Richard J. Wallace. Directed arc consistency preprocessing as a strategy for maximal constraint satisfaction. In M. Meyer, editor, *Constraint Processing*, pages 121–138. Springer Verlag Lect. Notes Comp. Sci. (923), 1995.
17. Richard J. Wallace and Eugene C. Freuder. Conjuctive width heuristics for maximal constraint satisfaction. In *Proc. Nat'l Conf. on Artif. Intell. (AAAI-93)*, pages 762–768, 1993.
18. Richard J. Wallace and Eugene C. Freuder. Heuristic methods for over-constrained constraint satisfaction problems. In *CP95 Wrksp. on Over-constrained Systems*, 1995.

Towards a More Efficient Stochastic Constraint Solver

Jimmy H.M. Lee, Ho-fung Leung and Hon-wing Won

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
{jlee, lhf, hwwon}@cs.cuhk.edu.hk

Abstract. E-GENET shows certain success on extending GENET for non-binary CSP's. However, the generic constraint representation scheme of E-GENET induces the problem of storing too many penalty values in constraint nodes and the min-conflicts heuristic is not efficient enough on some problems. To overcome these two weaknesses and further improve the performance, we propose several modifications. All of them together can boost the efficiency of E-GENET without resorting to modifying the underlying network model or the convergence procedure in an *ad hoc* manner. The performance of modified E-GENET also compares well against that of CHIP.

1 Introduction

Many problems in artificial intelligence and computer science in general can be formulated as *constraint satisfaction problems* (CSP's). Efficient algorithms for solving CSP's are thus very useful. In 1992, Minton *et al.* published a paper on a new approach for solving CSP's. The approach is known as heuristic repair method or iterative repair method [8]. Some standard problems such as N-queens and graph-coloring can be solved in orders of magnitude better than traditional backtracking techniques. The average solution time for the million-queens problem is reduced to less than one minute and a half on a SPARCstation1 [8].

A problem of this approach is that execution can easily be trapped in local minima (or maxima), a state in which no repair can be made but the current assignment is still inconsistent. When trapping occurs, execution has to be aborted. This situation is most likely to occur in highly constrained problems [2], especially non-binary CSP's, since reassigning one variable at each step usually cannot reduce number of constraint violations.

In a previous paper [7], we present the E-GENET, which bases on iterative repair approach, features a generic representation scheme for general constraints and adopts the heuristic learning rule from GENET [2]. Constraints ranging from disjunctive constraints to non-linear constraints to symbolic constraints can now be handled. However, being a first step for solving non-binary CSP's, E-GENET has two insufficiencies. For a complicated constraint, there may be a large number of penalty values stored in the corresponding constraint node. Besides, the underlying principle of E-GENET, the min-conflicts heuristic in

the iterative repair approach, cannot provide enough information to guide the search efficiently in some non-binary CSP's and hence the performance is not satisfactory.

In this paper, we describe several modifications to E-GENET such as a new type of nodes to deal with the problem of large constraint nodes and a novel assignment scheme of initial penalty values to improve the effectiveness of the min-conflicts heuristic on non-binary CSP's. We have also implemented an prototype to show the feasibility and efficiency of our proposal. The modified E-GENET compares well against CHIP [5] in most of the CSP's tested.

The rest of this paper is organized as follows. Section 2 briefly reviews E-GENET. In section 3, we explain the inadequacies of E-GENET. Section 4 describes the proposed modifications. Benchmarking results and related work are presented in section 5 and 6 respectively. Section 7 summarizes our contributions and sheds light on future work.

2 Brief Review of E-GENET

E-GENET extends GENET for general, binary and non-binary, constraint handling by a generic constraint representation scheme. It consists of a network model and a convergence procedure based on min-conflicts heuristic in iterative repair approach and the learning heuristic of GENET.

2.1 Network Architecture

E-GENET has two types of nodes: *variable nodes* and *constraint nodes*. Each variable in a CSP is represented by a variable node, which contains the domain associated with the variable. The *state* S_x of a variable node x is defined to be the current variable assignment. A *constraint node* is created for each constraint in the CSP. A variable node x is connected to a constraint node c if x occurs in c[1]. Consider the CSP "$x + y + z = 9 \wedge 3y - z = 4$," where the domains of x, y, and z are $\{1, \ldots, 10\}$. Figure 1 shows the CSP's network in E-GENET. The constraint node for $3y - z = 4$ is connected to relating variable nodes y and z; and the constraint node for $x + y + z = 9$ is connected to all of x, y, and z. The current state of the network represents the following variable assignment: $x = 3$, $y = 2$, and $z = 4$.

For any constraint $c(x_{i_1}, \ldots, x_{i_n})$, each combination (or tuple) $(v_1, \ldots, v_n)$ of possible values from domains of $x_{i_1}, \ldots, x_{i_n}$ is given a *penalty value* $\delta_{c(v_1,\ldots,v_n)}$. These penalty values, (conceptually) stored in the corresponding constraint node, may be modified as a result of heuristic learning. Initially, penalty values of prohibited tuples are set to -1 and others to 0.

Each value v in the domain of variable x has an input $I_{x=v}$ defined as:

$$I_{x=v} = \sum_{\forall c(x_{i_1},\ldots,x,\ldots,x_{i_k})} \delta_{c(S_{x_{i_1}},\ldots,v,\ldots,S_{x_{i_k}})}$$

[1] We relax terminology by naming a variable (a constraint) also by its variable (constraint) node name.

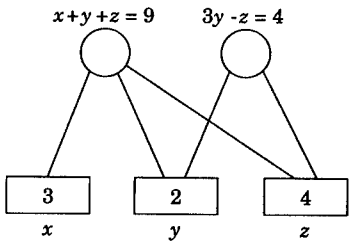


Fig. 1. An example network generated by E-GENET

A negative input indicates that the value is involved in the violation of some constraints in the network. When the sum of input of values in the current variable assignment is zero, no constraints are violated and the network is in a *solution state*. The variable assignment associated with a solution state is a *solution* to the corresponding CSP.

2.2 Convergence Procedure

Dynamics of E-GENET concerns how the network changes states of variable nodes and penalty values in constraint nodes before settling in solution state(s). Initially, a complete but possibly inconsistent variable assignment is generated. The value of each variable node x is then updated to reduce constraint violation by choosing the value with maximum input:

$$x := v' \text{ if } I_{x=v'} = \max\{I_{x=v} | v \in \text{domain of } x\}$$

If there are several values with the maximum input, a value with maximum input that is currently assigned to x stays assigned. Otherwise, pick a value with maximum input randomly. Following [2], we define a *repair* to be a variable update. Variables in the network are updated asynchronously until variable assignment remains unchanged.

E-GENET terminates execution if a solution state is reached. Otherwise, the network is trapped in a local minima and *heuristic learning* is activated to help escape from the local minima. Heuristic learning in E-GENET amounts to decreasing the penalty values of tuples violating some constraints. We leave the heuristic learning rule unspecified intentionally since application domain knowledge can usually assist us in designing good learning rule for specific problems. The *convergence procedure* is summarized in figure 2.

3 Inadequacies of E-GENET

3.1 Cumbrous Constraint Node

Consider the constraint $13x_1+4x_2+5x_3+9x_4 = 7y_1+8y_2$ where all variables have the domain $\{1,\ldots,100\}$. Since there are 100^6 possible tuples, we have to store 10^{12} penalty values in the corresponding constraint node and this is impractical.

```
repeat
    update all variable nodes asynchronously until no repair
    if (sum of inputs of values in current assignment is zero)
        terminate with success
    else
        activate heuristic learning
until (time limit or maximum number of cycles is exceeded)
```

Fig. 2. The convergence procedure of E-GENET

If we break down the constraint into $13x_1 + 4x_2 + 5x_3 = T$ and $T + 9x_4 = 7y_1 + 8y_2$ by using a new variable T, the domain of T after pruning would be $\{22, \ldots, 1491\}$. The number of possible tuples for these two constraints is then $2 \times 1470 \times 100^3$, *i.e.* 2.94×10^9. Although we can successfully lower the storage requirement, the performance would be affected by the addition of a new variable of relatively large domain. Further decomposition of the two constraints can cut the space requirement to a greater extent but induces severe drawback in performance. So this is not a good method to solve the problem. What we need is something that has properties similar to T but does not increase the number of variables.

3.2 Inefficiency of the min-conflicts heuristic

In this section, we extend the model in [8] to investigate the efficiency of E-GENET. Consider a CSP with variables $x_1, \ldots, x_n$, where each x_i has k possible values and involves exactly in c m-ary constraints. Assume that there is only one solution $(v_1, \ldots, v_n)$ and the randomly generated initial assignment has d variables assigned different from the solution. Denote the set of these d variables as Var. For any constraint $c'(x_{i_1}, \ldots, x_{i_m})$ in the CSP, if $\exists j$ such that $S_{x_{i_j}} \neq v_{i_j}$, let the probability of the constraint being violated be p.

Randomly choose a variable $x_k \in Var$ for repairing (the case for $x_k \notin Var$ is similar). For any one of the $k-1$ incorrect values ($\neq v_k$) of x_k, since all c related constraints have a probability p of being violated, the expected number of constraint violations is pc and hence the expected input to these values is $-pc$.

Consider the correct value of x_k. For an arbitrary constraint depending on the variable, the probability that all other $m-1$ variables are not elements of Var is $\binom{n-d}{m-1}/\binom{n-1}{m-1}$. The probability of the constraint being violated is thus $\left(1 - \binom{n-d}{m-1}/\binom{n-1}{m-1}\right) p$ and the expected input to the correct value of x_k is given by $-\left(1 - \binom{n-d}{m-1}/\binom{n-1}{m-1}\right) pc$.

From the expected inputs, we can see that the probability of making an incorrect repair would be decreased if c increases or d shrinks. In other words, if number of constraints is small or d is very near to n ($d > n - m$) at the beginning, the efficiency of E-GENET may be very low.

This result can also be observed from the benchmarking results of E-GENET in our previous paper [7]. We find that in some non-binary CSP's like systems of

linear equations and cryptarithmetic problems, the performance of E-GENET is worse than that of CHIP. Use the problem in figure 3 as an example. Here, $m = n = c = 4$ and p is equal to 1 approximately. If we try to repair a variable that is assigned incorrectly at start, in almost all cases, the input to any one of incorrect values of the variable is -4 and that to the correct value is -4 as well $\left(\binom{n-d}{m-1}/\binom{n-1}{m-1} = 0\right)$. Hence, variables are updated randomly and the network would wander over all possible states aimlessly until there are sufficient information provided by learning.

$$\begin{cases} 691x_1 + 81x_2 + 22x_3 + 629x_4 = 7007 \\ 519x_1 + 147x_2 - 971x_3 - 710x_4 = -8726 \\ 841x_1 - 527x_2 + 948x_3 - 589x_4 = -4357 \\ 899x_1 + 343x_2 - 877x_3 + 531x_4 = 4571 \\ 0 \leq x_1, x_2, x_3, x_4 \leq 10 \end{cases}$$

Fig. 3. A system of linear equations

4 Modifications

To overcome the two weaknesses of E-GENET, we propose four modifications. All of them together can boost the performance of E-GENET without resorting to modifying the underlying network model or the convergence procedure in an *ad hoc* manner.

4.1 Intermediate Node

The first modification is the introduction of a new type of nodes called *intermediate nodes* to address the problem of cumbrous constraint nodes. Figure 4 shows a general representation for a constraint with this modification.

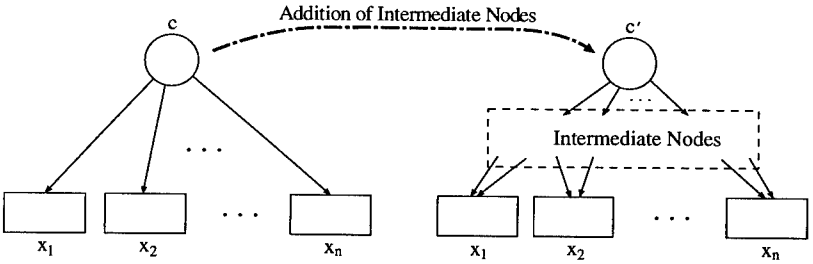


Fig. 4. General representation for a constraint after the addition of intermediate nodes

Formally, a representation for a constraint on n variables $\{x_1, \ldots, x_n\}$ is a directed acyclic graph (*dag*) with at least $n+1$ nodes. One of the nodes is called constraint node, denoted by c'. We have $indegree(c') = 0$ and $outdegree(c') \geq 1$. [2] n

[2] *indegree(Y)* is the number of edges entering node Y and *outdegree(Y)* is that leaving.

of the other nodes are called variable nodes. There exists a one-one mapping between variables and variable nodes. Thus we relax terminology and give a variable and its corresponding variable node the same name. For all x_i, $indegree(x_i) \geq 1$ and $outdegree(x_i) = 0$.

All other nodes, not in $\{c', x_1, \ldots, x_n\}$, are intermediate nodes. Let the set of intermediate nodes be $\{f_1, \ldots, f_m\}$. Each intermediate node f_i has the properties that $indegree(f_i) \geq 1$ and $outdegree(f_i) \geq 1$. Define $outbundle(f_i)$ as the set $\{a_1, \ldots, a_p\}$ such that for each a_j in the set, there is an edge from f_i to a_j. Intermediate node f_i is associated with a function defined on its outbundle, $F_i(a_1, \ldots, a_p)$. The state S_{f_i} of f_i is the value $F_i(S_{a_1}, \ldots, S_{a_p})$ and f_i's domain, $dom(f_i)$, is the range of the function F_i. In constraint node c', we store penalty values for combinations of values from domains of nodes in $outbundle(c')$ instead.

Consider the constraint $x^2 + x^2 y = 4$. Figure 5 shows a possible representation for this constraint. The intermediate node a is associated with the function x^2 and currently has the value (state) 4. If $x \in \{1, 2\}$ and $y \in \{3, 4\}$, after the addition of nodes a and b, the content of the constraint node is changed as shown in the figure.

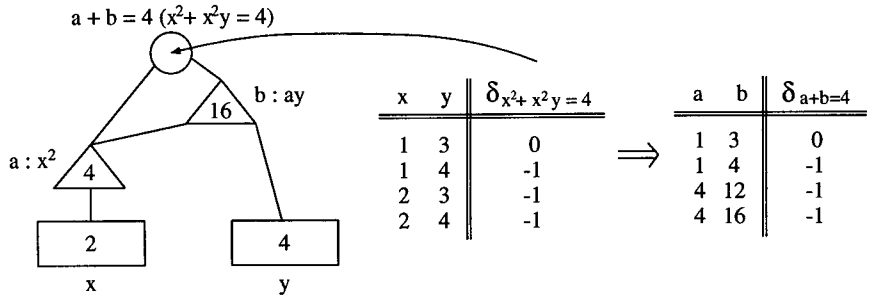


Fig. 5. An example representation with intermediate nodes

Let the outbundle for constraint node c' of the constraint $c(x_1, \ldots, x_n)$ be $\{f_1, \ldots, f_p\}$. Since for each combination $(v_1, \ldots, v_n)$ of possible values from domains of $x_1, \ldots, x_n$, there is a corresponding tuple $(S_{f_1}, \ldots, S_{f_p})$, a mapping $H : x_1 \times \ldots \times x_n \rightarrow B$ where $B \subseteq dom(f_1) \times \ldots \times dom(f_p)$ exists and range of $H = B$ (surjective). Each tuple in B is given a penalty value and all these penalty values form the content of c'.

There is one restriction on usage of intermediate nodes. For each $(u_1, \ldots, u_p)$ in B, all tuples in $H^{-1}(u_1, \ldots, u_p)$ must be either satisfying or violating the constraint unanimously. If all these tuples satisfy the constraint, the initial penalty value for $(u_1, \ldots, u_p)$ is 0. Otherwise, it is set to -1. The heuristic learning is similar to that in the original E-GENET. For example, a plausible learning rule is $\delta_{c'(S_{f_1}, \ldots, S_{f_p})} := \delta_{c'(S_{f_1}, \ldots, S_{f_p})} - (\delta_{c'(S_{f_1}, \ldots, S_{f_p})} < 0)$. [3]

The most essential usage of intermediate nodes is to reduce the number of

[3] $<$ is a boolean function returning 1 if the comparison is true and 0 otherwise.

nodes in outbundle of the constraint node and the size of domains of these nodes, i.e. the size of B. Although we allow any levels of intermediate nodes, one is sufficient to meet this objective.

Theorem Given a representation M for a constraint $c(x_1, \ldots, x_n)$, there exists another representation M' such that all paths in M' are of length ≤ 2 and the content of the constraint node in M' is the same as that in M.

The philosophy behind intermediate nodes is to divide tuples of a constraint into several groups (using the function H). Each group (a tuple in B) is given only one penalty value and during learning, we would treat a group as a unit and penalize a whole group.

Consider again the constraint $13x_1 + 4x_2 + 5x_3 + 9x_4 = 7y_1 + 8y_2$. It can be represented as shown in figure 6. Intermediate nodes a and b represent expressions at both sides of the constraint respectively. Each distinct combination (v_a, v_b) calculated with possible values from domains of $x_1, x_2, x_3, x_4, y_1, y_2$ is given one penalty value. When heuristic learning is activated, if $S_a \neq S_b$, we would decrease the penalty value $\delta_{(a=b)(S_a,S_b)}$ by 1. The pair $(60, 50)$ in the new constraint node, for example, represents the set of tuples $\{(1,1,5,2,6,1)$, $(1,2,6,1,6,1)$, $(1,6,1,2,6,1)$,$(1,7,2,1,6,1)$,$(2,5,1,1,6,1)\}$ in the original node. Updating the penalty value of the pair is equivalent to performing the same operation on penalty values of all tuples in the set.

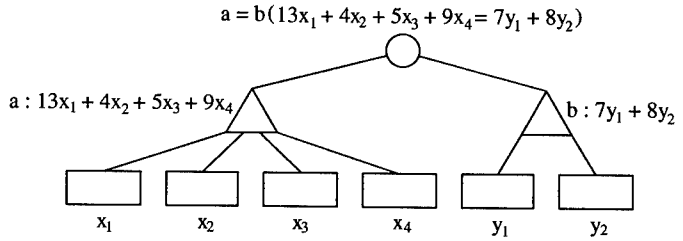


Fig. 6. A possible representation for the constraint $13x_1 + 4x_2 + 5x_3 + 9x_4 = 7y_1 + 8y_2$

Under this representation, the storage requirement for the constraint becomes 4×10^6 (compared to 10^{12} in the original one). If we divide all possible tuples into fewer groups, we can save more space. However, performance would be affected. Decreasing the penalty value of a tuple during learning is to reduce the probability that this tuple is assigned to corresponding variables again. With grouping, changing one penalty value would influence probabilities of all tuples in the group and this effect may not be desirable. To avoid any decrease in performance, we carefully design a suitable grouping for each type of constraints. Moreover, in practice, only penalty values of tuples that have been penalized during learning would be stored. The others would be computed on demand.

Besides reducing size of constraint node, intermediate node has two more advantages:

Eliminating Common Subexpression: In each constraint node, if we do not store all penalty values directly and, instead, derive some storage scheme to reduce storage requirement, we usually need to do constraint checks. Intermediate nodes can be used to store the results of common subexpressions and eliminate the need of re-calculation. The node a in figure 7 removes redundant work on both inter- and intra-constraint common subexpression.

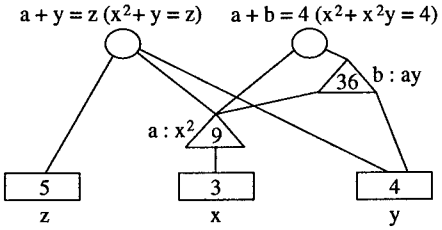


Fig. 7. An example network with the common subexpression x^2

Unifying Heuristic Learning: The heuristic learning rule is not fixed in original E-GENET. During learning, for any constraint $c(x_1, \ldots, x_n)$, if $\delta_{c(S_{x_1},\ldots,S_{x_n})} < 0$, users can choose a P where $\{(S_{x_1}, \ldots, S_{x_n})\} \subseteq P \subseteq \{(v_1, \ldots, v_k) | \delta_{c(v_1,\ldots,v_k)} < 0\}$ and decrease penalty values of all tuples in P by 1. As most advantages of this practice can be obtained by using intermediate nodes, heuristic learning rules can be unified with $P = \{(S_{x_1}, \ldots, S_{x_n})\}$.

Use the constraint $\texttt{atmost}(1, \{x_1, x_2, x_3, x_4\}, \{3\})$ as an example. This constraint states that at most one out of the four variables can take the value 3. We usually choose $P = \{(v_1, \ldots, v_k) | \delta_{c(v_1,\ldots,v_k)} < 0\}$. Constructing the representation as in figure 8, we can use the learning rule $\delta_{(a=0)(S_a)} := \delta_{(a=0)(S_a)} - (\delta_{(a=0)(S_a)} < 0)$ instead to get the same effect.

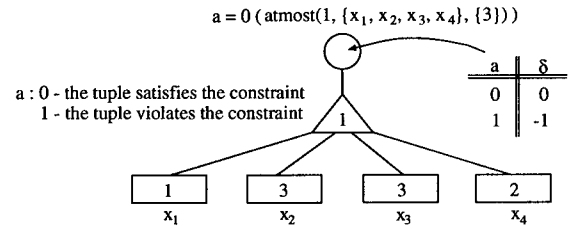


Fig. 8. The representation for the `atmost` constraint

4.2 New Assignment Scheme of Initial Penalty Values

E-GENET is based on the min-conflicts heuristic, which favors the value violating the minimum number of constraints. Before learning procedure is activated, the input to a value is $-n$ where n is number of constraints violated by assigning

this value to the corresponding variable. E-GENET would then change the value of the variable to the one that has the maximum input.

If the network is trapped in a local minima, heuristic learning is invoked. This amounts to decreasing the penalty values of tuples violating some constraints. Assume that in a particular learning process, the tuple $(v_1, \ldots, v_n)$ of a constraint $c(x_1, \ldots, x_n)$ is penalized. This can be considered as adding a redundant constraint $\texttt{illegal}((x_1, \ldots, x_n), (v_1, \ldots, v_n))$, which specifies that $(x_1, \ldots, x_n) \neq (v_1, \ldots, v_n)$: decreasing the penalty value $\delta_{c(v_1,\ldots,v_n)}$ is effectively the same as combining the content of the constraint node with that of the `illegal` constraint.

$x_1 \ldots x_n$	new $\delta_{c(x_1,\ldots,x_n)}$ = old $\delta_{c(x_1,\ldots,x_n)}$ + $\delta_{\texttt{illegal}((x_1,\ldots,x_n),(v_1,\ldots,v_n))}$		
$\vdots$	$\vdots$	$\vdots$	**0**
$v_1 \ldots v_n$	-2	-1	-1
$\vdots$	$\vdots$	$\vdots$	**0**

Hence decreasing penalty values can be interpreted as the accumulation of knowledge by introducing redundant constraints. It is well known that domain knowledge or some other information can be utilized to guide the search by adding appropriate redundant constraints. According to the the above discussion, we need not create new constraint nodes for these redundant constraints in the E-GENET model. What we need is simply a new assignment scheme of initial penalty values: for any constraint, initial penalty values of tuples satisfying the constraint are 0's, but those for prohibited tuples may be any negative integers determined by the user using his domain knowledge. [4] This resembles the fitness in "Evolutionary Model" of GENET [11].

4.3 Concept of Contribution

Consider a constraint $c(x_1, \ldots, x_n)$. If the current assignments of $x_1, \ldots, x_n$ are $v_1, \ldots, v_n$ respectively, $\delta_{c(v_1,\ldots,v_n)}$ is always used in computing any input $I_{x_i=v_i}$, where $1 \leq i \leq n$. However, this method is inadequate in some cases. Take the constraint $\texttt{atmost}(1, \{x_1, x_2, x_3, x_4\}, \{3\})$ and the tuple $(3, 4, 3, 3)$ as an example. That x_2 being assigned the value 4 does not contribute to the violation of the constraint. The penalty value of the tuple should not be added to the input $I_{x_2=4}$ to decrease the probability that x_2 takes the value 4. Thus, the definition of an input is given as:

$$I_{x=v} = \sum_{\forall c(x_{i_1},\ldots,x,\ldots,x_{i_k})} \mathit{contribute}_c(j, (S_{x_{i_1}}, \ldots, v, \ldots, S_{x_{i_k}})) \times \delta_{c(S_{x_{i_1}},\ldots,v,\ldots,S_{x_{i_k}})}$$

where the j^{th} argument of $(S_{x_{i_1}}, \ldots, v, \ldots, S_{x_{i_k}})$ is v; $\mathit{contribute}_c(i, (v_1, \ldots, v_n))$ is a function returning a value between 0 and 1. The magnitude of the value shows the contribution of v_i to the tuple $(v_1, \ldots, v_n)$ with respect to the constraint c.

[4] It should be noted that any finite domain constraints can be expressed as a conjunction of `illegal` constraints.

4.4 Learning Heuristic

As now the initial penalty value of a tuple can be much smaller than -1, the landscape of the search space becomes comparably rough. A penalty amount of -1 may not be sufficient to unstable the network. So if the same tuple of a constraint is penalized consecutively, penalty amount is increased exponentially at each time of learning $(-1, -1, -2, -4, -8, \ldots)$.

5 Benchmarking Results

To illustrate the feasibility and effectiveness of the modifications, we have implemented several types of constraints and test each of them on different problems. We compare our result with that of a constraint logic programming language Cosytec CHIP version 4.1.0 [9], which uses traditional constraint propagation and backtracking tree search for constraint solving.

All benchmarking is performed on a SUN SPARCstation 10 model 30. Timing (including network construction) and number of repairs results for both versions of E-GENET are median of 10 runs. For each problem, median number of penalty values stored in the modified E-GENET is also shown. A "–" symbol means that the execution fails due to either execution time exceeded or memory exhaustion.

5.1 Linear Arithmetic Constraint

A linear arithmetic constraint is a constraint of the form $U \Delta V$, where U and V are linear arithmetic expressions and $\Delta \in \{=, \neq, <, \leq, >, \geq\}$. The representation of the constraint is shown in figure 9. Intermediate nodes a and b are used to hold the current values of U and V. The initial penalty values for violating tuples are mainly based on the difference between values of a and b.

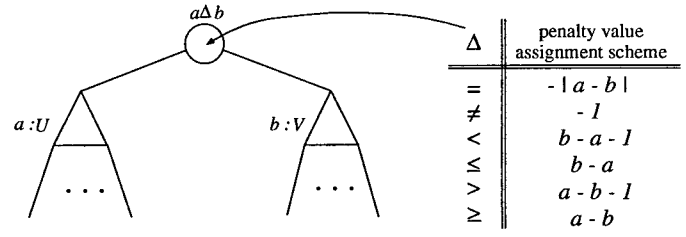


Fig. 9. The representation for a linear arithmetic constraint

Five traditional benchmark programs [3] have been used to show the ability of the modified E-GENET on solving linear equation problems. `send`, `donald` and `crypta` are cryptarithmetic problems of different size. `eq10` and `eq20` are systems of 10 and 20 linear equations respectively.

Since constraint propagation alone can solve the cryptarithmetic problems with little backtracking, CHIP outperforms both versions of E-GENET significantly in most cases. However, we can observe that the proposed modifications

Problem	CHIP CPU Time (sec)	Original E-GENET CPU Time (sec)	Modified E-GENET CPU Time (sec)	penalty values stored
send	0.010	0.300	0.085	53
donald	0.050	2.895	0.380	166
crypta	0.050	3.400	1.040	629
eq10	0.140	–	0.045	0
eq20	0.210	–	0.040	0

Table 1. Results on cryptarithmetic problems and systems of linear equations

can improve the performance of E-GENET. The modified E-GENET is more efficient than CHIP in the problems eq10 and eq20, which cannot be solved within 10 minutes by the original version.

5.2 Atmost Constraint

The $\texttt{atmost}(N, Var, Val)$ constraint specifies that no more than N variables taken from the variable set Var are assigned values in value set Val. Assume n is the number of variables currently having values in Var. If $n > N$, we would prefer a smaller n. Take the constraint $\texttt{atmost}(1, \{x_1, x_2, x_3, x_4\}, \{3\})$ as an example. The assignment $(1, 3, 3, 2)$ for variable tuple (x_1, x_2, x_3, x_4) is better than $(3, 4, 3, 3)$ as we only need to change the value of one variable to get the constraint satisfied.

To utilize this information, we use the representation in figure 8 with the intermediate node a changed for storing the number of variables currently assigned values in Val. The initial penalty value for each violating tuple is given by the formula $N - a$ as follows:

a	$\delta'_{\texttt{atmost}}$
0	0
⋮	⋮
N	0
$N+1$	-1
⋮	⋮
$\|Var\|$	$N - \|Var\|$

where $|Var|$ is the cardinality of Var. This can be regarded as adding the set of redundant constraints $\{\texttt{atmost}(N+1, Var, Val), \texttt{atmost}(N+2, Var, Val), \ldots, \texttt{atmost}(|Var|-1, Var, Val)\}$. Here we set $contribute_{atmost}(i, (v_1, \ldots, v_n)) = (v_i \in Val)$, where $\in$ is a function returning 1 if value v_i is in Val and 0 otherwise.

We compare our implementations with CHIP on the car-sequencing problem which involves scheduling cars onto an assembly line so that different options can be installed on these cars satisfying various utilization constraints [4]. 50 problems [5] are tested, 10 for each utilization percentage in the range 60% to 80%. Using the method described in [4], CHIP can only manage to solve 6 out of

[5] We thank Andrew Davenport for supplying the car-sequencing benchmarks.

50 problems within one hour. We have also tested the performance of CHIP with the new method in [1]. There is not much improvement. For the two versions of E-GENET, the execution limit is set to 1000 repairs and the results are summerized in table 2. The modified E-GENET can terminate in less than 10 minutes for all 500 runs, requiring no more than 1800 repairs (except in one run). In general, there are increases of 15% to 20% in percentages of successful runs.

utiliza-	Original E-GENET		Modified E-GENET		penalty values
tion %	% succ. runs	median repair	% succ. runs	median repair	stored
60	74	223.5	100	282.5	29
65	80	223.5	99	262	20
70	81	241	100	280.5	30
75	84	339	97	331	74
80	53	576	73	537	187

Table 2. Results on car sequencing problems

5.3 Disjunctive Constraint

To handle a disjunctive constraint $C_1 \vee C_2 \vee \ldots \vee C_n$ in the modified E-GENET, we can use the representation in figure 10. For each constraint C_i, there is a corresponding intermediate node a_i which holds the initial penalty value for the associated tuple under the assignment scheme of C_i. Actually, a_i can be regarded as the degree of violation of C_i. Since we only need one of them satisfied, we add one more intermediate node b to store the maximum value among all a_i's. The assignment scheme of the disjunctive constraint would then be based on the value of b. Use the constraint $x = y + z \vee x = 2y - z$ as an example. The representation is given in figure 11. It is constructed with also techniques described in section 5.1.

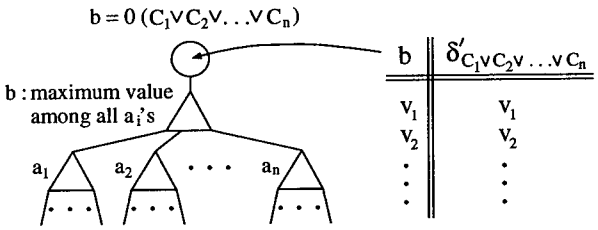


Fig. 10. The representation for a disjunctive constraint

The Hamiltonian path problem is used to test this handling method of disjunctive constraint. The problem can be summarized as follows: given a graph of n vertices, we have to find an ordering of these n vertices $\langle v_1, v_2, \ldots, v_n \rangle$ so that for all i, $1 \leq i < n$, there is an edge between v_i and v_{i+1}. We formulate the problem as in [7]. In CHIP and the original E-GENET, one kind of redundant constraints is used to speed up the execution. We omit these constraints in the modified E-GENET intentionally to test the efficiency of our proposal.

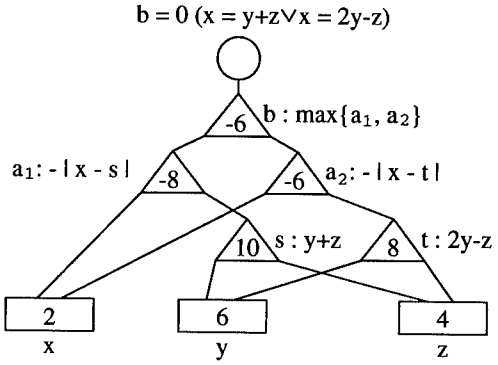


Fig. 11. The representation for the constraint $x = y + z \vee x = 2y - z$

The benchmarking results are listed in table 3. CHIP uses choices [10] to model disjunctive constraints. This approach induces a large number of backtracking and fails to solve a graph with 50 vertices in 10 hours. With more information provided by penalty values, the modified E-GENET is several times faster than the previous version.

graph node	CHIP CPU Time (sec)	Original E-GENET CPU Time (sec)	Modified E-GENET CPU Time (sec)	penalty values stored
30	581.640	54.000	6.550	215
40	4023.500	587.158	29.465	350
50	–	4897.092	783.790	2943

Table 3. Results on Hamiltonian path problems

5.4 Cumulative Constraint

The `cumulative` constraint [1] in CHIP is found to be useful as some real world problems like scheduling and placement problems can be stated more easily and directly. Thus, we implement it in modified E-GENET. The constraint has the form `cumulative`$([O_1, \ldots, O_m], [D_1, \ldots, D_m], [R_1, \ldots, R_m], L)$ and can be explained with a simple scheduling problem of m tasks and one kind of resources. We can treat $O_1, \ldots, O_m$ as starting times for the tasks. Each task i uses R_i amount of resources and lasts for D_i units of time. The constraint holds if at any time unit t, resources required are less than L, $\sum_{i|O_i \leq t \leq O_i + D_i - 1} R_i \leq L$.

To handle the constraint, we can break it down into several inequality constraints and apply the techniques described in section 5.1. Assume $\alpha(V)$ be the minimum value in the domain of variable V and $\beta(V)$ the maximum. Let $t' = min\{\alpha(O_1), \ldots, \alpha(O_m)\}$ and $t'' = max\{\beta(O_1) + \beta(D_1), \ldots, \beta(O_m) + \beta(D_m)\}$. For each time unit t between t' and t'', we use an intermediate node to hold the resources required at that unit. In figure 12, the intermediate node a_1 is for t', a_2 for $t' + 1$, and so on. Then we need to solve n constraints of $a_i \leq L$, where $n = t'' - t' + 1$.

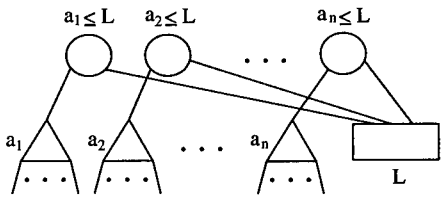


Fig. 12. The representation for the `cumulative` constraint

Simple scheduling problems of different number of tasks and resources utilization percentage (three problems for each setting) are used as testing examples and the results are shown in table 4. CHIP cannot solve two problems with 21 tasks within 1 hour. The performance of the original E-GENET is extremely poor as there is only one constraint in each problem. However, with the modifications, the new version of E-GENET can solve all problems efficiently.

task no.	utilization %	CHIP CPU Time (sec)			Modified E-GENET CPU Time (sec)			penalty values stored		
7	80	0.020	0.010	0.010	0.010	0.000	0.000	0	0	0
	85	0.025	0.010	0.010	0.010	0.040	0.000	0	0	0
	90	0.015	0.010	0.010	0.095	0.010	0.000	49	2	0
	95	0.015	0.010	0.010	0.050	0.010	0.010	37	3	5
14	80	0.205	0.025	0.015	0.090	0.095	0.020	3	2	1
	85	0.030	0.030	0.020	0.180	0.085	0.030	12	6	2
	90	4.630	0.020	0.070	0.410	0.375	0.030	38	43	2
	95	43.920	6.940	5.330	11.185	13.950	0.125	714	727	25
21	80	0.030	0.030	0.035	0.115	0.140	0.045	1	3	0
	85	0.055	0.045	0.030	0.435	0.265	0.080	13	4	2
	90	1269.560	0.125	0.030	2.420	0.915	0.135	57	28	9
	95	–	–	4.190	69.460	16.780	0.380	1501	407	23

Table 4. Results on simple scheduling problems

6 Related Work

Davenport *et al.* [2] proposed an extension to GENET for non-binary constraints, in which a hyper-edge is used to link up incompatible labels in an n-ary constraint. Our approach is different from theirs in three aspects:

1. In E-GENET, all constraints, binary or non-binary, have the same status and are represented homogeneously, while they are much smaller in size than their counterparts in the scheme of Davenport *et al.* For example, for the constraint $x+y = u+v$ where all variables have the domain $\{1, \ldots, 100\}$, their scheme needs around 100^4 constraint nodes whereas only **3** nodes are needed in E-GENET and penalty values can be computed on demand.
2. The cost of network construction in E-GENET is low compared with that for the extension of Davenport *et al* as huge amount of work on building connections is removed. For the above example, 4×100^4 connections are required in their scheme but we can just use 6 in E-GENET.

3. E-GENET also provides much greater flexibility. New types of constraints and domain knowledge can be incorporated easily, not the case for their scheme, by using intermediate nodes and suitable assignment schemes of initial penalty values. As a result, better performance can be obtained.

7 Conclusion

In this paper, several modifications to E-GENET are proposed. The new version of E-GENET has three advantages over the original one. First, memory requirement is largely reduced. Second, there is a great increase in performance, especially on highly constrained problems. Third, some very complex constraints can now be handled. Both versions of E-GENET outperform CHIP on CSP's requiring much backtracking tree search. However, more experiments have to be done to compare these two approaches in different regions of the problem space.

Interesting future work includes applying the `cumulative` and the `diffn` constraint [1] of the modified E-GENET on real world problems and further enhancement of E-GENET to solve partial constraint satisfaction problems [6] and optimization problems.

References

1. N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Journal of Mathematical and Computer Modelling*, 17(7):57–73, 1994.
2. A. Davenport, E. Tsang, C. J. Wang, and K. Zhu. GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proc. 12th National Conference on Artificial Intelligence*, 1994.
3. D. Diaz and P. Codognet. A minimal extension of the WAM for `clp(FD)`. In *Proc. 10th International Conference on Logic Programming*, pages 774–790, 1993.
4. M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving car sequencing problem in contraint logic programming. In *Proc. European Conference on AI*, pages 290–295, 1988.
5. M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proc. International Conference on Fifth Generation Computer Systems*, pages 693–702, December 1988.
6. E. C. Freuder. Partial constraint satisfaction. In *Proc. 11th International Joint Conference on AI*, pages 278–283, 1989.
7. J.H.M. Lee, H.F. Leung, and H.W. Won. Extending GENET for non-binary CSP's. In *Proc. 7th International Conference on Tools with Artificial Intelligence*, pages 338–343, 1995.
8. S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.
9. The COSYTEC Team. *CHIP V4.1 User Manuals*, 1994.
10. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.
11. T. Warwick and E.P.K. Tsang. Tackling car sequencing problems using a generic genetic algorithm. *Evolutionary Computation*, 3(3):267–298, 1995. (to appear).

A View of Local Search in Constraint Programming

Gilles Pesant[1] and Michel Gendreau[1,2]

[1] Centre for Research on Transportation,
Université de Montréal,
C.P. 6128, succ. Centre-ville, Montréal, Canada, H3C 3J7
[2] Département d'informatique et de recherche opérationnelle,
Université de Montréal,
C.P. 6128, succ. Centre-ville, Montréal, Canada, H3C 3J7
{pesant,michelg}@crt.umontreal.ca

Abstract. We propose in this paper a novel way of looking at local search algorithms for combinatorial optimization problems which better suits constraint programming by performing branch-and-bound search at their core. We concentrate on neighborhood exploration and show how the framework described yields a more efficient local search and opens the door to more elaborate neighborhoods. Numerical results are given in the context of the traveling salesman problem with time windows. This work on neighborhood exploration is part of ongoing research to develop constraint programming tabu search algorithms applied to routing problems.

Introduction

Local search methods in operations research (OR) date back to over thirty years ago ([Lin65]). Applied to difficult combinatorial optimization problems, this heuristic approach yields high-quality solutions by iteratively considering small modifications (called *local moves*) of a good solution in the hope of finding a better one. Used within a strategy designed to escape local optima such as simulated annealing and tabu search, it has been very successful in achieving near-optimal (and sometimes optimal) solutions to a variety of hard problems ([GLTdW93][Ree93]).

In solving real-life combinatorial optimization problems, constraint programming (CP) has to date almost invariably adopted the branch-and-bound strategy, a global and therefore complete search method.[3] When an exact algorithm proved too costly, approximate algorithms were often devised by heuristically discarding the least-promising edges in the branch-and-bound search tree. The lack of popularity of local search in constraint *logic* programming may be attributed to the apparent need to *modify* solution variables when performing a local move.

[3] Though incomplete search methods have been successfully introduced in the constraint *satisfaction* community.

A few exceptions are nevertheless found in the CP literature. In the context of a locomotive scheduling problem reduced to a traveling salesman problem with deadlines, [Pug92] opts for an iterative improvement method which repeatedly looks for a local move improving on the current best solution. Local moves are defined here as a particular replacement of three edges in the solution. Every possible move is performed and then checked against the constraints. [CSKA94] use the vehicle routing problem to compare a standard branch-and-bound approach to iterative improvement with local moves defined as the exchange of two vertices in the route. Again the constraints ensure that only feasible moves are considered. [CL95] describe a disjunctive scheduling system of which local search is a component. Two types of local moves are considered: "repair" moves swap two tasks scheduled on the same machine while "shuffle" moves only keep part of the solution and search through the rest of the solution space to complete it, guided by constraint propagation. The latter can be seen as branch-and-bound starting from some internal node of a search tree for the original problem. A limited number of backtracks is allowed since the purpose of local search in this context is to quickly improve an initial solution before resorting to (full) branch-and-bound search.

All of the above, with the exception of "shuffle" moves, essentially consider each of the possible moves individually to then assess their feasibility and cost. This can be a costly endeavor in both OR and CP when the nature of the local moves is such that a great number of possibilities must be considered. In a way, "shuffle" moves bear a resemblance to what we have in mind. We believe that local search is not so foreign to branch-and-bound search and that constraints can be more actively involved in the exploration of these local search spaces. We propose in this paper a clean integration of local search in constraint programming by keeping on doing what comes naturally, i.e. branch-and-bound, but on a different search space, though related to the original one. In contrast with OR, the resulting framework maintains a clear separation between the constraints of the problem and the actual search procedure. For both OR and CP, the potential pruning capabilities open the door to more elaborate local moves, which could lead to even better approximate results. In addition, it does not require modifying the value of (logic) variables.

The rest of the paper is organized as follows. Section 1 first gives an overview of local search methods in OR. Our general framework for local search in CP is then presented in section 2 to be followed by an instance of it in section 3. Finally, an experimental evaluation in section 4 provides insight into the potential gain of such an approach.

1 Local Search Methods in Operations Research

Local search methods generally involve repeatedly going from one solution to another through a *local move*. What constitutes a valid local move will vary according to the problem and even within it, as we shall see in section 3.1. The set of all solutions reachable from a solution s through a local move is termed

the *neighborhood* of s. The set of all feasible solutions in this neighborhood will be called its *feasible neighborhood.*

Given this framework, a simple strategy called *iterative improvement* moves to the best feasible neighbor (i.e. of lowest cost) every time until it does not improve on the current solution, reaching a local optimum. Some ways of alleviating the obvious drawback of this strategy have been proposed. *Multi-start iterative improvement* achieves local optima from a pool of solutions and returns the best one. *Genetic local search* builds upon the previous by recombining the local optima (in the fashion of genetic algorithms [Hol75]), applying iterative improvement, discarding the least-promising solutions and repeating the process until some stopping criterion is satisfied.

Two very successful strategies try to escape local optima by allowing moves which temporarily increase the cost of the solution. *Tabu search* ([Glo77]) moves to the best neighbor at each iteration, regardless of whether or not it improves on the current solution. To avoid cycling, a dynamic list of tabu solution attributes is kept. Typically, such a list covers recently examined solutions, which will remain forbidden for a certain number of iterations. *Simulated annealing* ([KGJV83]) randomly selects a neighbor at each iteration. If it improves on the current solution, the move is performed; otherwise, it will be performed with a certain probability which depends on the cost difference and which also decreases over time according to a cooling schedule. Both strategies iterate until some stopping criterion is satisfied.

One crucial aspect in all of these local search methods is obviously the choice of the neighborhood structure (see for example [GTdW93]). Ambitious neighborhoods increase the chances of success but are more expensive to explore for methods which need to do so. Small neighborhoods are both simple and fast to explore but may prevent us from ever reaching a particularly good solution: it could require a sequence of local moves (as opposed to a single one in a larger neighborhood), and in tabu search every such move would have to be the best one locally. A neighborhood which strictly includes another induces a search which encounters fewer local optima and thus facilitates their avoidance.

Because of the above, large neighborhoods seem to be the current trend, as exemplified by the recent CROSS exchange ([TBG+95]) which generates a neighborhood of size $\mathcal{O}(n^4)$. Several techniques have been developed to speed up this exploration. The size of the neighborhood can be somewhat reduced either by ignoring parts of it which are unlikely to produce good solutions or, in a more exact fashion, by interrupting a carefully engineered exploration when the remainder can only lead to worse (or infeasible) solutions. In addition, the feasibility of neighbors must be assessed. As an early example in routing problems, [Sav85] describes a way to verify time window constraints in constant time per neighbor, though assumptions about the neighborhood structure must be made. Others may perform approximate tests of feasibility to quickly identify promising neighbors which are then thoroughly investigated, with the potential risk of missing the best one.

So, the challenge of expressive neighborhoods has been met with specialized techniques embedded in the local search and sometimes with compromises.

2 Local Search as Branch-and-Bound

This adaptation of local search to the constraint programming paradigm stems from its perception as a generalization of traditional branch-and-bound search. In the latter case, we branch on the variables of the model and the neighborhood degenerates into the whole solution search space for which a single iteration becomes sufficient since the optimal solution will necessarily be found. In order to lift this approach to local search, the art then consists of choosing a representation for a particular neighborhood structure which CP branch-and-bound search can exploit. Each iteration of local search will simply be a branch-and-bound search, but on a different search space. In the remainder, we shall concentrate on the way neighborhood structures are explored during a single iteration. The arboreal exploration at the heart of a branch-and-bound strategy remains the way we examine the neighborhood of a solution. To complete the parallel, the active role of modeling constraints together with lower bounds on the cost of partial solutions will help prune the tree and thus reduce the search effort over the whole neighborhood.

We now formalize the idea presented above. Let $\mathcal{N}$ denote some neighborhood structure for solutions to a problem $\mathcal{P}$. A set of finite domain variables $\{\nu_1, \ldots, \nu_k\}$, usually distinct from the variables appearing in the model for $\mathcal{P}$, together with a (possibly empty) set of constraints on $\{\nu_1, \ldots, \nu_k\}$ is a *neighborhood model* for $\mathcal{N}$ if there is a one-to-one mapping between the set of feasible combinations of values for $\{\nu_1, \ldots, \nu_k\}$ and the neighbors in $\mathcal{N}$.

To illustrate this, consider again the neighborhood of [CSKA94] for a route on m cities. We introduce variables I and J both ranging over $1, 2, \ldots, m$ and with the constraint $I < J$ between them. If the solution is represented as $\langle c_1, \ldots, c_m \rangle$, the sequence of cities forming the route, a particular (feasible) combination of values for $\{I, J\}$ is interpreted as exchanging entries c_I and c_J in that solution to obtain a neighbor. One easily verifies that this constitutes a neighborhood model.

The requirement of a one-to-one mapping could be relaxed to a *surjective* mapping though this would mean that a neighbor may be examined more than once. For example without the constraint $I < J$, symmetries would lead to identical solutions. However, surjectivity is crucial since otherwise we would miss some of the neighbors.

So we shall *branch* on $\{\nu_1, \ldots, \nu_k\}$ and also *bound* the cost of partially constructed neighbors. As we saw in section 1, some heuristic algorithms based on local search, such as tabu search and variations on iterative improvement, are interested in acquiring the *best* solution in the neighborhood and hence do not require to manipulate the whole of it per se. We can therefore record the cost of the best neighbor found so far and compute lower bounds for partial neighbors to further reduce the portion of the neighborhood that has to be explored, as in traditional branch-and-bound.

We see two main advantages to such an approach:

- The modeling constraints for $\mathcal{P}$ (and often some of the variables) are kept separate from the neighborhood model. This is especially useful when lots of different side constraints are present in the problem: they will not clutter the local search with explicit feasibility tests as is often the case in operations research heuristics. They will rather be indirectly involved when some of the modeling variables will be further constrained as a result of choices made on $\{\nu_1, \ldots, \nu_k\}$ (section 3 describes in detail an instance of this). The end result is a *generic* neighborhood exploration method parameterized by the type of neighborhood but *not* by the modeling constraints.
- There is a strong relationship between the savings brought about by this branch-and-bound approach and how ambitious the neighborhood is. Typically, enlarging the neighborhood means increasing the degrees of freedom and so requires a greater number of variables to encode its structure. This translates into a greater depth of the (neighborhood) search tree and a potentially larger gain with every branch pruned, either from the lower bound at the particular node or the modeling constraints. This constitutes an asset in view of the current trend toward more ambitious neighborhood structures.

3 An Example for the Traveling Salesman Problem with Time Windows

This section provides a larger and more interesting instance of the general framework just described. It addresses a well-known problem on which several local search heuristics have been applied in the past. The traveling salesman problem with time windows (TSPTW) consists of finding a minimum cost (usually the total travel distance or total schedule time) tour of a set of cities where each city is visited exactly once and which starts and ends at a unique depot. In addition, each city must be visited within its own time window. Early arrival is allowed but implies a waiting time until the beginning of the window. [Sav85] showed that simply deciding whether there exists a feasible solution to an instance of the TSPTW is NP-complete. Nevertheless, the full problem has important applications in bank and postal deliveries, school-bus routing and scheduling, disjunctive scheduling with sequence-dependent processing times, automated manufacturing environments and as a subproblem of the vehicle routing problem with time windows (VRPTW).

We will sketch a constraint programming model for the TSPTW since some of its variables will be referred to later on. The actual details of this model appear in [PGPR96] but are not relevant here. Let $V = \{2, \ldots, n\}$ represent the cities to visit and duplicate the unique depot into an origin-depot and a destination-depot, identified as 1 and $n+1$, respectively. A tour thus becomes a Hamiltonian path starting at 1 and ending at $n+1$. At the heart of the model are variables S_i, $i = 1, \ldots, n$ associated to each of the cities (and the origin-depot) and which represent their successor in the tour. Their domain will therefore be an integer

in the range $2, \ldots, n+1$. A valid tour assigns a distinct successor to each city and avoids sub-tours. We also define predecessor variables P_j, $j = 2, \ldots, n+1$ which represent the symmetric counterpart of the S_i's ($S_i = j \Leftrightarrow P_j = i$). To account for the scheduling component of the problem, variables T_i are introduced to represent the time at which we visit each city. These must take a value within their respective time window while being coherent with the order in which cities appear on the tour and taking into consideration the travel time between cities.

3.1 Some Neighborhoods for Routing Problems

Given the inherent difficulty of the problem, several OR heuristic algorithms have been designed and among them some based on local search. We briefly describe the most popular neighborhoods in the context of *routing* problems, which include the TSPTW but may allow solutions consisting of a set of routes and involve other restrictions such as capacity constraints.

Neighborhoods generated by modifications at the level of vertices include re-inserting a vertex or exchanging two vertices. *GENI*, a generalized insertion procedure ([GHL92]), directs the re-insertion between some of its p (typically 5) nearest neighbors. In the *λ-interchange* ([Osm93]), subsets of vertices S_i, S_j (with $|S_i| \leq \lambda$ and $|S_j| \leq \lambda$) are selected each from a different route and then swapped. The possibility of an empty subset allows simply re-inserting vertices. For efficiency reasons, λ rarely exceeds 2.

At the level of edges, a *k-interchange* ([Lin65]) replaces k edges in the solution by k others that reconnect the route(s). We explore the associated neighborhood to find a *k-opt* solution, which cannot be improved by a k-interchange and is therefore a local optimum. Various reported experiments use $k = 2$ or 3. An *Or-opt* move ([Or76]) relocates a string of one, two or three consecutive vertices. A *2-opt** move ([PR95]) replaces two edges (v_i, v_j), (v_k, v_ℓ) from different routes with edges (v_i, v_ℓ), (v_k, v_j). This exchanges the end portion of one route with the end of the other. The *CROSS exchange* ([TBG+95]) goes further by exchanging a middle portion of one route with a middle portion of the other and includes the two previous types of moves as special cases.

3.2 3-Opt Edge Exchange

We will give a neighborhood model for the 3-interchange neighborhood. Let $\mathcal{T}$ be the current tour. If I represents a city on this tour, then I^+ (resp. I^-) denotes its successor (resp. predecessor) on $\mathcal{T}$. We define the following binary relation on cities: $I \prec_{\mathcal{T}} J$ holds if I appears before J on $\mathcal{T}$ (the subscript will be dropped unless necessary to avoid confusion). $I \preceq J$ will be used as shorthand for "$I \prec J$ or I is the same as J".

Without loss of generality, let $I \prec J \preceq K$. A *3-interchange move* from $\mathcal{T}$ deletes the three edges (I, I^+), (J^-, J), (K, K^+) and then reconnects the tour by introducing three new ones on the vertices (cities) I, I^+, J^-, J, K and K^+. An *orientation-preserving* 3-interchange move reconnects the tour in the only possible way which does not reverse any of the original route segments, by

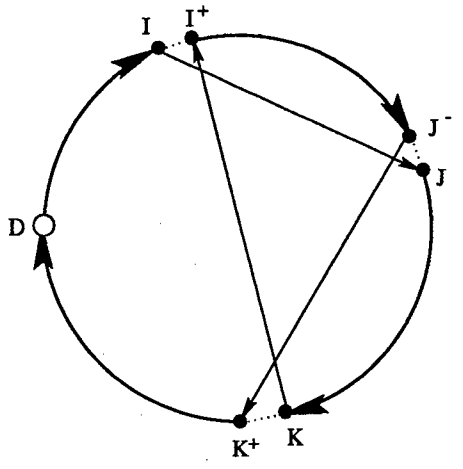


Fig. 1. The orientation-preserving 3-interchange.

adding edges (I, J), (K, I^+), (J^-, K^+) (see figure 1). This is desirable when a scheduling component is present in the problem, such as time windows. The new tour would then start at the depot D, follow the same path to I as in $\mathcal{T}$, go to J, proceed to K as in $\mathcal{T}$, go to I^+, proceed to J^- as in $\mathcal{T}$, go to K^+ and finally return to D as in $\mathcal{T}$. Two segments of the original route have been swapped but the ordering within a segment is kept.

The neighborhood defined by orientation-preserving 3-interchange moves can be encoded through three finite domain variables I, J, K, ranging from 1 to n. Their interpretation is the one given above. Because of the original constraint $I \prec J \preceq K$, each neighbor is in one-to-one correspondence with a 3-tuple of values describing it. We therefore have a neighborhood model.

Initially, the CP model for the problem is stated and the variables of that model (S_i's, P_j's and T_i's) are only constrained to that extent — in other words, they are not bound to the current solution $\mathcal{T}$. Then the local search takes place. Within our tree search, the variable ordering will be fixed to I, J, K. We now describe the information exploited at each level of this tree (refer to figure 1 throughout).

level 0, the root of the tree:

Initially, we simply have that $I \in \{1, \ldots, n\}$, $J, K \in \{2, \ldots, n\}$, $I \prec J \preceq K$. That latter constraint will become active as the domains of I, J, K shrink (and eventually hold a single value).

level 1, I fixed:

1. The path from D to I is identical to that in $\mathcal{T}$. We can perform the following variable bindings: $S_D = D^+, S_{D^+} = D^{++}, \ldots, S_{I^-} = I$.

2. I^+ no longer belongs to the domain of S_I since that edge will be removed, and the domain of J can be identified with that of S_I.
3. Similarly, I no longer belongs to the domain of P_{I+} and the domain of K can be identified with that of P_{I+}.

level 2, I, J fixed:

1. Edge (I, J) is added and the path from I^+ to J^- is identical to that in $\mathcal{T}$: variables S_I and $S_{I+}, S_{I++} \ldots, S_{J--}$ become bound.
2. K^+ must be one of the values in the domain of S_{J-}. Consequently, the domain of K can be further tightened: any value for K must appear in $\{E^- \mid E \in$ domain of $S_{J-}\}$.

level 3, I, J, K fixed:

We have reached a leaf of our tree — the rest of the tour can be completed by fixing the rest of the S_i's to their appropriate value.

Applying branch-and-bound yields an orientation-preserving 3-opt move (we will discuss lower bounds in section 3.3). At different stages of the search, we manage to express constraints that restrict the set of allowable values for the three variables on which that search is performed but, more importantly, we constrain as well the principal modeling variables (the S_i's) which may independently prune our search tree by propagating these changes through modeling constraints we need not know anything about. Note that the time variables T_i were not directly involved either.

By the way, the neighborhood structure just described also corresponds to the one in [Pug92]. As for the other ones we encountered in the introduction, the "repair" moves of [CL95] could be modeled as those of [CSKA94] with an extra variable, say M, to represent the choice of a machine; to fit "shuffle" moves into our framework, since branching is done not on the value of a variable but on the ordering of a pair of tasks, one could associate a neighborhood model binary variable ν_{ij} to every pair t_i, t_j left unordered and introduce constraints to propagate the transitivity of the partial order relation. The other neighborhood structures described in section 3.1 are easily modeled.

3.3 Lower Bounds on the Cost of a 3-Interchange Neighbor

Associate a cost c_{ij} to every edge (i, j) and let $\mathcal{C} = \sum_{(i,j)\in\mathcal{T}} c_{ij}$ be the total cost of $\mathcal{T}$. The cost of a neighbor (I, J, K) of $\mathcal{T}$ is given by the following formula, replacing the costs of the old edges by that of the new:

$$\mathcal{C} + (c_{IJ} - c_{II+}) + (c_{KI+} - c_{KK+}) + (c_{J-K+} - c_{J-J}). \qquad (1)$$

Given I, J, K, we can therefore compute this cost in constant time. In the spirit of branch-and-bound search, we will seek lower bounds on the cost of incomplete neighbors as we traverse the tree. The following formulas are easily derived for the two interesting levels, 1 and 2. If only I has been fixed (level 1), we have:

$$\mathcal{C} + (\min_J\{c_{IJ}\} - c_{II+}) + \min_{J,K}\{(c_{KI+} - c_{KK+}) + (c_{J-K+} - c_{J-J})\}. \qquad (2)$$

Similarly, if both I and J have been fixed (level 2):

$$C + (c_{IJ} - c_{II+}) + (\min_{K}\{(c_{KI+} - c_{KK+}) + c_{J-K+}\} - c_{J-J}). \quad (3)$$

While the latter can be computed in $\mathcal{O}(n)$ time, (2) requires $\mathcal{O}(n^2)$ time because of the last term in the sum. On the other hand, the number of possible values for J and K once I has been selected could often be a small fraction of n. We may achieve a tighter lower bound by insisting that the minimizing J be the same in the second and third terms of (2), yielding:

$$C + (\min_{J,K}\{c_{IJ} + (c_{KI+} - c_{KK+}) + (c_{J-K+} - c_{J-J})\} - c_{II+}). \quad (4)$$

From an implementation point of view, the effect of (1)-(4)-(3) can be achieved by initially posting a constraint relating the cost of the neighbor to the value of I, J, K.

4 Experimental Results

In order to evaluate the potential of the ideas developed in the paper, the neighborhood model detailed in the previous section was tested on problem instances taken from the literature. The TSPTW constraint programming model of [PGPR96] was used to describe the problem.

We considered two sets of symmetric Euclidean problems with travel times between cities taken as the distance separating them. The first set comes from [DDGS95] and features instances on 20 cities uniformly distributed on the $[0, 50] \times [0, 50]$ grid with time windows of maximum width 20, 40, 60, 80 and 100. These problems tend to be fairly constrained. The second set uses subproblems of the well-known VRPTW test bed in [Sol87]. The original problem instances feature 100 cities distributed on the $[0, 100] \times [0, 100]$ grid and require several vehicles to service them while obeying the side constraints. Some partition of the cities such that each group may be visited by a single vehicle yields our subproblems. Their resulting size varies from 16 to 49 cities. We used some of the instances in the C2, R2 and RC2 classes of problems for which the cities are respectively clustered, uniformly distributed and a mixture of the two. This second set is more heterogeneous and instances sometimes include very few meaningful time windows.

We introduce two types of indicators to analyze the results:

$$\gamma = \frac{|\text{feasible neighborhood}|}{|\text{neighborhood}|}$$

will indicate how constrained a problem instance is by evaluating for a given solution the proportion of neighbors which are feasible;

$$\epsilon = \frac{\#\text{backtracks}}{|\text{neighborhood}|}$$

will measure the effort put into our neighborhood exploration by comparing the number of backtracks required with the size of the neighborhood. Both γ and ϵ necessarily range between 0 and 1. Tests were conducted with and without making use of lower bounds on the cost of partial solutions in order to evaluate their impact. The search effort for the case without lower bounds will be denoted ϵ^-.

The results on the first set are summarized in figure 2 and in figures 3, 4 and 5 for the three classes in the second set. Each value reported represents an average over a few problem instances and solutions to them.

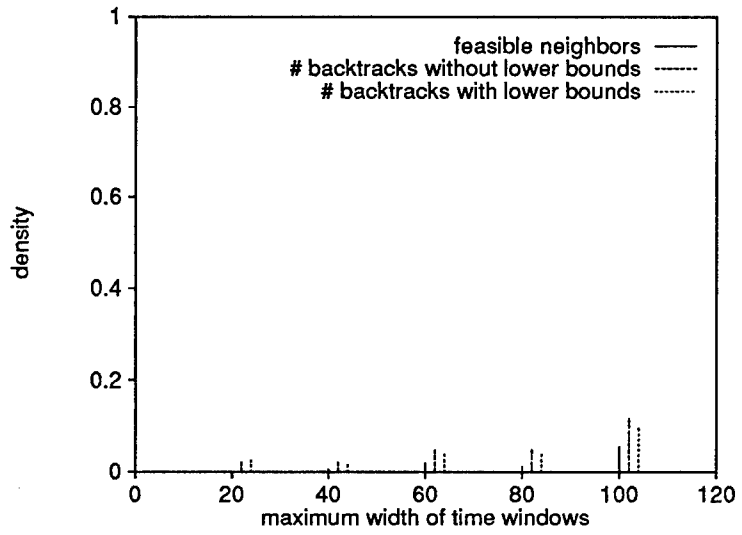


Fig. 2. Indicators γ, ϵ^- and ϵ for some instances in the first set of problems.

Our first observation is that practically all these problem instances are quite constrained in the neighborhood of the solutions considered: γ averages 0.03 and never rises above 0.18. Avoiding the systematic exploration of the whole neighborhood therefore appears advantageous. The relatively small extent of this first experiment makes it premature to conclude that the problem instances have quite constrained neighborhoods for all their solutions but it would be interesting to verify this. Secondly, our search effort is often a small fraction of the neighborhood search space: ϵ^- and ϵ average 0.13 and 0.09 respectively. In the worst case, ϵ^- approaches 0.5 (figure 5, rc208) but ϵ does not exceed 0.29 (figure 4, r204). The beneficial effect of maintaining lower bounds in the neighborhood model goes as far as cutting down by half the number of backtracks on some instances. Though the relative merits of ϵ^- and ϵ have little to do with the CP model used here, their actual values do. The model used includes redundant

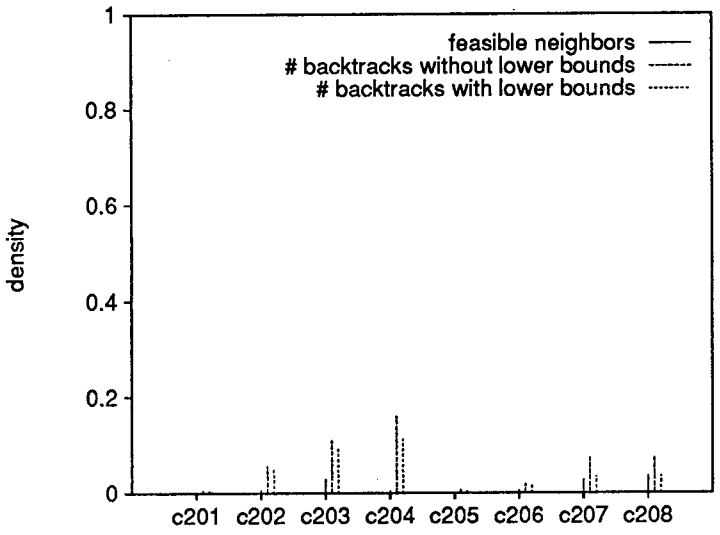


Fig. 3. Indicators γ, ϵ^- and ϵ for some instances in C2.

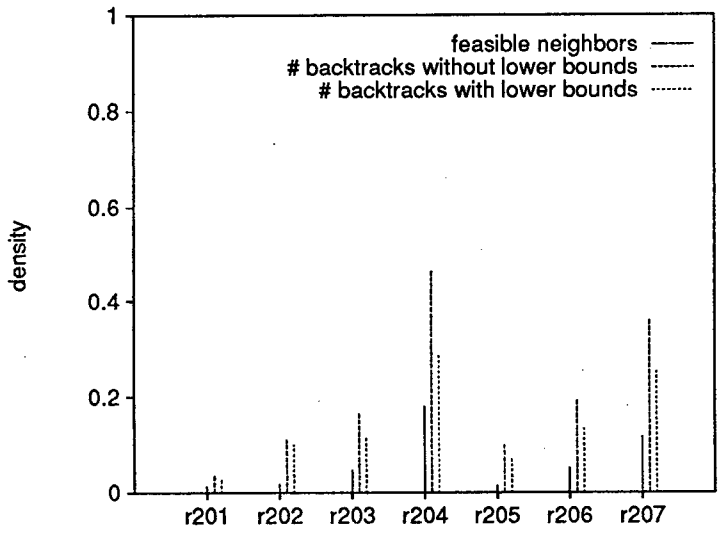


Fig. 4. Indicators γ, ϵ^- and ϵ for some instances in R2.

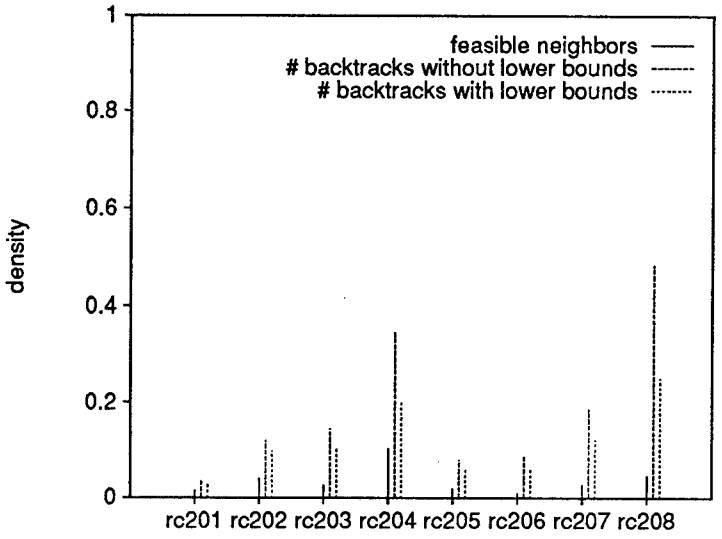


Fig. 5. Indicators γ, ϵ^- and ϵ for some instances in RC2.

constraints for more powerful propagation which ends up producing smaller ϵ's but also slows down the exploration. As usual, a reasonable compromise must be sought. It is worth mentioning that sometimes ϵ was even smaller than γ (e.g. figure 3, c208) — in such cases, the bounds were particularly productive in pruning subtrees containing unattractive feasible neighbors.

Looking at the sets of problems individually, all three indicators in figure 2 tend to increase with the width of the time windows, as one would expect. The intriguing periodicity apparent in figures 3, 4 and 5 is probably due to the way these problems were originally generated. [Sol87] mentions that different percentages of cities with time windows were used, namely 100, 75, 50 and 25%. A greater proportion of constraining time windows will likely decrease the density of feasible neighbors.

Conclusion

We have proposed a novel way of looking at local search algorithms in constraint programming. By maintaining branch-and-bound search at their core, we believe that a cleaner and more natural integration is achieved. In addition, the familiar pruning which can take place yields substantial savings on the number of neighbors that actually need to be considered.

The extra efficiency brought about by lower bounds on the cost of partial solutions makes more attractive methods which explore the whole neighborhood in search of the best local move, such as tabu search. We certainly advocate the

use of a local search method more subtle than plain iterative improvement, as others within the CP community have already suggested.

This work on neighborhood exploration is part of ongoing research to develop CP tabu search algorithms applied to routing problems.

Acknowledgments

We wish to thank the two anonymous referees for their constructive comments. Financial support for this research was provided by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[CL95] Y. Caseau and F. Laburthe. Disjunctive Scheduling with Task Intervals. Technical Report 95-25, Laboratoire d'informatique de l'École Normale Supérieure, Département de mathématiques et d'informatique, 45 rue d'Ulm, 75230 Paris Cedex 05, France, 1995.

[CSKA94] N. Christodoulou, E. Stefanitsis, E. Kaltsas, and V. Assimakopoulos. A Constraint Logic Programming Approach to the Vehicle-Fleet Scheduling Problem. In *Proceedings of Practical Applications of Prolog*, 1994.

[DDGS95] Y. Dumas, J. Desrosiers, É. Gélinas, and M.M. Solomon. An Optimal Algorithm for the Traveling Salesman Problem with Time Windows. *Operations Research*, 43(2):367–371, 1995.

[GHL92] M. Gendreau, A. Hertz, and G. Laporte. New Insertion and Postoptimization Procedures for the Traveling Salesman Problem. *Operations Research*, 40:1086–1094, 1992.

[Glo77] F. Glover. Heuristic for Integer Programming Using Surrogate Constraints. *Decision Sciences*, 8:156–166, 1977.

[GLTdW93] F. Glover, M. Laguna, É. Taillard, and D. de Werra. Tabu Search. volume 41 of *Annals of Operations Research*. 1993.

[GTdW93] F. Glover, É. Taillard, and D. de Werra. A User's Guide to Tabu Search. *Annals of Operations Research*, 41:3–28, 1993.

[Hol75] J.H. Holland. *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, MI, 1975.

[KGJV83] S. Kirkpatrick, C.D. Gelatt Jr, and M.P. Vecchi. Optimization by Simulated Annealing. *Science*, 220:671–680, 1983.

[Lin65] S. Lin. Computer Solutions of the Traveling Salesman Problem. *Bell System Technical Journal*, 44:2245–2269, 1965.

[Or76] I. Or. *Traveling salesman-type combinatorial problems and their relation to the logistics of regional blood banking.* PhD thesis, Northwestern University, Evanston, IL, 1976.

[Osm93] I.H. Osman. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annals of Operations Research*, 41:421–451, 1993.

[PGPR96] G. Pesant, M. Gendreau, J.-Y. Potvin, and J.-M. Rousseau. An Optimal Algorithm for the Traveling Salesman Problem with Time Windows using Constraint Logic Programming. Publication CRT-96-15, Centre de recherche sur les transports, Université de Montréal, Montréal, 1996.

[PR95] J.-Y. Potvin and J.-M. Rousseau. An Exchange Heuristic for Routing Problems with Time Windows. *Journal of the Operational Research Society*, 46(12):1433–1446, 1995.

[Pug92] J.-F. Puget. Object-Oriented Constraint Programming for Transportation Problems. In *Proceedings of Advanced Software Technology in Air Transport (ASTAIR)*, 1992.

[Ree93] C.R. Reeves. *Modern Heuristic Techniques for Combinatorial Problems.* Halsted Press, 1993.

[Sav85] M.W.P. Savelsbergh. Local Search in Routing Problems with Time Windows. *Annals of Operations Research*, 4:285–305, 1985.

[Sol87] M.M. Solomon. Algorithms for the Vehicle Routing and Scheduling Problem with Time Window Constraints. *Operations Research*, 35:254–265, 1987.

[TBG+95] É. Taillard, P. Badeau, M. Gendreau, F. Guertin, and J.-Y. Potvin. A New Neighborhood Structure for the Vehicle Routing Problem with Time Windows. Publication CRT-95-66, Centre de recherche sur les transports, Université de Montréal, Montréal, 1995.

From Quasi-Solutions to Solution: An Evolutionary Algorithm to Solve CSP

María Cristina Riff Rojas
INRIA, CERMICS
BP 93. 06902 Sophia-Antipolis, France
mcriff@sophia.inria.fr

Abstract. This paper describes an Evolutionary Algorithm that repairs to solve Constraint Satisfaction Problems. Knowledge about properties of the constraints network can permit to define a fitness function which is used to improve the stochastic search. A selection mechanism which exploits this fitness function has been defined. The algorithm has been tested by running experiments on randomly generated 3-colouring graphs, with different constraints networks. We have also designed a specialized operator "permutation", which permits to improve the performance of the classic crossover operator, reducing the generations number and a faster convergence to a global optimum, when the population is staying in a local optimum. The results suggest that the technique may be successfully applied to other CSP.

Keywords: Constraint satisfaction, Evolutionary algorithms,Fitness evaluation

1 Introduction

Constraint satisfaction problems, or CSP are widely used in artificial intelligence. The goal is to find the values for problem variables that satisfy the imposed constraints. Genetic Algorithms have been applied to solving CSOP, [15], and CSP, [5], [4], [12], [9], [1], [6], however the researchers have been concentrated in studying the genetic representation and the reproduction mechanisms more than the evaluation function definition, which in general, have been defined as the number of satisfied constraints.

This paper introduces a new evaluation function. This fitness function uses information about the connectivity of the constraints network, which is represented by a constraints matrix. Such network offers potential advantages in terms of knowledge on variables interaction (epistasis). In addition, a fitness-proportionate selection algorithm has been defined to improve the genetic search. The remainder of this paper is organized as follows. After defining what we mean by CSP in relation with Genetic Algoritms (GA) in section 2, an evolutionary algorithm with a new approach to calculate the evaluation function is presented in section 3. We then address the graph 3-colouring problem subject to the restriction that adjacent nodes in the graph must be colored differently in section 4, it also contains the definition of a new operator "permutation" which permits to improve

the performance of the classic crossover operator. Pointers to future research and conclusions are given in section 5.

2 Constraint Satisfaction Problems and Genetic Algorithms

A *Constraint Satisfaction Problem* (CSP) is composed of a set of *variables* $V = X_1, \ldots, X_n$, their related *domains* $D_1, \ldots, D_n$ and a set containing NC *constraints* on these variables. The domain of a variable is a group of values to which the variable may be instantiated. The domain sizes are $m_1, \ldots, m_n$, respectively,and we let m denote the maximum of the m_i. Each constraint C_α is *relevant* to a subset of variables $X_{j_1}, \ldots, X_{j_k}$ where $\{j_1, \ldots, j_k\}$ is some increasing subsequence of $\{1, 2, \ldots, n\}$. It may be regarded as containing all tuples of values over this set of variables that are allowed with respect to C_α. A constraint which is relevant to exactly one variable is called a *unary constraint*. Similarly, a *binary constraint* is relevant to exactly two variables. A solution to the CSP consists of an instantiation of all the variables which does not violate any of the constraints,i.e.,a consistent labeling of each variable with a value from its domain. The simplest algorithm is the brute force algorithm (generate and test), which simply tries every possible combination of values being instantiated to the variable. In Genetic Algorithms the variables will be the genes in our representation of the chromosome, the variables values will be the alleles and the goal is to find an instantiation of the chromosome which does not violate any constraint. A generate and test algorithm, is equivalent, in the worst case, to generate a population size m^n. Obviously in this population we will have at least one chromosome solution (if the CSP has a solution). However, the most CSP are computationally NP-complete, which implies that there are no known polynomial time algorithms which can guarantee finding a solution, owing in part to the size of domains, to the size of variables and to the structure of the constraints network. Traditionally the effort of the research community on constraints has attemped to develop techniques to improve the algorithm performance using the knowledge on the constraints,[3], [11], [7], for example, by pruning the search space. In order to use the knowledge on constraints in genetic algorithms, we concentrate our attention on the constraints network, which we have represented by a constraints matrix. For simplicity we restrict our attention here to binary CSPs, in which the constraints involve two variables.

- **Definition 1:** A *Constraints Matrix* R is an $NC \times n$ rectangular array with elements, which presents the information inherent in a constraints graph. The n columns correspond to variables and the NC rows correspond to constraints. An element in the ith row and jth column will be "1" if the variable j is relevant with respect to constraint i, and it will be "0" if it is irrelevant. Figure 1 presents a constraints matrix and its constraints graph.

An evolutionary algorithm has been created, which is a *genetic algorithm transformed* to suit the CSP, in order to use the knowledge on constraints and vari-

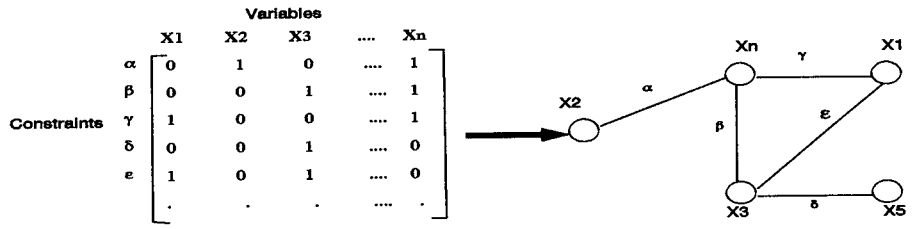


Fig. 1. Constraint Network

ables. Our approached is based on the first evolution programming principle: appropriate data structures should do the job of taking care of constraints, [10].

3 An Evolutionary Algorithm for CSP: FCA

Designing an evolutionary algorithm (EA) involves the following six components:

1. An initial population
2. A genetic representation of chromosomes
3. Genetic operators:crossover, mutation, other specialized operators.
4. An evaluation function
5. A selection algorithm
6. Parameters: population size, probabilities of genetic operators.

The structure of an evolutionary algorithm is:

```
Begin /* procedure Evolutionary Algorithm */
t = 0
initialize population P(t)              (1)
evaluate fitness of indivuals in P(t)   (4)
while (not termination-condition) do
      begin
      t=t+1
     Parents = select-parents-from P(t-1) (5)
      Children = alter Parents          (3)
      P(t) = Children
     evaluate P(t)                      (4)
      end
endwhile
End /* procedure Evolutionary Algorithm */
```

Fig. 2. Structure of an EA and its components

Our research effort has been principally concentrated on a genetic representation of chromosomes, on the evaluation function (or fitness function), on the

selection algorithm, and on the genetic operator crossover, i.e., components 2, 3, 4, 5.

3.1 Initial Population

The initial population is generated randomly (component 1).The variable's values are selected from their domains with an uniform probability distribution.

3.2 Genetic Representation

We have selected a non-binary genetic representation taking into consideration that the variables domains involved in a CSP could be of different types,i.e., a constraint satisfaction problem, in general, has variables with real domains, boolean domains, class domains. Therefore our genetic representation has the structure shown in Figure 3.

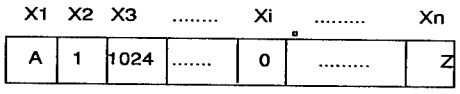


Fig. 3. Chromosome representation

3.3 Genetic Operators

We have worked in our first set of tests with the classical mutation and crossover genetic operators, for the component 3 (alter Parents), [8]. After we have incorporated a "permutation" operator which helps to crossover operator when the population is staying in a local optimum. In the second set of tests we have used alone a good specialized asexual operator (#,r,b), defined by Eiben in [6], for the 3-colouring graph problems to alter the population.

3.4 Evaluation Function

Different fitness functions for the CSP have been defined in the CSP literature, for example, Eiben [5] proposed for the N-Queens problem a minimization of the number of unsatisfied diagonal constraints. For the graph colouring problem [5], [9] proposed to minimize the number of violated constraints. Thorton [14] proposed to minimize $Y = \Sigma d_i^2$ where d_i is the normalized error for the constraint i.

However, few researchers have take into account the constraints network structure. Inspite of that it has been very important in the research of conventional search techniques for CSPs. In a recent research Dozier [4] proposed an evaluation function which determines an individual's fitness by subtracting the weights of

all violated breakouts from the number of constraints satisfied by the candidate solution that the individual represents. To consider only the number of violated constraints, implicitly means that a preference between the constraints doesn't exist, and it doesn't take into account the number of variables in a constraint. Intuitively a first preference may be: *The more important constraints are those that involve more variables.*
In a binary constraint network we will have every constraint with the same preference. In this point we incorporate another concept, that is, *involved variables.*

- **Definition 2:**
  We shall now define the sets $\vartheta_\alpha \in V, \alpha = 1, \ldots, NC$ roughly speaking ϑ_α will contain variables involved in constraints that are unsatisfied. More precisely $X_j \in \vartheta_\alpha$ if and only if one of the following conditions is satisfied:
  - X_j appears in the constraint C_α (i.e.$R[\alpha, j] = 1$), and C_α is not satisfied.
  - For some l, X_l appears in the constraint C_α ($R[\alpha, l] = 1$) and in the constraint C_β ($R[\beta, l] = 1$), X_j appears in the constraint C_β ($R[\beta, j] = 1$), and C_α is not satisfied.

This definition shows that there are variables whose values may involve more than one constraint, more precisely, the effect of change the value of one variable would be reflected in other constraints. It is this effect that we have incorporated in the evaluation function. To define our evaluation function the following definition is necessary:

- **Definition 3:** The following be given:
  R constraint matrix(Def 1), C_α the unsatisfied constraint that contains the variables X_k and X_l ($R[\alpha, k] = R[\alpha, l] = 1$).
  The *Error-evaluation* EC_α *for* C_α, is defined as

$$EC_\alpha = \text{Number of variables in } \vartheta_\alpha \text{ (Def 2), i.e.}$$

$$EC_\alpha = (\text{Number of variables in } C_\alpha) + (\text{Propagation Effect } X_k \text{ and } X_l)$$

  where Propagation Effect X_k and X_l in a binary constraint network, is defined as the number of constraints C_β, $\beta = 1, \ldots, NC, \beta \neq \alpha$ that include either X_k or X_l (i.e. $R[\beta, l] = 1$ or $R[\beta, k] = 1$).
  The Error-evaluation for an unsatisfied constraint C_α can be represented in terms of the constraint matrix R as:

$$EC_\alpha = \left(\sum_{j=1}^{n} R[\alpha, j] \right) + \left(\sum_{\beta \neq \alpha, \beta=1}^{NC} R[\beta, k] + \sum_{\beta \neq \alpha, \beta=1}^{NC} R[\beta, l] \right)$$

  If C_γ is satisfied then we define $EC_\gamma = 0$

In order to make Definition 3 a bit clearer, suppose that we try repairing a chromosome which satisfies all except the constraint C_i. We must modify the values instantiated of X_k and/or X_l, this change of genes has an effect which

will be propagated to other connected variables by constraints in the network to X_k and/or X_l.
Now we extend this definition to n-ary constraints network. The idea is: if we have a chromosome whose alleles unsatisfy a constraint, in order to repair it, in the worst case, we must change both all variables values participating in this constraint, and the variables values connected to them by the others constraints in the network . So, the Propagation Effect uses the same idea that for binary constraints network, that is:

- **Definition 4:**
  The following be given:

  R constraint matrix(Def 1), C_α the unsatisfied constraint that contains the variables X_{k_i}, $i \in [1, \ldots, l]$, $l \leq n$ $(R[\alpha, k_i] = 1 \; \forall i)$.
  The *Error-evaluation* n-ary EC_{α_n} *for* C_α, is defined as

  $$EC_{\alpha_n} = \text{(Number of variables in } C_\alpha) + \text{(Propagation Effect } X_{k_i} \forall i)$$

  where Propagation Effect X_{k_i} in a n-ary constraint network, is defined as the number of variables in the constraint C_β, $\beta = 1, \ldots, NC, \beta \neq \alpha$ which includes X_{k_1} or X_{k_2} .. or X_{k_l} (i.e. $R[\beta, k_1] = 1$ or $R[\beta, k_2] = 1$ ... or $R[\beta, k_l] = 1$).
  The Error-evaluation n-ary for an unsatisfied constraint C_α can be represented in terms of the constraints matrix R as:

  $$EC_{\alpha_n} = \left(\sum_{w=1}^{n} R[\alpha, w]\right) + \left(\sum_{w=1}^{n} R[\alpha, w] \left[\sum_{\beta \neq \alpha, \beta=1}^{NC} R[\beta, w] \left(\sum_{j \neq w, j=1}^{n} R[\beta, j]\right)\right]\right)$$

  If C_γ is satisfied then we define $EC_\gamma = 0$

Finally, our fitness function is the sum of the Error-evaluations of the constraints, that is:

- **Definition 5:**
  Given the constraint matrix R (Def 1),and
  EC_α the Error-evaluation for constraint C_α (Def 3) $\alpha = 1, \ldots, NC$, we define an *Evaluation Function* Z for a binary constraint satisfaction problem as:

  $$Z = \sum_{\alpha=1}^{NC} EC_\alpha$$

The goal of the search is to minimize our evaluation function, which equals zero when all constraints are satisfied.
This Evaluation Function can be viewed in a binary network as a form of quantify our preference for the chromosomes whose values of variables satisfy more arcs and paths.

Note: In the same way for a n-ary CSP: $Z = \sum_{\alpha=1}^{NC} EC_{\alpha_n}$

For instance, suppose us randoms CSP caracterized by four parameters: n, the number of variables, m the number of values in each variable's domain; p_1, the probability that there is a constraint between a pair of variables, and p_2, the conditional probability that a pair of values is inconsistent for a pair of variables, given that there is a constraint between the variables. We can see that our fitness function depends stronger of p_1 than the common fitness function of Number of constraints violated, that is, given NC_v = Number of constraints violated,
the value expected of Z in a random CSP is: $2NC_v p_1(n-1)$. We can see that in the Figure 4 and Figure 5.

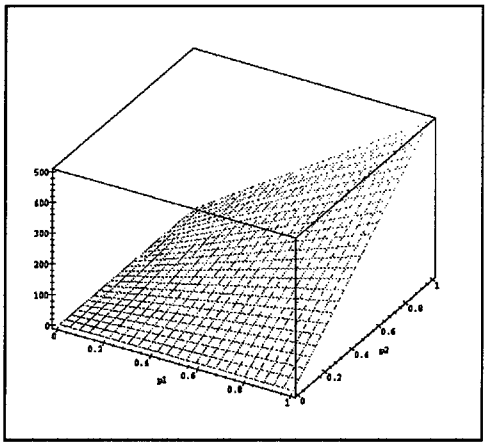


Fig. 4. Common fitness function v/s p_1 and $p2$

3.5 Selection Algorithm

The fitness function allows us to define a new selection algorithm. First of all the best chromosome of a generation is selected for the next generation, elitist approach. We wish to privilege the chromosomes with lower fitness more importantly that the standard selection method of roulette wheel, [8]. However the illegal individuals, in our algorithm, will also have a probability to be selected, because stated legal individuals often require the production of illegal individuals as intermediate structures, [10].

We have designed the selection strategy of Figure 6.

In selecting α and β we compared for different constraints networks the number of generations required to find a solution. The best results have been obtained with $\alpha = 0.5$ and $\beta = 0.85$.

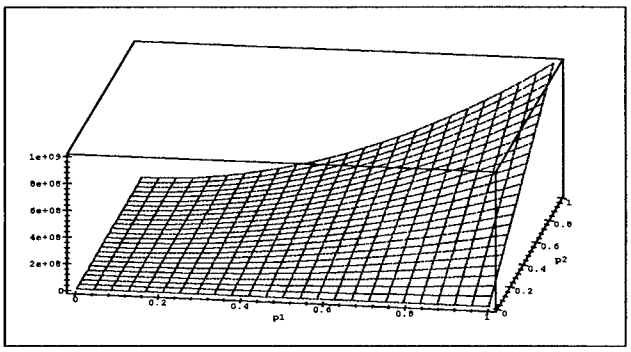


Fig. 5. New fitness function v/s p_1 and $p2$

```
Begin /* Selection Algorithm SA-I */
Choose j from U[0, 1]
if (j < α) then
   Choose chromosome with fitness<= average-fitness
else
   if (j < β) then
   Choose chromosome with fitness< average-fitness+Standard-Desviation
   else
     Choose chromosome from U[1, population_size]
   endif
endif
End /* Selection Algorithm SA-I */
```

Fig. 6. Selection Algorithm

Statistic Properties of SA-I Considering the values of α and β the population has been divided in three regions A,B,C. It is shown in the Figure 7. Suppose that we have a population size N, with n_1 chromosomes in region A, n_2 in region B and n_3 in region C, such $n_1+n_2+n_3 = N$, then we have the following selection probabilities for a chromosome:

- from region $A = \alpha + (\beta - \alpha) * \frac{n_1}{n_1+n_2} + (1 - \beta) * \frac{n_1}{N}$
- from region $B = (\beta - \alpha) * \frac{n_2}{n_1+n_2} + (1 - \beta) * \frac{n_2}{N}$
- from region $C = (1 - \beta) * \frac{n_3}{N}$

Figure 7 shows that we have a preference for the chromosomes in the region A, i.e., we prefer individuals whose fitness function is lower than or equal to average. However,there exists the probability of selecting chromosomes from region C, whose fitness function is greater than average+standard deviation.

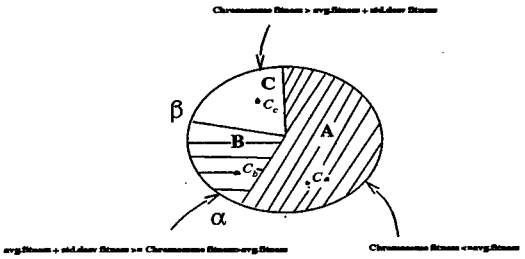


Fig. 7. Selection regions

4 An example: Graph Colouring Problem

In order to ilustrate our evolutionary algorithm suppose that we consider a small graph colouring problem, subject to the restriction that adjacent nodes must be colored differently.
The graph is shown in Figure 8 and it consists of seven variables and eleven constraints. In coloring the graph, we can use the three colors red, white and blue.

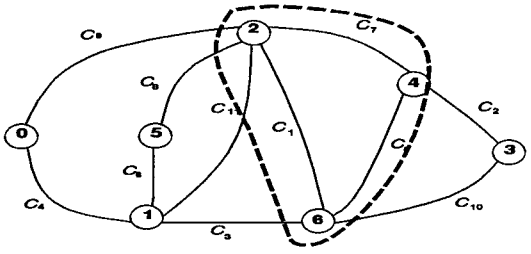


Fig. 8. Graph example

4.1 First Analysis

This graph in particular would be reduced concentrating our attention in the constraints between nodes 2,4,6. by applying the reduction algorithm proposed by Cheeseman [2]. If we have a consistent instantiation for 2,4,6, it is easy to find a value which satisfies all constraints, for the other variables. However, in order to ilustrate our algorithm, the search has been realized with all nodes considering the new fitness function. With our fitness function we are given a preference hierarchy for constraints to be satisfied. For example, the constraint C_1 between $node_2$ and $node_6$ is more important to satisfy than constraint C_{10} between $node_3$ and $node_6$. Both involve two variables. Analysing the network

structure, $node_2$, that is relevant to C_1, is strongly connected (nodes 0,5,1), on the other hand $node_3$, that is relevant to C_{10}, is only connected to $node_4$.

4.2 Results

For this first set of tests we have worked with the classical mutation and crossover genetic operators. This example is almost trivially simple, of course; the thing to note is that the structure of the constraints network is very important to improve the efficiency of the search. Further we have tested for the same problem different orders for the variables in the chromosome. For that different topologies with 7 variables and 11 constraints have been tested. For all of them 30 initial different orders of variables in the chromosome have been generated. The maximum number of generations has been fixed in 100. Figure 9 shows the graph of the performance of FCA. The x-axis represents the variables orders and the y-axis represents the number of generations for each order. We have observed the existence of orders that lead to performance degradation.
This analysis suggests that a specialized operator of the type "permutation" should also help to increase the search performance. Intuitively a permutation operator is justified by the high degree of relation between the variables in the CSP. For example an order could become not good when the constraints are unsatisfied by neighbour variables in the chromosome, because it is difficult to break their union for the crossover operator. To understand precisely that means that, we consider the important concept of a *schema* [8]. A schema is a similarity template describing a subset of chromosomes with similarities at certain chromosome positions. If we have a chromosome with m=4 variables an example of schema could be * 8 2 *, where "*" is a don't care symbol. We are interested in the chromosomes whose alleles are matched by the schema, i.e., in every chromosome with the second variable value is equal to 8, and the third variable value is equal to 2. A schema Sch has the following properties: schema *order* $o(Sch)$ and *defining length* $\delta(Sch)$, where the order of a schema is the number of fixed positions and the *defining length* is the distance between the first and last specific chromosome position. The schema example has $o(Sch) = 2$ and $\delta(Sch) = 3 - 2 = 1$. Suppose us that the constraints are violated by the values 8 and 2, then the chromosomes which are matched by this schema are not wished, because these values will not be contained in any solution. We know in the schema fundamental theorem that the destruction probability in one-point crossover of a schema Sch is: $p_d(Sch) = \frac{\delta(Sch)}{m-1}$. In other words, it is the probability that after of crossover the next generation will not have chromosomes matched by this schema. A permutation operator will permit to modify the value of $\delta(Sch)$ by changing the variable positions in the chromosome, in consequence, the probability of destruction will be also altered.
The permutation operator goal is to increase the crossover potentiality.

Permutation Operator The permutation operator will be only actived when we realize that the order chosen is not good, because if a good order has been

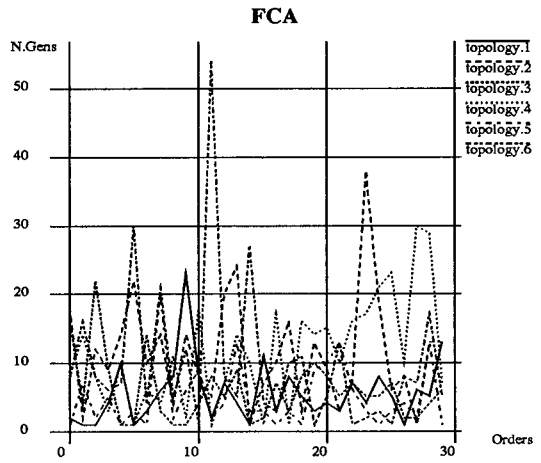


Fig. 9. Network constraint: 7 variables and 11 constraints

chosen the algorithm naturally will converge without additional help. However, if the order chosen is a *bad order*, the algorithm will need a help, that is our new operator, to converge faster to the solution (if that exists). We introduce an other parameter, that is, a permutation probability which basically works with the same idea that the mutation and crossover probabilities. It is the probability that the population will change the order of the variables in the chromosome. The principal difference with the others operator probabilities is that all the population is concerned in, i.e., if we decide to apply the permutation operator it will affect to each member of the population, moreover in the same way. If the permutation operator has not been actived, the permutation probability is zero. In order to identify if an initial order needs to active the permutation operator, we require the following notion of *stability*.

- **Definition 6**:
  An order is experiencing *stability* if the algorithm has found the same *best chromosome* in the last S generations.

Figure 10 shows when our permutation operator could be activated during the evolution process. The structure of permutation procedure is shown in the Figure 11. Once the permutation operator is actived the evolution process continues applying it according to the permutation probability in every new generation. We have included this operator in our algorithm FCA. It has been tested with $S = 25$ for the CSPs analyzed in the section 4. A permutation probability equal to 0.3 has been used. The results show an increasing of search performance at least 20% for the *worst initial orders*, while for the *best initial orders* the performance is the same.

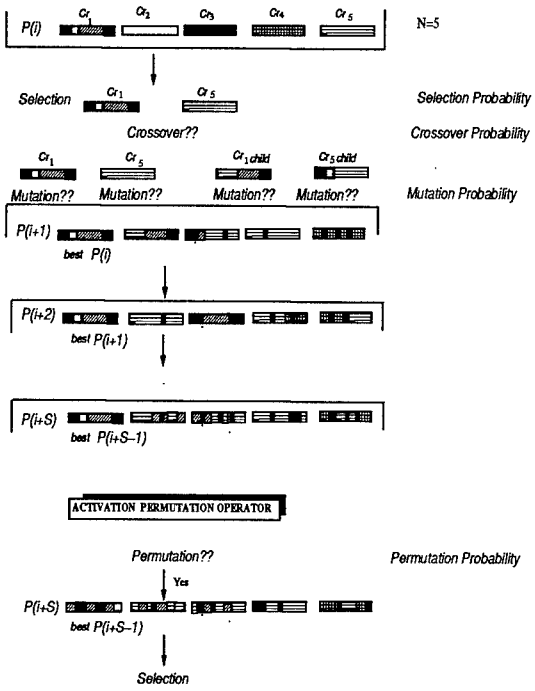


Fig. 10. Activation of the genetic permutation operator

```
Begin /* permutation operator */
if permutation operator is active
   Generate a random number r from the range [0..1]
   if r <probability of permutation
     Generate a random chromosome new order for [1..Number-of-variables-1]
     for all the chromosomes in the population
        Change the place of the variable values according to the new order
     end for
   end if
end if
End /* permutation operator */
```

Fig. 11. Structure of permutation procedure

4.3 Comparison

We have also generated 1000 random CSPs with different topologies with a degree of connectivity between $[4, 6]$ for 30 variables. We have compared three algorithms which differ in fitness functions and in selection algorithms(Figure 12). All of them work with a good specialized asexual operator defined by Eiben in {cf. [6]} as: The heuristic asexual operators are based on the idea of improving

an individual by changing some of its genes. An asexual operator selects a number of positions in the parent, then selects new values for these positions. The number of modified values, the criteria for identifying the position of the values to be modified and the criteria for defining the new values for the child are the defining parameters of the asexual operators. Therefore an asexual operator is denoted by the triple (n,p,g) where n indicates the number of modified values, and the values for p and g are chosen from the set {r,b}, where r indicates uniform random selection and b indicates some heuristic-based biased selection. For this kind of problems the best parameters for this operator were (n,p,g)=(#,r,b) where # meaning that the number of values to be modified is chosen randomly but is at most 1/4 of all positions.

Algorithm	Fitness Function	Selection algorithm
Common_ff	Number of constraints violated	Roulette wheel
New_ff	Fitness of Definition 5	Roulette wheel
New_ff + new_select	Fitness of Definition 5	SA-I

Fig. 12. Three algorithms: Common_ff, New_ff and New_ff + new_select

For each connectivity we have generated 100 different random graphs. Figure 13 shows the percentage of solutions found by the three algorithms. The new fitness function(New_ff) with SA-I give the best results. It has found in the worst case a 70% of solutions, in contrast to Common_ff which in the worst case found 20% of solutions. Figure 14 shows the average generations for each graph connectivity. The number of average generations for Common_ff is greater than for the other algorithms. Furthermore we can observe that the New_ff is best when it uses the selection algorithm SA-I.

5 Conclusion

A new evolutionary algorithm has been presented which repares pre-solutions to find a solution of a CSP. It takes into account the structure of constraints network, in order to define a better evaluation function for CSP. This evaluation function has been used to construct a selection algorithm that strongly privileges the better individuals, however it doesn't avoid the production of illegal individuals. Our research allows us to conclude that the structure of a constraints network is very important to guide the search. Furthermore the order of nodes in the chromosome for the problems with high degree of interaction (epistasis),such as the CSPs, can lead to degradation of performance when the algorithm uses a crossover operator to alter the population. This CSP caracteristic has been also considered to design a new operator "permutation", which is actived when the initial order of variables is not good. It has permitted to reduce in average 20% the generations number for the worst orders.

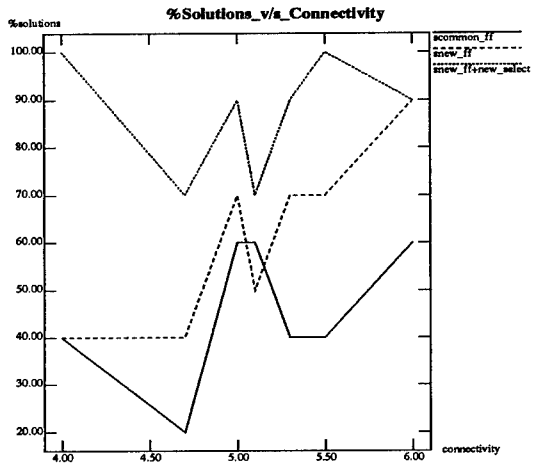


Fig. 13. Percentage of solutions found by Common fitness,New fitness and New fitness with new selection algorithm for different connectivities

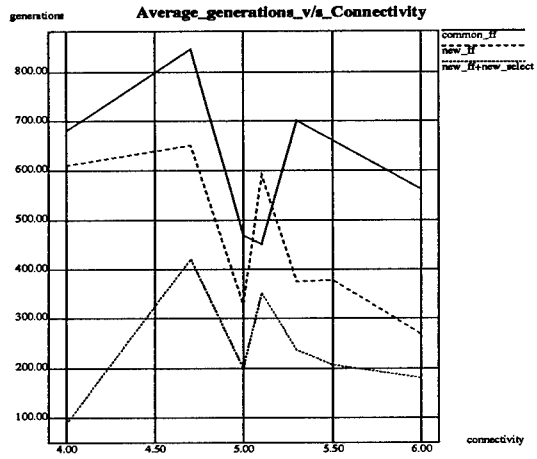


Fig. 14. Comparison: Common fitness, New fitness and New fitness with new selection algorithm for different connectivities

There are a variety of ways in which the techniques that we have presented can be extended. The principal advantage of our method is that it is general, i.e., the approach to estimate the evaluation function is not related to a particular CSP. Now our research is directed towards defining better genetic operators which consider the structure of the constraints network.

Acknowledgements

I wish to gratefully acknowledge the discussions of B.Neveu

References

[1] Bowen James, Gerry Dozier, Solving Constraint Satisfaction Problems Using A Genetic/Systematic Search Hybrid That Realizes When to Quit Proceedings of the Sixth International Conference on Genetic Algorithms pp. 122-129, 1995.

[2] Cheeseman Peter, Bob Kanefsky, William Taylor, Where the Really Hard Problems Are Proc. of IJCAI-91, pp. 163-169, 1991

[3] Dechter Rina, Enhancement schemes for constraint processing: backjumping, leaarning, and cutset decomposition. Artificial Intelligence 41, pp. 273-312, 1990.

[4] Dozier Gerry, James Bowen, Dennis Bahler, Solving Small and Large Scale Constraint Satisfaction Problems Using a Heuristic-Based Microgenetic Algorithm Proc. of the First IEEE Conf on Evolutionary Computation, Orlando, pp 306-311, 1994.

[5] Eiben A.E., P-E Raué, Zs. Ruttkay, Solving Constraint Satisfaction Problems Using Genetic Algorithms Proc. of the First IEEE Conf on Evolutionary Computation, Orlando, pp 542-547, 1994.

[6] Eiben Ágoston, Paul-Erik Raué, Zsófia Ruttkay, GA-easy and GA-hard Constraint Satisfaction Problems Constraint Processing, Ed. Manfred Meyer, pp. 267-283, 1995.

[7] Freuder Eugene. C, A sufficient condition of backtrack-free search J. ACM. 29, pp. 24-32, 1982.

[8] Goldberg D.E., Genetic Algorithms in Search, Optimization and Machine Learning Ed. Addison-Wesley, 1989.

[9] Michalewicz Zbigniew, Cezary Janikow, Handling Constraints in Genetic Algorithms Proc. of 4th Conference on GA, Morgan Kaufmann Publishers Los Altos, CA, pp 151-157, 1991.

[10] Michalewicz Zbigniew, Genetic Algorithms + Data Structures = Evolution Programs Ed. Springer-Verlag, Artifial Intelligence Series, 1994.

[11] Montanari U, Networks of constraints: Fundamental properties and applications to picture processing. Inf. Sci. 7, pp. 95-132, 1974.

[12] Paredis Jan, Co-evolutionary Constraint Satisfaction Proceedings PPSN III, Int. Conference on Evolutionary Computation Israel, pp. 46-55, Oct. 1994.

[13] Riff María-Cristina, Improving fitness function for CSP in an Evolutionary Algorithm. Rapport de Recherche CERMICS, Dec. 1995.

[14] Thorton A.C., Genetic Algorithms versus Simulated Annealing: Satisfaction of Large Sets of Algebraic Mechanical Design Constraints Artifial Intelligence in Design, pp 381-398, 1994.

[15] Tsang Edward, Applying Genetic Algorithms to Constraint Satisfaction Optimization Problems Proc. of ECAI-90, Pitman Publishing, pp 649-654 , 1990

Existential Variables and Local Consistency in Finite Domain Constraint Problems

Francesca Rossi

Computer Science Department
University of Pisa
Corso Italia 40
56125 Pisa, Italy
E-mail: rossi@di.unipi.it

Abstract. In this paper we introduce the concept of *existential variables* in finite domain constraint problems. A variable is existential if, once instantiated the other variables, one can always find a value for it such that all constraints are satisfied. In other words, existential variables (and the constraints they are connected to) do not add any information to the rest of the constraint problem. We use the notion of existential variable to achieve what we call *incremental k-consistency*, which means that different levels of (strong) consistency are obtained on different subparts of the problem, but all lower than or equal to k. At the end, the overall problem can be solved by a backtrack-bounded search, and the complexity of the search will be as if (or smaller) the CSP were k-consistent everywhere. We also consider (1,k)-consistency, which is a form of local consistency which is more powerful than arc-consistency while still removing just domain elements (and thus never adding any new constraint), and we discuss how to get a similar algorithm also for this form of consistency.

1 Introduction

Finite domain constraint problems (CSPs) [Mon74, Mac92] are a very powerful knowledge representation formalism. In fact, many real-life situations can be easily cast as CSPs. However, sometimes it is not easy to model a real-life problem as a CSP, and one of the results is that some level of redundancy is introduced during the modelling phase. This can happen, for example, while defining a constraint, and specifying too many tuples as allowed by it, or also while using variables and constraints that in reality do not add any information to the rest of the problem. The first kind of redundancy, which may be called *constraint redundancy*, can be dealt with by the local consistency algorithms [Mon74, Mac77, MF85, DP88, Dec92], whose task is exactly that of identifying, and removing, some tuples in some constraints which are not allowed by the rest of the problem. The second type of redundancy can be called *variable redundancy*, and it is the issue we consider in this paper. That is, we are interested in understanding how we can recognize (at least some of) these redundant vari-

ables (which we call *existential variables*), and how we can use this information to solve CSPs in a more efficient way.

More precisely, a variable is existential if, once instantiated (some or all of) the other variables, one can always find a value for it such that all constraints are satisfied. This means that any solution of a subpart of the CSP (the part without existential variables) can always be extended to get a solution of the entire CSP without too much effort. In fact, the extension will only need the time to try, for each of the existential variables, at most all the values in their domains. Thus it is linear in the number of such variables, with a multiplicative factor which includes the size of their domain and their degree.

We consider a sufficient condition for a variable to be existential, and we exploit it to develop more efficient local consistency algorithms. More precisely, if a variable is connected to a number of constraints which is smaller than the level of consistency of the problem, then it is existential. In fact, by definition of k-consistency [Fre78, Fre88], once $k-1$ variables are instantiated in a compatible way, any k-th variable can also be instantiated compatibly to them.

We also observe that the subCSP not containing the variable discovered as existential, say P, is still strong k-consistent if the entire CSP was so. This makes it possible to recognize as existential also variables that do not have a degree smaller than the level of local consistency if looking at the entire problem, but do have it if looking at the subCSP P. In this way, more and more variables can be defined as existential, untill all remaining variables have a degree greater than or equal to the level of consistency.

Therefore, if a given CSP is k-consistent, then a certain set S of variables will be found to be existential. This set can be obtained by considering all variables of degree less than k, then those variables which get a degree with this property after removing the first ones, and so on. Now, if we want to find a solution of the entire CSP, it is enough to first find an instantiation of the variables in $V - S$ (if V is the set of all variables) which satisfies all constraints among them, and then an instantiation of the existential variables which satisfies all constraints. The first phase can be exponential in the cardinality of $V - S$, while the second phase is linear in the cardinality of S.

However, if the CSP is not k-consistent, to get the same complexity we must make it k-consistent, which in general takes a time polynomial in the number of variables of the CSP (and exponential in k). Nevertheless, since not all existential variables really need k-consistency to be recognized as existential, one could achieve lower levels of consistency on some parts of the CSP while maintaining the same search complexity. In this way, we develop an algorithm which does not achieve k-consistency over the whole CSP, but only on those parts where it is needed. The kind of local consistency achieved is called *incremental k-consistency*. The resulting set of existential variables however is greater than or equal to that found after a standard k-consistency algorithm.

We also consider weaker sufficient conditions for existentiality. However, they cannot be used to develop more efficient algorithms to obtain the same set of existential variables, mainly due to the fact that such weaker notions rely on

local properties instead of global ones.

We also consider (1,k)-consistency [Fre88], which is a form of local consistency which is more powerful than arc-consistency [Mac77] while still removing just domain elements (and thus never adding any new constraint), and we give some ideas on how to develop an algorithm to achieve this form of consistency which follows the lines of incremental k-consistency.

The paper is organized as follows. Section 2 relates our work to the results already presented in the literature. Then, Section 3 provides the necessary notions about CSPs, graphs, and local consistency that will be needed in the paper, and Section 4 defines the concept of existential variables. Section 5 proposes the algorithm which achieves incremental k-consistency and yields the same search complexity as full k-consistency. Then, Section 6 discusses a similar algorithm for (1,k)-consistency. Finally, Section 7 summarizes the results of the paper and discusses topics for future work.

2 Related Work

The concept of existential variables is close to that of *redundant hidden variables* [Ros95]. In CSPs with hidden variables, the solution concerns only a subset of the variables, the visible one. All other variables are hidden. Then, some of the hidden variables can be found to be redundant, in the sense that their removal, together with the removal of the constraints connecting them, does not change the solution set of the CSP. Sufficient conditions similar to those considered in this paper also hold for redundant hidden variables, as well as similar algorithms to find them. However, in a CSP with hidden variables, this concept has not been used to make the preprocessing phase more efficient.

The concept of width, and the fact that a CSP whose level of consistency is greater than its width has a backtrack-free search [Fre88], are also very related to our work. In fact, the subCSP spanned by the existential variables can be proved to have a width smaller than the level of local consistency. However, our analysis is at a more local level, since we consider some variables at a time, and not the whole CSP.

The adaptive consistency algorithm [DP88] is based on the observation that, to get a backtrack-free search over an ordering of the variables, one needs to consider each variable, from the last one in the ordering to the first one, and achieve directional k-consistency over the subCSP spanned by this variable and its neighbors before it in the ordering, if they are $k-1$. Our algorithm instead does not obtain a backtrack-free search, but a backtrack-bounded one [Fre88], where the bound is given by the number of non-existential variables. Thus it can be seen as an adaptive consistency algorithm where however a limit is put on the amount of preprocessing (no more than k-consistency).

If a tuple $\langle d_1, \ldots, d_n \rangle$ is in the solution of a CSP whose variables are $v_1, \ldots, v_n$, then sometimes we will write it as the set of equations $\{v_1 = d_1, \ldots, v_n = d_n\}$.

The structure of a CSP can be easily pictured as a (hyper)graph, which is usually called a *constraint graph* [DP88], where nodes represent variables and hyperarcs represent constraints. The representation of a CSP by a hypergraph is very convenient, because many notions typical of (hyper)graphs can be used in the CSP context. For example, in this paper we will use the concept of *degree* of a node (variable), which is the number of arcs (constraints) incidents in that node (variable), and of CSP *spanned* by a certain set of variables.

Definition 3 (graph and related). A graph is a triple $G = \langle N, A, f \rangle$, where N is the set of nodes, A is the set of arcs, and function $f : A \rightarrow \bigcup_k N^k$ specifies which nodes are connected to which arc. In a graph $G = \langle N, A, f \rangle$, consider a node $n \in N$. The degree of n, written degree(n), is defined as the cardinality of the set $\{n' \in N$ such that $\exists a \in A$ with $\langle n, n' \rangle \in f(a)^2\}$. Also, given a subset of the nodes $N' \subseteq N$, we define the graph spanned by N' as $G(N') = \langle N', A', f' \rangle$, where $A' = \{a \in A$ such that $f(a) \subseteq N'\}$ and $f' = f_{|A'}$. Given a CSP $P = \langle V, D, C, con, def \rangle$, its constraint graph is $G(P) = \langle V, C, con \rangle$. □

3.2 Local Consistency

Local consistency algorithms remove from a CSP some domain elements or also some tuples from constraint definitions if these objects are found to be inconsistent with some other object in the CSP. This is safe (that is, it does not change the set of solutions of the CSP), because local inconsistency implies global inconsistency, and thus such objects would never appear in any solution of the CSP. However, there may be objects (tuples and/or domain elements) which are inconsistent but are not recognized as such and therefore are not removed. Thus in general only *local* consistency is achieved (and not global consistency, which would mean that the problem is solved).

The first local consistency algorithms have been called arc-consistency [Mac77] and path-consistency [Mon74]. Later, both of them were generalized to the concept of k-consistency [Fre88]: a CSP is k-consistent if, for each k-1 variables, and values for them which are allowed by all the constraints involving subsets of them, and for each other variable, there is at least a value locally allowed for such k-th variable which is compatible with the values chosen for all the other k-1 variables. In this line, arc-consistency [Mac77] is just 2-consistency and path-consistency [Mon74] is 3-consistency. Formally, k-consistency can be defined as follows:

Definition 4 ((strong) k-consistency [Fre88]). A CSP $\langle V, D, C, con, def \rangle$ is said to be k-consistent if, for all k-1 variables $v_1, \ldots, v_{k-1} \in V$, values $d_1, \ldots, d_{k-1}$
$\in D$, and k-th variable $v_k \in V$, there is at least a value $d_k \in D$ such that,

[2] Here we mean that n and n' appear in a tuple which belongs to $f(a)$.

3 Background

Here we will give the basic notions on finite domain constraint problems, k-consistency, graph structure, and search, that will be useful in the following of the paper.

3.1 Finite Domain Constraint Problems

A (finite domain) constraint problem (CSP) consists of a set of variables ranging over a finite domain, and a set of constraints among such variables. Each constraint involves a subset of the variables and specifies a set of tuples of values, for such variables, which satisfy the constraint. A solution of a CSP is an instantiation of all the variables such that all the constraints are satisfied.

Definition 1 (constraint satisfaction problem). A (finite domain) constraint satisfaction problem (CSP) is a tuple $\langle V, D, C, con, def \rangle$ where

- V is a finite set of *variables* (i.e., $V = \{v_1, \ldots, v_n\}$);
- D is a finite set of values, called the *domain*;
- C is a finite set of *constraints* (i.e., $C = \{c_1, \ldots, c_m\}$); C is ranked, i.e. $C = \bigcup_k C_k$, such that $c \in C_k$ if c involves k variables;
- *con* is called the *connection function* and it is such that $con : \bigcup_k (C_k \rightarrow V^k)$, where $con(c) = \langle v_1, \ldots, v_k \rangle$ is the tuple of variables involved in $c \in C_k$;
- def is called the *definition function* and it is such that $def : \bigcup_k (C_k \rightarrow \wp(D^k))$.

Given a CSP $P = \langle V, D, C, con, def \rangle$, consider a subset $V' \subseteq V$. Then the subCSP spanned by V' is $CSP(P, V') = \langle V', D, C', con', def' \rangle$, where C' is the set of constraints involving only variables in V', and con' and def' are the restrictions of *con* and def to C'. □

Function *con* describes which variables are involved in which constraint, while function def gives the meaning of a constraint in terms of a set of tuples of domain elements, which represent the allowed combinations of values for the involved variables. Then, the solution $Sol(P)$ of a CSP $P = \langle V, D, C, con, def \rangle$ is defined as the set of all instantiations of the variables in V (seen as tuples of values) which are consistent with all the constraints in C.

Definition 2 (CSP solution). The solution $Sol(P)$ of a CSP $P = \langle V, D, C, con, def \rangle$ is defined as the set $\{\langle v_1, \ldots, v_n \rangle\}$[1] such that

- $v_i \in D$ for all i;
- $\forall c \in C$, $\langle v_1, \ldots, v_n \rangle_{|con(c)} \in def(c)$. □

[1] Here we assume to have given an order to the variables in V.

if $v_i = d_i$ for all $i = 1, \ldots, k-1$ belongs to the solution of the CSP $\langle VK1 = \{v_1, \ldots, v_{k-1}\}, D, C_{|VK1}, con_{|CK1}, def_{|CK1}, \rangle$, then $v_i = d_i$ for all $i = 1, \ldots, k$ belongs to the solution of the CSP $\langle VK = \{v_1, \ldots, v_k\}, D, C_{|VK}, con_{|CK}, def_{|CK} \rangle$. A CSP is said to be *strong k-consistent* whenever it is j-consistent for all j $\leq$ k. □

There are many ways to achieve k-consistency (for example, many algorithms have been proposed for achieving arc- and path-consistency). However, it has been proved that in the worst case a k-consistency algorithm is $O(n^k)$ [Fre88], where n is the number of variables of the given CSP. Thus, if k is much smaller than n, such algorithms are polynomial. Also, it is important to recall that achieving strong k-consistency may involve adding constraints with at most k-1 variables [Fre78].

Since CSPs are NP-hard problems, they are usually solved via a backtracking search, where partial assignments are extended by one variable at a time while checking the satisfiability of the subset of constraints connecting the already assigned variables. Whenever the new variable cannot be assigned to any value compatibly to the constraints, the assignment of the latest variable is backtracked and another value is tried. This search is exponential in the worst case. It can however be improved if some search branches are cut in advance, which is usually done by using a local consistency algorithm, whose aim is to make constraints stronger while not changing the set of solutions, or, in other words, to remove redundant tuples from the constraint definitions. In fact, if a tuple is removed, then some failure branches are cut from the search tree. In other words, the role of the local consistency algorithms is to obtain a gain in the process of finding a solution via a backtrack search (that is, to eliminate some of the trashing behaviour of such search [Mac77]).

Sometimes, the pruning achieved by the local consistency algorithm is so much that the subsequent search does not need any backtracking to find a solution of the given problem. For example, it has been recognized that the sparseness of the constraint graph and the level of consistency of the CSP have a strong relationship with the fact that a search-based solution algorithm could solve the CSP without backtracking at all or with a bounded amount of backtracking [Fre88]. More precisely, if a CSP is strong k-consistent and has "width" less than k, then it is backtrack-free. Here, the width is a notion related to how sparse the CSP is: the sparser it is, the smaller its width is.

Definition 5 (width [Fre88]). Given a CSP and its constraint graph, consider the ordered graph obtained by putting an order on the nodes of the constraint graph. Then

- the width of a node in an ordering O is the number of arcs that connect such node to nodes previous to it in the order O: if $O = v_1, \ldots, v_n$, then the width of v_k in O is the number of arcs a such that $\{v_i, v_k\} \subseteq con(a)$ and $i \in \{1, \ldots, k-1\}$.
- the width of an ordering is the maximum of the widths of the nodes in that ordering.

– the width of a constraint graph is the minimum of the widths of all the orderings on that graph. □

Theorem 6 (width, k-consistency, backtrack-free search [Fre88]). *Given a CSP, there exists a backtrack-free search if the level of strong consistency of the CSP is greater than the width of its constraint graph.* □

4 Existential Variables

In a given CSP, an existential variable is a variable whose removal, together with that of all the constraints involving it, does not change the set of solutions of the CSP spanned by all variables but them. More precisely:

Definition 7 (existential variables). Given a CSP $P = \langle V, D, C, con, def \rangle$ and a variable $x \in V$, x is an existential variable for a P if and only if $Sol(P)_{|_{V-\{x\}}} = Sol(CSP(P, V - \{x\}))$. □

Consider for example the CSP in Figure 1. This CSP has three variables, x, y, and z, and three binary constraints. The removal of variable x, together with all the constraints involving it, does not change the solution of the subCSP spanned by y and z, which is $\{y = c, z = b\}$. Thus x is existential for this CSP[3].

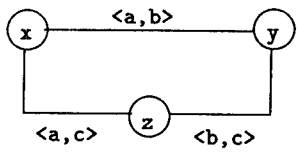


Fig. 1. Existential variables.

Existential variables are very important in CSPs, since they allow us to reduce the task of solving a given CSP to the task of solving an equivalent but smaller CSP. In fact, the following theorem can be proved.

Theorem 8. *Consider a CSP $P = \langle V, D, C, con, def \rangle$, an existential variable x for P, and the CSP $P' = CSP(P, V - \{x\})$. Then, taken any tuple in the solution of P', there is a way to extend it to a tuple in the solution of P.*

Proof: Were it not possible to extend the tuple, it would contradict the assumption that x is an existential variable (which means that $Sol(P)_{|_{V-\{x\}}} = Sol(P')$). □

[3] The same is also for y and z, as we will see later.

The above theorem can be extended to the case of more than one existential variable. That is, if we have a set S of variables each found individually to be existential, then the solution of the entire CSP can be found by just solving the CSP not containing any of the variables in S.

Theorem 9. *Consider a CSP $P = \langle V, D, C, con, def \rangle$, a set S of existential variables for P, and the CSP $P' = CSP(P, V - S)$. Then, taken any tuple in the solution of P', there is a way to extend it to a tuple in the solution of P.*

Proof: If one can extend a longer tuple to an additional variable, then *a fotiori* it is possible to extend a shorter tuple to the same variable, since a smaller number of constraints have to be satisfied. □

In other words, if we assume to solve CSPs via a backtracking search process, existential variables may be left at the end of the instantiation process, since we can be sure that, when their time will come up to be considered, it will be possible to find an instantiation for them compatible with the already assigned values. Note that postponing a variable means in practice that we don't have to worry about that variable any more. Thus we can just concentrate on the solution of the remaining problem, that is, the CSP spanned by V minus S, if S is the set of existential variables. In fact, once obtained in some way a solution of such CSP, we can be sure that a solution of the given problem can be found without too much additional effort (just linear in the number of existential variables). This means that we have reduced the problem of solving the given CSP to the problem of solving an equivalent CSP with a smaller search space.

Consider now a CSP which is strong k-consistent, and consider any variable, say x, with degree smaller than k. Then it is easy to see that, if all variables except x have been instantiated, then there exists an instantiation of x such that all constraints are satisfied. Actually, since the CSP is k-consistent, then instantiating only the variables connected to x (and not all variables but x), which are less than k by assumption, is enough to assure us that x has a possible instantiation. Thus x is an existential variable. Moreover, consider the CSP spanned by all variables but x. Then such CSP is still strong k-consistent, so again all its variables of degree smaller than k are existential variables. And so on until all variables have degree greater or equal to k. Thus the discover of an existential variables may lead to that of other existential variables which may not appear so at the beginning.

Theorem 10 (strong k-consistency and existential variables). *Consider a strong k-consistent CSP $P = \langle V, D, C, con, def \rangle$, and any of its variables, say $v \in V$, such that degree(v) $\leq k - 1$. Then v is existential for P.*

Proof: To show that v is existential for P we have to show that the solution of the CSP is not changed by the removal of v. Therefore we consider $Sol(P')$, where $P' = \langle V' = V - \{v\}, D, C' = C - \{c \mid v \in con(c)\}, con_{|C'}, def_{|C'} \rangle$. Obviously $Sol(P) \subseteq Sol(P')$, since the removal of some constraints can only enlarge the set of allowed tuples for the remaining variables. Consider now any

tuple $t' \in Sol(P')$. This tuple involves all involving all variables in $V - \{v\}$. Let us now see if t' can be extended to v while satisfying all the constraints in P. This is indeed so, since P is strong k-consistent, and degree(v) $\leq k-1$. In fact, consider the projection of t', say tp, to the variables connected to v, say $v_1, \ldots, v_i$, with $i \leq k-1$. Then, by strong k-consistency, tp can be extended to v while satisfying all constraints involving $v, v_1, \ldots, v_i$, yielding tuple tv. Therefore tv is an extension of t' which satisfies all constraints in P: that is, t is in the solution of P. Thus, $Sol(P') \subseteq Sol(P)$. As a result, $Sol(P) = Sol(P')$. Therefore v is existential for P. □

A nice application of Theorem 10 is the problem of coloring a graph with k colors. In fact, it can be proved that such a problem is k-consistent [vBD94], and thus any variable with degree $k-1$ or less is existential.

Theorem 11 (existential variables and strong k-consistency). *Consider a strong k-consistent CSP $P = \langle V, D, C, con, def \rangle$, and any of its variables, say $v \in V$, such that degree(v) $\leq k-1$. Consider also the problem $P' = CSP(V - \{v\})$. Then P' is strong k-consistent as well.*

Proof: Since $v' \subseteq V$, all variables of P' are also in P, and all constraints of P' are also in P. Therefore all the properties of subsets of such variables and constraints, which hold in P, will *a fortiori* hold also in P'. □

An algorithm that postpones variables according to the results of Theorem 10 and 11, called Algorithm 1, can be seen in Table 1. This algorithm takes a strong-k-consistent CSP and returns a partial ordering O of postponed variables. This ordering is partial because no ordering is given among the variables postponed at the same stage. Thus it is represented as an ordered list of sets. If the returned ordering is $O = S_i, S_{i-1}, \ldots, S_1$, it means that S_1 is the firstly discovered set of existential variables (which thus are the last ones in the ordering).

Algorithm 1:
Input: a strong k-consistent CSP $P = \langle V, D, C, con, def \rangle$.
Output: an ordering O.

$O := \emptyset$;
1. $V' = \{v \in V$ **such that degree**$(v) < k\}$, $P' = CSP(P, V - V')$;
2. **If** $P \neq P'$ **then** $P := P'$, $O := V'.O$, **goto 1**

Table 1. Algorithm 1

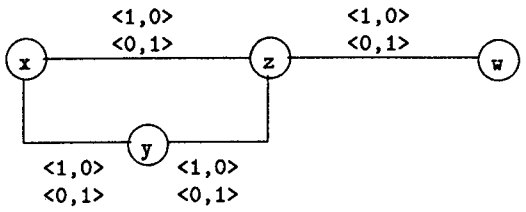


Fig. 2. Existentiality and variable degree.

As an example of the application of algorithm 1, consider the CSP in Figure 2. Assuming that the domain of each variable contains (or it is represented by a unary constraint containing) the values 0 and 1, this CSP is 2-consistent but not 3-consistent, since there is no way to extend any instantiation of any two variables among x, y and z to the third variable. For this CSP, the algorithm postpones only variable w, which has degree 1, since no other variable gets a degree smaller than 2 after w is removed. Consider now the CSP in Figure 1, which is 3-consistent. Here all variables have degree less than 3, thus the algorithm would postpone all of them, meaning that the whole CSP has a backtrack-free search. Finally, consider the CSP in Figure 3 (here unary constraints are denoted by arrows pointing to the involved variable). This problem is not 3-consistent, but it is 2-consistent. In the first iteration the algorithm postpones only variable w, since this is the only variable with degree 1. Then, it removes also v, since v gets degree 1 after the first pass of the algorithm.

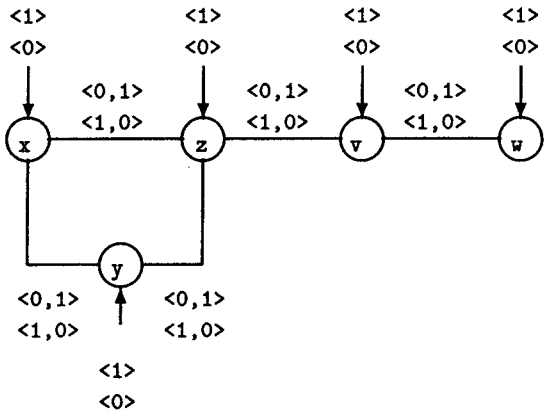


Fig. 3. A CSP with two existential variables: w and v.

Now, it is possible to prove that the CSP spanned by the variables we postpone according to algorithm 1 in a strong k-consistent CSP has width less than

k. Therefore, our result could be seen as an application of the result in [Fre88] to (dinamically chosen) subCSPs instead of entire CSPs.

Theorem 12 (redundancy and width). *Given a strong k-consistent CSP P, consider the CSP P' which is spanned by the variables postponed by P according to algorithm 1. Then, P' has width less than k.*

Proof: To show that P' has width less than k it is enough to find an ordering with width less than k, that is, where all the nodes are connected to at most $k-1$ previous nodes. Consider then the ordering O which is returned by algorithm 1. In O, each node is connected to at most $k-1$ nodes which are later in the ordering, otherwise it could not be postponed by the algorithm. Therefore, the reverse of ordering O has the desired feature. □

5 K-consistency and Existential Variables

From the previous section we know that, if a CSP is k-consistent, then algorithm 1 may find some existential variables, by iterating the process of considering those variables with degree smaller than k. Then, the problem of solving the entire CSP will be reduced to the problem of solving the CSP spanned by the remaining variables (those not recognized as existential). Thus a backtrack search will backtrack only over these variables. That is, if V is the set of all variables, V' the set of variables found to be existential, and D the variable domain, then the worst-case time complexity of such search will be $O(\mid D \mid^{\mid V-V' \mid} + (\mid D \mid \times \mid V' \mid))$. In fact, once an instantiation for the variables in $V - V'$ has been found (and this may take exponential time), we are sure that the variables in V' can be compatibly instantiated without backtracking (and thus in linear time).

However, if the CSP is not already k-consistent, then k-consistency has to be achieved, with a complexity which is in general $O(\mid V \mid^k)$. Here we propose a more convenient way to achieve the same search complexity (that is, exponential in $\mid V - V' \mid$ and linear in $\mid V' \mid$), or a smaller one.

The convenience lies in the fact that we do not achieve (strong) k-consistency on the whole problem, but just on a subpart of it. This is allowed by the exploitation of the concept of existential variables. In fact, it is possible to observe that not all variables recognized as existential by algorithm 1 need k-consistency to be so. Consider for example any variable with degree much smaller than k, say i. Then such variable would be existential also in a $(i+1)$-consistent CSP. Thus obtaining k-consistency is too much for variables like this one. It would be enough to achieve $(i+1)$-consistency.

This observation immediately leads to the algorithm in Table 2, which basically achieves only the necessary level of consistency in the parts of the CSP where this is needed. Thus, at the end, different parts of the CSP will have different levels of consistency, depending on the structure of the graph. More precisely, the sparser the subCSP will be, the smaller its level of consistency will result.

Algorithm 2:
Input: a CSP $P = \langle V, D, C, con, def \rangle$.
Output: a partial ordering O.

$O := \emptyset$;
for $j = 1$ **to** k **do**
1. **achieve j-consistency on CSP(P,V);**
2. $V' = \{v \in V$ **such that degree**$(v) < j\}$;
3. **if** $V' \neq \emptyset$ **then** $V := V - V'$, $O := O \cup V'$, **goto 2**
endfor

Table 2. Algorithm 2

In reality, the statement of Theorem 10 could be weakened. In fact, consider a variable x which has degree i, and consider the i other variables it is connected to, say $V_x = \{x_1, \ldots, x_i\}$. Then, achieve $(i + 1)$-consistency on the sub-CSP spanned by $V_x \cup \{x\}$. At this point it is possible to see that the removal of x does not change the solution of the remaining problem. That is, x is existential.

Theorem 13. *Consider a CSP $P = \langle V, D, C, con, def \rangle$, and any of its variables, say $x \in V$. Consider also the set of variables connected to x via some constraints, say V_x, and the problem $P' = CSP(P, V_x \cup \{x\})$. If P' is k-consistent and the degree of x is smaller than k, then x is existential for P.*

Proof: By definition of k consistency, once the variables in V_x have an instantiation, it is possible to find also a compatible instantiation for x. □

An even weaker condition can be obtained by considering that existential variables are put at the end of the search ordering. Thus directional k-consistency [DP88] is enough to make a variable with degree less than k existential. In fact, it is enough to assure that, once the variables in V_x have been instantiated, x can be instantiated as well. This is the same observation that lead to the definition of the concept of adaptive consistency [DP88].

However, both these weaker sufficient conditions do not lead to any improvement in algorithm 2, since this algorithm looks for existential variables only after achieving j-consistency. Thus we cannot recognize the suparts where to achieve j-consistency, because we don't know where we will find the existential variables. This is due to the fact that achieving j-consistency, as noted above, may add new constraints to the problem, thus modifying the degree of some variable (and thus, possibly, its existentiality).

Algorithm 2 does not make the CSP k-consistent, of course, because k-consistency is achieved only on the part of the CSP where all variables have degree k or larger. However, the complexity of the search process is the same (or

less) as if we achieved k-consistency on the whole problem. To prove that the complexity of the search remains the same or decreases, we just need to show that the variables recognized as existential by the two methods (k-consistency + algorithm 1, or algorithm 2) are the same, or that those postponed by the second method are a superset of those postponed by the first one.

Theorem 14. *Consider a CSP P, and the CSP P_1 obtained by applying a k-consistency algorithm to P. Consider also the order O_1 returned by algorithm 1 on P_1. Then, consider the partial order O_2 returned by applying algorithm 2 to P. Let $O_1 = S_i S_{i-1} \ldots S_1$ and $O_2 = I_j I_{j-1} \ldots I_1$. Then we have that $\bigcup_{l=1,\ldots,i} S_l \subseteq \bigcup_{l=1,\ldots,j} I_l$.*

Proof: Assume a variable is postponed by algorithm 1 (applied after a k-consistency algorithm). It means that such variable has degree $< k$, say j. Consider now the same variable during algorithm 2. Since obtaining any level of consistency lower than k may not add more constraints then obtaining k consistency, the degree of such variable would be smaller than or equal to j. Thus, at iteration j of algorithm 2 at the latest, such variable will be postponed.

Consider now a variable which has degree j in P, with $j < k$. Then it is postponed at some iteration of algorithm 2. Consider now the same variable after applying a k-consistency algorithm to P. Since achieving k-consistency may add constraints of arity $k-1$, this variable may gain $k-1$ new neighbors, thus getting a degree $j+k-1$, which is greater than k is j is greater than 1. Thus such variable is not postponed by algorithm 1. □

The complexity of algorithm 2 depends on how many variables are discovered as existential at each of the k iterations. If the set of existential variables discovered at iteration j is V_j (thus we have $V_1 + \ldots + V_k = V'$), then the complexity is $O(\mid V \mid + (\mid V - V_1 \mid)^2 + (\mid V - V_1 - V_2 \mid)^3 + \ldots + (\mid V - V' \mid)^k)$, instead of $O(\mid V \mid^k)$ which is the complexity of a k-consistency algorithm.

A special case of the use of algorithm 2 is when one knows that a CSP is polynomially solved by a k-consistency algorithm. This is for example the case of simple temporal constraint problems (STCSPs) [DMP89], which are solved by 3-consistency. This means that, after one has achieved 3-consistency, the variables can be instatiated compatibly to the constraints without any backtracking. In this case, applying algorithm 2 would obtain the same resulting backtrack-free search, although with a restriction on the order of the instantiations.

6 (1,k)-consistency and Existential Variables

The concept of (i,j)-consistency [Fre88] is a generalization of that of k-consistency: a CSP is (i,j)-consistent if, given an instantiation of i variables which is compatible with all the constraint among the i variables, it is possible to extend it to other j variables such that all constraints among the $i+j$ variables are satisfied. Now, as achieving k-consistency may add constraints of arity at most $k-1$, achieving (i,j)-consistency may add constraints of arity at most i. Adding new

constraints may be too much of a burden on the space and time requirement of the local consistency algorithm, since it means creating new data structures. This is one of the reasons of the great success of arc-consistency (that is, 2-consistency). In fact, achieving arc-consistency may involve adding constraints at most of arity 1, which means just removing elements from variable domains. In this way, no new data structure is needed, but just a modification of an already existing data structure, the domain of a variable (which in most cases is just a bit vector).

Therefore, it is important to study properties and behaviour of algorithms which just remove domain values. Now, it is obvious that the class of (1,k)-consistency algorithms fits in this category. Notice that (1,1)-consistency is just 2-consistency (thus arc-consistency), while (1,k)-consistency, with k greater than 1, is obviously more powerful than 2-consistency. Thus we have an algorithm which achieves more pruning than arc-consistency but still removing just domain elements.

As for k-consistency, achieving (i,j)-consistency may yield a great gain during a subsequent search process to find a solution of the given CSP. In particular, there is a relationship between the amount of backtracking to be done during the search and the level of (1,k)-consistency that the problem has (see [Fre88] for more details). Therefore, it is natural to try to exploit the concept of existential variables also with respect to this kind of algorithms, so that some variables may be postponed during the search. In particular, it is possible to find sufficient conditions similar to those in Section 4. One of them is the following one.

Theorem 15 ((1,k)-consistency and existential variables). *Consider a CSP $P = \langle V, D, C, con, def \rangle$ which is (1,k)-consistent, and any set $S = \{x_1, \ldots, x_i\} \subseteq V$ such that $i \leq k$. Assume also that there is a variable $x \notin S$ with $V_{x_j} \subseteq S \cup \{x\}$ for all $j = 1, \ldots, i$, where V_{x_j} is the set of neighbors of x_j. Then all variables in S are existential for P.*

Proof: Similar to that of Theorem 10, and left out for reasons of space. □

The above theorem basically says that any set containing less than k variables which are either connected among them or to another variable x is a set of variables which can be instantiated without any backtracking after all other variables have been instantiated. Actually, only the instantiation of variable x is needed.

Starting from this theorem, an algorithm similar to algorithm 2 may easily be derived, so that a search complexity smaller than or equal to that needed for a (1,k)-consistent CSP may be obtained without actually computing (1,k)-consistency everywhere in the CSP.

7 Conclusions and Future work

We proposed an algorithm which achieves different levels of consistency, all lower than or equal to k, on different parts of a CSP, but which yields a subsequent

search process with a complexity smaller than or equal to that one would have in a k-consistent CSP. That is, we got a smaller complexity of the solution search by using a less costly algorithm for local consistency.

We plan to experiment with our algorithm for achieving incremental k-consistency and see how it behaves with respect to standard k-consistency algorithms (as a start, we will consider $k = 3$, so that we can compare it to the many existing algorithms for path-consistency).

We also plan to investigate the relationship between constraint tightness and variable existentiality, following some recent studies on the relationship between such notion and backtrack-free search [vBD94]. In fact, our conjecture is that, in a CSP which is $(m + 2)$-consistent, any variable which is involved only in constraints with tightness smaller than or equal to m is existential. This could be combined with the sufficient condition we consider in this paper to discover more existential variables during the algorithm.

We also plan to combine this work with that on CSPs with hidden variables, so that in a CSP with both visible and hidden variables, some hidden variables are removed because found to be redundant [Ros95], and some visible variables are postponed because found to be existential.

References

[Dec92] Rina Dechter. From local to global consistency. *Artificial Intelligence*, 55:87–107, 1992.

[DMP89] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. In *Proc. International Conference on Knowledge Representation.* Morgan-Kaufmann, 1989.

[DP88] R. Dechter and J. Pearl. Network-Based Heuristics for Constraint-Satisfaction Problems. In Kanal and Kumar, editors, *Search in Artificial Intelligence.* Springer-Verlag, 1988.

[Fre78] E. C. Freuder. Synthesizing constraint expressions. *Communication of the ACM*, 21(11), 1978.

[Fre88] E. C. Freuder. Backtrack-free and backtrack-bounded search. In Kanal and Kumar, editors, *Search in Artificial Intelligence.* Springer-Verlag, 1988.

[Mac77] A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1), 1977.

[Mac92] A.K. Mackworth. Constraint satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of AI (second edition)*, volume 1, pages 285–293. John Wiley & Sons, 1992.

[MF85] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25, 1985.

[Mon74] U. Montanari. Networks of constraints: Fundamental properties and application to picture processing. *Information Science*, 7, 1974.

[Ros95] F. Rossi. Redundant hidden variables in finite domain constraint problems. In M.Meyer, editor, *Constraint Processing.* Springer-Verlag, LNCS 923, 1995.

[vBD94] P. van Beek and R. Dechter. Constraint tightness versus global consistency. In *Proc. KR94.* Morgan Kaufmann, 1994.

Logical Semantics of Concurrent Constraint Programming

Paul Ruet[1,2]

[1] LIENS, Ecole Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France
[2] Thomson - LCR, Domaine de Corbeville, 91404 Orsay, France

Email: ruet@dmi.ens.fr

Abstract. This paper investigates logical characterizations of some aspects of concurrent constraint (cc) computations. It contains both negative and positive results.
We show that intuitionistic logic enables to observe the so-called *stores* of a concurrent constraint agent, but neither its successes nor its suspensions, even in the monotonic and deterministic case. On the other hand, IMALL (intuitionistic multiplicative and additive linear logic) does enable the observation of successes (but not that of suspensions): we consider a non-monotonic and non-deterministic version of cc, lcc, and we show that the successes of an lcc computation can be characterized logically; this holds also for cc, since cc can be faithfully translated into lcc.

Keywords
Concurrent constraint programming, intuitionistic logic, linear logic.

1 Introduction

Concurrent constraint programming cc [21] is a model of concurrent computation, where concurrent agents communicate through a shared store, represented by a constraint, which expresses some partial information on the values of the variables involved in the computation. An agent may add a constraint c to the store, or ask the store to entail a given constraint ($c \rightarrow A$). Communication is *asynchronous*: agents can remain idle, and senders (constraints c) are not blocking. Computation is *monotonic* (the constraints in the store are not consumed): this allows to provide cc with a denotational semantics, viewing agents as closure operators on the semi-lattice of constraints [23].

Syntactically, concurrent constraint programming is an extention of *constraint logic programming* [9, 14] with a suspension mechanism $c \rightarrow A$, and the operational semantics of cc is the same as that of constraint logic programming, except for $c \rightarrow A$ which blocks until the amount of accumulated information (the store) is strong enough to entail c (in intuitionistic or classical logic), in which case $c \rightarrow A$ evolves to A.

Besides, Saraswat and Lincoln have proposed a *non-monotonic* version of cc [22], further studied in [4, 25] where the logic of constraints is linear logic [8]: in this version, constraints can be consumed, and the language is therefore closer to process calculi like Milner's π-calculus [17].

In constraint logic programming (with, or without negation by failure, or constructive negation), the logical nature of the constraint system extends to the goals and program declarations, and states strong connections between operational semantics and entailment (in classical logic or 3-valued logic) [14, 12, 24, 6]. For instance, success constraints (i.e. final states of computations) can be observed logically: any success entails the initial state (modulo the completed program $P^\star$ and the constraint system $\mathcal{C}$); conversely any constraint c entailing a goal G is covered (again modulo $P^\star$ and $\mathcal{C}$) by a finite set of successes $c_1 \dots c_n$, i.e. $\mathcal{C} \vdash \forall(c_1 \dots c_n \Rightarrow c)$. Such results make easier the design and understanding of programs, and provide useful tools for reasoning about them.

In concurrent constraint programming, the situation is less clear. In [13] Lincoln and Saraswat give an interesting connection between the observation of the stores of cc agents and entailment in intuitionistic logic. However it tells just part of the story of a cc computation: for instance, it does not say anything about eventual suspending agents. Actually let a *success* of an agent A be a store c such that A evolves to c, and let a *suspension* be an agent $B = c \wedge (d \rightarrow A)$ such that A evolves to B and c does not entail d (the exact definition is slightly longer): in section 3 we shall show, through counter-examples, that the observation of successes or suspensions is not expressible in intuitionistic logic. Roughly speaking, the interpretation of cc agents as intuitionistic formulas stumbles against the structural rule of (left) *weakening*:

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}$$

Girard's linear logic is a fine proof-theoretical study of the weakening and *contraction* structural rules of classical and intuitionistic logics. While moving to linear logic, it is very natural to move to a non-monotonic version of cc at the same time. This has been done by Saraswat and Lincoln in a higher-order setting. Here we define a first-order non-monotonic variant, lcc, for which we distinguish successes and suspensions (section 4), and we prove a completeness result on successes: we show that the successes of an lcc computation can be characterized in IMALL (intuitionistic multiplicative and additive linear logic) (section 5), whereas the suspensions cannot (section 6). We show that cc can be faithfully translated into lcc, so this result holds also for cc.

Finally we discuss in section 6 the limits of the correspondence between linear logic and lcc computations, and its signficance for linear logic and concurrency.

2 Monotonic cc

A *monotonic constraint system* is a pair $(\mathcal{C}, \Vdash_{\mathcal{C}})$, where: $\mathcal{C}$ is a set of formulas (the *constraints*) built from a set V of variables, a set Σ of function and relation symbols, and logical operators $\top$ (*true*), $\wedge$ and $\exists$; and $\Vdash_{\mathcal{C}} \subseteq \mathcal{C}^\star \times \mathcal{C}$. We assume $\wedge$ has neutral $\top$. Instead of $((c_1 \ldots c_n), c) \in \Vdash_{\mathcal{C}}$, we write $c_1 \ldots c_n \Vdash_{\mathcal{C}} c$.

$\vdash_{\mathcal{C}}$ is the least reflexive relation $\subseteq \mathcal{C}^\star \times \mathcal{C}$ containing $\Vdash_{\mathcal{C}}$ and closed by the following rules of intuitionistic logic:

$$\frac{\Gamma, d, d \vdash c}{\Gamma, d \vdash c} \qquad \frac{\Gamma \vdash c}{\Gamma, d \vdash c} \qquad \frac{\Gamma, c \vdash d \qquad \Gamma \vdash c}{\Gamma \vdash d}$$

$$\frac{\Gamma \vdash c}{\Gamma \vdash \exists x c} \qquad \frac{\Gamma, A \vdash c}{\Gamma, \exists x A \vdash c}\ x \notin fv(\Gamma, c)$$

$$\frac{\Gamma \vdash c_1 \qquad \Gamma \vdash c_2}{\Gamma \vdash c_1 \wedge c_2} \qquad \frac{\Gamma, c_1 \vdash c}{\Gamma, c_1 \wedge c_2 \vdash c} \qquad \frac{\Gamma, c_2 \vdash c}{\Gamma, c_1 \wedge c_2 \vdash c}$$

The set $\mathcal{A}$ of cc agents is given by the following grammar:

$$A ::= c \mid c \rightarrow A \mid A \wedge A \mid A \vee A \mid \exists x A \mid p(\boldsymbol{x})$$

where c stands for a constraint, $\wedge$ for parallel composition, $\vee$ for non-determinism and $\exists x$ for hiding of a variable x. In an agent $A = c \wedge A_1 \wedge \cdots \wedge A_n$, if c is a constraint, the main constructor of each A_i is not $\wedge$, and no A_i is a constraint, we call c the *store* of A. It is the 'constraint' part of A.

Recursion is obtained with declarations:

$$D ::= \epsilon \mid p(\boldsymbol{x}) = A \mid D, D$$

The operational semantics is given in the style of the Chemical Abstract Machine [3] (see also [19]). This presentation, though different from a logic programming one, has the advantage of keeping track of the variable bindings, and we find it therefore cleaner to manage logically.

• The *structural congruence* $\equiv$ is the least congruent equivalence such that $(\mathcal{A}/\equiv, \wedge, \top)$ is an abelian monoid, $(\mathcal{A}/\equiv, \vee)$ is an abelian semi-group, $\wedge$ and $\vee$ are distributive with respect to each other, and such that, for all agents A and B:

$$\exists x \top \equiv \top \qquad \exists x \exists y A \equiv \exists y \exists x A$$

$$\frac{A \text{ and } B \text{ are } \alpha\text{-convertible}}{A \equiv B} \qquad \frac{x \text{ is not free in } A}{\exists x (A \wedge B) \equiv A \wedge \exists x B}$$

• The *transition* relation $\longrightarrow$ is the least reflexive transitive congruence such that:

$$c \wedge (c \to A) \longrightarrow c \wedge A \qquad \frac{c \vdash_C d}{c \longrightarrow d} \qquad \frac{(p(x) = A) \in P}{p(x) \longrightarrow A}$$

$$\frac{A' \equiv A \qquad A \longrightarrow B \qquad B \equiv B'}{A' \longrightarrow B'} \qquad \frac{A \longrightarrow C \qquad B \longrightarrow C}{A \vee B \longrightarrow C}$$

Remarks:

▶ This version of cc is monotonic in the following sense: for any transition $c \wedge A \longrightarrow d \wedge B$ between agents $c \wedge A$ and $d \wedge B$ with respective stores c and d, there exists a transition $c \wedge A \longrightarrow c \wedge d \wedge B$ (even though d might not entail c, as constraints may vanish because of the second rule for the transition relation).

▶ The non-deterministic construct $\vee$ is not the angelic non-deterministic construct of [10], it is close to that of Lincoln and Saraswat in [13]: intuitively $A \vee B$ can be either A or B, but you cannot decide, your vision is 'blurred', so to be able to do a 'sharp' observation on the rest of the computation, both possibilities must have some common result C (a 'coincidence'); it is a form of hiding at the level of agents, in the same way as $\exists$ is a hiding of variables.

Agents and declarations not involving $\vee$ are said *deterministic.*

▶ Concurrent constraint programming languages are parameterized by a constraint system, which is often not mentioned explicitly in the operational semantics as well as in the constraint entailment. Declarations and constraint entailment are also implicit in the operational semantics, but the context will not allow any confusion.

Examples:

▶ $c \wedge (d \to A \vee e \to A) \equiv (c \wedge d \to A) \vee (c \wedge e \to A)$ (distributivity)
$\longrightarrow (d \wedge d \to A) \vee (e \wedge e \to A)$ if $c \vdash_C d$ and $c \vdash_C e$
$\longrightarrow (d \wedge A) \vee (e \wedge A)$
$\longrightarrow (\top \wedge A) \vee (\top \wedge A)$
$\equiv A \vee A \equiv A$ (transition rule for $\vee$)

▶ $c(x) \wedge \exists x(c(x) \to c(x))$ suspends because $\exists x(c(x) \to c(x))$ is not a constraint and therefore cannot be erased; whereas:

$c(x) \wedge \exists x(\top \to c(x)) \equiv c(x) \wedge \exists x(\top \wedge \top \to c(x))$ ($\top$ neutral for $\wedge$)
$\longrightarrow c(x) \wedge \exists x(\top \wedge c(x)) \equiv c(x) \wedge \exists x(c(x)) \longrightarrow c(x)$ ($\exists x(c(x))$ is a constraint)

3 Observations with intuitionistic logic

3.1 Monotonic stores

In concurrent constraint programming, the permanent information is expressed by the store, therefore the stores are very natural observations on cc agents. Indeed this is the approach of Lincoln and Saraswat [13]: the stores of monotonic cc agents can be observed with intuitionistic logic IL.

Let $(\mathcal{C}, \Vdash_{\mathcal{C}})$ be a monotonic constraint system, and $\mathcal{D}$ a set of declarations. Let IL$(\mathcal{C},\mathcal{D})$ be the deduction system obtained by extending IL with:

- elements of $\Vdash_{\mathcal{C}}$ as non-logical axioms,
- for each declaration $p(\boldsymbol{x}) = A$ in $\mathcal{D}$, the sequent $p(\boldsymbol{x}) \vdash A$ as non-logical axiom.

$\dashv$ denotes the inverse of $\vdash$ and $A \dashv\vdash B$ stands for $A \dashv B$ and $A \vdash B$.

Theorem 1 (Soundness) *Let A and B be* cc *agents. If $A \equiv B$ then $A \dashv\vdash_{IL(\mathcal{C},\mathcal{D})} B$. If $A \longrightarrow B$ then $A \vdash_{IL(\mathcal{C},\mathcal{D})} B$.*

Proof Trivial induction on $\equiv$ and $\longrightarrow$. ■

Theorem 2 (Observation of monotonic stores) *Let A be a* cc *agent and c a constraint. If $A \vdash_{IL(\mathcal{C},\mathcal{D})} c$ then there are constraints $c_1 \ldots c_n$ and agents $B_1 \ldots B_n$ such that $A \longrightarrow (c_1 \wedge B_1) \vee \cdots \vee (c_n \wedge B_n)$ and for all i, $c_i \vdash_{\mathcal{C}} c$.*

Sketch of proof It is simpler to prove the result for multisets of agents $A_1 \ldots A_n$: if $A_1 \ldots A_n \vdash c$ in IL$(\mathcal{C},\mathcal{D})$, then $A_1 \wedge \ldots \wedge A_n \longrightarrow c$. We proceed by induction on a (sequent calculus) proof of $A_1 \ldots A_n \vdash c$: remark that this works because the formula on the right is a constraint in each sequent of the proof. Each logical rule simulates a transition rule of `1cc`, where commas on the left of sequents stand for parallel composition. For axioms, cut, contraction and $\wedge$-rules, it is evident. Idem for the left introductions of $\vee$ and $\top$. Since a constraint c contains only $\wedge$ and $\exists$, the only other right rule to consider is the right introduction of $\exists$, for which the induction hypothesis applies, by the definition of constraint entailment. For weakening, just remark that conjunction $\wedge$ is distributive with respect to $\vee$, and the hypothesis applies. The only non-trivial cases are:

1. the $\rightarrow$ left introduction:

$$\frac{\Gamma, A \vdash c \qquad \Delta \vdash d}{\Gamma, \Delta, d \rightarrow A \vdash c}$$

By induction hypothesis, $\Delta \longrightarrow (d_1 \wedge E_1) \vee \cdots \vee (d_k \wedge E_k)$, and for all j, $d_i \vdash_{\mathcal{C}} d$, so for all j, $d_j \wedge E_j \wedge (d \rightarrow A) \longrightarrow d_j \wedge E_j \wedge A$. Set $F = (d_1 \wedge E_1) \vee \cdots \vee (d_k \wedge E_k)$. Then $\Gamma \wedge \Delta \wedge (d \rightarrow A) \longrightarrow \Gamma \wedge F \wedge (d \rightarrow A) \longrightarrow \Gamma \wedge F \wedge A$. Again by induction hypothesis, $\Gamma \wedge A \longrightarrow (c_1 \wedge B_1) \vee \cdots \vee (c_n \wedge B_n)$ and for all i, $c_i \vdash_{\mathcal{C}} c$. Set $G = (c_1 \wedge B_1) \vee \cdots \vee (c_n \wedge B_n)$. Then $\Gamma \wedge \Delta \wedge d \rightarrow A \longrightarrow F \wedge G$, and the distributivity of $\wedge$ w.r.t. $\vee$ enables to conclude.

2. the $\exists$ left introduction:

$$\frac{\Gamma, A \vdash c}{\Gamma, \exists x A \vdash c}\ x \text{ not free in } \Gamma, c$$

By induction hypothesis, $\Gamma \wedge A \longrightarrow (c_1 \wedge B_1) \vee \cdots \vee (c_n \wedge B_n)$, so $\exists x(\Gamma \wedge A) \longrightarrow \exists x((c_1 \wedge B_1) \vee \cdots \vee (c_n \wedge B_n))$. Now x is not free in Γ, so $\exists x(\Gamma \wedge A) \equiv \Gamma \wedge \exists x A$, hence $\Gamma \wedge \exists x A \longrightarrow \exists x((c_1 \wedge B_1) \vee \cdots \vee (c_n \wedge B_n))$, with $\exists x((c_1 \wedge B_1) \vee \cdots \vee (c_n \wedge B_n)) \vdash \exists x c$. Now x is not free in c, so $\exists x c \equiv c$, and the result is proved. ■

3.2 Denotational semantics?

In this section we show through examples that the fixed points of (monotonic) cc agents (hence their denotational semantics [23, 10]) cannot be characterized via intuitionistic logic, even in the deterministic case.

Let $(\mathcal{C}, \Vdash_{\mathcal{C}})$ be a monotonic constraint system, and suppose, to simplify, that the set of declarations is empty. Again to simplify, we suppose that agents are deterministic. In [23], deterministic cc agents are viewed as continuous closure operators on the set of constraints $\mathcal{C}$; an agent A is determined by its set $[A]$ of fixed points:

$[c] = \{d \in \mathcal{C} \mid d \vdash_{\mathcal{C}} c\}$,
$[c \rightarrow A] = \{d \in \mathcal{C} \mid d \vdash_{\mathcal{C}} c \text{ implies } d \in [A]\}$,
$[A \wedge B] = [A] \cap [B]$,
$[\exists x A] = \{c \in \mathcal{C} \mid$ there exists $d \in [A]$ such that $\exists x c \dashv\vdash_{\mathcal{C}} \exists x d\}$.

By the first equality above, a logical characterization of the denotational semantics of cc agents should be: $c \in [A]$ iff $c \vdash_{IL(\mathcal{C},\mathcal{D})} A$. But then take $A = d \rightarrow B$: by the second equality, $c \vdash_{IL(\mathcal{C},\mathcal{D})} A$ iff $c \in [A]$ iff $(c \not\vdash_{\mathcal{C}} d$ or $c \in [B])$ iff $(c \not\vdash_{\mathcal{C}} d$ or $c \vdash_{IL(\mathcal{C},\mathcal{D})} B)$. This is impossible, since $c \vdash d \rightarrow B$ is by no means equivalent to $c \not\vdash d$ or $c \vdash B$: for instance, if c is $x > 3$, d is $x > 4$ and B is just the constraint $x > 5$, it is true that $c \not\vdash d$, but $c \not\vdash d \rightarrow B$. The problem is the instanciation of free variables.

Another approach to the denotational semantics of cc is the partial correctness criterion of [7].

3.3 Successes and suspensions?

The declarative nature of usual logic programming and constraint logic programming relies essentially on the logical observation of the successes (and failures) of a program. It is therefore natural to look for a similar characterization in the cc setting. We define the *successes* of a cc computation starting with A to be the stores c such that $A \longrightarrow c$. Note that, since $c \vdash_{\mathcal{C}} 1$ holds for any constraint c, a success may just be part of a final store. Other interesting observations would be the *suspensions* of a computation, i.e. the agents $B = c \wedge (d_1 \rightarrow A_1) \wedge \cdots \wedge (d_n \rightarrow A_n)$ such that $A \longrightarrow B$ and for no i, $c \vdash_{\mathcal{C}} d_i$.

Intuitionistic logic does not enable the observation of successes, neither that of suspensions, even in the deterministic case and without program declarations.

Let $(\mathcal{C}, \Vdash_{\mathcal{C}})$ be a constraint system.

▶ $\dashv$: It is not true in general that $A \dashv \Gamma$ (for Γ a success or a suspension) implies $A \longrightarrow \Gamma$. For instance $c \rightarrow d \dashv d$ but $c \rightarrow d$ can suspend, and have thus no success. Besides $d \dashv d \wedge (c \rightarrow d)$ and if we do not have $d \vdash c$, $d \wedge (c \rightarrow d)$ is a suspension, whereas the constraint c is not a suspension.

▶ $\vdash$: Similar problems arise with $\vdash$. $d \wedge (c \rightarrow A) \vdash d$ whereas $d \wedge (c \rightarrow A)$ suspends as soon as we do not have $d \vdash c$. Besides $d \wedge (d \rightarrow e) \vdash d \rightarrow e$, but $d \wedge (d \rightarrow e)$ has a success $(d \wedge e)$ and does not suspend.

▶ $+\!\!+$: Equivalence $+\!\!+$ is of no help. Suppose we do not have $d \vdash c$, and consider the following equivalence: $d \wedge (c \rightarrow d) +\!\!+ d$. It does not allow to conclude anything about the operational behaviour of the agents d and $d \wedge (c \rightarrow d)$.

The obstacle is the structural rule of (left) *weakening*. Therefore we move to linear logic. At the same time it is natural to move to a non-monotonic version of `cc`, `lcc`, already introduced by Saraswat and Lincoln [22] and further studied by [4, 25].

4 Non-monotonic cc

A *linear constraint system* is a pair $(\mathcal{C}, \Vdash_{\mathcal{C}})$ where: $\mathcal{C}$ is a set of formulas (the *linear constraints*) built from a set V of variables, a set Σ of function and relation symbols, and logical operators 1, the multiplicative conjunction $\otimes$, the existential quantifier $\exists$ and the exponential !; and $\Vdash_{\mathcal{C}} \subseteq \mathcal{C}^\star \times \mathcal{C}$. We assume $\otimes$ has neutral 1. Instead of $((c_1 \ldots c_n), c) \in \Vdash_{\mathcal{C}}$, we write $c_1 \ldots c_n \Vdash_{\mathcal{C}} c$.

$\vdash_{\mathcal{C}}$ is the least reflexive and transitive relation $\subseteq \mathcal{C}^\star \times \mathcal{C}$ containing $\Vdash_{\mathcal{C}}$ and closed by the following rules:

$$\frac{\Gamma, c_1, c_2 \vdash c}{\Gamma, c_1 \otimes c_2 \vdash c} \qquad \frac{\Gamma \vdash c_1 \qquad \Delta \vdash c_2}{\Gamma, \Delta \vdash c_1 \otimes c_2} \qquad \frac{\Gamma, c \vdash d \qquad \Delta \vdash c}{\Gamma, \Delta \vdash d}$$

$$\frac{\Gamma \vdash c}{\Gamma \vdash \exists x c} \qquad \frac{\Gamma, A \vdash c}{\Gamma, \exists x A \vdash c}\ x \notin fv(\Gamma, c)$$

$$\frac{!\Gamma \vdash c}{!\Gamma \vdash !c} \qquad \frac{\Gamma \vdash d}{\Gamma, !c \vdash d} \qquad \frac{\Gamma, c \vdash d}{\Gamma, !c \vdash d} \qquad \frac{\Gamma, !c, !c \vdash d}{\Gamma, !c \vdash d}$$

They are the rules of intuitionistic linear logic (ILL) for $\otimes$, $\exists$ and !, plus the cut rule. The syntax of `lcc` agents is given by the following grammar:

$$A ::= c \mid c \multimap A \mid A \otimes A \mid A \& A \mid \exists x A \mid p(\boldsymbol{x})$$

where $\otimes$ stands for parallel composition, and $\multimap$ for suspension. In an agent $A = c \otimes A_1 \otimes \cdots \otimes A_n$, if c is a constraint, the main constructor of each A_i is not $\otimes$, and no A_i is a constraint, we call c the *store* of A. It is the 'constraint' part of A.

Recursion is obtained with declarations:

$$D ::= \epsilon \mid p(\boldsymbol{x}) = A \mid D, D$$

• The *structural congruence* $\equiv$ is the least congruent equivalence such that $(\mathcal{A}/\equiv, \otimes, 1)$ is a abelian monoid, $(\mathcal{A}/\equiv, \&)$ is an abelian semi-group, and such that, for all agents A and B:

$$\exists x 1 \equiv 1 \qquad \exists x \exists y A \equiv \exists y \exists x A \qquad \frac{A \text{ and } B \text{ are } \alpha\text{-convertible}}{A \equiv B}$$

$$\frac{x \text{ is not free in } A}{\exists x (A \otimes B) \equiv A \otimes \exists x B} \qquad c \multimap (A \& B) \equiv (c \multimap A) \& (c \multimap B)$$

• The *transition* between agents $\longrightarrow$ is the least reflexive transitive congruence such that:

$$c \otimes (c \multimap A) \longrightarrow A \qquad \frac{c \vdash_C d}{c \longrightarrow d} \qquad \frac{(p(x) = A) \in P}{p(x) \longrightarrow A}$$

$$\frac{A' \equiv A \qquad A \longrightarrow B \qquad B \equiv B'}{A' \longrightarrow B'}$$

$$A \& B \longrightarrow A \qquad A \& B \longrightarrow B \qquad A \otimes (B \& C) \longrightarrow (A \otimes B) \& (A \otimes C)$$

Remarks:

▶ Constraints are 'consumed' by suspensions; therefore the rule for $c \multimap A$ involves *non-determinism* since several stores may satisfy the condition of the rule.

▶ The non-deterministic $A \& B$ (already considered in [22, 4]) can behave either like A or like B, it has both capabilities. Note that this non-determinism is different from that of $\vee$ in monotonic cc.

Agents and declarations not involving & are said *deterministic*.

▶ The exponential ! allows to recover the monotonic case cc; for this reason, we just allow it on constraints.

Translation of deterministic cc into lcc

In the next section we show that IMALL (intuitionistic multiplicative additive linear logic) enables the observation of the successes of lcc agents. To be able to observe the successes of monotonic cc agents, one has to prove that cc agents can be translated into lcc ones, in such a way that the operational semantics is preserved. We do it for deterministic cc agents. The non-deterministic case can be translated into lcc as well, but it requires the use of the additive disjunction $\oplus$, which we do not consider in this paper to avoid confusion. We postpone the treatment of full monotonic cc to further work.

Let $(\mathcal{C}, \Vdash_C)$ be a constraint system. The linear constraint system $(\mathcal{C}, \Vdash_C)^!$ has the same set of atomic constraints (with $\top$ renamed 1). The associated compound constraints and deterministic cc agents are translated into linear constraints and lcc agents:

$$c^\dagger = !c, \text{ if } c \text{ is atomic} \quad p(\boldsymbol{x})^\dagger = p(\boldsymbol{x})$$
$$(c \to A)^\dagger = c \multimap A^\dagger \quad (\exists x A)^\dagger = \exists x A^\dagger$$
$$(A \wedge B)^\dagger = A^\dagger \otimes B^\dagger$$

Observe that c is a (monotonic) constraint iff $c^\dagger$ is a (linear) constraint. The proof of the following proposition is then straightforward:

Proposition 1 *Let c and d be monotonic constraints: $c \vdash_{cc} d$ iff $c^\dagger \vdash_{lcc} d^\dagger$. Let A and B be deterministic* cc *agents: $A \equiv_{cc} B$ iff $A^\dagger \equiv_{lcc} B^\dagger$, $A \longrightarrow_{cc} B$ iff $A^\dagger \longrightarrow_{lcc} B^\dagger$.*

For the atoms, $\wedge$ and $\exists$, our translation is Girard's second tranlation of intuitionistic logic into linear logic [8, p.81]. The only difference is the translation of $\to$ ($c \multimap A^\dagger$ instead of $!(c^\dagger \multimap A^\dagger)$), which essentially forbids the erasure of suspensions.

5 Observing successes with IMALL

Let $(\mathcal{C}, \Vdash_\mathcal{C})$ be a fixed linear constraint system, and $\mathcal{D}$ be a fixed set of declarations.

Let $(\mathcal{C}, \Vdash_\mathcal{C})$ be a linear constraint system, and $\mathcal{D}$ a set of declarations. Let IMALL($\mathcal{C}$,$\mathcal{D}$) be the deduction system obtained by extending IMALL with:

- elements of $\Vdash_\mathcal{C}$ as non-logical axioms,
- for each declaration $p(\boldsymbol{x}) = A$ in $\mathcal{D}$, the sequent $p(\boldsymbol{x}) \vdash A$ as non-logical axiom.

We define the *successes* of an `lcc` computation starting with A to be the stores c such that $A \longrightarrow c$. The *suspensions* of A are the agents $B = c \otimes (d_1 \multimap A_1) \otimes \cdots \otimes (d_n \multimap A_n)$ such that $A \longrightarrow B$ and for no i, $c \vdash_\mathcal{C} d_i$.

Theorem 3 (Soundness) *Let A and B be* `lcc` *agents.*
If $A \equiv B$ then $A \vdash_{IMALL(\mathcal{C},\mathcal{D})} B$. If $A \longrightarrow B$ then $A \vdash_{IMALL(\mathcal{C},\mathcal{D})} B$.

Proof Trivial induction on $\equiv$ and $\longrightarrow$. ■

A *success* for an `lcc` agent A is a linear constraint c such that $A \longrightarrow c$.

Theorem 4 (Observation of successes) *Let A be an* `lcc` *agent, and c be a linear constraint. If $A \vdash_{IMALL(\mathcal{C},\mathcal{D})} c$, then $A \longrightarrow c$, i.e. c is a success for A.*

Sketch of proof It is simpler to prove the result for multisets of agents $A_1 \ldots A_n$: if $A_1 \ldots A_n \vdash c$ in IMALL($\mathcal{C}$,$\mathcal{D}$), then $A_1 \otimes \ldots \otimes A_n \longrightarrow c$. We proceed by induction on a (sequent calculus) proof of $A_1 \ldots A_n \vdash c$. Each logical rule simulates a transition rule of `lcc`, where commas on the left of sequents stand for parallel composition. For axioms, cut, $\otimes$-rules and !-rules, it is evident. Since

a constraint c contains only $\otimes$, $\exists$ and !, the only other right rule to consider is the right introduction of $\exists$, for which the induction hypothesis applies, by the definition of constraint entailment. It is evident for the left introductions of & and 1:

$$\frac{\Gamma, A \vdash c}{\Gamma, A\&B \vdash c} \qquad \frac{\Gamma, B \vdash c}{\Gamma, A\&B \vdash c} \qquad \frac{\Gamma \vdash c}{\Gamma, 1 \vdash c}$$

The only non-trivial cases are:

1. the $\multimap$ left introduction:

$$\frac{\Gamma, A \vdash c \qquad \Delta \vdash d}{\Gamma, \Delta, d \multimap A \vdash c}$$

By induction hypothesis, $\Delta \longrightarrow d$, so $(\Gamma \otimes \Delta \otimes d \multimap A) \longrightarrow (\Gamma \otimes d \otimes d \multimap A) \longrightarrow (\Gamma \otimes A)$. Again by induction hypothesis, $(\Gamma \otimes A) \longrightarrow c$, so $(\Gamma \otimes \Delta \otimes d \multimap A) \longrightarrow c$.

2. the $\exists$ left introduction:

$$\frac{\Gamma, A \vdash c}{\Gamma, \exists x A \vdash c}\ x \text{ not free in } \Gamma,\ c$$

By induction hypothesis, $(\Gamma \otimes A) \longrightarrow c$, so $\exists x(\Gamma \otimes A) \longrightarrow \exists x c$ But x is not free in c, so $\exists x c \equiv c$. And as x is not free in Γ, $\exists x(\Gamma \otimes A) \equiv (\Gamma \otimes \exists x A)$, hence $(\Gamma \otimes \exists x A) \longrightarrow c$. ■

Thanks to the translation from `cc` to `lcc`, the result holds for `cc` agents as well.

It is worth noting that a success is not proved to be a constraint entailing the initial agent (as in constraint logic programming), but entailed by the initial agent. This change of perspective is not very surprising in fact, since suspensions $c \multimap A$ contain implicitly a kind of negation (under the form of linear implication $\multimap$ as we shall see), which reverses the sense of deduction.

6 Discussion

We have shown that linear logic enables to do finer observations on `cc` and `lcc` agents than intuitionistic logic.

6.1 The limits of the correspondence

This result emphasizes the correspondence between the transition relation $\longrightarrow$ on agents and the entailment relation $\vdash$ in intuitionistic linear logic, and it is therefore a first step towards viewing concurrent constraint programming as (a fragment of some version of) linear logic. The purpose of this paragraph is to make more precise the limits of this approach.

A look at the proof of Theorem 4 shows that the left introduction rules of IMALL sequent calculus just correspond to transitions in `lcc`. The rules for $\otimes$

express the monoidal structure of $\otimes$. Program declarations have been oriented ($p(\boldsymbol{x}) = A$ is translated into $p(\boldsymbol{x}) \vdash A$): then an axiom $p(\boldsymbol{x}) \vdash A$ just corresponds to the replacement of the agent $p(\boldsymbol{x})$ by the agent A.

On the contrary, the right introductions of &, $\multimap$ and $\exists$ do not correspond to transitions of agents. Let us look at them more closely:

- The right introduction of &:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \& B}$$

  expresses the *combination of two experiments* starting with the same agent. You know that $A\&B$ can behave either like A or like B (left introduction), and now you postulate the converse, i.e. that observing either A or B from Γ means $\Gamma \longrightarrow A\&B$. The right introduction of & is equivalent to $A\&A \dashv\vdash A$.

- The right introduction of $\multimap$:

$$\frac{\Gamma A \vdash B}{\Gamma \vdash A \multimap B}$$

  expresses a '*porosity*' of suspensions: for instance, $A \otimes (c \multimap B) \vdash c \multimap (A \otimes B)$... A strong form of porosity is considered in process calculi like the π-calculus [16], namely *enablement*: for a guard ω such that no free variable in A is bound in ω, it states that $\omega(A \otimes B) \equiv A \otimes (\omega B)$. But it is not considered in `cc`, and as we shall see in the next paragraph, it forbids the observation of suspensions.

- The right introduction of $\exists$:

$$\frac{\Gamma \vdash A[t/x]}{\Gamma \vdash \exists x A}$$

  expresses *blindness*. The agent refuses the communication with the environment through variable (or channel) x, what can lead it to deadlock.

The significance of these remarks is twofold:

1. they are limits to the correspondence between deduction in IMALL and the operational semantics of `cc` and `1cc` languages,
2. their rather intuitive operational interpretation (at least for the first two ones) deserves further consideration.

6.2 Suspensions

Concurrent constraint programming suggests another interesting observation, namely the suspensions of an agent: a *suspension* for an `1cc` agent A is an agent $B = c \otimes (d_1 \multimap A_1) \otimes \cdots \otimes (d_n \multimap A_n)$ such that $A \longrightarrow B$ and for no i, $c \vdash_C d_i$.

The above remark on the right introduction of $\multimap$ shows that suspensions cannot be observed in the setting of IMALL: for instance, $c \otimes (c \multimap 1) \vdash c \multimap$

$(c \otimes 1)$, a suspension, whereas $c \otimes (c \multimap 1)$ succeeds with 1. This 'porosity' is linked to the lack of a *sequential composition* connective in ILL. Such a connective needs to be non-commutative. Abrusci [1] studied a pure non-commutative version LL, without commutative and additive (&) connectives, and we need at least a commutative multiplicative connective (the 'parallel' connective). Retoré's *before* connective $<$ [20] is not a solution either, since $A \otimes (B < C) \vdash B < (A \otimes C)$.

The next step of our investigation will be to define a non-commutative version of linear logic, which copes with this difficulty.

6.3 The significance for linear logic and concurrency

Other (linear) logical approaches have been proposed to study concurrency with the approach of logic programming. Andreoli and Pareschi [2] point out that the 'proof-search as computation' analogy for linear logic corresponds to a reactive paradigm, but in Linear Objects, synchronous message passing involves extra-logical operators ('tell markers'), whereas concurrent constraint programming is asynchronous, what avoids the resort to such extra-logical operators. Miller [15] describes a connection between the π-calculus and linear logic, but uses non-logical constants as well; a connection *à la* Miller between Boudol's asynchronous version of the π-calculus [5] and our work should be interesting. Perrier [18] proposes a denotational semantics based on the phase semantics, to model the interaction capability of a process. Our paper focuses on concurrent constraint programming, so the two approaches are different, but we think a thorough comparison would be interesting. Kobayashi and Yonezawa [11] define several concurrent semantics for linear logic processes, including bisimulation; we believe the relationship with concurrent constraint programming through our work should be worth investigating.

Acknowledgements

I am greatly indebted to François Fages for many useful discussions on this work. I thank the anonymous referees for their comments on a preliminary version of this paper.

References

1. M. Abrusci. Phase semantics and sequent calculus for pure non-commutative classical linear logic. *Journal of Symbolic Logic*, 56(4), 1991.
2. J.M. Andreoli and R. Pareschi. Linear objects: logical processes with built-in inheritance. *New Generation Computing*, 9, 1991.
3. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96, 1992.
4. E. Best, F.S. de Boer, and C. Palamidessi. Concurrent constraint programming with information retrieval. Esprit project ACCLAIM final report, 1994.
5. G. Boudol. Asynchrony and the π-calculus. Technical Report RR 1702, INRIA, 1992.

6. F. Fages. Constructive negation by pruning. *To appear in J. of Logic Programming*, 1996.
7. F.S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. In *Proceedings of the ACM International Conference on Principles of Programming Languages*, 1994.
8. J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50, 1987.
9. J. Jaffar and J-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on Principles of Programming Languages, Munich, Germany*, pages 111–119. ACM, January 1987.
10. R. Jagadeesan, V. Shanbhogue, and V.A. Saraswat. Angelic non-determinism in concurrent constraint programming. Technical report, Xerox Parc, 1991.
11. N. Kobayashi and A. Yonezawa. Logical, testing and observation equivalence for processes in a linear logic programming. Technical Report 93-4, Department of Computer Science, University of Tokyo, 1993.
12. K. Kunen. Negation in logic programming. *Journal of Logic Programming*, 4(3):289–308, 1987.
13. P. Lincoln and V.A. Saraswat. Proofs as concurrent processes. Manuscript, 1992.
14. M.J. Maher. Logic semantics for a class of committed-choice programs. In *Proceedings of ICLP'87, International Conference on Logic Programming*, 1987.
15. D. Miller. The π-calculus as a theory in linear logic: preliminary results. In *Proceedings Workshop on Extensions of Logic Programming*, Springer LNCS 660, 1992.
16. R. Milner. The polyadic π-calculus. Technical Report ECS-LFCS-91-180, Edinburgh University, 1991.
17. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I and II. *Information and Computation*, 100(1), 1992.
18. G. Perrier. Concurrent programming in linear logic. Technical Report CRIN 95-R-052, INRIA-Lorraine, 1995.
19. A. Podelski and G. Smolka. Operational semantics of constraint logic programming with coroutining. In *Proceedings of ICLP'95, International Conference on Logic Programming*, Tokyo, 1995.
20. Ch. Retoré. *Réseaux et séquents ordonnés.* PhD thesis, Université Paris 7, 1993.
21. V.A. Saraswat. *Concurrent constraint programming.* ACM Doctoral Dissertation Awards. MIT Press, 1993.
22. V.A. Saraswat and P. Lincoln. Higher-order linear concurrent constraint programming. Manuscript, 1992.
23. V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *POPL'91: Proceedings 18th ACM Symposium on Principles of Programming Languages*, 1991.
24. P.J. Stuckey. Constructive negation for constraint logic programming. *Information and Computation*, 118(1), 1995.
25. C. Tse. The design and implementation of an actor language based on linear logic. Master Thesis, MIT, 1994.

Solving Non-binary Convex CSPs in Continuous Domains

Djamila Sam-Haroud and Boi V. Faltings

Artificial Intelligence Laboratory (LIA),
Computer Science Department (DI),
Swiss Federal Institute of Technology (EPFL),
EPFL,Ecublens
CH-1015 Lausanne

Abstract. A globally consistent labeling is a compact representation of the complete solution space for a constraint satisfaction problem (CSP). Constraint satisfaction is NP-complete and so is the construction of globally consistent labelings for general problems. However, for binary constraints, it is known that when constraints are *convex*, path-consistency is sufficient to ensure global consistency and can be computed in polynomial time . We show how in continuous domains, this result can be generalized to ternary and in fact arbitrary n-ary constraints using the concept of (3,2)-relational consistency. This leads to polynomial-time algorithms for computing globally consistent labelings for a large class of numerical constraint satisfaction problems.

1 Introduction

Many problems, ranging from resource allocation and scheduling to fault diagnosis and design, involve numerical constraint satisfaction as an essential component. These problems often represent complex decision processes where the set of variables and constraints involved is not independent of particular solutions and where relevant information, in the form of active constraints and variables, is revealed only as the task proceeds and decisions are taken. In the case where variables and constraints are numerical, the search space for such problems becomes of an unbounded size; each numerical value may trigger a different active context and thus, potentially lead to a different solution.

Figure 1 shows an example from civil engineering where different values for beam depth and beam span lead to different design options and constraints. Choosing the values of the beam's depth and span within regions 3 or 4 would increase the susceptibility to vibrations and involve installing bridging (lateral reinforcements) for damping the floor (Figure 1, -a-). Choosing these values within region 1,2 or 3 makes it possible to have the ventilation ducts go under the beams while choosing them in region 4 would dictate to make opening in the beams to allow passage of the ducts.

Identifying single point solutions, possibly optimal according to some criterion is the viewpoint adopted by almost all the existing mathematical solvers

ranging from linear and non-linear programming to numerical analysis and stochastic techniques. Alternatively, consistency techniques offer the possibility of producing a compact description of the space of *all* solutions by assigning labels (sets of legal values) to individual variables or combination of variables. This is essential for reasoning about design alternatives, as in the example of Figure 1.

While in general, computing globally consistent labeling is NP-hard, recent results [8] show that in the case where constraints are *convex*, low orders of consistency are equivalent to global consistency. For binary constraints (involving at most 2 variables), it has been shown that 3-consistency (also called *path*-consistency and computable in polynomial time) is equivalent to global consistency [8], [2]. However in discrete domains, it has been shown [9] that the generalization of these results to ternary (and higher arity) constraints may involve significantly higher degrees of consistency and thus complexity. In this paper, we show that much more positive results can be obtained in continuous domains.

In fact, we introduce a concept of (3,2)-relational consistency which can be computed in polynomial time and proven equivalent to global consistency for constraint networks containing ternary constraints as well.

We also show how these results can be reliably implemented in practice using an appropriate representation of continuous constraints.

2 Problem statement

In this work we consider constraint satisfaction problems in continuous domains. Variable domains are *intervals* over the reals and constraints are numerical *equalities* and *inequalities* of arbitrary types and arities. For practical considerations, the methods developed target problems where both variables and constraints have physical interpretations and can be handled with limited degrees of precision, as is the case in many engineering applications.

A *continuous CSP* (CCSP),($\mathcal{P} = (V, D, R)$), is defined as a set V of *variables* $x_1, x_2, \ldots x_n$, taking their values respectively in a set D of continuous *domains* $D_1, D_2, \ldots, D_n$ and constrained by a set of *relations* $R_1, \ldots, R_m$. A *domain* is an interval of $\Re$ and a *relation* is defined intensionally by a set of arbitrary equalities and inequalities.

Given a CCSP, we require that for each subset of variables $(x_1, \ldots x_k)$, a unique relation $R(x_1 \ldots x_k)$ exists in the underlying constraint network. In words, each hyper-arc of the constraint network will be labeled by a *total constraint* [3]. We recall that a total constraint between a set S of variables is given as the region formed by combining all mathematical constraints involving S.

We define:

Definition 1. (Convex relation)
Let $\mathcal{P} = (V, C, D)$ be a CCSP. A relation $R(x_1, ..x_k)$ over C is convex if it determines a convex solution space in the domain $D_{x_1} \ldots \times D_{x_k}$, where $D_i \in D$.

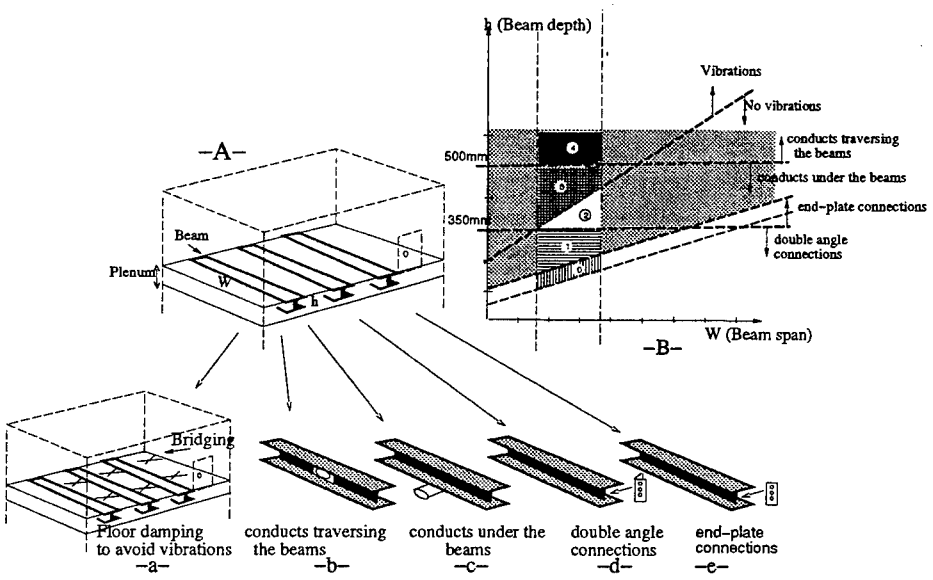


Fig. 1. *Many CCSPs are embedded within complex decision processes*

A CCSP is called convex when all its relations are convex. In a *globally consistent* network, any partial consistent instantiation of a subset of variables can be extended to a *solution* with no backtracking [1], a process which can generally carried out in linear time. For both simple temporal problems and row-convex discrete problems, it has been observed that *convexity* of the constraint relations means that path-consistency is sufficient to ensure a globally consistent labeling. This result is proven using Helly's Theorem:

Theorem 2 (Helly). *Let F be a finite family of at least $n+1$ convex sets in R^n such that every $n+1$ sets in F have a point in common. Then all the sets have a point in common.*

Helly's Theorem can be applied to show that for each assignment of n variables $x_1, x_2, ..., x_n$, there exists a consistent value which can be assigned to x_{n+1} in the following way. Since the constraint network is binary, the only constraints existing between $x_1, ..., x_n$ and x_{n+1} are individual constraints between $x_i, i \in \{1.., n\}$ and x_{n+1}. Since the constraints are convex, every variable x_i already assigned constrains x_{n+1} to a single interval. Path-consistency ensures that every pair of such intervals intersect each other. Thus, by Helly's Theorem,

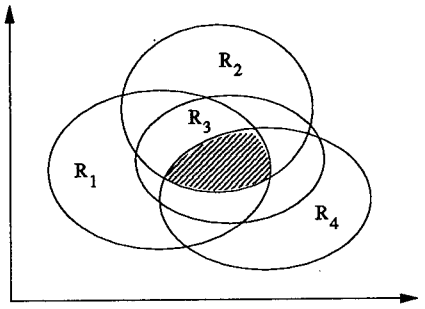


Fig. 2. *Helly's Theorem in $\Re^2$: if a finite set of binary convex regions is such that each triplet of regions has a non-null intersection, then the whole set of regions has a non-null common intersection (shaded area)*

there must exist a common intersection of all the intervals (i.e., at least one value for x_{n+1}) which is consistent with all previous assignments, and consequently the assignment can be extended.

3 Convex n-ary CCSPs

N-ary continuous and discrete CSPs can be translated into ternary ones without loss of information [4]. To generalize the result from binary discrete networks to ternary discrete CSPs, van Beek and Dechter [9] have introduced the notion of relational path-consistency for discrete problems:

Definition 3 (van Beek & Dechter). Let $\mathcal{R}$ be a network of relations over a set of variables X, and let $\mathcal{R}_S$ and $\mathcal{R}_T$ be two relations in $\mathcal{R}$, where $S, T \subseteq X$. We say that $\mathcal{R}_S$ and $\mathcal{R}_T$ are relationally path-consistent relative to variable x iff any consistent instantiation of the variables in $(S \bigcup T) - \{x\}$, has an extension to x that satisfies $\mathcal{R}_S$ and $\mathcal{R}_T$ simultaneously. A pair of relations $\mathcal{R}_S$ and $\mathcal{R}_T$ is relationally path-consistent iff it is relationally path-consistent relative to each variable in $(S \bigcap T)$. A network is relationally path-consistent iff every pair of relations is relationally path-consistent.

By definition, relational path-consistency guarantees for each set of relations having a variable x in common, that the pairwise intersections of their unary projections over the x axis are non-empty. Helly's Theorem becomes thereby applicable, which results in the following Theorem [9]:

Theorem 4 (van Beek & Dechter). *Let $\mathcal{R}$ be a network of relations that is relationally path-consistent. If there exists an ordering of the domains $D_1 \ldots D_n$ of $\mathcal{R}$ such that the relations are row convex, the network is globally consistent.*

Relational path-consistency ensures pairwise non-null intersection of *unary projections*, with the objective of applying Helly's Theorem in *one dimension*

```
Procedure 3-2-rel-con(V,C,D)
   repeat
   changed ← false
   for each pair (u,v), u,v ∈ V do
      for each ternary tuple (i,j,k), i,j,k ∈ V do
         begin
         c'(i,j,k) ← c(i,j,k) ⊕ ∏_(i,j,k) c(i,u,v) ⊗ c(j,u,v) ⊗ c(k,u,v)
         if c'(i,j,k) ≠ c(i,j,k) then
            begin
            c(i,j,k) ← c'(i,j,k)
            changed ← true
            end
         end
   until changed = false
```

Fig. 3. *Algorithm for computing a (3-2)-relationally consistent labeling.*

to individual variables. Composing *pairs* of ternary relations (with a variable in common) results in a relation of arity five. Thus, for global consistency it might be necessary to guarantee that a set of four variables is extensible to a fifth one. But this means that it might be necessary to ensure relational path-consistency for relations of arity four —and recurrently, for relations of unbounded arity, thus possibly engendering an intractable complexity in the most general case.

The alternative generalization we propose is based on the observation that the extensibility of a ternary set of variables to a *binary* region (rather than to a *unary* one like for relational path-consistency) does not involve relations with arity greater than 3 and thus removes the causes behind combinatorial explosion. This approach implies that Helly's Theorem must be applied in two dimensions rather than one (see Figure 2). For the case of ternary networks, we introduce the notion of (3,2)-relational consistency which guarantees that each *triplet of relations*, with *two variables in common*, has a non-null intersection.

Definition 5. (Extension)
Let $\mathcal{P} = (V, C, D)$ be a constraint satisfaction problem. Let V_1 and V_2 be subsets of V. $V1$ has an extension to V_2 if any consistent instantiation of the variables in V_1 can be extended to a consistent instantiation of the variables in $V_1 \bigcup V_2$.

Definition 6. ((3,2)-relational consistency)
Let $\mathcal{P}$ be a ternary network of relations over a set of variables X. Let $R_{I_1}(x_1, u, v)$, $R_{I_2}(x_2, u, v)$ and $R_{I_3}(x_3, u, v)$ be three relations of N which share two variables u and v, where u might be identical to v. R_{I_1}, R_{I_2} and R_{I_3} are (3,2)-relationally consistent relative to $\{u, v\}$ if and only if any consistent instantiation of the 3 variables in x_1, x_2, x_3 has an extension to $\{u, v\}$ that satisfies R_{I_1}, R_{I_2} and R_{I_3} simultaneously.

Since (3,2)-relational consistency only requires labels between at most three variables, it does not add to the arity of a ternary constraint network. Provided

that each binary projection is convex, (3,2)-relational consistency enables the application of Helly's Theorem in two dimensions. However, Helly's Theorem only guarantees that each *pair* of variables has a non-empty domain. It remains to show that the constraints on each *individual* variable are also non-empty.

The simple algorithm of Figure 3 terminates with a set of (3,2)-relational consistent set of labels.

This algorithm takes as input a ternary CCSP, $\mathcal{P} = (V, C, D)$ where V is the set of variables, C the set of constraints, and D is the set of variable domains. c denotes the label of a relation in C. R denotes relations in C.

Using Helly's Theorem in two dimensions we can state the following result:

Theorem 7. *For any convex ternary network $\mathcal{P}$, (3,2)-relational consistency will either decide that $\mathcal{P}$ is inconsistent, or else compute an equivalent globally consistent labeling of $\mathcal{P}$, in time $O(n^5)$ where n is the number of variables of $\mathcal{P}$.*

Informally, the proof consists of the following steps:

1. we first prove that when applied to a ternary network $\mathcal{P}$, the algorithm for (3,2)-relational consistency results in an empty network if a given pair of variables has an empty label,
2. in order to show that non-empty labels on *each pair* of variables implies global consistency, we introduce a *binary* dual representation of the original ternary problem. This dual representation, by the fact that it is binary, makes it easier to show how an instantiation process can be carried out backtrack free when binary labels are non-empty,
3. the dual representation is shown to be globally consistent and equivalent to the primal one.

The binary dual representation of a ternary network $\mathcal{P}(V, C, D)$ is a binary network, $\mathcal{P}_d(V_d, C_d, D_d)$, such that:

- $V_d = \{\alpha_1, \ldots \alpha_m\}$. A variable of V_d, α_j, represents a pair of variables in the original network (an element $(x_{(j,1)}, x_{(j,2)})$ of V^2 so that $x_{(j,1)} \neq x_{(j,2)}$)
- a domain of D_d is an element of D^2
- a relation between two variables, α_u and α_v, of V_d is the relation $R(x_{(u,1)}, x_{(u,2)}, x_{(v,1)}, x_{(v,2)})$ resulting from the composition of the relations between $x_{(u,1)}, x_{(u,2)}, x_{(v,1)}, x_{(v,2)}$ in the original problem $\mathcal{P}$.

In the following, the fact that *each pair* of variables has a non-empty label will be referred to as the *binary-extensibility* property:

Definition 8. (Binary-extensibility)
Let $\mathcal{P}$, (V, C, D), be a network of relations. $\mathcal{P}$ is said to be binary-extensible if any subset of V's variables has an extension to any pair of variables of V.

We now present the intermediate results needed for stating Theorem 7 along with sketches of their proofs.

Lemma 9. *Let $\mathcal{P}'$ be a (3,2)-relationally consistent ternary network and let $\mathcal{P}'_d$ be its dual representation. The following propositions are verified:*
i. each partial solution of $\mathcal{P}'$ corresponds to a partial solution of $\mathcal{P}'_d$
ii. each partial solution of $\mathcal{P}'_d$ corresponds to a partial solution of $\mathcal{P}'$

This results follows immediately from the definition of the dual network representation.

Corollary 10. *Let $\mathcal{P}'$ be a (3,2)-relationally consistent ternary network and let $\mathcal{P}'_d$ be its dual representation. $\mathcal{P}'$ is equivalent to $\mathcal{P}'_d$.*

$\mathcal{P}'$ and $\mathcal{P}'_d$ are equivalent in the sense that each solution of the first network is also a solution of the latter one, and vice versa. Since Lemma 9 is stated for arbitrary partial instantiations it also hold for a global instantiation.

Lemma 11. *Let $\mathcal{P}$ be a ternary network of relations, $\mathcal{P}'$ be its (3,2)-relationally consistent counterpart and $\mathcal{P}'_d$ be the dual representation of $\mathcal{P}'$. If $\mathcal{P}$ has no solution, $\mathcal{P}'_d$ is empty.*

Sketch of proof. An inconsistent network $\mathcal{P}$ is a fortiori not binary-extensible (i.e there exists at least one pair of variables with an empty label). Since the algorithm for (3,2)-relational consistency computes the closure of $\mathcal{P}$ with respect to binary-extensibility, it will therefore necessarily results in an empty (3,2)-relationally consistent representation $\mathcal{P}'$. $\mathcal{P}'$ being equivalent to $\mathcal{P}'_d$ (Corollary 10), $\mathcal{P}'_d$ is also empty □.

Lemma 12. *Let $\mathcal{P}$ be a ternary network of relations, $\mathcal{P}'$ be its (3,2)-relationally consistent counterpart and $\mathcal{P}'_d$ be the dual representation of $\mathcal{P}'$. If $\mathcal{P}'_d$ is non-empty, $\mathcal{P}'_d$ is globally consistent.*

Sketch of proof. Suppose that $k-1$ variables of $\mathcal{P}'_d$ have been consistently instantiated. Using Helly's Theorem in two dimensions, we first show [5] that in the case where the relations of $\mathcal{P}'_d$ are not empty, each consistent instantiation of three variables in $\mathcal{P}'_d$ can be consistently extended to a fourth one. This guarantees that each subset of three $\mathcal{P}'_d$'s relations, $R(\alpha_k, \alpha_a), R(\alpha_k, \alpha_b), R(\alpha_k, \alpha_c)$, (where $(a, b, c) \in [1..k-1]$) has a non-null projection over α_k. Since each relation $\prod_{\alpha_k} R(\alpha_k, \alpha_i)$ is a convex, non-empty region of $\Re^2$ (variables $\alpha_i, i \in [1..k-1]$ are instantiated), Helly's Theorem is applicable and guarantees that $\prod_{\alpha_k}^{i:[1..k-1]} R(\alpha_k, \alpha_i) \neq \emptyset$. This means that α_k can be instantiated consistently. This result holds for an arbitrary k, hence $\mathcal{P}'_d$ is globally consistent □.

Theorem 7 follows immediately from Lemmas 9, 11 and 12. $\mathcal{P}$ being a ternary convex network of relations, $\mathcal{P}'$ its (3,2)-relationally consistent counterpart and $\mathcal{P}'_d$ the dual representation of $\mathcal{P}'$, Lemma 9 guarantees the equivalence of $\mathcal{P}'$ and $\mathcal{P}'_d$, Lemma 11 ensures that an inconsistent $\mathcal{P}$ results in an empty $\mathcal{P}'_d$ (and hence $\mathcal{P}'$) representation, and finally, Lemma 12 guarantees that if a solution exists, $\mathcal{P}'_d$ (and hence $\mathcal{P}'$) is globally consistent.

Complexity. The number of relations checked for binary-extensibility is initially in $O(n^3 + n^2) = O(n^3)$. Each time a relation is modified, $O(n^2 + n) = O(n^2)$ new (3,2)-relational compositions are computed. The global time complexity of (3,2)-relational consistency is therefore $O(n^5)$ □.

Given that k variables $\{x_1, x_2 \ldots x_k\}$ of $\mathcal{P}$ have already been instantiated, finding a value for a third variable x_{k+1} is always possible: it amounts to finding a value, in $\mathcal{P}'_d$, for a node $(x_{k+1}, x_j), j = [1..k]$. A possible backtrack-free instantiation procedure for deriving the solutions of $\mathcal{P}$ would be as follows:

1. Choose a value X_1 of x_1 that satisfies $R(x_1)$
2. For $i \leftarrow 2$ to n do
3. $\quad I_i \leftarrow \bigcap_{j:1..i-1} \prod_{x_i} R(xi, X_1, X_j)$
4. $\quad X_i \leftarrow$ choose a value for x_i in I_i

4 Partial and directional convexity properties

Constraint convexity is a rather strong condition, but it turns out that weaker forms of convexity are often sufficient to satisfy the conditions of a globally consistent labeling.

Partially convex binary CCSPs We introduce in [4, 5] a new category of partial convexity called (x)-*convexity* for binary relations. This property is more restrictive than path and simple connectivity but guarantees that convexity is maintained while enforcing relational consistency.

Definition 13. ((x)-Convexity [4])
Let R be a binary relation defined by a set of algebraic or transcendental constraints on two variables x_1, x_2. R is said to be x_k-convex in the domain D_{x_k} if for any two points q_1 and q_2 of r such that the segment $\overline{q_1 q_2}$ is parallel to x_k, $\overline{q_1 q_2}$ is entirely contained in r.

A network is said to be (x)-convex if each of its relations $R(x, y)$ is (x)-convex.

(x)-convexity guarantees the convexity of any unary projection of a given relation [5]. This allows the formulation of the following result [4].

Theorem 14. *A binary CCSP which is (x)-convex and path-consistent is minimal and decomposable*

The (x)-convexity property is non-conservative with respect to intersection. Hence, the global consistency property stated by Theorem 14 is only guaranteed for the *a posteriori* network computed by path-consistency.

Directional (x)-convexity In the case where a discrete network does not satisfy the row-convexity property, van Beek shows that directional row-convexity remains a useful property for obtaining backtrack-free solutions. Similar results generalizes to the case of (x)-convex relations. The following Theorem states [4] that a partial (x)-convexity of the network is sufficient to ensure that a solution can be determined without backtracking.

Theorem 15. *Let $\mathcal{N}$ be a path-consistent binary constraint network. If there exists an ordering of the variables $x_1, \ldots x_n$ such that each relation $R(x_i, x_j)$ of $\mathcal{N}$ with $1 \leq j \leq i$, is (x_i)-convex, then a consistent instantiation can be found without backtracking.*

Directional row-convexity imposes ordering conditions on both variables and variable domains. Since discrete CSPs do not have the strictly ordered domains characterizing continuous CSPs, the fact that a constraint network is row-convex can sometimes be hidden. This is obviously not the case for continuous CSPs concerning (x)-convexity.

Partially convex n-ary relations Similarly to the case of binary constraints, a less restrictive convexity condition can be defined for n-ary constraints. We first propose the following generalization of the (x)-convexity property:

Definition 16. ($(x_1, \ldots x_k)$-Convexity)
Let R be an n-ary relation between n variables $x_1 \ldots x_n$. R is said to be $(x_1, \ldots x_k)$-convex in the domains $D_1 \times \ldots D_{x_k}$ if for any two points q_1 and q_2 of r, such that the segment $\overline{q_1 q_2}$ is on a plane parallel to $x_1 \ldots \times x_k$, $\overline{q_1 q_2}$ is entirely contained in r

Informally, this means that a relation is $(x_1, \ldots x_k)$-convex if any sub-projection over the subset $(x_1, \ldots x_k)$ yields a convex k-ary region. In the case of networks of arity r, the composition of two maximal arity constraints having at least one variable in common, results in a relation of arity $2r-1$. In analogy to the case of ternary networks, we observe that the extension of an r-ary set of variables to a region of arity $r-1$ does not involve relations with arity greater than r. To apply Helly's Theorem we must introduce the notion of (r,r-1)-relational consistency which guarantees that each set of r relations having $r-1$ variables in common has a non-null intersection:

Definition 17. ((r,r-1)-relational consistency)
Let $\mathcal{P}$ be a network of relations over a set of variables X, of arity r. Let $R_{I_1}(x_1, y_1, ..., y_{r-1}), \ldots R_{I_r}(x_r, y_1, ..., y_{r-1})$ be r relations of N sharing the r-1 variables $\{y_1, \ldots y_{r-1}\}$. The relations are (r,r-1)-relationally consistent relative to the shared variables if and only if any consistent instantiation of the variables in $\{x_1, ..., x_r\}$ has an extension to $\{y_1, \ldots y_{r-1}\}$ that satisfies all relations simultaneously. The network $\mathcal{P}$ is relationally (r,r-1) consistent if and only if all relations are (r,r-1)-consistent with respect to all subsets of shared variables.

Hence, the following generalization of Theorem 14 can be proposed:

Theorem 18. *Let $\mathcal{P}$ be a constraint network of arity r at most, $(x_1, \ldots, x_{r-1})$-convex. If $\mathcal{P}$ is (r,r-1)-relationally consistent, then it is globally consistent.*

Proof. The proof is similar to the one given for Theorem 7. In the general case, the nodes in the binary dual representation of $\mathcal{P}$ represent $(r-1)$-ary subsets of $\mathcal{P}$'s variables.

Directional (x,y)-convexity In the proof of Theorem 7, convexity of the constraints is used only in the application of Helly's Theorem to ensure extensibility of partial solutions of the dual network. Here, it would be sufficient to have convexity hold only in two of the three dimensions involved in the constraint. Thus, we have the following Theorem:

Theorem 19. *Let $\mathcal{P}$ be a (3,2)-relationally consistent ternary constraint network. If there exists an ordering of the variables $x_1, \ldots x_n$ such that for any $i,j,k : 1 \leq i < j \leq k \leq n$, $R(x_i, x_j, x_k)$ is (x_j, x_k)-convex, then the network is globally consistent and a consistent instantiation can be found without backtracking.*

Sketch of proof. According to Helly's Theorem in two dimensions, the fact that each ternary relation $R(x_i, x_j, x_k)$ is (x_j, x_k)-convex guarantees that the binary relations $R(x_j, x_k)$ derived from the problem are non-empty ($R(x_j, x_k) = \bigcap_{j:i..k} \prod_{x_i} R(x_i, x_j, x_k)$). By construction, these binary relations are convex and have non-null pairwise intersections. Consequently, a similar argument as the one given for the proof of Theorem 7 hold and instantiation can be carried out backtrack-free □.

5 Exploiting convexity in practice

In discrete domains, relations are represented simply as enumerations of values or value combinations. In continuous domains, sets of individual values are often compact and can be represented by one or a small collection of intervals. However, representing and manipulating labels of *several variables*, as it is necessary for implementing algorithms for higher degree of consistency than two is more involved as they may be complex geometric shapes.

Constraint representation In [4], we propose to represent numerical constraints using 2^k-trees (a hierarchical representation of space commonly used in vision and spatial reasoning [7]). The 2^k-trees representation of constraints is based on the observation that in most practical applications each variable takes its values in a bounded domain (bounded interval) and there exists a maximum precision with which results can be used.

Provided that these two assumptions hold, a relation defined by inequalities can be approximated by carrying out a hierarchical binary decomposition of its solution space into 2^k-trees (quadtrees for binary relations, octrees for ternary ones etc...) (see Figure 4). In order to provide a unified framework for handling both inequalities and equalities, we propose in [4] to translate equalities into a weaker form called *toleranced equalities* [6]: the final grey nodes of the 2^k-tree decomposition for an equality constraint are replaced by white nodes. This amounts to replacing each equality by two inequalities close to each other and is acceptable in practice as long as a the results can be identified with a limited precision.

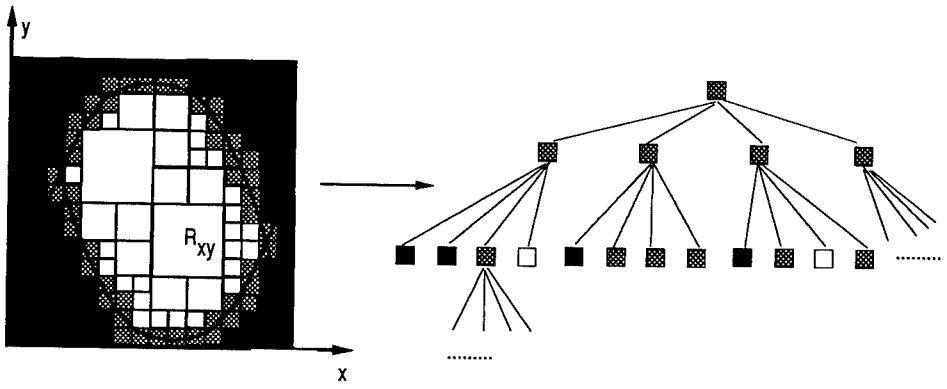


Fig. 4. *A continuous relation can be approximated by carrying out a hierarchical binary decomposition of its solution space into a 2^k-tree where: white nodes represent completely legal solution regions, grey nodes partially legal and partially illegal regions and black nodes completely illegal ones*

Using this discretized representation, the sets of feasible value combinations can be interpreted and manipulated explicitly. The main advantage is that the use of complex numerical tools for solving *sets* of simultaneous constraints (stated implicitly by their mathematical expressions) can be avoided. The approximate solution region defined by a set of several constraints is constructed by projecting, composing and intersecting their individual 2^k-tree representations. These operations on 2^k-trees are easy to implement and extensively studied in computer vision, computer graphics and image processing. Moreover, constructing the 2^k-tree representation for a single constraint only requires evaluating an individual constraint equation at certain points in the space [6]. As we show in [4, 5], the explicit handling of solution regions using 2^k-trees conveys a simple implementation for path- and higher degrees of consistency in continuous domains.

Correctness of the representation A 2^k-tree representation can be interpreted as providing two different approximations for a feasible region:

- the inner content approximation, $\mathcal{I}(\mathcal{S})$, is given by the white nodes (interior nodes) and is entirely enclosed within the solution space. Since all values within $\mathcal{I}(\mathcal{S})$ are consistent, it is a *sound* approximation. However, some solutions maybe missing from this representation,
- the closest outer content approximation, $\mathcal{O}(\mathcal{S})$, is given by the union of the white and grey nodes (interior nodes $\cup$ boundary nodes). This approximation is guaranteed to contain all solutions, but it may be *not sound* since the grey nodes contain inconsistent values.

With regard to equalities, remember that this method only allows toleranced equalities and soundness will only hold with respect to these tolerances. For the

initial 2^k-tree representations of *individual* constraints, we can always guarantee that $\mathcal{I}(\mathcal{S})$ and $\mathcal{O}(\mathcal{S})$ are as close as possible to the actual solution region. However, constructing total constraints and enforcing consistency involves composition and intersection operations on constraints.

Letting $\mathcal{S}_1 \oplus \mathcal{S}_2$ denote the solution space resulting from the intersection of $\mathcal{S}_1$ and $\mathcal{S}_2$. We can show the following properties (see [5]):

- The inner content approximation is exact with respect to intersection: $\mathcal{I}(\mathcal{S}_1) \oplus \mathcal{I}(\mathcal{S}_2) = \mathcal{I}(\mathcal{S}_1 \oplus \mathcal{S}_2)$
- The outer content approximation may contain spurious nodes after intersection: $\mathcal{O}(\mathcal{S}_1 \oplus \mathcal{S}_2) \subseteq \mathcal{O}(\mathcal{S}_1) \oplus \mathcal{O}(\mathcal{S}_2)$

For composition, it is possible to show that the projection of a constraint into a higher-dimensional space is exact for both inner and outer approximations. Therefore, the $\mathcal{I}(\mathcal{S})$ representation of total constraints is exact, even after executing consistency algorithms. This means that it is both sound (not containing any spurious inconsistent values) as well as maximal in the sense that there is no larger sound $\mathcal{I}(\mathcal{S})$ approximation, for a given precision. On the other hand, the $\mathcal{O}(\mathcal{S})$ representation computed by logical combination of simultaneous constraints is complete but not sound with respect to the minimal enclosing approximation $\mathcal{O}(\mathcal{S})$ — spurious grey nodes can be created by intersections.

2^k-trees and convexity Since the 2^k-tree decomposition generates stepwise approximations of the boundaries, convexity is obviously not preserved in the strict mathematical sense. In [5], we show that when the resolution chosen is insufficient, situations may occur where a connected solution space is represented by disconnected or even empty $\mathcal{I}(S)$ representation. However, these limitations are compensated by the fact that:

- the $\mathcal{I}(S)$ representation of a convex solution space can be empty or disconnected only when the solution of the CCSP falls within the limit of resolution chosen for the 2^k-tree representation. This situation is therefore restricted to limit cases,
- when the $\mathcal{I}(S)$ representation of a convex solution space is disconnected, a single additional level of decomposition is then sufficient to make the representation connected again.

Hence, if a disconnection occurs (limit cases), it is consequently possible either to resort to further refinements of the quadtrees or to neglect the solution region within the disconnected area—considering that its identification requires a precision having no significance for the application. Moreover, a particular class of minimal convexity deficiencies can be identified which precludes the risk of disconnection (see [5]). The 2^k-trees having minimal deficiencies of this type are said to be convex.

Checking for convexity When constraints are approximated using quadtrees, we show in [5] that the (x)-convexity property can checked for in $O(Nlog_4(N)+$

$2^{D_y/\varepsilon}$) where N is the number of feasibility nodes, D_y is the domain size of variable x (i.e. interval length) and ε the minimal interval length of x in the quadtree decomposition. Similarly, convexity can be checked for in $O(2.Nlog_4(N)+2^{\frac{D}{\varepsilon}+1})$ where N is the number of feasibility nodes, D is the maximal domain size in the quadtree (i.e. interval length) and ε the minimal interval length in the quadtree decomposition (for a fixed precision, this complexity is $O(Nlog_4(N))$. Finally, using analog procedures, the (x_1, x_2)-convexity property, useful for solving ternary problems, can be checked for in $O(N + Nlog_4(N))$, for a fixed precision. These simple convexity checking procedures examine exhaustively the boundary nodes of the 2^k-tree representations. For detailed descriptions we refer the reader to [5].

Comparison with the discrete case— 2^k-trees and backtrack-free search It is worth mentioning that the results on n-ary constraints (see section 3) are not directly transferable to discrete domains. Ensuring backtrack-free search in ternary constraints requires convexity conditions to hold in $\Re^2$ rather that $\Re$. We have shown that (3,2)-relational consistency is equivalent to global consistency, but the backtrack-free instantiation might require refining the resolution of different variables. This is possible in continuous domains, but not possible if we use a continuous domain to represent a discrete problem.

Consider the following example where two matrices representing discrete relations are understood as showing convex solution regions. When we intersect the two regions, the result is:

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \oplus \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \longrightarrow \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

i.e. the intersection has been lost as it is smaller than the resolution limit. In a continuous problem, we can now refine the resolution to make this problem go away, and continue with the instantiation. But in a discrete problem, we do not have this possibility as the maximum resolution is fixed. A path-consistent labeling does not guarantee that we can in fact successfully complete a backtrack-free instantiation without need for increasing the resolution, and hence does not guarantee global consistency in a discrete problem where this possibility does not exist.

On the other hand, extending the row-convexity property to 2D, so that Helly's Theorem can ensure the binary extensibility condition, would dictate that each discrete ternary relation yields a universal matrix (with 1s only) as binary projection, which is probably too restrictive for practical use.

6 Example

We now sketch out how the introductory example of Figure 1 is solved using our method. In this example, four main independent variables, beam depth(H_b), slab

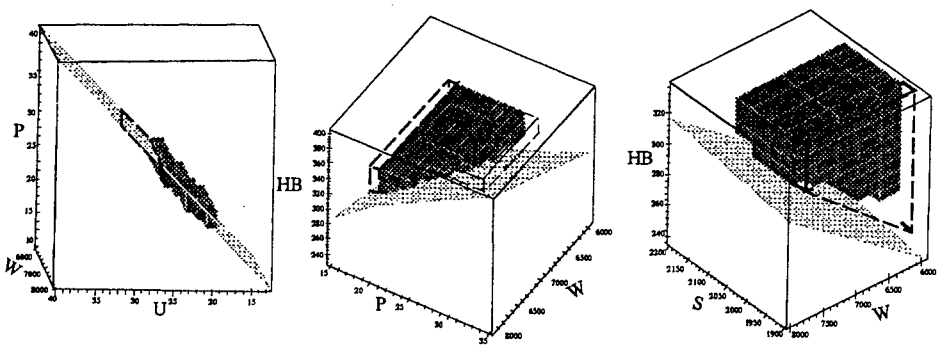


Fig. 5. *Three constraints from the steel structure problem. The areas within the dashed lines are those removed while enforcing global consistency*

thickness(H_s), beam span(W) and beam spacing(S) are linked together through the following non-linear constraints:

$$H_s > 137.70 - 8633.10^{-3}S + 5511.10^{-8}S^2 - 835810^{-10}S^3$$

$$H_b > 41.383\left(\frac{1.1 * 2.35 * p * W^2}{10^6 C * f_y}\right)^{0.3976}$$

$$p = 1.3\left[\frac{(P_{sw(slab)} + P_{dl}) * S}{1000} + 0.000074\left(\frac{W}{24}\right)^{1.5161}\right]$$

$$H_b > \left[\frac{P_{llst} * S}{1000} \cdot \frac{D * 350 * W^3}{0.1545 * 384 * E_s}\right]^{0.2839}$$

Traditionally, an engineer works through these equations hierarchically; at no time is the complete solution set known. Exploration of possible solutions is done point by point according to the experience of the designer.

Using our system, a prototype Lisp implementation calculates the globally consistent solutions in $\sim$ 1800 seconds on a Silicon Graphics Indigo with an R4000 processor. Figure 5 shows a set of constraints derived from the problem after global relaxation: the areas within the dashed lines are those removed by (3,2)-relational consistency. (u is an intermediate variable used when transforming the original problem into a ternary one, $u = 318.10^{-8}H_s * S + 0.0054$).

This shows that the problem admits in fact a large space of potential solutions, of which current mathematical methods only find a single one.

7 Conclusion

Convexity has been shown to be a useful property for efficiently solving binary discrete and temporal constraint satisfaction problems. In this paper, we

propose a generalization of these results to continuous constraints of arbitrary arities. While in discrete domains it has been shown that the generalization of the results on convexity to ternary (and higher arity) constraints may pose complexity problems, we introduce a concept of (3,2)-relational consistency which can be computed in polynomial time and proven equivalent to global consistency for constraint networks containing ternary constraints as well. Since n-ary constraint problems can always be transformed into equivalent ternary ones, these results guarantee polynomial-time solution for a large class of continuous n-ary problems. We also show how these results can be exploited in practice. The applicability condition of these results is that a limited precision must exist under which the results have no significance. This condition holds for almost all the engineering applications manipulating physical entities.

8 Acknowledgment

We would like to thank Sylvie Boulanger (Steel Structure lab., EPFL) for interesting discussions related to civil engineering, Peter van Beek (University of Alberta, CA), and Gaston Gonnet (ETHZ) for many helpful comments on this work, as well as the Swiss National Science Foundation for sponsoring this research under contract No.5003-034269.

References

1. Dechter R. From local to global consistency. In *Proceedings of the 8th Canadian Conference on AI*, 1990.
2. Dechter R., Meiri i., and Pearl J. Temporal constraint networks. In *Artificial Intelligence 49(1-3)*, 1990.
3. Faltings B. Arc consistency for continuous variables. In *Artificial Intelligence 65(2)*, 1994.
4. Haroud D. and Faltings B.V. Global consistency for continuous constraints. In Alan Borning, editor, *Lecture notes in computer Science 874: Principles and Practice of constraint programming.* Springer Verlag, 1994.
5. Sam-Haroud D. *Constraint consistency techniques for continuous domains.* PhD thesis, Ecole Polytechnique Fédérale de Lausanne, Switzerland, 1995.
6. Sam-Haroud D. and Faltings B.V. Consistency techniques for continuous constraints. In *Constraints Journal, 1*, 1996.
7. Samet H. *Applications of spatial data structures — Computer graphics, image processing and GIS.* Addison-Wesley Publishing Company, 1993.
8. van Beek P. On the minimality and decomposability of constraint networks. In *Proceedings of the 10th National Conference on AI*, 1992.
9. van Beek P. and Dechter R. On the minimality and global consistency of row convex constraint networks. In *Journal of the ACM*, 1995.

An Experimental Comparison of Three Modified DeltaBlue Algorithms

Tetsuya Suzuki, Nobuo Kakinuma, and Takehiro Tokuda*

Dept. of Comp. Science, Tokyo Inst. of Tech.
Ohokayama, Meguro, Tokyo 152, Japan
{tetsuya, kakinuma, tokuda}@cs.titech.ac.jp
tel: +81-3-5734-3213 fax: +81-3-5734-2912
http://tokuda-www.cs.titech.ac.jp/

Abstract. We present an experimental comparison of three modified DeltaBlue algorithms for local-propagation-based constraint solving. Our three modified methods are respectively called DeltaDown method, DeltaUp method and DeltaCost method. These methods were designed to speed up the planning phase or the evaluation phase of the original DeltaBlue method using additional cost functions to break a tie of the walkabout strength. Our cost functions are respectively called up cost and down cost. These cost functions can give us information about the upstream and the downstream constraints. Our experiments show that DeltaUp method brings us a considerable improvement of the total performance of DeltaBlue method using a small overhead of keeping the cost function.

Keywords: Constraint solving algorithm, Local propagation, DeltaBlue method, DeltaDown method, DeltaUp method, and DeltaCost method

1 Introduction

DeltaBlue algorithm is a widely used constraint solving method based an local propagation [1-6]. The purpose of this paper is to present an experimental comparison of three modified DelataBlue algorithms.

These modified algorithms are respectively called DeltaDown method, DeltaUp method and DeltaCost method. These methods were designed to hopefully speed up the planning phase or the evaluation phase of DeltaBlue method without using expensive bookkeeping operations. Our experiments show that the planning phase of DeltaUp method works at least as fast as that of DeltaBlue method and that DeltaUp method brings us a considerable improvement of the total performance of DeltaBlue method.

The organization of the rest of this paper is as follows. In Section 2 we review basic definitions of constraint problems and DeltaBlue method . In Section 3 we present our three modified DeltaBlue methods. In Section 4 we give comparisons of our methods with DeltaBlue method. In Section 5 we give our conclusion.

* Corresponding author

2 Preparation

2.1 Definitions

We summarize basic definitions for constraint problems and constraint solving methods based on local propagation.

A *constraint problem* P consists of a set of variables *VSET*, a set of constraints *CSET*, and a set of methods *MSET*.

A *constraint* C of *CSET* is a relation of values of variables V_1, ..., V_n of *VSET*. A constraint C has a priority level, which is an integer from 0 to k. A priority level is also called the strength of a constraint. We assume smaller integers have stronger priorities. A constraint of priority level 0 is called a required constraint. Required constraints must be satisfied in any constraint problem. A stay constraint is a constraint to keep the same value. An input constraint is a constraint that the value is given as an input value from outside.

A *method* M for a constraint C is a function to realize the constraint relation of C such that some of variables of C are output variables and the rest of variables of C are input variables. The number of output variables is called output degree of the method. For each constraint we may have one or more methods. In what follows, we deal with methods whose output degree is exactly one.

We can represent a constraint C of variables V_1, ..., V_n by an undirected hyperedge connecting points respectively representing variables V_1, ..., V_n. We can also represent a method M of C by a directed hyperedge where we add an outgoing arrow into the output variable to an undirected hyperedge of C.

A *constraint graph* of a constraint problem P is an undirected hypergraph such that points represent variables of P and undirected hyperedges represent constraints of P. We show an example of such a constraint graph in Fig.1. This undirected hypergraph represents a constraint problem of four variables a, b, c and d such that $a + b = c$ and $c \times d = e$.

A *data-flow graph* of P is a directed hypergraph obtained from a constraint graph of P by replacing each undirected hyperedge of C by a directed hyperedge representing one method of C.

A solution graph of P is a data-flow graph of P such that there exist no directed cycles or no two arrows outgoing into one same point. In Fig.2 we show one solution graph for the constraint graph of Fig.1.

A locally-predicate-better solution of constraint problem P is a maximally better solution using the following better relation for solutions x and y of P.

A solution x is better than a solution y if and only if there exists i satisfying following conditions.

1. For constraints of levels 0 to i-1, solutions x and y satisfy same constraints.
2. For constraints of level i, constraints of y satisfying P is a proper subset of constraints of x satisfying P.

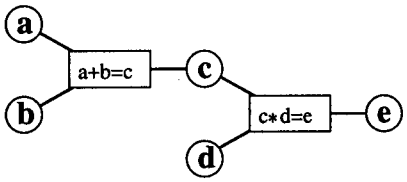


Fig. 1. A constraint graph

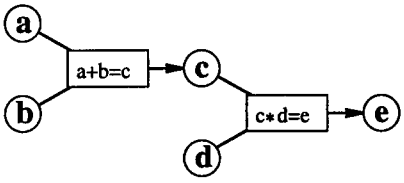


Fig. 2. A solution graph

2.2 DeltaBlue Method

DeltaBlue method is an incremental constraint solving method based on local propagation. DeltaBlue algorithm uses the walkabout strength to determine the direction of propagation. The walkabout strength is the weakest level of priorities of constraints existing in the upstream of the variable of a solution graph.

Definition 1 (Walkabout strength) The walkabout strength of a variable V is the weakest of walkabout strengths of input variables of the method whose output variables is V, and the level of priority of the constraint of that method. If such V has no input variables and no associated constraint, then the walkabout strength of V is the weakest priority level.

The outline of DeltaBlue method is as follows.

Method 1 (DeltaBlue method)
Input : A solution graph and an added constraint C.

1. We select a method of C whose output variable has the weakest walkabout strength and whose output variable is not yet used. Let the output variable of this method be V. If the walkabout strength of V is not weaker than the strength of C, then we are done. If C was required, then we also declare an error.
2. We record that C was enforced by the selected method and V was used.
3. We update the walkabout strength of V and its downstream variables. If we find any of variables consumed by C among the downstream variables, then we declare an error.
4. If a constraint D had previously determined V, then we retract D and attempt to enforce D by performing steps 1-3 on D. Otherwise we are done.

3 Three Modified Methods

DeltaBlue algorithm is an efficient constraint solving method using the walkabout strength. However in DeltaBlue algorithm, if walkabout strengths are all equal, then we have no further criterion. Fig.3 shows a situation where walkabout strengths are equal. Here priority levels are strong, medium, weak and weakest.

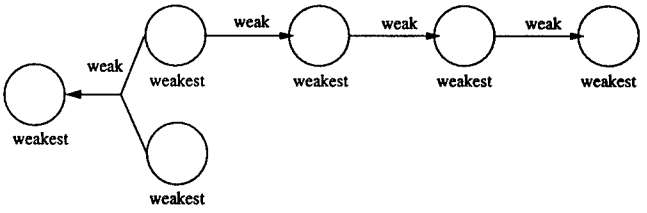


Fig. 3. A situation where walkabout strengths are equal

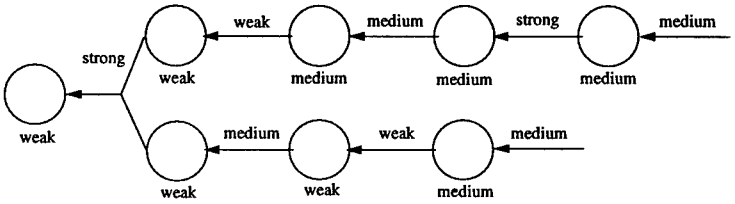


Fig. 4. Another tie situation of walkabout strengths

Fig.4 shows another tie situation of the walkabout strength. In order to choose a desirable direction to hopefully speed up the planning phase or the evaluation phase, we introduce two cost functions called down cost and up cost. The down cost of a variable is the number of methods existing in the downstream of the variable. The up cost of a variable is the number of methods which are to be selected when we determine the value of the variable by other method.

Definition 2 (Down cost) The down cost of a variable V is the sum of down costs of output variables of methods whose input variables have V, and the number of methods whose input variables have V.

Definition 3 (Up cost) The up cost of a variable V, whose value is determined by method M of constraint C, is the sum of the following values.

1. Let W be an input variable of M such that it has the weakest walkabout strength among input variables of M. If there exist two or more such input variables, then we choose an input variable having the smallest up cost U as W. The value is 0, if such W does not exist at all or the walkabout strength of W is not weaker than the strength of C. The value is U, otherwise.
2. The number of methods whose output variable is V.

In Fig.5 and 6 we show solution graphs with down costs and up costs of variables.

We can state DeltaDown method as follows. Our DeltaDown method was designed to reduce the number of recomputations of methods in the evaluation phase.

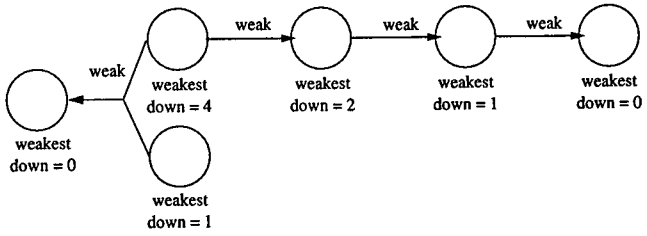


Fig. 5. A solution graph with down cost

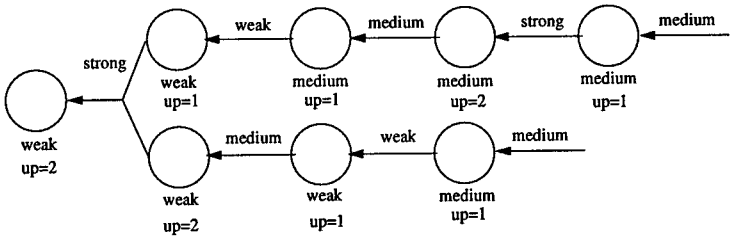


Fig. 6. A solution graph with up cost

Method 2 (DeltaDown method)
Input : A solution graph and an added constraint C.

1. We select a method of C whose output variable has the weakest walkabout strength and whose output variable is not yet used. If there exist two or more such methods, then we select a method which has the smallest down cost. Let the output variable of this method be V. If the walkabout strength of V is not weaker than the strength of C, then we are done. If C was required, then we also declare an error.
2. We record that C was enforced by the selected method and V was used.
3. We update the walkabout strength of V and its downstream variables. If we find any of variables consumed by C among the downstream variables, then we declare an error.
4. If a constraint D had previously determined V, then we retract D and attempt to enforce D by performing steps 1-3 on D. Otherwise we update the down cost of all the necessary variables.

In Fig.7 we show how DeltaDown method works for the solution graph of Fig.5.

We can state DeltaUp method as follows. Our DeltaUp method was designed to reduce the number of reselections of methods in the planning phase.

Method 3 (DeltaUp method)
Input : A solution graph and an added constraint C.

1. We select a method of C whose output variable has the weakest walkabout strength and whose output variable is not yet used. If there exist two or more such methods, then we select a method which has smallest up cost. Let the output variable of this method be V. If the walkabout strength of V is not weaker than the strength of C, then we are done. If C was required, then we also declare an error.
2. We record that C was enforced by the selected method and V was used.
3. We update the walkabout strength and up cost of V and its downstream variables. If we find any of variables consumed by C among the downstream variables, then we declare an error.
4. If a constraint D had previously determined V, then we retract D and attempt to enforce D by performing steps 1-3 on D. Otherwise we are done.

In Fig.8 we show how DeltaUp method works for the solution graph of Fig.6.

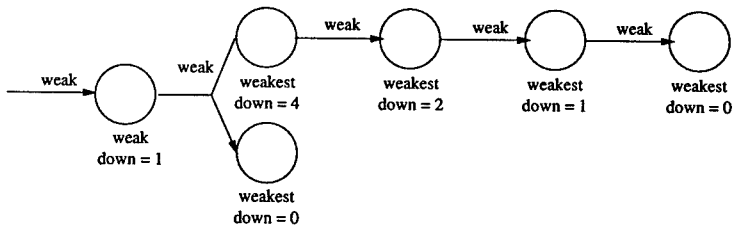


Fig. 7. DeltaDown method

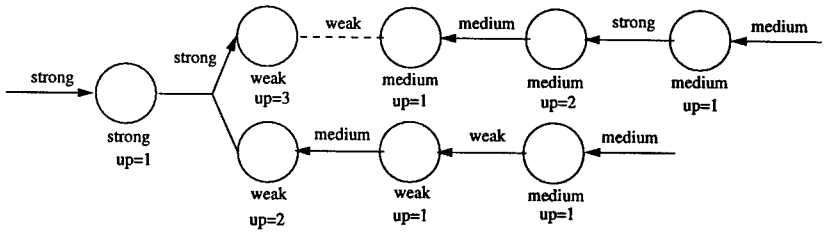


Fig. 8. DeltaUp method

Finally we state DeltaCost method which uses the sum of down cost and up cost as the cost function. Our DeltaCost method was designed to reduce the sum of the number of reselections of methods in the planning phase and the number of recomputations of methods in the evaluation phase.

Method 4 (DeltaCost method)
Input : A solution graph and an added constraint C.

1. We select a method of C whose output variable has the weakest walkabout strength and whose output variable is not yet used. If there exist two or more such methods, then we select a method which has the smallest sum of up cost and down cost. Let the output variable of this method be V. If the walkabout strength of V is not weaker than the strength of C then we are done. If C was required, then we also declare an error.
2. We record that C was enforced by the selected method and V was used.
3. We update the walkabout strength and up cost of V and its downstream variables. If we find any of variables consumed by C among the downstream variables, then we declare an error.
4. If a constraint D had previously determined V, then we retract D and attempt to enforce D by performing steps 1-3 on D. Otherwise we update the down cost of all the necessary variables.

4 Comparison

For a constraint problem such that the number of constraints is N and the maximum number of methods for a constraint is M, the complexity of the planning phase of four constraint solving methods are as follows.

DeltaBlue method	$O(MN)$
DeltaDown method	$O(MN^2)$
DeltaUp method	$O(MN)$
DeltaCost method	$O(MN^2)$

Hence we need to measure the overhead of keeping down cost and up cost in the planning phase as well as the total performance of these methods.

First we used a linear chain constraint problem of Fig. 9 to measure the overhead time in the planning phase. Because the linear chain constraint problem has no ambiguity from a viewpoint of the walkabout strength, this case is considered to be the worst case for our modified methods. In Fig.10 we measured the time to create a long linear chain from left to right. In Fig.11 we measured the time to add an input constraint to the left end of a chain. In Fig.12 we measured the time to remove the input constraint from a chain. These results show that DeltaUp method works at least as efficiently as DeltaBlue method during the planning phase in spite of the worst bookkeeping overhead.

Secondly we used a binary tree maintenance problem of Fig.13 to measure the total performance. In Fig.14 we measured the time of nine different types of operations to maintain a binary tree. Operations from number 1 to 5 are for binary tree creation. Operations from number 6 to 7 are for binary tree moving. Operations from number 8 to 9 are for binary tree swapping. In Table 1 and 2 we respectively show the number of reselections of methods in the planning phase and the number of recomputations of methods in the evaluation phase.

These results show that DeltaUp method achieves a considerable improvement of total time in comparing with DeltaBlue method.

Our interpretation of the performance of DeltaUp method is as follows.

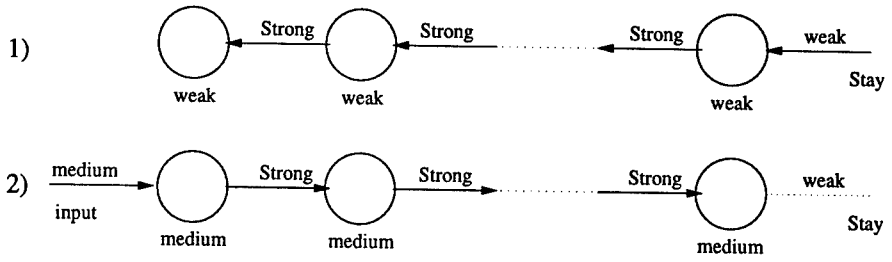


Fig. 9. A long chain of constraints

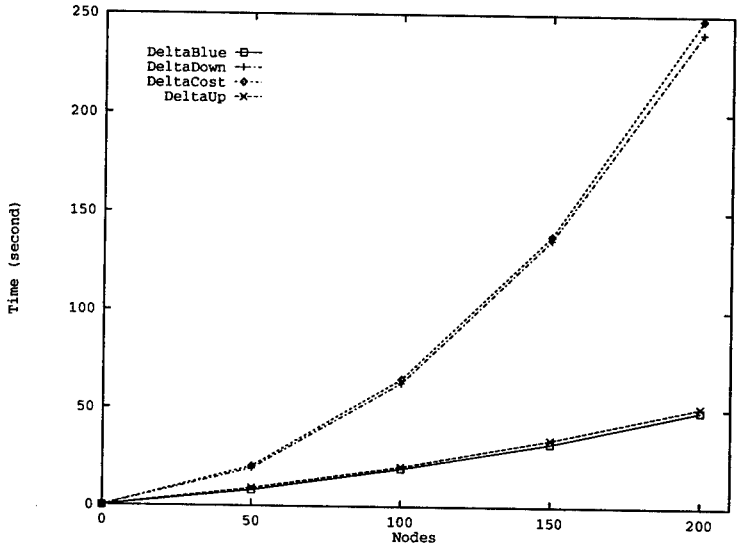


Fig. 10. Time to create a long chain

1. When constraint problems have ambiguity from a viewpoint of the walkabout strength, the variance of the number of reselections of methods in the planning phase of DeltaBlue method and the variance of the number of recomputations of methods in the evaluation phase of DeltaBlue method tend to get larger.
2. When constraint problems have ambiguity from a viewpoint of the walkabout strength, DeltaUp method reduces both the number of reselections of methods in the planning phase and the time for the planning phase, even if we include the overhead time.The time for the evaluation phase may decrease

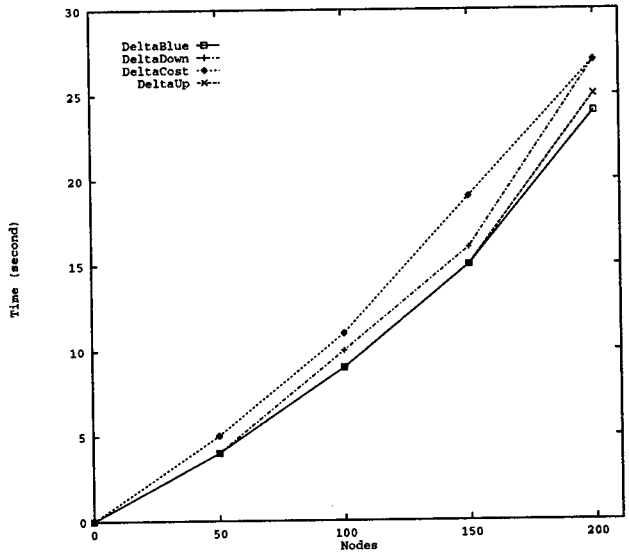


Fig. 11. Time to add an input constraint to a chain

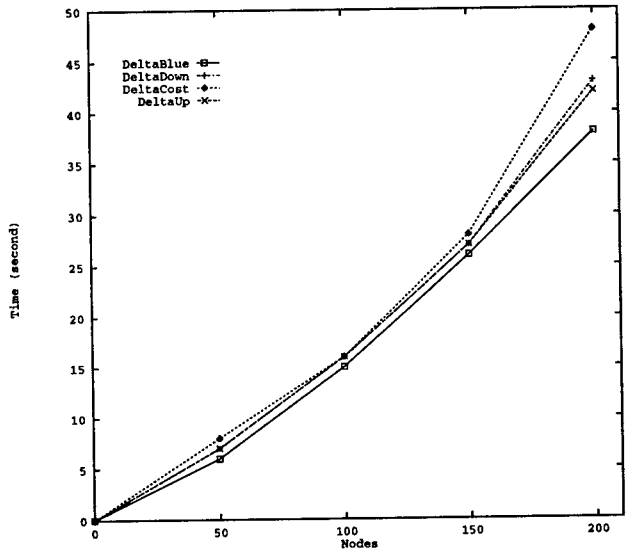


Fig. 12. Time to delete an input constraint from a chain

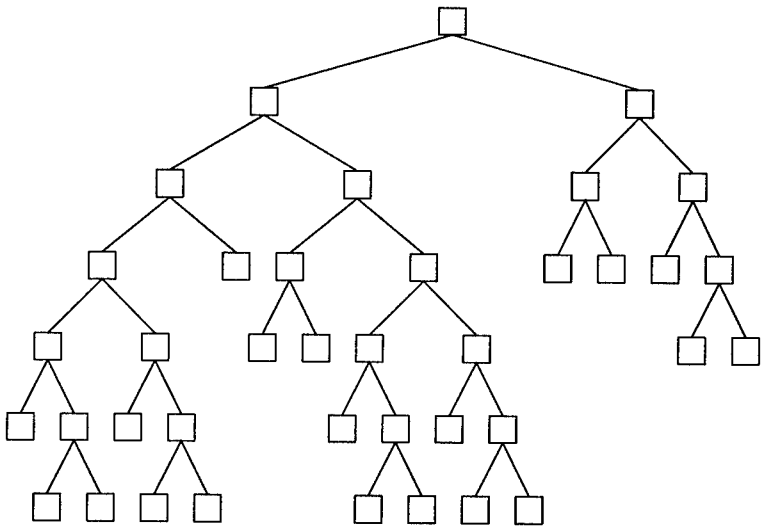


Fig. 13. A binary tree maintenance problem

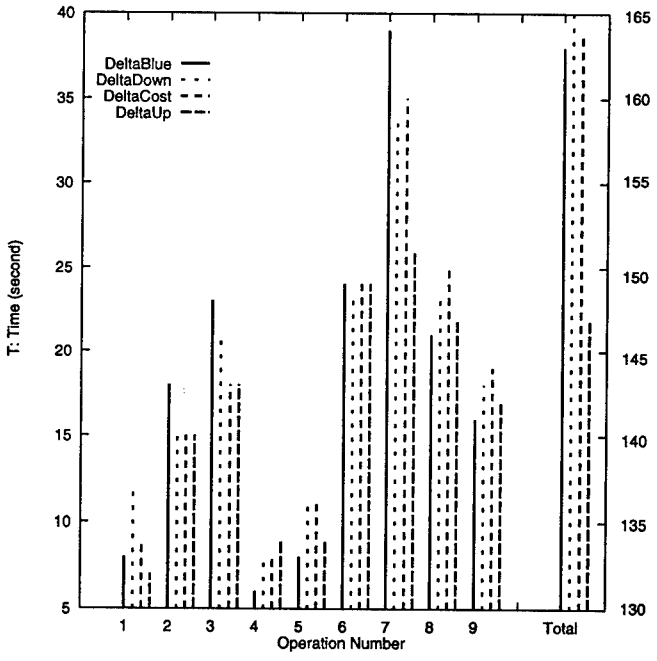


Fig. 14. Performance of four constraint solving methods

Table 1. The number of reselections of methods in the planning phase

Operation number	1	2	3	4	5	6	7	8	9
DeltaBlue	39	58	73	11	18	23	47	7	16
DeltaDown	47	63	80	11	18	25	37	7	16
DeltaCost	42	62	76	11	18	25	37	7	16
DeltaUp	34	49	57	8	18	13	23	4	11

Table 2. The number of recomputations of methods in the evaluation phase

Operation number	1	2	3	4	5	6	7	8	9
DeltaBlue	79	162	207	65	86	230	255	248	176
DeltaDown	90	98	151	65	86	230	243	248	176
DeltaCost	56	88	105	65	86	230	243	248	176
DeltaUp	70	125	156	88	140	230	241	250	190

or may increase.The reduction of the time for the planning phase contributes to the reduction of the total execution time.

5 Conclusion

We have presented three modified DeltaBlue algorithms to speed up the planning phase or the evaluation phase. DeltaUp method brings us a considerable improvement of the performance of DeltaBlue method using a small bookkeeping overhead. DeltaDown method and DeltaCost method work more slowly than DeltaBlue method. This is because they use the down cost and increase the time for the planning phase more than they reduce the time for the evaluation phase.

References

1. Borning, A. H.: *ThingLab – A Constraint-Oriented Simulation Laboratory.* PhD Dissertraiton, Stanford University(March 1979).
2. Borning, A. H., Duisberg R., Freeman-Benson B., Kramer A., and Woolf, M.: Constraint Hierarchies, Proceedings of the 1987 ACM Conference on Object-Oriented Programming Systems, Languages and Applications, pp. 48-60, 1987.
3. Freeman-Benson, B. N. , Maloney, J. and Borning, A.: An Incremental Constraint Solver, Communications of the ACM, 33:1, 54-63, 1990.
4. Leler, W.: Constraint Programming Languages, Addison-Wesley, 1988.
5. Maloney, J.H.: Using Constraints for User Interface Construction, PhD thesis 91-08-12, Department of Computer Science and Engineering FR-35, University of Washington, 1991.
6. Sannella M.: The SkyBlue Constraint Solver, Technical Report 92-07-02, Department of Computer Science and Engineering, University of Washington, 1993.

Constraint Logic Programming over Unions of Constraint Theories

Cesare Tinelli and Mehdi Harandi

Department of Computer Science
University of Illinois at Urbana-Champaign
1304 W. Springfield Ave.
Urbana, IL 61801 – USA
{tinelli,harandi}@cs.uiuc.edu
tel: +1(217)333-5821
fax: +1(217)333-3501

Abstract. In this paper, we propose an extension of the Jaffar-Lassez Constraint Logic Programming scheme that operates with unions of constraint theories with different signatures and decides the satisfiability of *mixed* constraints by appropriately combining the constraint solvers of the component theories. We describe the extended scheme and provide logical and operational semantics for it along the lines of those given for the original scheme. Then we show how the main soundness and completeness results of Constraint Logic Programming lift to our extension.

Keywords: Constraint Logic Programming, combination of satisfiability procedures.

1 Introduction

The Constraint Logic Programming scheme was originally developed in [8] by Jaffar and Lassez as a principled way to combine the two computational paradigms of Logic Programming and Constraint Solving. The scheme extends conventional Logic Programming by replacing the notion of *unifiability* with that of *constraint solvability* over an underlying constraint domain. As originally proposed, the CLP scheme extends immediately to the case of multiple constraint domains as long as they do not share function or predicate symbols. The scheme though does not account for the presence of *mixed* terms, terms built with functors from different signatures, and corresponding mixed constraints. The reason for this is that, although the CLP scheme allows in principle multiple, separate constraint theories, each with its own constraint solver, it is not designed to operate on their *combination*, to which mixed constraints belong.

This paper proposes an extension of the scheme to include constraint domains built as the combination of a number of independent domains, such as, for instance, the domains of finite trees and real numbers, the domains of lists, strings, and integers, and the like.

In principle, we can always instantiate the CLP scheme with a suitable constraint domain once we have a constraint solver for it, no matter whether the

domain is simple or "composite". For composite constraint domains however it is desirable not to have to build a solver from scratch if a constraint solver is already available for each component domain.

A lot of research has been done in recent years on domain combination (see, for instance [2, 3, 13, 16]) although most of the efforts have been concentrated on unification problems and equational theories (see [1, 4, 5, 6, 10, 14, 19], among others). The results of these investigations are still limited in scope and a deep understanding of many model- and proof-theoretic issues involved is still out of reach. In spite of that, we try to show the effectiveness of combination techniques by choosing one of the most general and adapting it so that it can be incorporated in the CLP scheme with few modification of the scheme itself.

1.1 Notation and Conventions

We adhere rather closely to the notation and definitions given in [15] for what concerns mathematical logic in general and [9] for what concerns constraint logic programming in particular. We report here the most notable notational conventions followed. Other notation which may appear in the sequel follows the common conventions of the two fields.

In this paper, we use v, x, y, z as meta-variables for the logical variables, s, t for first-order terms, p, q for predicate symbols, f, g for function symbols, a, b, h for atoms, A for a multi-set of atoms, c, d for constraints, C for a multi-set of constraints, φ, ψ for first order formulas, and ϑ for a value assignment, or valuation, to a set of variables. Some of these symbols may be subscripted or have an over-tilde which will represent a finite sequence. For instance, $\tilde{x}$ stands for a sequence of the form $(x_1, x_2, \ldots, x_n)$ for some natural number n. When convenient, we will use the tilde notation to denote sets of symbols (as opposed to sequences). Where $\tilde{s}$ and $\tilde{t}$ have both length n, the equation $\tilde{s} = \tilde{t}$ stands for the system of equations $\{s_1 = t_1 \wedge \cdots \wedge s_n = t_n\}$.

In general, $var(\varphi)$ is the set of φ's free variables. The shorthand $\exists_{-\tilde{x}} \varphi$ stands for the existential quantification of all the free variables of φ that are not contained in $\tilde{x}$, while $\vec{\exists} \varphi$ stands for the existential closure of φ.

We will identify union of multi-sets of formulas with their logical conjunction. We will also identify first-order theories with their deductive closure. Where $\mathcal{S}, \mathcal{T}$ are sets of Σ-sentences, for some signature Σ, $Mod(\mathcal{T})$ is the set of all the Σ-models of $\mathcal{T}$. The notation $\mathcal{T} \models \varphi$ means that $\mathcal{T}$ logically entails the universal closure of φ, while $\mathcal{S}, \mathcal{T} \models \varphi$ stands for $\mathcal{S} \cup \mathcal{T} \models \varphi$.

We will say that a formula φ is *satisfiable in* a theory $\mathcal{T}$ iff there exists a model of $\mathcal{T}$ that satisfies $\vec{\exists} \varphi$.

If P is a CLP program we will denote with P^* its Clark completion.

1.2 Organization of the Paper

In Section 2, we briefly describe the original CLP scheme and motivate the need for constraints with mixed terms, which the CLP scheme does not explicitly

accommodate. In Section 3, we mention a method for deriving a satisfiability procedure for a combination of theories admitting mixed terms from the satisfiability procedures of the single theories. In Section 4, we explain how one can use the main idea of that combination method to extend the CLP scheme and allow composite constraint domains and mixed terms over them. In Section 5 we prove some soundness and completeness results for the new scheme. In Section 6, we summarize the main contribution of this paper, outlining directions for further development.

2 The CLP Scheme

A thorough description of CLP($\mathcal{X}$), the CLP scheme, can be found in [9]. We recall here that, in essence, an instance of CLP($\mathcal{X}$) is obtained by assigning the parameter $\mathcal{X}$ with a quadruple that specifies the chosen constraint domain, its axiomatization, and the features of the constraint language. More specifically, $\mathcal{X} := \langle \Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T} \rangle$ where Σ is a signature, $\mathcal{D}$ is a Σ-structure representing the constraint domain over which computation is performed, $\mathcal{L}$ is the constraint language, that is, the class of Σ-formulas used to express the constraints, and $\mathcal{T}$ is a first-order Σ-theory (with equality) describing the relevant properties of the domain. A number of assumptions are generally made about $\mathcal{X}$. The most important are:

- Σ contains the equality symbol which $\mathcal{D}$ interprets as the identity in the underlying domain.
- $\mathcal{L}$ contains an identically true and an identically false predicate and a set $\mathcal{L}_P$ of *primitive constraints*.
- $\mathcal{L}$ is closed under variable renaming and logical conjunction.
- $\mathcal{D}$ and $\mathcal{T}$ *correspond* on $\mathcal{L}$, that is, $\mathcal{D}$ is a model of $\mathcal{T}$ and every formula of $\mathcal{L}$ that is satisfiable in $\mathcal{D}$ is satisfiable in every model of $\mathcal{T}$.

For some applications $\mathcal{T}$ is required to be *satisfaction complete* with respect to $\mathcal{L}$: for every $c \in \mathcal{L}$, either $\mathcal{T} \models \tilde{\exists}\, c$ or $\mathcal{T} \models \neg\tilde{\exists}\, c$.

Prolog itself can be seen as an instance of the CLP scheme, specifically as CLP($\mathcal{FT}$), where $\mathcal{FT}$ is the constraint domain of finite trees represented as first-order terms. Actually, all the CLP($\mathcal{X}$) systems in which $\mathcal{X}$ is not $\mathcal{FT}$ or an extension of it[1] still retain the possibility of building uninterpreted terms and so are at least CLP($\mathcal{FT}, \mathcal{X}$) systems. Furthermore, many systems support several constraint domains. They can be seen as CLP($\tilde{\mathcal{X}}$) systems, with $\tilde{\mathcal{X}} := \{\mathcal{X}_1, \ldots, \mathcal{X}_n\}$, where the $\mathcal{X}_i$'s are built over disjoint signatures and their constraints are processed by different, specialized solvers. In these systems, predicate or function symbols in one signature are applicable, with few exceptions, only to (non-variable) terms entirely built with symbols from the same signature.

Thus, although in one way or another all CLP systems use more than one constraint domain, they do not freely allow *mixed* terms or predicates, that is,

[1] PrologII, for instance, works with rational trees instead of finite trees.

expressions built with symbols from different signatures. Meaningful constraints over such heterogeneous expression, however, arise naturally in many classes of applications. An example of a constraint with mixed terms in the theory of lists and natural numbers is

$$v = (x + 9) :: y \ \wedge \ head(y) > z + u$$

where :: is the list constructor and *head* returns the first element of a list. An example, adapted from [12], in the theory of finite trees and real numbers, is

$$f(f(x) - f(y)) \neq f(z) \ \wedge \ y + z \leq x \ .$$

Proper instances of CLP($\mathcal{X}$) cannot deal with these types of constraints simply because the CLP computational paradigm does not consider them. In the rest of this paper, we will show a method for extending the CLP scheme to a new scheme, MCLP($\vec{\mathcal{X}}$), that offers a systematic and consistent treatment of constraints with mixed terms. Specifically, we will show how to convert a CLP($\vec{\mathcal{X}}$) system into a MCLP($\vec{\mathcal{X}}$) system which operates on the constraint structure generated by a suitable combination of the various $\mathcal{X}_i$'s.

3 Combining Satisfiability Procedures

The main idea of our extension is to adapt and use in CLP a well-known method, originally proposed by Nelson and Oppen [13], for combining first-order theories and their satisfiability procedures. Although the method applies to the combination of any finite number of theories, for simplicity we will consider the case of just two theories here.

Definition 1. We say that a formula is in *simple Conjunctive Normal Form* if it is a conjunction of literals. Given a Σ-theory $\mathcal{T}$, we will denote with $sCNF(\mathcal{T})$ the set of simple Conjunctive Normal Form Σ-formulas.

Definition 2. A consistent theory Σ-$\mathcal{T}$ is called *stably-infinite* iff any quantifier-free Σ-formula is satisfiable in $\mathcal{T}$ iff it is satisfiable in an infinite model of $\mathcal{T}$.

Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be two stably-infinite theories with respective signatures Σ_1, Σ_2 such that $\Sigma_1 \cap \Sigma_2 = \emptyset$.[2] The simplest combination of $\mathcal{T}_1$ and $\mathcal{T}_2$ is the $(\Sigma_1 \cup \Sigma_2)$-theory $\mathcal{T}_1 \cup \mathcal{T}_2$ defined as (the deductive closure of) the union $\mathcal{T}_1$ and $\mathcal{T}_2$.

If for each $i = 1, 2$ we have a procedure Sat_i that decides the satisfiability in $\mathcal{T}_i$ of the formulas of $sCNF(\mathcal{T}_i)$, we can generate a procedure that decides the satisfiability in $\mathcal{T}_1 \cup \mathcal{T}_2$ of any formula $\varphi \in sCNF(\mathcal{T}_1 \cup \mathcal{T}_2)$ by using Sat_1 and Sat_2 modularly. Clearly, because of the expanded signature, φ cannot in general be processed directly by either satisfiability procedure, unless it is of the form $\varphi_1 \wedge \varphi_2$—call it *separate form*—where φ_i is a (possibly empty) formula of $sCNF(\mathcal{T}_i)$.[3] In that case, each Sat_i will process φ_i separately.

[2] We consider the equality symbol "=" as a logical constant.

[3] Since φ_i does not contain symbols from the other signature, we say that it is *pure*.

If φ is not already in separate form, it is always possible to apply a procedure that, given φ, returns a formula $\tilde{\varphi}$ which is in separate form and is satisfied exactly by the same models and variable assignments that satisfy φ.[4] Some examples of formulas and their separate forms are given in Figure 1.

$$v = (x + 9) :: y \wedge head(y) > z + u \quad (1)$$
$$f(f(x) - f(y)) \neq f(z) \wedge y + z \leq x \quad (2)$$

$$(v = x_1 :: y \wedge head(y) = x_2) \wedge (x_1 = x + 9 \wedge x_2 > z + u) \quad (3)$$
$$(f(x_1) \neq f(z) \wedge x_2 = f(x) \wedge x_3 = f(y)) \wedge (x_1 = x_2 - x_3 \wedge y + z \leq x) \quad (4)$$

Fig. 1. Formulas (3) and (4) are separate forms of formulas (1) and (2), respectively.

A description of the Nelson-Oppen combination procedure is not necessary here. For our present purposes it is enough to say that global consistency between the separate satisfiability procedures is achieved by propagation, between the procedures, of the entailed equalities between the variables of the input formula. That this approach is correct and sufficient is justified by the model-theoretic result given in the following.

Definition 3. If P is any partition on a set of variables V and R is the corresponding equivalence relation, we call *arrangement of V (determined by P)* the set $ar(V)$ made of all the equations between any two equivalent words of V and all the disequations between any two non-equivalent words. Formally, $ar(V) := \{x = y \mid x, y \in V \text{ and } xRy\} \cup \{x \neq y \mid x, y \in V \text{ and not } xRy\}$.

Proposition 4 [18]. *Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be as above. Consider $\varphi_1 \in sCNF(\mathcal{T}_1)$, $\varphi_2 \in sCNF(\mathcal{T}_2)$ and let $\tilde{x} := var(\varphi_1) \cap var(\varphi_2)$. Then, $\varphi_1 \wedge \varphi_2$ is satisfiable in $\mathcal{T}_1 \cup \mathcal{T}_2$ iff there exists and arrangement $ar(\tilde{x})$ such that $\varphi_1 \wedge ar(\tilde{x})$ is satisfiable in $\mathcal{T}_1$ and $\varphi_2 \wedge ar(\tilde{x})$ is satisfiable in $\mathcal{T}_2$.*

4 Extending CLP($\mathcal{X}$)

We want to extend the CLP scheme so that we go from a language of type CLP($\tilde{\mathcal{X}}$), where $\mathcal{X} := \{\mathcal{X}_1, \ldots, \mathcal{X}_n\}$ is a set of signature-disjoint constraint structures, to a language of type MCLP($\tilde{\mathcal{X}}$), where $\tilde{\mathcal{X}}$ is a combination of the previous

[4] The gist of the separation procedure is to repeatedly replace in φ each term t of the "wrong" signature with a new variable and add the equation $x = t$ to φ. More details can be found in [17].

structures in the sense that it allows any computation performed in CLP($\bar{\mathcal{X}}$) and, furthermore, poses no signature restriction on term construction.

The reason why we are interested in combinations of satisfiability procedures is that CLP systems already utilize *separate* satisfaction procedures, the constraint solvers, to deal with the various constraint theories they support and so already have a main module to drive the goal reduction process and control the communication with the solvers.

Intuitively, if we rewrite MCLP($\bar{\mathcal{X}}$) statements in a separate form similar to that mentioned earlier, we may be able to use the various constraint solvers much the same way the Nelson-Oppen combination procedure uses the various satisfiability procedures. Moreover, the machinery we will need for a MCLP($\bar{\mathcal{X}}$) system will be essentially the same we would need for a corresponding CLP($\bar{\mathcal{X}}$) system. The only necessary addition, to realize the solvers combination, will be a mechanism for generating equations and disequations between variables shared by the different solvers and propagating them to the solvers themselves. More precisely, we will need a procedure that, each time a new constraint is given to one solver, (a) identifies the variables that that constraint shares with those in the other solvers, (b) creates a backtrack point in the computation and chooses a (novel) arrangement of those variables, and (c) passes the arrangement to all the other solvers.

We clarify and formalize all this in the following subsections.

4.1 The Extended Scheme

The first issue we are confronted with in extending the CLP scheme is the impossibility of fixing a single domain of computation. Recall that the CLP scheme puts primacy on a particular structure which represents the intended constraint domain. The combination procedure we are considering, however, combines *theories*, not structures[5]: it succeeds when the input formula is satisfiable in *some* model of the combined theory. For this reason, our extension will use as "constraint domain" a whole class of structures instead of a single one. In this respect, our scheme is actually a restriction of the Höhfeld-Smolka constraint logic programming framework [7]. The restriction is achieved along two dimensions: the constraint language and the set of solution structures. We use only sCNF formulas as constraints and elementary classes[6] as the class of structures over which constraint satisfiability is tested. In particular, the class associated to a given MCLP($\bar{\mathcal{X}}$) language is the set of the models of the union of the component theories.

Formally, the constraint structure $\bar{\mathcal{X}}$ for the MCLP($\bar{\mathcal{X}}$) scheme is defined as the tuple

$$\bar{\mathcal{X}} := \langle\langle \Sigma_1, \mathcal{T}_1\rangle, \ldots, \langle \Sigma_n, \mathcal{T}_n\rangle\rangle$$

[5] For some very recent work on the combination of structures, see [3].

[6] Recall that a class of structures is called *elementary* if it coincides with the set of models of some first-order theory.

where $\Sigma_1, \ldots, \Sigma_n$ are pairwise disjoint signatures and $\mathcal{T}_i$ is a stably-infinite Σ_i-theory, for each $i \in \{1, \ldots, n\}$. The constraint theory for MCLP($\bar{\mathcal{X}}$) is $\mathcal{T} := \mathcal{T}_1 \cup \cdots \cup \mathcal{T}_n$, the constraint language is $sCNF(\mathcal{T})$, and the set of solution structures is $Mod(\mathcal{T})$.

Next, we describe a logical and a top-down operational model of MCLP($\bar{\mathcal{X}}$) where we assume that, for each $\mathcal{T}_i$, a solver Sat_i is available that decides satisfiability of sCNF formulas in $\mathcal{T}_i$.

4.2 Logical Semantics

The format of MCLP($\bar{\mathcal{X}}$) statements is identical to that of CLP statements except that mixed constraints are allowed with no restrictions. As a consequence, MCLP($\bar{\mathcal{X}}$) adopts CLP($\mathcal{X}$)'s logical semantics for both its programs and their completion. The only difference concerns the notation used to describe MCLP($\bar{\mathcal{X}}$) programs.

Since we want to apply the available solvers modularly, it is convenient to look at each MCLP($\bar{\mathcal{X}}$) statement as if it had first been converted into an appropriate separate form. After we define the computation transitions, the careful reader will observe that it is not necessary to actually write MCLP($\bar{\mathcal{X}}$) programs in separate form because a separation procedure can be applied on the fly during subgoal expansion. We use the separate form here simply for notational convenience. Specifically, instead of a standard CLP rule of the form, $p(\tilde{x}) \leftarrow B$, as defined in [9] for instance, we consider the *separate form*

$$p(\tilde{x}) \leftarrow \ddot{B}$$

obtained by applying the procedure mentioned in Section 3 to the body of the rule[7]. Analogously, instead of a goal G we consider the corresponding goal $\ddot{G}$.

It should be clear that, under the CLP logical semantics, a MCLP($\bar{\mathcal{X}}$) statement and its separate form are equivalent.

4.3 Operational Semantics

We will only consider the case of two component theories here, as the n-component case is an easy generalization.

As with CLP($\mathcal{X}$), computation in MCLP($\bar{\mathcal{X}}$) can be described as a sequence of state transitions. Each state in turn is described by either the symbol *fail* or a tuple of the form $\langle A, C_1, C_2 \rangle$ where A is a set of pure atoms and constraints, and for $i = 1, 2$ C_i, the constraint store, is a set of Σ_i-constraints.[8] We assume the presence of a function, *select*, that when applied to the first element A of

[7] This is always possible as the body of a MCLP($\bar{\mathcal{X}}$) rule is a sCNF formula.

[8] For simplicity, we have decided to ignore the issue of delayed constraints here. We would like to point out however that our extension could be easily applied with comparable results to an operational model including delayed constraints such as the one described in [9].

a transition returns a member of A non-deterministically. State transitions are defined as follows.

1. $$\langle A, C_1, C_2\rangle \rightarrow_r \langle A \cup B - a(\tilde{x}), C_1', C_2'\rangle$$

where $a(\tilde{x}) := \mathit{select}(A)$ is an atom, the program P contains the rule $a(\tilde{y}) \leftarrow B$, and $C_i' := C_i \cup \tilde{x} = \tilde{y}$ for $i = 1, 2$.

2. $$\langle A, C_1, C_2\rangle \rightarrow_r \mathit{fail}$$

where $a(\tilde{x}) := \mathit{select}(A)$ is an atom and no rule in P has a as the predicate symbol of its head.

3. $$\langle A, C_1, C_2\rangle \rightarrow_c \langle A - c, C_1', C_2'\rangle$$

where $c := \mathit{select}(A)$ is a constraint literal and, for $i = 1, 2$, $C_i' := C_i \cup c$ if c is a Σ_i-constraint, $C_i' = C_i$ otherwise.

4. $$\langle A, C_1, C_2\rangle \rightarrow_s \langle A, C_1', C_2'\rangle$$

where $ar(\tilde{v})$ is an arrangement of the variables shared by C_1 and C_2 and $C_i' := C_i \cup ar(\tilde{v})$ and $Sat_i(C_i)$ succeeds for $i = 1, 2$.

5. $$\langle A, C_1, C_2\rangle \rightarrow_s \mathit{fail}$$

where $ar(\tilde{v})$ is an arrangement of the variables shared by C_1 and C_2, $C_i' := C_i \cup ar(\tilde{v})$ for $i = 1, 2$, and either of $Sat_1(C_1)$ or $Sat_2(C_2)$ fails.

Similarly to CLP($\mathcal{X}$), transitions of type $\rightarrow_r$ are just goal reduction steps. The difference here is that the variable equalities produced by matching the selected predicate with the head of some rule go to both constraint solvers as, by definition, an equality predicate with variable arguments belongs to both $\mathcal{L}_1$ and $\mathcal{L}_2$.

Transitions of type $\rightarrow_c$ feed the constraint solvers with a new constraint, where each constraint goes to the relative solver (variable equalities going to both solvers).

Transitions of type $\rightarrow_s$ differ more significantly from the corresponding transitions in CLP($\mathcal{X}$) as they actually implement, in an incremental fashion, the combination procedure. They deserve a more detailed explanation then.

In terms of the procedure, for every $\rightarrow_s$ transition, we consider the constraint stores C_1 and C_2 as the i-pure halves of sCNF formulas whose satisfiability must be checked. For each constraint store, we use the constraint solver "as is" but we make sure that *global consistency* information is shared by the two solvers by choosing an arrangements of the variables they have in common. It goes without saying that, like the transitions of type $\rightarrow_r$, transitions of type $\rightarrow_s$ are non-deterministic. With the first type, the choice is among the possible reductions of the selected predicate, with the second, it is among the possible arrangements of the shared variables. In actual implementations of the scheme then, backtracking

mechanisms similar to those used for $\rightarrow_r$ transitions must be used. For space limitations we cannot include a more complete discussion on the implementation of $\rightarrow_r$ transitions and its various possible optimization. The interested reader is again referred to [17].

The concepts of derivation, final state, fair/failed/successful derivation, answer constraint, and computation tree can be defined analogously to the corresponding CLP($\mathcal{X}$) concepts (see [9]).

5 Computational Properties of MCLP($\bar{\mathcal{X}}$)

To discuss the main computational properties of MCLP($\bar{\mathcal{X}}$) it is necessary to specify a more detailed operational semantics than the one given in the previous section. Since any implementation of MCLP($\bar{\mathcal{X}}$) is a deterministic system, a particular computation rule has to be defined. For us, this amounts to specifying the behavior of the *select* function and the order in which the various types of transitions are applied. We will need to further restrict our attention to specific classes of MCLP($\bar{\mathcal{X}}$) systems to prove some of the properties of MCLP($\bar{\mathcal{X}}$).

Definition 5. Let $\rightarrow_{cs}$ be the two-transition sequence $\rightarrow_c\rightarrow_s$. We say that a MCLP($\bar{\mathcal{X}}$) system is *quick-checking* if all of its derivations are sequences of $\rightarrow_r$ and $\rightarrow_{cs}$ transitions only.

A quick-checking CLP($\mathcal{X}$) system verifies the consistency of the constraint store immediately after it modifies it. Analogously, a quick-checking MCLP($\bar{\mathcal{X}}$) system verifies the consistency of the union of all the constraint stores (by means of equality sharing among the solvers) immediately after it modifies at least one of them.

Definition 6. A MCLP($\bar{\mathcal{X}}$) system is *ideal* if it is quick-checking and uses a fair computation rule.[9]

In both schemes, we can define the concept of finite failure if we restrict ourselves to the class of ideal systems. We say that a goal G is *finitely failed* for a program P if, in any ideal system, every derivation of G in P is failed.

5.1 Comparing CLP($\mathcal{X}$) with MCLP($\bar{\mathcal{X}}$)

To show that the main soundness and correctness properties of the CLP schema lift to our extension, we will consider, together with the given MCLP($\bar{\mathcal{X}}$) system, a corresponding CLP($\mathcal{X}$) system that, while accepting the very same programs, supports the combined constraint theory directly (i.e., with a single solver) and show that the two systems have the same computational properties.

[9] This definition differs from that given in [9] because we adopt a slightly different definition of derivation (see [17] for more details), but it refers to the same class of systems.

Actually, MCLP($\bar{\mathcal{X}}$) systems cannot have a *corresponding* CLP($\mathcal{X}$) system since the original scheme and ours define constraint satisfiability in a different manner. In CLP($\mathcal{X}$), the satisfiability test on the constraint store is successful if the store is satisfiable in the *fixed* structure $\mathcal{D}$. In MCLP($\bar{\mathcal{X}}$) instead, the satisfiability test is successful if the (union of all) constraint store(s) is satisfiable in *any* structure among those modeling the constraint theory. However, correspondence becomes possible if we "relax" the CLP($\mathcal{X}$) system by testing satisfiability within the class of structures $Mod(\mathcal{T})$, where $\mathcal{T}$ is the chosen constraint theory, instead of a single structure.

The relaxed CLP scheme is more general than the original scheme but less general than that proposed by Höhfeld and Smolka in [7]. In fact, its soundness is derivable as a consequence of the soundness of Höhfeld and Smolka's. Its completeness, however, cannot be derived from their scheme because it is essentially a consequence of the choice of a *first-order* constraint language and theory, which Höhfeld and Smolka do not require.

In Section 5.2, we will see that not only is the relaxed CLP scheme sound and complete, but also, and more importantly, it has logical properties no weaker than those exhibited by the CLP scheme.

It should not be difficult to see now that, once we have shown that CLP($\mathcal{X}$) maintains its nice properties even with a satisfiability test over an elementary class of structures, soundness and completeness results of MCLP($\bar{\mathcal{X}}$) easily follow—the intuitive justification being that it is immaterial whether we check for satisfiability in the union constraint theory utilizing a combined procedure or a non-combined one.

5.2 Soundness and Completeness of Relaxed CLP

We will now consider CLP($\mathcal{X}$) systems that are instances of the relaxed CLP scheme mentioned earlier. For these systems, the tuple $\mathcal{X}$ is defined as in Section 2 with the difference that $\mathcal{D}$ is replaced by $Mod(\mathcal{T})$, and the satisfiability test succeeds if and only if the input constraint is satisfiable in some element of $Mod(\mathcal{T})$.

Assuming a relaxed CLP($\mathcal{X}$) system with constraint language $\mathcal{L}$ and theory $\mathcal{T}$, we have formulated the following results after those given in [9] for CLP($\mathcal{X}$). For lack of space we forgo their proofs here; they are very similar, however, to those given in [8] and [11] for the corresponding CLP results and can be found in [17].

Proposition 7 Soundness. *Given a program P and a goal G:*

1. *If G has a successful derivation with answer constraint c, then $P, \mathcal{T} \models c \rightarrow G$.*
2. *When $\mathcal{T}$ is satisfaction complete wrt to $\mathcal{L}$, if G has a finite computation tree with answer constraints $c_1, \ldots, c_n$, then $P^*, \mathcal{T} \models G \leftrightarrow c_1 \vee \cdots \vee c_n$.*

Proposition 8 Completeness. *Given a program P, a simple goal G and a constraint c:*

1. *If $P, \mathcal{T} \models c \rightarrow G$ and c is satisfiable in $\mathcal{T}$, then there are $n > 0$ derivations of G with respective answer constraint $c_1, \ldots, c_n$ such that $\mathcal{T} \models c \rightarrow c_1 \vee \cdots \vee c_n$.*
2. *When $\mathcal{T}$ is satisfaction complete wrt to $\mathcal{L}$, if $P^*, \mathcal{T} \models G \leftrightarrow c_1 \vee \cdots \vee c_n$ then G has a computation tree with answer constraints $c'_1, \ldots, c'_m$ such that $\mathcal{T} \models c_1 \vee \cdots \vee c_n \leftrightarrow c'_1 \vee \cdots \vee c'_m$.*

In [8], Jaffar and Lassez show that Negation-as-Failure can be used correctly in their scheme provided that the constraint theory is satisfaction complete with respect to the constraint language. We discovered that we can still use negation as failure properly in the relaxed CLP scheme and, in addition, we do not need satisfaction completeness of the component theories at all. As before, a sufficient condition for this result is that we use a first-order constraint language.

Proposition 9 Soundness and Completeness of Negation-as-Failure. *In an ideal system, a goal G is finitely failed for a program P iff $P^*, \mathcal{T} \models \neg G$.*

5.3 Main Results

In the following, we consider a MCLP($\bar{\mathcal{X}}$) system, where $\bar{\mathcal{X}}$ is defined as in Section 4 but limited, again for simplicity, to the combination of only two stably-infinite theories. We will assume that, while the system satisfies the general implementation requirements given earlier, its computation rule is flexible enough with respect to the order in which the various transitions can be applied. Such assumption is not necessary for our results but makes their proofs easier and more intuitive.

We will also assume that programs and goals are all given in separate form. For notational ease, we will use $\rightarrow_{r/c}$ to denote either a $\rightarrow_r$ or a $\rightarrow_c$ transition. We start with some easy to prove lemmas.

Lemma 10. *If goal G has a successful derivation in a MCLP($\bar{\mathcal{X}}$) program P, then it has a successful derivation with the same answer constraint and such that all of its transitions are $\rightarrow_{r/c}$ transitions, except the last one which is a $\rightarrow_s$ transition.*

Essentially, the lemma states that a successful derivation can be always rearranged into a derivation of the form $\langle G, \emptyset, \emptyset \rangle \stackrel{*}{\rightarrow}_{r/c} \langle \emptyset, C_1, C_2 \rangle \rightarrow_s \langle \emptyset, C'_1, C'_2 \rangle$ by first reducing the goal to the empty set and then testing the consistency of the collected constraints.

Lemma 10 also entails that a successful derivation in MCLP($\bar{\mathcal{X}}$) is not just a finite derivation not ending with *fail*, but one whose answer constraint is satisfiable in the union theory. In fact, according to the MCLP($\bar{\mathcal{X}}$) operation model, a necessary condition for the above derivation to be successful is that C'_i be satisfiable in $\mathcal{T}_i$ for $i = 1, 2$. From Prop. 4 then, we can infer that $\exists_{-\mathrm{var}(G)}(C'_1 \wedge C'_2)$, the answer constraint of the derivation, is satisfiable in $\mathcal{T}_1 \cup \mathcal{T}_2$.

Lemma 11. *Given a MCLP($\bar{\mathcal{X}}$) program P and a goal G, if a derivation of the form $\langle G, \emptyset, \emptyset \rangle \stackrel{*}{\rightarrow}_{r/c} \langle \emptyset, C_1, C_2 \rangle$ exists in P, then $\mathcal{T}, P \models \exists_{-\mathrm{var}(G)}(C_1 \wedge C_2) \rightarrow G$.*

Proposition 12 Soundness of MCLP($\bar{\mathcal{X}}$). *Given a program P and a goal G,*

1. *If G has a successful derivation with answer constraint c, then $P, \mathcal{T} \models c \rightarrow G$.*
2. *When $\mathcal{T}$ is satisfaction complete wrt to $\mathcal{L}$, if G has a finite computation tree with answer constraints $c_1, \ldots, c_n$, then $P^*, \mathcal{T} \models G \leftrightarrow c_1 \vee \cdots \vee c_n$.*

Proof. Let $\tilde{x} := var(G)$.

1. By Lemma 10, we can assume without loss of generality that the derivation of G is of the form $\langle G, \emptyset, \emptyset \rangle \stackrel{*}{\rightarrow}_{r/c} \langle \emptyset, C_1, C_2 \rangle \rightarrow_s \langle \emptyset, C_1', C_2' \rangle$ where c is then $\exists_{-\tilde{x}} (C_1' \wedge C_2')$. Recalling the definition of $\rightarrow_s$ transitions, it is immediate that $\models c \rightarrow \exists_{-\tilde{x}} (C_1 \wedge C_2)$. The claim is then a direct consequence of Lemma 11.

2.[sketch] Let us call the MCLP($\bar{\mathcal{X}}$) system S and assume a corresponding relaxed CLP system S_{rel}. With no loss of generality we assume that, for each $i \in \{1, \ldots, n\}$, the derivation δ_i in S, having answer constraint c_i, is of the form $\langle G, \emptyset, \emptyset \rangle \stackrel{*}{\rightarrow}_{r/c} \langle \emptyset, C_{1i}, C_{2i} \rangle \rightarrow_s \langle \emptyset, C_{1i}', C_{2i}' \rangle$. Since S_{rel} has the same computational rule as S, for each δ_i there is a corresponding derivation $h(\delta_i)$ in S_{rel} of the form $\langle G, \emptyset, \emptyset \rangle \stackrel{*}{\rightarrow}_{r/c} \langle \emptyset, D_i \rangle \rightarrow_s \langle \emptyset, D_i \rangle$, where $D_i = C_{1i} \cup C_{2i}$.

Let $\sim$ be an equivalence relation over $\{1, \ldots, n\}$ such that $i \sim j$ iff δ_i and δ_j coincide up to the last transition. Notice that $h(\delta_i) = h(\delta_j)$ if $i \sim j$. Recalling the definition of $\rightarrow_s$ transitions, it should not be difficult to see that for each $j \in \{1, \ldots, n\}$, if $[j]$ is the equivalence class of j with respect to $\sim$, the following chain of logical equivalences holds,

$$\bigvee_{i \in [j]} c_i \leftrightarrow \bigvee_{i \in [j]} \exists_{-\tilde{x}} (C_{1i}' \wedge C_{2i}') \leftrightarrow \exists_{-\tilde{x}} (C_{1j} \wedge C_{2j}) \leftrightarrow \exists_{-\tilde{x}} D_j. \qquad (5)$$

By an analogous of Lemma 10 for the relaxed CLP scheme, we can show that the tree made by all the $h(\delta_i)$'s above is indeed the finite computation tree of G in S_{rel}. By the soundness of the relaxed CLP scheme, we then have that

$$P^*, \mathcal{T} \models G \leftrightarrow \bigvee_{j \in \{1, \ldots, n\}} \exists_{-\tilde{x}} D_j. \qquad (6)$$

The claim follows then immediately, combining (5) and (6) above. □

The following lemma is also easy to prove.

Lemma 13. *Consider a program P, a MCLP($\bar{\mathcal{X}}$) system S, and a corresponding relaxed CLP system S_{rel}. Then, for any transition t in S_{rel} of the form $\langle A, C \rangle \rightarrow_{r/c} \langle A', C' \rangle$ there is a transition t' in S of the form $\langle A, C_1, C_2 \rangle \rightarrow_{r/c} \langle A', C_1', C_2' \rangle$ such that $\mathcal{T} \models C \leftrightarrow C_1 \wedge C_2$ implies $\mathcal{T} \models C' \leftrightarrow C_1' \wedge C_2'$.*

Proposition 14 Completeness of MCLP($\bar{\mathcal{X}}$). *Given a MCLP($\bar{\mathcal{X}}$) system, a program P, a simple goal G and a constraint c,*

1. *If $P, \mathcal{T} \models c \rightarrow G$ and c is satisfiable in $\mathcal{T}$, then there are $n > 0$ derivations of G with respective answer constraint $c_1, \ldots, c_n$ such that $\mathcal{T} \models c \rightarrow c_1 \vee \cdots \vee c_n$.*

2. *When $\mathcal{T}$ is satisfaction complete wrt to $\mathcal{L}$, if $P^*, \mathcal{T} \models G \leftrightarrow c_1 \vee \cdots \vee c_n$, then G has a computation tree with answer constraints $c'_1, \ldots, c'_m$ such that $\mathcal{T} \models c_1 \vee \cdots \vee c_n \leftrightarrow c'_1 \vee \cdots \vee c'_m$.*

Proof: 1. Let us call the MCLP($\bar{\mathcal{X}}$) system S and assume a corresponding relaxed CLP system S_{rel}. Let $\tilde{x} := var(G)$. To simplify the notation, if δ is a successful derivation, we will denote its answer constraint by $ans(\delta)$.

Now, by the completeness of the relaxed CLP scheme, there exists a set D of successful derivations of G in S_{rel} such that $\mathcal{T} \models c \rightarrow \bigvee_{\delta \in D} ans(\delta)$. We show that, for each $\delta \in D$, there is a set D_δ of successful derivations of G in S such that $\mathcal{T} \models ans(\delta) \leftrightarrow \bigvee_{\gamma \in D_\delta} ans(\gamma)$. Then, the claim follows immediately by taking $c_1 \vee \cdots \vee c_n$ as $\bigvee_{\delta \in D}(\bigvee_{\gamma \in D_\delta} ans(\gamma))$.

Consider any $\delta \in D$. We generate a derivation δ' in S with initial state $\langle G, \emptyset, \emptyset \rangle$ such that δ' has a $\rightarrow_{r/c}$ transition for each $\rightarrow_{r/c}$ transition of δ in the way given in Lemma 13, and an *empty* transition for each $\rightarrow_s$ transition of δ. Using Lemma 13 and the fact that $\rightarrow_s$ transitions in S_{rel} preserve equivalence of the constraint stores, it is easy to show that if $\langle \emptyset, C \rangle$ is the final state of δ, then the last state of δ' has the form $\langle \emptyset, C_1, C_2 \rangle$ with $\mathcal{T} \models C \leftrightarrow C_1 \wedge C_2$.

We obtain the set D_δ mentioned above by completing δ' with one $\rightarrow_s$ transition from $\langle \emptyset, C_1, C_2 \rangle$ for each possible arrangement of the shared variables $\tilde{v}$ between C_1 and C_2 that is consistent with both stores. Observe that, since $C_1 \cup C_2$ is satisfiable, for being equivalent to the final constraint store of a successful derivation, we are guaranteed by Prop. 4 that at least one arrangement of $\tilde{v}$ is consistent with both C_1 and C_2 and, consequently, that D_δ is non-empty. It follows that, for every $\gamma \in D_\delta$, $\mathcal{T} \models ans(\gamma) \leftrightarrow \exists_{-\tilde{x}} (C_1 \wedge C_2 \wedge ar(\tilde{v}))$, for some arrangement $ar(\tilde{v})$.

Observing that the disjunction of all the arrangements of $\tilde{v}$ is a valid formula, it is then easy to deduce the following chain of logical equivalences in $\mathcal{T}$

$$\begin{aligned} \exists_{-\tilde{x}} C \leftrightarrow \exists_{-\tilde{x}} (C_1 \wedge C_2) &\leftrightarrow \exists_{-\tilde{x}} (C_1 \wedge C_2 \wedge \textstyle\bigvee_{ar(\tilde{v})} ar(\tilde{v})) \\ \leftrightarrow \textstyle\bigvee_{ar(\tilde{v})} \exists_{-\tilde{x}} (C_1 \wedge C_2 \wedge ar(\tilde{v})) &\leftrightarrow \textstyle\bigvee_{\gamma \in D_\delta} ans(\gamma) \end{aligned}$$

which concludes our proof.

2. The result follows as a consequence of the corresponding result for relaxed CLP and the construction in the proof of case 1 above. □

Observe that the necessity to consider multiple derivations to show the completeness of the system is not generated is already present in the CLP scheme itself. Our extension, however, may increase the number of necessary derivations because $\rightarrow_s$ transitions can generate multiple successful derivations, instead of just one, whenever more than one arrangement of variables is consistent with both the constraint stores.

By essentially the same arguments given for the relaxed CLP scheme, we can also prove soundness and completeness of Negation-As-Failure in MCLP($\bar{\mathcal{X}}$).

Proposition 15 Negation-as-Failure. *In an ideal MCLP($\bar{\mathcal{X}}$) system, a goal G is finitely failed for a program P iff $P^*, \mathcal{T} \models \neg G$.*

6 Conclusions

In this paper, we have described a way of extending the CLP($\mathcal{X}$) scheme to admit constraint theories generated as the union of several stably-infinite theories with pairwise disjoint signatures. The main idea of the extension is to incorporate in the scheme a well-known method of obtaining a satisfiability procedure for a union theory as the combination, by means of variable equality sharing, of the satisfiability procedures of each component theory.

By adopting a non-deterministic equality sharing mechanism, we have been able to prove that the main properties of our extension directly compare to those of the original scheme, provided that the CLP($\mathcal{X}$) consistency test on the constraint store is relaxed from satisfiability in a single structure to satisfiability in an elementary class thereof.

Specifically, we have first claimed that the relaxation of the satisfiability test (which gives rise to what we called a *relaxed CLP scheme*) does not modify the original soundness and completeness properties, even in the case of the negation-as-failure inference rule. Then, we have shown how the results given for the relaxed CLP scheme lift to our extension.

Finally, we would like to point out the advantages of adopting a non deterministic version of the original equality-sharing mechanism by Nelson and Oppen. On the theoretical side, our version fits rather nicely into the CLP scheme as it simply adds another level of "don't-know" non-determinism (corresponding to the choice of a variable arrangement) into the computational paradigm. On the practical side, where incremental solvers are already available for each constraint theory, not only does this scheme preserve their incrementality, a key computational feature for the implementation of any CLP system, but also allows one to use them as they are, with no modification whatsoever to their code or interface.

7 Acknowledgments

We would like to thank Michele Zito and Joshua Caplan for a number of discussions on some model-theoretic issues related to this work. This work is partially supported by grant DACA88-94-0014 from the US Army Construction Engineering Laboratories.

References

1. Franz Baader and Klaus U. Schulz. Unification in the union of disjoint equational theories: Combining decision procedures. In *Proceedings of the 11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 50–65. Springer-Verlag, 1992.
2. Franz Baader and Klaus U. Schulz. Combination of constraint solving techniques: An algebraic point of view. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, 1995.

3. Franz Baader and Klaus U. Schulz. On the combination of symbolic constraints, solution domains, and constraint solvers. In *Proceedings of the First International Conference on Principles and Practice of Constraint Programming, Cassis (France)*, September 1995.
4. Alexandre Boudet. Combining unification algorithms. *Journal of Symbolic Computation*, 16(6):597–626, December 1993.
5. E. Domenjoud, F. Klay, and C. Ringeissen. Combination techniques for non-disjoint equational theories. In A. Bundy, editor, *Proceedings 12th International Conference on Automated Deduction, Nancy (France)*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 267–281. Springer-Verlag, 1994.
6. A. Herold. Combination of unification algorithms. In J. Siekmann, editor, *Proceedings 8th International Conference on Automated Deduction, Oxford (UK)*, volume 230 of *Lecture Notes in Artificial Intelligence*, pages 450–469. Springer-Verlag, 1986.
7. Markus Höhfeld and Gert Smolka. Definite relations over constraint languages. *Journal of Logic Programming*, 1991.
8. Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. Technical Report 86/74, Monash University, Victoria, Australia, June 1986.
9. Joxan Jaffar and Michael Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
10. Hélène Kirchner and Christophe Ringeissen. A constraint solver in finite algebras and its combination with unification algorithms. In K. Apt, editor, *Proc. Joint International Conference and Symposium on Logic Programming*, pages 225–239. MIT Press, 1992.
11. Michael Maher. Logic semantics for a class of committed-choice programs. In Jean-Louis Lassez, editor, *ICLP'87: Proceedings 4th International Conference on Logic Programming*, pages 858–876, Melbourne, May 1987. MIT.
12. Greg Nelson. Combining satisfiability procedures by equality-sharing. *Contemporary Mathematics*, 29:201–211, 1984.
13. Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. on Programming Languages and Systems*, 1(2):245–257, October 1979.
14. Manfred Schmidt-Schauß. Combination of unification algorithms. *Journal of Symbolic Computation*, 8(1–2):51–100, 1989.
15. Joseph. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, MA, 1967.
16. Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31:1–12, 1984.
17. Cesare Tinelli. Extending the CLP scheme to unions of constraint theories. Master's thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, Illinois, October 1995.
18. Cesare Tinelli and Mehdi Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In *(to appear in) Proceedings of the 1st International Workshop "Frontiers of Combining Systems", Munich (Germany)*, Applied Logic. Kluwer, 1996.
19. K. Yelik. Unification in combinations of collapse-free regular theories. *Journal of Symbolic Computation*, 3(1–2):153–182, April 1987.

Analysis of Hybrid Systems in CLP($\mathcal{R}$)

Luis Urbina

Department of Electrical Engineering, University of Magdeburg, and
Department of Computer Science, University of Rostock, Germany

Abstract. This paper presents the formalization of the symbolic simulation and analysis technique for hybrid systems developed in [9, 7, 8]. The main advantage of this technique is the close relation between the hybrid–systems model *Hybrid Automata* [1, 8] and the execution model CLP($\mathcal{R}$) [3]. Our rule–based description is naturally suited for hybrid systems allowing (a) to lift CLP($\mathcal{R}$) definitions and results for the theory of hybrid systems, and therefore (b) to apply — in addition to forward/backward fixpoint computation and symbolic model–checking — CLP($\mathcal{R}$) intelligent search and backtracking procedures [2] in their analysis, since the depth–first search strategy of CLP($\mathcal{R}$) is incomplete on infinite trees. These techniques were implemented in part on top of the CLP($\mathcal{R}$) prototype system [4]. We illustrate our method with a variant of the reactor temperature control system from [1]. More realistic examples can be found in [9, 7, 8, 6].

1 Hybrid Systems

Hybrid systems [1, 8] describe the behavior of physical components that interact directly with an environment. Such systems usually consist of both a discrete component and an analog component. They model continuous state changes by means of differential equations, or inequations, over time intervals (analog behavior) as well as discrete state changes by means of nondeterministic guarded assignments that are instantaneous (discrete behavior). Typical examples of hybrid systems are real–time process–control systems such as automated factories or automated transportation systems. The correctness of such systems is more subtle and harder to verify than that of traditional systems because of their real–time aspect.

In this paper we show how a hybrid system H modeled by a hybrid automaton [1], Sect. 1.1, can be seen as a CLP($\mathcal{R}$) program [4], Sect. 1.2. We also describe some CLP($\mathcal{D}$) systems for evaluating some classes of hybrid systems, Sect. 2, allowing to verify reachability, safety, liveness, time–bounded and duration properties of H, written in the Integrator Computation Tree Logic (ICTL) [1], by applying top–down/bottom–up evaluation methods, symbolic model–checking and intelligent search and backtracking strategies, Sect. 3.

1.1 Hybrid Automata Representation

Informally, a *hybrid automaton* consists of a finite–state automaton, real–valued variables, discrete relations, activities, invariants and an initial condition imposed on the initial location. The finite–state automaton describes the system structure; the synchronization symbols together with the discrete relations describe the interaction of the system with its environment. The activities are conjunctions of differential (in)equations over the real–valued variables and describe the time–dependent changes of the system. The invariants are linear formulae permitting that the system remains in a location while its invariant is true. The initial condition describes the set of possible values for the system when it starts in the initial location.

Example 1 Reactor Temperature Control System. The system controls the temperature of a reactor core. It consists of three hybrid automata, of a reactor core controller CONTROLLER and two control rods ROD_i $(i = 1, 2)$ (cf Fig. 1). The real–valued variable θ describes the temperature. The goal is to keep θ between a minimal temperature θ_m and a maximal temperature θ_M. If θ reaches θ_M, then θ is to decrease by introducing one of the control rods into the reactor core. At the beginning θ is θ_m degrees and both control rods are outside of the reactor core. In this case θ raises according to the differential equation $\dot{\theta} = \frac{\theta}{10} + 50$, location '*no_rod*'. θ decreases according to the differential equations $\dot{\theta} = \frac{\theta}{10} - 56$ (location 'rod_1') and $\dot{\theta} = \frac{\theta}{10} - 60$ (location 'rod_2') depending on the control rod used. A control rod may be used again, if $T \geq 0$ time units have elapsed since it was last removed. If θ cannot decrease because no control rod is available, then a shutdown of the reactor is necessary. A shutdown of the system should be prevented. The value of the clock x_i represents the time having elapsed since the last use of the i–th control rod.

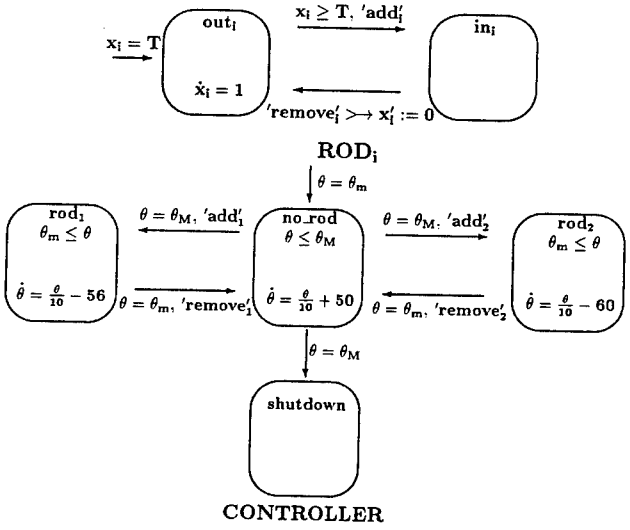


Fig. 1. ROD_1, ROD_2 and CONTROLLER for the temperature control system

Finite–state Automaton. The locations in ROD_i have the meaning: 'out_i' ('in_i')— i–th control rod is outside (inside) of the reactor core. The transitions have the meaning: (out_i, add_i, in_i) $((in_i, remove_i, out_i))$— i–th control rod is introduced (removed) into (from) the reactor core. In CONTROLLER the locations have the meaning: 'rod_i' — i–th control rod is inside of the reactor core; '*no_rod*' — there is no control rod inside of the reactor core; '*shutdown*' — a shutdown of the reactor occurs. The transitions have the meaning: (no_rod, add_i, rod_i) $((rod_i, remove_i, no\_rod))$ — i–th control rod is introduced (removed) into (from) the reactor core; $(no\_rod, \epsilon, shutdown)$ — shutdown of the reactor (ϵ is the *empty synchronization symbol*). 'out_i' and '*no_rod*' are the initial locations for ROD_i and CONTROLLER, respectively.

Data Variables. The real–valued variables $\mathbf{x} \equiv x_1, \ldots, x_n$ are called *data variables*. The natural number n is the *dimension* of the hybrid automaton. x_i and θ are data variables of ROD_i and CONTROLLER, respectively. In ROD_i T is a parameter. In CONTROLLER θ_m and θ_M are parameters as well.

Discrete Behavior. In addition, a hybrid automaton consists of *nondeterministic guarded assignments (ndga's)* of the form $\psi(\mathbf{x})$, $'a' \rightarrowtail \mathbf{x}' := \mathbf{f}(\mathbf{x})$, where $\psi(\mathbf{x})$ is a convex linear formula over $\mathbf{x}$, a is a synchronization symbol and $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), \ldots, f_n(\mathbf{x}))$ is a sequence of linear terms $f_i(\mathbf{x})$ over $\mathbf{x}$. A ndga in a transition (l, a, l') is instantaneous and may be taken when the convex linear formula $\psi(\mathbf{x})$ is true and the synchronization symbol a occurs. Then the assignments $\mathbf{x}' := \mathbf{f}(\mathbf{x})$ to the prime data variables $\mathbf{x}'$ are made. A ndga $\psi(\mathbf{x})$, $'a' \rightarrowtail \mathbf{x}' := \mathbf{f}(\mathbf{x})$ for a transition (l, a, l') generates a binary *discrete relation* $\alpha_{(l,a,l')}(\mathbf{x}, \mathbf{x}') \subseteq \mathbb{R}^n \times \mathbb{R}^n$ between the values before (nonprime data variables $\mathbf{x}$) and after (prime data variables $\mathbf{x}'$) the transition. In the graphical representation of hybrid automata empty synchronization symbols ϵ, true formulae and identity assignments are left out. In ROD_i, $x_i \geq T$, $'add_i' \rightarrowtail x_i' := x_i$ and *true*, $'remove_i' \rightarrowtail x_i' := 0$ are the ndga's for the transitions (out_i, add_i, in_i) and $(in_i, remove_i, out_i)$, respectively. In CONTROLLER, $\theta = \theta_M$, $'add_i' \rightarrowtail \theta' := \theta$, $\theta = \theta_m$, $'remove_i' \rightarrowtail \theta' := \theta$ and $\theta = \theta_M$, $'\epsilon' \rightarrowtail \theta' := \theta$ are the ndga's for the transitions (no_rod, add_i, rod_i), $(rod_i, remove_i, no\_rod)$ and $(no\_rod, \epsilon, shutdown)$, respectively. For the parameters T, θ_m and θ_M following identity assignments can be considered: $T' := T$, $\theta_m' := \theta_m$, and $\theta_M' := \theta_M$. The generated discrete relations have the form:

$$\alpha_{(out_i, add_i, in_i)}(x_i, x_i') \equiv x_i \geq T \wedge x_i' = x_i,\ \alpha_{(in_i, remove_i, out_i)}(x_i, x_i') \equiv x_i' = 0,$$
$$\alpha_{(no\_rod, add_i, rod_i)}(\theta, \theta') = \alpha_{(no\_rod, \epsilon, shutdown)}(\theta, \theta') \equiv \theta = \theta_M \wedge \theta' = \theta,$$
$$\alpha_{(rod_i, remove_i, no\_rod)}(\theta, \theta') \equiv \theta = \theta_m \wedge \theta' = \theta\ . \qquad (1)$$

Analog Behavior. For each location l there exists an invariant $\mathrm{Inv}(l)$ and an activity $\mathrm{Act}(l)$. An *invariant* is a convex linear formula over $\mathbf{x}$; an *activity* is a conjunction of differential (in)equations of the form $\dot{x} \# f(x)$, where $f(x)$ is a linear term, x is a data variable and $\# \in \{<, \leq, =, >, \geq\}$. A location describes the time–dependent actions of the system. Here the system may continuously change the values of $\mathbf{x}$ according to the differential (in)equations $\dot{x} \# f(x)$ if for these values the invariant is true. A *hybrid automaton* is *linear* if for each differential (in)equation $\dot{x} \# f(x)$, $f(x)$ is a constant term. Otherwise, the hybrid automaton is *non–linear*.

In the graphical representation of hybrid automata true formulae (invariants) and zero differential equations $\dot{x} = 0$ are omitted. For instance, in ROD_i all locations have the invariant *true* and the activities $\text{Act}(out_i) \equiv \dot{x}_i = 1$ and $\text{Act}(in_i) \equiv \dot{x}_i = 0$. In CONTROLLER the locations have the invariants $\text{Inv}(rod_i) \equiv \theta_m \leq \theta$, $\text{Inv}(no\_rod) \equiv \theta \leq \theta_M$ and $\text{Inv}(shutdown) \equiv true$, while they have the following activities: $\text{Act}(no\_rod) \equiv \dot{\theta} = \frac{\theta}{10} + 50$, $\text{Act}(rod_1) \equiv \dot{\theta} = \frac{\theta}{10} - 56$, $\text{Act}(rod_2) \equiv \dot{\theta} = \frac{\theta}{10} - 60$ and $\text{Act}(shutdown) \equiv \dot{\theta} = 0$. A parameter p can be considered as a data variable with differential equation $\dot{p} = 0$. CONTROLLER is a non–linear hybrid automaton, while both ROD_i are linear.

Initial Condition. An *initial condition* $\phi(\mathbf{x})$ is a convex linear constraint (a convex linear formula) over $\mathbf{x}$ imposed on the initial location. It describes the set of all possible initial values for $\mathbf{x}$ when the system starts at the initial location. In ROD_i and CONTROLLER $x_i = T$ and $\theta = \theta_m$ are their corresponding initial conditions.

1.2 CLP($\mathcal{R}$) Representation

Besides the hybrid automata representation hybrid systems can also be modeled as CLP($\mathcal{R}$) programs. Locations are interpreted as *predicates* over the data variables $\tilde{x} \equiv \mathbf{x}, time$, where $time \in \mathbb{R}^+$ is the non–negative *time variable* saving the time having elapsed since the system started. Discrete relations, invariants, activities and initial conditions are CLP($\mathcal{R}$) constraints. A hybrid system H consists of an *initial fact* and a set of *discrete* and *analog rules*. We now write for Example 1 the corresponding CLP($\mathcal{R}$) program. For the initial conditions $x_i = T$ and $\theta = \theta_m$ of the initial locations 'out_i' and 'no_rod' we set the initial facts as follows:

% Initial fact for ROD_i	% Initial fact for CONTROLLER
$out_i(x_i, time) \leftarrow x_i = T,\ time = 0.$	$no\_rod(\theta, time) \leftarrow \theta = \theta_m,\ time = 0.$

In the body of each initial fact we write the initial condition as a conjunction of convexed linear constraints. Each system starts at time 0. For each transition with corresponding discrete relation we set a *discrete rule* (cf Equation (1)):

% Discrete rules for ROD_i
$out_i(x_i, time) \leftarrow x_i \geq T,\ 'add_i',\ x_i' = x_i,\ in_i(x_i', time).$
$in_i(x_i, time) \leftarrow 'remove_i',\ x_i' = 0,\ out_i(x_i', time).$

% Discrete rules for CONTROLLER
$no\_rod(\theta, time) \leftarrow \theta = \theta_M,\ 'add_i',\ \theta' = \theta,\ rod_i(\theta', time).$
$rod_i(\theta, time) \leftarrow \theta = \theta_m,\ 'remove_i',\ \theta' = \theta,\ no\_rod(\theta', time).$
$no\_rod(\theta, time) \leftarrow \theta = \theta_M,\ '\epsilon',\ \theta' = \theta,\ shutdown(\theta', time).$

Time does not progress in discrete rules. Empty synchronization symbols and identity (copy) functions can be omitted. Moreover, if a prime variable x' is assigned a (linear) term t, in the *target location* l' x can be substituted for t and the assignment $x' = t$ can be discarded. Nonprime (prime) data variables $\tilde{x}$ ($\tilde{x}'$) in the *source* (*target*) *location* stand for any values in that location. $\tilde{x}$ ($\tilde{x}'$) variables stand for any values of $\tilde{x}$ ($\tilde{x}'$) before (after) the discrete rule has been applied. If we want to express a *parameterized hybrid system*, all parameters have to occur in each atom in the same position, such that they can always be copied by each rule.

Now, consider the differential (in)equation $\dot{x}_i \# f(x_i)$ for the data variable x_i in the location l. We assume $f(x_i)$ is integrable on the closed intervall $[0, \delta_l]$[1], where $\delta_l = time' - time \geq 0$ is the *delay* of the location l. Then

$$x_i'(t) - x_i'(0) \# \int_0^t f(x_i)dt \quad \text{for } t \in [0, \delta_l] \tag{2}$$

computes the value of the prime variable x_i' when $t \in [0, \delta_l]$ time units have passed since the system entered the location l. It is $x_i'(0) = x_i$ and for non–linear hybrid systems with $f(x_i) = c_1 * x_i + c_2$, $c_1, c_2 \in \mathbb{R}$ with $c_1 \neq 0$, (2) yields thereby:

$$x_i'(t) - x_i \# - \frac{c_2}{c_1} + \frac{c_2}{c_1} * e^{c_1 * t} \quad \text{for } t \in [0, \delta_l] \ . \tag{3}$$

The linear case is trivial. A location l with invariant $\mathrm{Inv}(l)$, activity $\mathrm{Act}(l) \equiv \bigwedge_{i \in I}(\dot{x}_i \# f(x_i))$ and delay δ_l generates an *analog relation* $\beta_l(\tilde{x}, \tilde{x}') \subseteq (\mathbb{R}^n \times \mathbb{R}^+)^2$:

$$\beta_l(\tilde{x}, \tilde{x}') \equiv \mathrm{Inv}(l)(\mathbf{x}') \wedge \delta_l \geq 0 \wedge \bigwedge_{i \in I}(x_i'(t) \# x_i + \int_0^t f(x_i)dt) \quad \text{for } t \in [0, \delta_l] \ . \tag{4}$$

By (3) and (4) we set for ROD_i and CONTROLLER the following *analog rules*:

% Analog rules for ROD_i
$out_i(x_i, time) \leftarrow time \leq time'$, $x_i' = x_i + (time' - time)$, $out_i(x_i', time')$.
$in_i(x_i, time) \leftarrow time \leq time'$, $x_i' = x_i$, $in_i(x_i', time')$.

% Analog rules for CONTROLLER
$rod_1(\theta, time) \leftarrow \theta_m \leq \theta'$, $time \leq time'$,
$\theta' = \theta - 560 * \exp((time' - time)/10) + 560$, $rod_1(\theta', time')$.
$no\_rod(\theta, time) \leftarrow \theta' \leq \theta_M$, $time \leq time'$,
$\theta' = \theta - 500 * \exp((time' - time)/10) + 500$, $no\_rod(\theta', time')$.
$rod_2(\theta, time) \leftarrow \theta_m \leq \theta'$, $time \leq time'$,
$\theta' = \theta - 600 * \exp((time' - time)/10) + 600$, $rod_2(\theta', time')$.
$shutdown(\theta, time) \leftarrow time \leq time'$, $shutdown(\theta, time')$.

In analog rules time progresses since $time \leq time'$. $time' - time \geq 0$ is the delay. To summarize, we set following proposition (For the sake of simplicity we set $\alpha_{(l,a,l')}(\tilde{x}, \tilde{x}') = \alpha_{(l,a,l')}(\mathbf{x}, \mathbf{x}') \wedge time' = time$ and consequently $l'(\tilde{x}') = l'(\mathbf{x}', time)$.).

Proposition 1. *A hybrid system H consists of one fact and a set of rules:*

$$\{l_0(\tilde{x}) \leftarrow \phi(\tilde{x}).\} \cup \bigcup_{(l,a,l')} \{l(\tilde{x}) \leftarrow 'a', \alpha_{(l,a,l')}(\tilde{x}, \tilde{x}'), l'(\tilde{x}').\}$$
$$\cup \bigcup_{l} \{l(\tilde{x}) \leftarrow \beta_l(\tilde{x}, \tilde{x}'), l(\tilde{x}').\} \ .$$

[1] This also applies to piecewise–continuous functions.

Product of Hybrid Systems. Let H_i, for $i = 1, 2$, be two hybrid systems over the common set of data variables $\mathbf{x}$. Both hybrid systems synchronize on the common set of synchronization symbols $T_1 \cap T_2$. The product of H_1 and H_2 is the hybrid system $H = H_1 \times H_2$ consisting of the discrete rules:

$$\begin{aligned}(l_1, l_2)(\tilde{x}) \leftarrow {}& 'a', \alpha_{(l_1,a,l_1')}(\tilde{x}, \tilde{x}'), \alpha_{(l_2,a,l_2')}(\tilde{x}, \tilde{x}'), (l_1', l_2')(\tilde{x}'). \in H \text{ iff} \\ l_i(\tilde{x}) \leftarrow {}& 'a', \alpha_{(l_i,a,l_i')}(\tilde{x}, \tilde{x}'), l_i'(\tilde{x}'). \in H_i \text{ for } i = 1, 2\ ,\end{aligned}$$

$$\begin{aligned}(l_1, l_2)(\tilde{x}) \leftarrow {}& 'a_1', \alpha_{(l_1,a_1,l_1')}(\tilde{x}, \tilde{x}'), (l_1', l_2)(\tilde{x}'). \in H \text{ and} \\ (l_1, l_2)(\tilde{x}) \leftarrow {}& 'a_2', \alpha_{(l_2,a_2,l_2')}(\tilde{x}, \tilde{x}'), (l_1, l_2')(\tilde{x}'). \in H \text{ iff} \\ l_i(\tilde{x}) \leftarrow {}& 'a_i', \alpha_{(l_i,a_i,l_i')}(\tilde{x}, \tilde{x}'), l_i'(\tilde{x}'). \in H_i \\ & \text{for } i = 1, 2, a_1 \in T_1 \setminus T_2 \text{ and } a_2 \in T_2 \setminus T_1\ ,\end{aligned}$$

and of the analog rules:

$$\begin{aligned}(l_1, l_2)(\tilde{x}) \leftarrow {}& \beta_{l_1}(\tilde{x}, \tilde{x}'), \beta_{l_2}(\tilde{x}, \tilde{x}'), (l_1, l_2)(\tilde{x}'). \in H \text{ iff} \\ l_i(\tilde{x}) \leftarrow {}& \beta_{l_i}(\tilde{x}, \tilde{x}'), l_i(\tilde{x}'). \in H_i \text{ for } i = 1, 2\ .\end{aligned}$$

For the initial fact we have:

$$\begin{aligned}(l_{01}, l_{02})(\tilde{x}) \leftarrow {}& \phi_1(\tilde{x}), \phi_2(\tilde{x}). \text{ is the initial rule of } H \text{ iff} \\ l_{0i}(\tilde{x}) \leftarrow {}& \phi_i(\tilde{x}). \in H_i \text{ for } i = 1, 2\ .\end{aligned}$$

A *data state* for the data variables $\mathbf{x}$ ($\mathbf{x}'$ resp) is a function ϖ that assigns a real–value $\varpi(x_i) \in \mathbb{R}$ to each data variable x_i. The behavior of a hybrid system in each time instant is described through data states ϖ and locations l. The pair $\sigma =< l, \varpi >$ is a *state*. Intuitively, a discrete rule $d(l, a, l')$ is a brief description of the *discrete successor states* $< l', \varpi' >$ reachable from the state $< l, \varpi >$ at time point *time*. An analog rule $a(l)$ is a concise description of all possible *analog successor states* $< l, \varpi' >$ reachable from the state $< l, \varpi >$ since $(time' - time \geq 0)$ time units have passed after the last entry from the system in the location l at the time point *time*. A possible behavior of a hybrid system H is described by a trajectory.

Trajectory. A *trajectory* ρ is a finite or infinite sequence

$$\rho : < l_0, \varpi_0 >^{time_0'} \mapsto < l_1, \varpi_1 >^{time_1'} \mapsto < l_2, \varpi_2 >^{time_2'} \mapsto \ldots \tag{5}$$

of states $< l_i, \varpi_i >$ and time points $time_i'$ such that for all $i \geq 0$ for all $time' \in \mathbb{R}^+$ with $time_i' \leq time' \leq time_{i+1}'$, and data state ϖ', $\beta_{l_i}(\varpi_i, time_i', \varpi', time')$ is true, $< l_i, \varpi' >$ is an analog successor state of $< l_i, \varpi_i >$, and $< l_{i+1}, \varpi_{i+1} >$ is a discrete successor state of $< l_i, \varpi' >$.

The *position* π of a trajectory ρ is a pair $\pi = (i, r)$, where $i \in \mathbb{N}$ and $r \in \mathbb{R}^+$ with $r \leq \delta_{l_i}$. i is the i–th place in ρ and r the time of an analog successor state $< l_i, \varpi_i' >$ of $< l_i, \varpi_i >$ before the $i+1$–th state $< l_{i+1}, \varpi_{i+1} >$ in ρ. The relation '$\leq$' between two positions π and π' is defined as follows: $(i, r) = \pi \leq \pi' = (i', r')$ iff $i < i'$ or $i = i'$ and $r < r'$. The *state at position* (i, r) is $sap_\rho(i, r) =< l_i, \varpi_i' >$. A *region* is a pair $< l, \Psi >$ with location l and formula Ψ over $\mathbf{x}$. It is $< l, \Psi >= \{< l, \varpi > \mid \Psi[\mathbf{x}/\varpi]\}$.

Symbolic Trajectory. A *symbolic trajectory* ρ is a finite or infinite sequence

$$\rho : < l_0, \Psi_0 > \mapsto < l_1, \Psi_1 > \mapsto < l_2, \Psi_2 > \mapsto \ldots \tag{6}$$

of regions $< l_i, \Psi_i >$ such that for all $i \geq 0$ $< l_{i+1}, \varpi_{i+1} > \in < l_{i+1}, \Psi_{i+1} >$ iff $\exists$ $< l_i, \varpi_i >, < l_i, \varpi_i' > \in < l_i, \Psi_i >$ such that $< l_i, \varpi_i' >$ is an analog successor state of $< l_i, \varpi_i >$ and $< l_{i+1}, \varpi_{i+1} >$ is a discrete successor state of $< l_i, \varpi_i' >$.

The number of applied discrete rules (transition steps) in a (symbolic) trajectory ρ is called the length of ρ. The duration δ_ρ of a trajectory ρ is the sum $\sum_{i=0}^{\infty} \delta_{l_i}$. A trajectory is *divergent* if $\delta_\rho = \infty$. Clearly, a symbolic trajectory (6) represents a set of trajectories of the form (5) where $< l_i, \varpi_i > \in < l_i, \Psi_i >$ for all $i \geq 0$. Besides, every trajectory (5) is represented by some symbolic trajectory of the form (6).

Nonzeno Hybrid System. A state $< l, \varpi >$ is *admissible* if $\mathrm{Inv}(l)([\mathbf{x}/\varpi])$ is true. The hybrid system H is *nonzeno* if for each admissible state $< l, \varpi >$ there exists a divergent trajectory ρ of H which begins at $< l, \varpi >$, i.e., $sap_\rho(0,0) =< l, \varpi >$. Intuitively, H is nonzeno iff every finite prefix of a trajectory is a prefix of a divergent trajectory. In the following we only consider nonzeno hybrid systems.

1.3 Integrator Computation Tree Logic, ICTL

We specify safety, liveness, time–bounded and duration requirements of hybrid systems in ICTL [1]. Let H be a hybrid system with data variables $\mathbf{x}$ and set of locations L, and let $\mathbf{z} = z_1, \ldots, z_m$ be a vector of non–negative real–valued variables $z_i \in \mathbb{R}^+$ called *integrators*. An *integrator* is a clock which continues only in a subset I of L. I is called the *type* of z. A $\mathbf{z}$*–extended data predicate* of H is a formula over $\mathbf{x} \biguplus \mathbf{z}$. A $\mathbf{z}$*–extended state predicate* of H is a collection of $\mathbf{z}$–extended data predicates, one for each location in L. The formulae of ICTL are defined by the following grammar:

$$\varphi ::= \psi \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \, \exists\mathcal{U} \, \varphi_2 \mid \varphi_1 \, \forall\mathcal{U} \, \varphi_2 \mid (z : I).\varphi \ , \tag{7}$$

where ψ is a $\mathbf{z}$–extended state predicate that contains only integer constants, z is an integrator from $\mathbf{z}$, $I \subseteq L$ is the type of z, and $(z : I).\varphi$ is an *integrator reset quantifier*. The ICTL formula φ is *closed* if every occurrence of an integrator in φ is bound by a integrator reset quantifier. It is assumed that different integrator reset quantifiers in φ bind different integrators. We construct the hybrid system $H_{\mathbf{z}}$ by extending H with integrators $\mathbf{z}$ such that for the interpretation of the formula φ in $H_{\mathbf{z}}$ there is in H a corresponding interpretation. For each integrator z_i and each location $l \in L$ the analog relation of l has the form (We set $\mathbf{y} = \mathbf{x}, \mathbf{z}$ and $\tilde{y} = \mathbf{x}, \mathbf{z}, time$):

$$\beta_l'(\tilde{y}, \tilde{y}') = \beta_l(\tilde{x}, \tilde{x}') \wedge \begin{cases} z_i' = z_i + \delta_l & : \text{ if } l \in I_i, \\ z_i' = z_i & : \text{ otherwise} \end{cases} . \tag{8}$$

By (8) it is clear that the analog rule $a(l)$ in $H_{\mathbf{z}}$ is the analog rule of H extended with $\mathbf{z}$, i.e., $a(l) \equiv l(\tilde{y}) \leftarrow \beta_l'(\tilde{y}, \tilde{y}'), l(\tilde{y}')$. For the discrete rule $d(l, a, l')$ of $H_{\mathbf{z}}$ we get: $d(l, a, l') \equiv l(\tilde{y}) \leftarrow {}'a', \alpha'_{(l,a,l')}(\tilde{y}, \tilde{y}'), l'(\tilde{y}')$. with

$$\alpha'_{(l,a,l')}(\tilde{y}, \tilde{y}') = \alpha_{(l,a,l')}(\tilde{x}, \tilde{x}') \wedge \bigwedge_{i=1}^{m} z_i' = z_i \ .$$

For a set Ω of trajectories in H the $\mathbf{z}$-*extension* $\Omega_{\mathbf{z}}$ consists of all $\mathbf{z}$-*extended trajectories* of $H_{\mathbf{z}}$ the $\mathbf{x}$-projections of which are in Ω. For a state σ of $H_{\mathbf{z}}$ and set Ω of trajectories in H the satisfaction relation $\sigma \models_\Omega \varphi$ is defined inductively on the subformulae of φ (cf Equation (7)).

$\sigma \models_\Omega \psi$ iff $\sigma \in [\![\psi]\!]$ ($[\![\psi]\!]$ defines the region $\bigcup_{l \in L} < l, \Psi_l >$) .
$\sigma \models_\Omega \neg\varphi$ iff $\sigma \not\models_\Omega \varphi$.
$\sigma \models_\Omega \varphi_1 \vee \varphi_2$ iff $\sigma \models_\Omega \varphi_1$ or $\sigma \models_\Omega \varphi_2$.
$\sigma \models_\Omega \varphi_1 \exists\mathcal{U} \varphi_2$ iff for some trajectory $\rho \in \Omega_{\mathbf{z}}$ with $sap_\rho(0,0) = \sigma$ there is a position π of ρ such that $sap_\rho(\pi) \models_\Omega \varphi_2$, and for all positions π' of ρ, if $\pi' \leq \pi$ then $sap_\rho(\pi') \models_\Omega \varphi_1 \vee \varphi_2$.
$\sigma \models_\Omega \varphi_1 \forall\mathcal{U} \varphi_2$ iff for all trajectories $\rho \in \Omega_{\mathbf{z}}$ with $sap_\rho(0,0) = \sigma$ there is a position π of ρ such that $sap_\rho(\pi) \models_\Omega \varphi_2$, and for all positions π' of ρ, if $\pi' \leq \pi$ then $sap_\rho(\pi') \models_\Omega \varphi_1 \vee \varphi_2$.
$\sigma \models_\Omega (z : I).\varphi$ iff $\sigma[z/0] \models_\Omega \varphi$.

Example 2 Reactor Temperature Control System. The hybrid system H for Example 1 is the product H = CONTROLLER × ROD_1× ROD_2. In the ICTL specification language the safety requirement for the temperature control system: **'A shutdown of the reactor never occurs'** is specified as:

$$\phi_{initial} \Rightarrow \forall\Box\, \neg\varphi_{final} \ , \tag{9}$$

where $\phi_{initial} \equiv \ell = (no\_rod, out_1, out_2) \wedge \theta = \theta_m \wedge x_1 = T \wedge x_2 = T$ and $\varphi_{final} \equiv \ell = (no\_rod, out_1, out_2) \wedge \theta = \theta_M \wedge x_1 \leq T \wedge x_2 \leq T$. H starts at the initial location (no_rod, out_1, out_2) with $\theta = \theta_m$ and $x_1 = T \wedge x_2 = T$. To meet the safety requirement (9) H must not reach a state $< (no\_rod, out_1, out_2), [\theta, x_1, x_2] >$ with $\theta = \theta_M$ and no control rod available because $x_1 \leq T$ and $x_2 \leq T$.

2 A CLP($\mathcal{R}$) System for Hybrid Systems

Firstly, we describe the general structure $\mathcal{R}$ and show some special cases for $\mathcal{R}$, Sect. 2.1. Secondly, the equivalent semantics of CLP($\mathcal{R}$) are considered and some results for hybrid systems are lifted from the CLP theorie surveyed in [3], Sect. 2.2.

2.1 The CLP($\mathcal{R}$) Structure

The structure $\mathcal{R}^2 = (\Sigma, \mathbf{R}, \mathcal{L}, \mathcal{T})$ defines the underlying domain of discourse $\mathbb{R}$ and the operations and relations on it, where $\Sigma = (\mathbb{R}, \{0,1\}, \{+, \exp\}, \{=, <, \leq\} \cup L)$. If we omit exp in the definition of $\mathcal{R}$, we write $\mathcal{R}_{lin}$. Thus, linear (non-linear) hybrid systems are CLP($\mathcal{R}_{lin}$) (CLP($\mathcal{R}$)) programs.

2.2 Semantics of Hybrid Systems

Logical Semantics. The two logical semantics of CLP($\mathcal{R}$) programs applied to hybrid systems lead to the following proposition.

Proposition 2. *For a hybrid system H the Σ-theory, also denoted by H, and the Clark-Completion $H^\star$ are the sets (cf Theorem 1):*

$$H = \{\forall\tilde{x}.(\phi(\tilde{x}) \Rightarrow l_0(\tilde{x}))\} \cup \bigcup_{d(l,a,l')\in H} \{\forall\tilde{x},\tilde{x}'.(\alpha_{(l,a,l')}(\tilde{x},\tilde{x}') \wedge l'(\tilde{x}') \Rightarrow l(\tilde{x}))\} \cup \bigcup_{a(l)\in H} \{\forall\tilde{x},\tilde{x}'.(\beta_l(\tilde{x},\tilde{x}') \wedge l(\tilde{x}') \Rightarrow l(\tilde{x}))\} \ , \tag{10}$$

$$H^\star = \{\ \forall\tilde{x}.(l_0(\tilde{x}) \iff \exists\tilde{x}'.\phi(\tilde{x}') \vee \bigvee_{d(l_0,a,l')\in H} \exists\tilde{x}'.(\alpha_{(l_0,a,l')}(\tilde{x},\tilde{x}') \wedge l'(\tilde{x}')) \vee \exists\tilde{x}'.(\beta_{l_0}(\tilde{x},\tilde{x}') \wedge l_0(\tilde{x}')))\ \} \cup \bigcup_{l\in L\setminus\{l_0\}} \{\ \forall\tilde{x}.(l(\tilde{x}) \iff \bigvee_{d(l,a,l')\in H} \exists\tilde{x}'.(\alpha_{(l,a,l')}(\tilde{x},\tilde{x}') \wedge l'(\tilde{x}')) \vee \exists\tilde{x}'.(\beta_l(\tilde{x},\tilde{x}') \wedge l(\tilde{x}')))\ \} \ . \tag{11}$$

Fixpoint Semantics. The fixpoint semantics is based on two one–step functions $T_H^{\mathcal{R}}$ and $S_H^{\mathcal{R}}$. The closure operator generated by $T_H^{\mathcal{R}}$ is denoted by $[\![H]\!]$. $T_H^{\mathcal{R}}$ and $[\![H]\!]$ map over $\mathbf{R}$–interpretations. The set of $\mathbf{R}$–interpretations forms a complete lattice under $'\subseteq'$. Both $T_H^{\mathcal{R}}$ and $[\![H]\!]$ are continuous on $\mathcal{B}_{\mathcal{R}} = \{l(\varpi) \mid l \in L, \varpi \in \mathbb{R}^n \times \mathbb{R}^+\}$. For a set of facts X denotes $[X]_{\mathcal{R}}$ the set $\{v(l) \mid (l \leftarrow c.) \in X, \mathbf{R} \models v(c)\}$.

$$T_H^{\mathcal{R}}(X) = \{l(\varpi) \mid l(\tilde{x}) \leftarrow c, l'. \in H, a \in X,\ v \text{ is a valuation on } \mathbf{R} \text{ such that } \mathbf{R} \models v(c),\ v(\tilde{x}) = \varpi \text{ and } v(l') = a\} \ . \tag{12}$$

$S_H^{\mathcal{R}}$ is defined on sets of facts, which form a complete lattice under $'\subseteq'$.

$$S_H^{\mathcal{R}}(X) = \{l(\tilde{x}) \leftarrow c. \mid l(\tilde{x}) \leftarrow c', l'. \in H,\ a \leftarrow c''. \in X \text{ renamed to new variables and } \mathbf{R} \models (c \iff c' \wedge c'' \wedge l' = a)\} \ . \tag{13}$$

The closure operator generated by $S_H^{\mathcal{R}}$ is denoted by $\ll H \gg$. Both $S_H^{\mathcal{R}}$ and $\ll H \gg$ are continuous. Between $T_H^{\mathcal{R}}$ and $S_H^{\mathcal{R}}$ there exists the relation: $[S_H^{\mathcal{R}}(X)]_{\mathcal{R}} = T_H^{\mathcal{R}}([X]_{\mathcal{R}})$.

Proposition 3. *For hybrid systems H, H_1, H_2 and set of facts Q over the constraint domain $\mathcal{R}$ with corresponding Σ-theory T the following holds:*

1. $T_H^{\mathcal{R}} \uparrow \omega = lfp(T_H^{\mathcal{R}}) = [lfp(S_H^{\mathcal{R}})]_{\mathcal{R}} = [\![H]\!](\emptyset)$.
2. $lm(H, \mathbf{R}) = [\{l \leftarrow c \mid H^\star, \mathbf{R} \models (c \Rightarrow l)\}]_{\mathcal{R}} = [\{l \leftarrow c \mid H^\star, T \models (c \Rightarrow l)\}]_{\mathcal{R}}$.

[2] Our results also hold for a constraint domain isomorph to $\mathcal{R}$.

3. $lm(H^\star, \mathbf{R}) = lm(H, \mathbf{R}) = lfp(T_H^{\mathcal{R}})$.

4. $gm(H^\star, \mathbf{R}) = gfp(T_H^{\mathcal{R}})$.

5. $[\![H]\!]([Q]_{\mathcal{R}}) = [\![H \cup Q]\!](\emptyset) = lm(H \cup Q, \mathbf{R})$.

6. $\ll H \gg (Q) = \ll H \cup Q \gg (\emptyset) = lfp(S^{\mathcal{R}}_{H \cup Q})$.

7. $\mathbf{R} \models (H_1 \iff H_2)$ *iff* $[\![H_1]\!] = [\![H_2]\!]$.

Top–Down Semantics. The operational semantics is given as a transition system of CLP($\mathcal{R}$) states together with a computation rule. The *computation rule* consists — starting with the initial fact — of an **'alternating selection of analog and discrete rules in a top-down left-to-right Prolog style'**, top–down for the selection of an applicable rule and left–to–right for the subgoal selection. We give a specific transition system for hybrid systems that computes a symbolic trajectory of length m by calling the hybrid system with the goal $trajectory([L_0, L_1, \ldots, L_m])$. L_i are *location metavariables* running on the set L.

Transition System. The transition system consists of the *transition rules* $\rightarrow_{init}$, $\rightarrow_a$, $\rightarrow_d$, $\rightarrow_c$, $\rightarrow_i$ and $\rightarrow_s$ such that :

- $< trajectory[L_0, \ldots, L_m]), C, S > \rightarrow_{init}$
  $< \phi(\tilde{x}) \bullet l_0(\tilde{x}) \bullet trajectory([L_1, \ldots, L_m]), C, S >$,
  if L_0 is a location metavariable, $l_0(\tilde{x}) \leftarrow \phi(\tilde{x})$. is the initial fact renamed to new variables, and both L_0 and l_0 are the same. *Meaning: Symbolic trajectories begin in the initial location.*
- $< trajectory([L_0, \ldots, L_m]), C, S > \rightarrow_{init} fail$,
  if L_0 is a location metavariable, and for the initial fact $l_0(\tilde{x}) \leftarrow \phi(\tilde{x})$. L_0 and l_0 are different. *Meaning: Only symbolic trajectories starting in the initial location may be accepted.*
- $< l_i(\tilde{x}) \bullet trajectory([L_{i+1}, \ldots, L_m]), C, S > \rightarrow_a$
  $< \beta_{l_i}(\tilde{x}, \tilde{x}') \bullet l_i(\tilde{x}') \bullet trajectory([L_{i+1}, \ldots, L_m]), C, S >$,
  if l_i is a location, $l(\tilde{x}) \leftarrow \beta_l(\tilde{x}, \tilde{x}'), l(\tilde{x}')$. an analog rule renamed to new variables, and both l and l_i are the same. *Meaning: Application of an analog rule means firstly to select the analog rule and secondly to solve the constraints* $\beta_{l_i}(\tilde{x}, \tilde{x}')$.
- $< l_i(\tilde{x}) \bullet trajectory([L_{i+1}, \ldots, L_m]), C, S > \rightarrow_a fail$,
  if l_i is a location, and for each analog rule $l(\tilde{x}) \leftarrow \beta_l(\tilde{x}, \tilde{x}'), l(\tilde{x}')$. l and l_i are different. *Meaning: There must be at most one applicable analog rule for each location.*
- $< l_i(\tilde{x}) \bullet trajectory([]), C, S > \rightarrow_a < \beta_{l_i}(\tilde{x}, \tilde{x}'), C, S >$,
  if l_i is a location, $l(\tilde{x}) \leftarrow \beta_l(\tilde{x}, \tilde{x}'), l(\tilde{x}')$. an analog rule renamed to new variables, and both l and l_i are the same. *Meaning: The computation ends by the application of the analog rule corresponding to the last location.*
- $< l_i(\tilde{x}) \bullet trajectory([L_{i+1}, \ldots, L_m]), C, S > \rightarrow_d$
  $< \alpha_{(l,a,l')}(\tilde{x}, \tilde{x}') \bullet l'(\tilde{x}') \bullet trajectory([L_{i+2}, \ldots, L_m]), C, S >$,
  if l_i is a location, $l(\tilde{x}) \leftarrow \alpha_{(l,a,l')}(\tilde{x}, \tilde{x}'), l'(\tilde{x}')$. is a discrete rule renamed to new variables, and l and l_i, l' and L_{i+1} are the same, respectively. *Meaning: Application of a discrete rule means firstly to select an applicable discrete rule and secondly to solve the constraints* $\alpha_{(l,a,l')}(\tilde{x}, \tilde{x}')$.

- $< l_i(\tilde{x}) \bullet trajectory([L_{i+1}, \ldots, L_m]), C, S > \rightarrow_d fail$,
  if l_i is a location, and for each discrete rule $l(\tilde{x}) \leftarrow \alpha_{(l,a,l')}(\tilde{x}, \tilde{x}'), l'(\tilde{x}')$. l and l_i are different. *Meaning: If there is no applicable discrete rule as required for the computation, then the computation fails.*

The transition rules $\rightarrow_c$, $\rightarrow_i$ and $\rightarrow_s$ are defined as usual. The transition system can easily be extended to handle more general goals like $G \equiv c(\tilde{x}) \wedge L_m(\tilde{x}) \wedge trajectory([L_0, \ldots, L_m])$ or $G \equiv c(\tilde{x}) \wedge L_m(\tilde{x})$ expressing the reachability of the property (state predicate) $\varphi \equiv c(\tilde{x}) \wedge L_m(\tilde{x})$ in the symbolic trajectory through $L_0, \ldots, L_m$. The exhaustive search method of CLP($\mathcal{R}$) can be prohibited for certain properties and hybrid systems. Thus, to analyse hybrid systems we propose in Sect. 3 the use of some evaluation techniques tailored to hybrid systems.

Example 3. Consider the reactor temperature control system from Example 1 and 2. $trajectory([(no\_rod, out_1, out_2), (rod_1, in_1, out_2), (no\_rod, out_1, out_2)])$ computes the symbolic trajectory $< (no\_rod, out_1, out_2), \Psi_0 > \mapsto < (rod_1, in_1, out_2), \Psi_1 > \mapsto < (no\_rod, out_1, out_2), \Psi_2 >$ with constraints Ψ_i. To compute a set of symbolic trajectories we call, for instance, the goal $trajectory([(no\_rod, out_1, out_2), L_1, L_2, L_3, (no\_rod, out_1, out_2)])$ which will by backtracking compute a set of symbolic trajectories with $L_1, L_3 \in \{(rod_1, in_1, out_2), (rod_2, out_1, in_2)\}$ and $L_2 = (no\_rod, out_1, out_2)$. To check whether the prohibited state $(no\_rod, out_1, out_2) \wedge \theta = \theta_M \wedge x_1 \leq T \wedge x_2 \leq T$ can be reached by some (symbolic) trajectory of length m we could call $\theta = \theta_M \wedge x_1 \leq T \wedge x_2 \leq T \wedge (no\_rod, out_1, out_2)(\theta, x_1, x_2) \wedge trajectory([(no\_rod, out_1, out_2), L_1, \ldots, L_{m-1}, (no\_rod, out_1, out_2)])$ which will fail for every $m \geq 0$.

Proposition 4. *The computation rule mentioned above is a fair computation rule.*

Theorem 5 Soundness and Completeness. *For a hybrid system H the goal $trajectory([L_0, \ldots, L_m])$ with location metavariables L_i ($i = 0, \ldots, m$, $m \geq 0$) is successful in a CLP($\mathcal{R}$) system with answer constraint $\{c_0(\tilde{x}_0, \tilde{x}_0'), \ldots, c_m(\tilde{x}_m, \tilde{x}_m')\}$ iff the sequence of regions $< L_0, \Psi_0(\tilde{x}_0, \tilde{x}_0') > \mapsto \cdots \mapsto < L_m, \Psi_m(\tilde{x}_m, \tilde{x}_m') >$ with $\mathbf{R} \models (c_i(\tilde{x}_i, \tilde{x}_i') \iff \Psi_i(\tilde{x}_i, \tilde{x}_i'))$ for all $i = 0, \ldots, m$ is a symbolic trajectory of H.*

Now, for a hybrid system H we consider the *success set* $\mathrm{SS}(H) = \bigcup_{i=0}^{\infty} \mathrm{SS}(H)^i$ which collects the answer constraints of all finite and infinite symbolic trajectories. $\mathrm{SS}(H)^m$ is the set $\{l_0(\tilde{x}) \leftarrow c_0., \ldots, l_m(\tilde{x}) \leftarrow c_m.\}$ of facts $l_i(\tilde{x}) \leftarrow c_i.$ which are built of the answer constraint $\{c_0(\tilde{x}_0, \tilde{x}_0'), \ldots, c_m(\tilde{x}_m, \tilde{x}_m')\}$ of the corresponding symbolic trajectory of length m. The *finite failure set* $\mathrm{FFS}(H) = \bigcup_{i=0}^{\infty} \mathrm{FFS}(H)^i$ collects the set of finite failed goals of the form $c(\tilde{x}) \wedge l_m(\tilde{x}) \wedge trajectory([L_0, \ldots, l_m])$. It describes the set of all (symbolic) trajectories in a hybrid system which finitely fail.

$$\mathrm{FFS}(H)^m = \{l_m(\tilde{x}) \leftarrow c. \mid \text{ for every fair derivation } < l_m(\tilde{x}) \wedge trajectory([L_0, \ldots, L_{m-1}, l]), \{c(\tilde{x})\}, \emptyset > \stackrel{*}{\rightarrow} fail\} \ . \quad (14)$$

Proposition 6. *Let CLP($\mathcal{D}$) be an ideal system where $\mathcal{D}$ and $\mathcal{T}$ correspond on $\mathcal{L}$. Then:*

1. *$SS(H) = lfp(S_H^{\mathcal{D}})$ and $[SS(H)]_{\mathcal{D}} = lm(H, \mathbf{D})$.*
2. *If the goal G has a successful derivation with answer constraint c, then $H, T \models (c \Rightarrow G)$.*
3. *Now, suppose T is satisfaction complete wrt $\mathcal{L}$. If G has a finite computation tree, with answer constraints $c_1, \ldots, c_m$, then, $H^\star, T \models (G \iff c_1 \vee \cdots \vee c_m)$.*
4. *If $H, T \models (c \Rightarrow G)$ then there are derivations for the goal G with answer constraints $c_1, \ldots, c_m$ such that $T \models (c \Rightarrow \bigvee_{i=1}^m c_i)$. If, in addition, $(\mathcal{D}, \mathcal{L})$ has independence of negated constraints property, then the result holds for $m = 1$.*
5. *Now, suppose T is satisfaction complete wrt $\mathcal{L}$. If $H^\star, T \models (G \iff c_1 \vee \cdots \vee c_m)$, then G has a computation tree with answer constraints $c_1' \vee \cdots \vee c_m'$ such that $T \models (c_1 \vee \cdots \vee c_m \iff c_1' \vee \cdots \vee c_m')$.*
6. *Now, suppose T is satisfaction complete wrt $\mathcal{L}$. Then the goal G is finitely failed for H iff $H^\star, T \models \neg G$.*
7. *Suppose $(\mathcal{D}, \mathcal{L})$ is solution compact. Then $T_H^{\mathcal{D}} \downarrow \omega = \mathcal{B}_{\mathcal{D}} \setminus [FFS(H)]_{\mathcal{D}}$.*

Bottom–Up Semantics The bottom–up semantics starts with a set of facts and computes step by step — supposing the iteration terminates — a representation of the least model of H, $lm(H, \mathbf{R})$. The bottom–up execution of a hybrid system is defined as a transition system between sets of facts. For each rule $l(\tilde{x}) \leftarrow c, l'. \in H$ and sets of facts A, B the relation $A \rightsquigarrow B$ is defined as follows:

$$A \rightsquigarrow B \text{ iff } B = A \cup \{l(\tilde{x}) \leftarrow c'. \mid l(\tilde{x}) \leftarrow c, l'. \in H \text{ and there exists a fact } l_1(\tilde{x}) \leftarrow c_1. \in A \text{ with } \mathbf{R} \models (c' \iff c \wedge c_1 \wedge l_1 = l')\} \ . \quad (15)$$

By definition of the operator $S_H^{\mathcal{R}}$ in (13) it is clear that $A \rightsquigarrow B$ iff $B = A \cup S^{\mathcal{R}}_{\{l(\tilde{x}) \leftarrow c, l'.\}}(A)$ for a rule $l(\tilde{x}) \leftarrow c, l'. \in H$.

Execution. An *execution* is a sequence of transitions of the form (15). An execution is *fair*, if it can be applied infinitely often. The *execution* $A_0 \rightsquigarrow A_1 \rightsquigarrow \cdots \rightsquigarrow A_i \rightsquigarrow \ldots$ *terminates*, if there exists a m and for each $k > m$ $A_k = A_m$ is true. For a hybrid system H we say: H is *finitely computable*, if for each finite initial set A_0 of facts and for each fair execution there is a m such that $[A_k]_{\mathcal{R}} = [A_m]_{\mathcal{R}}$ for all $k \geq m$. An execution can be non–terminating even for *finitely computable* hybrid systems and finite initial set A_0.

Theorem 7. *Let H be a hybrid system, Q a set of facts and A the result of a fair bottom–up execution. Then $A = SS(H \cup Q) = \ll H \gg (Q)$ and $[\![H]\!]([Q]_{\mathcal{R}}) = [A]_{\mathcal{R}}$.*

Example 4. To verify the safety requirement (9) of the reactor temperature control system we can show by bottom–up evaluation that if the execution (starting with $\varphi_{final} \equiv \ell = (no\_rod, out_1, out_2) \wedge \theta = \theta_M \wedge x_1 \leq T \wedge x_2 \leq T$)

$$\{(no\_rod, out_1, out_2)(\theta, x_1, x_2) \leftarrow \theta = \theta_M \wedge x_1 \leq T \wedge x_2 \leq T.\} \rightsquigarrow \cdots \rightsquigarrow A$$

finishes with the set of facts A, then for A the initial goal (corresponding to the

set of initial states $\phi_{initial} \equiv \ell = (no\_rod, out_1, out_2) \wedge \theta = \theta_m \wedge x_1 = T \wedge x_2 = T)$
$G \equiv \theta = \theta_m \wedge x_1 = T \wedge x_2 = T \wedge (no\_rod, out_1, out_2)(\theta, x_1, x_2).\}$ fails.

The top–down and bottom–up evaluation methods require a method to check if the fixpoint is reached. Srivastava gave in [5] several algorithmus for subsumption check in CLP($\mathcal{R}_{lin}$). These are considered in Sect. 3.7, but in more details in [8].

3 Evaluation Techniques for Hybrid Systems

In this section we present several verification techniques for hybrid systems. The most important of them are the reachability analysis Sect. 3.1, the proof of safety requirements Sect. 3.2, the proof of duration properties Sect. 3.3, the parameterized reachability analysis Sect. 3.4, and the symbolic model–checking for ICTL formulae Sect. 3.5. The delay of constraints strategy Sect. 3.6, the subsumption and indexing of contraints [5] Sect. 3.7, and the intelligent backtracking strategy [2] Sect. 3.8, however, might be useful for improving the efficiency of the proof methods.

3.1 Reachability Analysis

The problem to check whether a property φ represented as a $\mathbf{z}$–extended state predicate is reachable in a given hybrid system H can be solved by top–down/bottom–up fixpoint computation. Let $\varphi_{final} = \bigcup_{l \in L} < l, \psi_l >$ be a *final region* describing a set of states. As every ψ_l is a convex linear formula φ_{final} can be rewritten as a *final goal* $G_{final} \equiv \bigvee_{l \in L}(\psi_l \wedge l(\tilde{y}))$ or as a set of *final facts* $A_{final} = \bigcup_{l \in L}\{l(\tilde{y}) \leftarrow \psi_l.\}$. Correspondingly, for the *initial region* $\varphi_{initial}$ we get the *initial goal* $G_{initial}$.

Proposition 8. *1. If $SS(H) = lfp(S_H^{\mathcal{R}})$ terminates and $\mathcal{R}$ is satisfaction complete, then φ_{final} is in H reachable iff G_{final} is in $SS(H)$ successful.*

2. Let A be the result of the fair bottom–up execution $A_{final} \rightsquigarrow \cdots \rightsquigarrow A$, then φ_{final} is in H reachable iff $G_{initial}$ is in A successful.

3. If $SS(H \cup A_{final}) = lfp(S_{H \cup A_{final}}^{\mathcal{R}})$ terminates and $\mathcal{R}$ is satisfaction complete. Then φ_{final} is in H reachable iff $G_{initial}$ is in $SS(H \cup A_{final})$ successful.

Proof. 1. "$\Longrightarrow$": By Definition of SS(H) and since φ_{final} reachable in H there exists an $m \geq 0$ such that for $\mathrm{SS}(H)^m = \{l_0(\tilde{y}) \leftarrow c_0., \ldots, l_m(\tilde{y}) \leftarrow c_m.\}$ the final foal $G_{final} \equiv \bigvee_{l \in L}(\psi_l \wedge l(\tilde{y}))$ is successful. "$\Longleftarrow$": Since $G_{final} \equiv \bigvee_{l \in L}(\psi_l \wedge l(\tilde{y}))$ in SS(H) successful, then there exists a computation tree from G_{final} with answer constraints $c_1, \ldots, c_s$, $s = 1, \ldots |L|$ with $\mathbf{R} \models (c_1, \ldots, c_s \iff \bigvee_{l \in L}(\psi_l \wedge l(\tilde{y})))$. Consequently, φ_{final} is in H reachable.

2. and 3. are consequences from 1. and Theorem 7. □

3.2 Proof of Safety Requirements

To prove the safety requirement $\varphi_{initial} \Rightarrow \forall\Box\, \neg\varphi_{final}$ it is transformed into a reachability problem. Then Proposition 8 can be applied.

Proposition 9. *Let H be a hybrid system. H meets the safety requirement $\varphi_{initial} \Rightarrow \forall\Box\, \neg\varphi_{final}$ iff φ_{final} is in H unreachable.*

3.3 Proof of Duration Properties

Duration properties can be proved by reachability analysis. This is done by displacing the clocks and integrators occurring in an ICTL duration formula into the hybrid system description. Then the duration property is transformed into a reachability problem and verified for the modified hybrid system as mentioned in Proposition 8.

Example 5. The duration property: 'Each control rod will be used at most 1/3 of the allowed time T' can be specified in ICTL as follows [1]:

$$\phi_{initial} \Rightarrow (z : L)(z_1 : (\_, in_1, \_))(z_2 : (\_, \_, in_2)).\, \forall\Box\, (3z_1 \leq z \wedge 3z_2 \leq z)\ , \qquad (16)$$

where $\phi_{initial} \equiv \ell = (no\_rod, out_1, out_2) \wedge \theta = \theta_M \wedge x_1 \leq T \wedge x_2 \leq T$, z is a clock, and z_1, z_2 are integrators. To verify this property the verification problem is rewritten as a reachability problem. The clock z proceeds as the global system time. For z the analog and discrete relations take the form: $\beta_l'(\tilde{y}, \tilde{y}') = \beta_l(\tilde{x}, \tilde{x}') \wedge z' = z + \delta_l$ and $\alpha'_{(l,a,l')}(\tilde{y}, \tilde{y}') = \alpha_{(l,a,l')}(\tilde{x}, \tilde{x}') \wedge z' = z$, respectively. The integrators z_1 and z_2 advance parallelly with the system time only in the set of locations $\{(\_, in_1, \_)\}$ and

$$\beta_l'(\tilde{y}, \tilde{y}') = \begin{cases} \beta_l(\tilde{x}, \tilde{x}') \wedge z_1' = z_1 + \delta_l & : \text{ if } l \in \{(\_, in_1, \_)\} \\ \beta_l(\tilde{x}, \tilde{x}') & : \text{ otherwise} \end{cases},$$

$$\beta_l'(\tilde{y}, \tilde{y}') = \begin{cases} \beta_l(\tilde{x}, \tilde{x}') \wedge z_2' = z_2 + \delta_l & : \text{ if } l \in \{(\_, \_, in_2)\} \\ \beta_l(\tilde{x}, \tilde{x}') & : \text{ otherwise} \end{cases}.$$

$\{(\_, \_, in_2)\}$. Otherwise, they do not change.

$$\alpha'_{(l,a,l')}(\tilde{y}, \tilde{y}') = \begin{cases} \alpha_{(l,a,l')}(\tilde{x}, \tilde{x}') \wedge z_1' = z_1 & : \text{ if } l \in \{(\_, in_1, \_)\} \\ \alpha_{(l,a,l')}(\tilde{x}, \tilde{x}') & : \text{ otherwise} \end{cases},$$

$$\alpha'_{(l,a,l')}(\tilde{y}, \tilde{y}') = \begin{cases} \alpha_{(l,a,l')}(\tilde{x}, \tilde{x}') \wedge z_2' = z_2 & : \text{ if } l \in \{(\_, \_, in_2)\} \\ \alpha_{(l,a,l')}(\tilde{x}, \tilde{x}') & : \text{ otherwise} \end{cases}.$$

Equation (16) yields thereby $\phi_{initial} \Rightarrow \forall\Box\neg\varphi_{final}$ with $\varphi_{final} \equiv 3z_1 \geq z \vee 3z_2 \geq z$. For this hybrid system it must be checked, whether the unsave states $[\![3z_1 \geq z \vee 3z_2 \geq z]\!]$ are reachable. That is, (16) holds iff $\phi_{initial}$ is unreachable.

3.4 Parameterized Reachability Analysis

The top–down and bottom–up fixpoints can also be used to verify parameterized hybrid systems. They offer a method to compute necessary and sufficient conditions for the parameters under which the reachability of a property can be ensured. Here,

the corresponding CLP($\mathcal{R}$) program description of the hybrid system has to be adequately adapted. For a vector $\mathbf{p} = p_1, \ldots, p_s$ of parameters the resulting $\mathbf{p}$-*extended hybrid system* $H_{\mathbf{p}}$ is built in that way as $H_{\mathbf{z}}$, but here p_i are parameters and therefore $\beta_l'(\tilde{w}, \tilde{w}') = \beta_l(\tilde{x}, \tilde{x}') \wedge \bigwedge_{i=1}^{s} p_i' = p_i$ and $\alpha_l'(\tilde{w}, \tilde{w}') = \alpha_l(\tilde{x}, \tilde{x}') \wedge \bigwedge_{i=1}^{s} p_i' = p_i$, where $\mathbf{w} = \mathbf{x}, \mathbf{p}$ and $\tilde{w} = \tilde{w}, time$.

Proposition 10. *Let $\varphi_{initial} \Rightarrow \forall\Box\, \neg\varphi_{final}$ be a safety property, $\mathbf{p}$ a vector of parameters, A_{final} the final set of facts and $A_{final} \rightsquigarrow \cdots \rightsquigarrow A$ the bottom-up fixpoint of the parameterized hybrid system. Then the $\mathbf{p}$-projection of the answer constraint of the initial goal $G_{initial}$ executed in A provides a sufficient and necessary condition φ over $\mathbf{p}$[3], such that for valuations of $\mathbf{p}$ which make φ true the safety requirement is met. In other words, the safety property does not hold iff $\neg\varphi$ is true.*

3.5 Symbolic Model Checking

The symbolic model-checking procedure SMC of [1] verifies for an ICTL formula φ and a linear hybrid system H whether H meets φ. The SMC procedure computes for a linear hybrid system H and an ICTL formula φ the *characteristic region* $[\![\varphi]\!]_H$ by providing a state predicate ψ which defines the answer, i.e., $[\![\varphi]\!]_H$. That is: $[\![\psi]\!] = [\![\varphi]\!]_H$. The state predicate ψ is called a *characteristic predicate* of φ. A characteristic predicate in general does not exist, and it is undecidable if a given state predicate is a characteristic predicate of a given ICTL formula φ. Our SMC procedure computes a set of facts which indirectly defines the characteristic predicate ψ. In the SMC procedure we mainly use the bottom-up semantics of hybrid systems.

$$|\psi| := \bigcup_{l \in L} \{l(\tilde{x}) \leftarrow \psi \wedge \mathrm{Inv}(l).\}, \text{ where } \psi = \bigcup_{l \in L} < l, \psi_l > \ . \tag{17}$$

$$|\neg\varphi| := |\varphi|^c \ . \tag{18}$$

$$|\varphi_1 \vee \varphi_2| := |\varphi_1| \cup |\varphi_2| \ . \tag{19}$$

$$|\varphi_1\, \exists\mathcal{U}\, \varphi_2| := \bigcup_{i \geq 0} \chi_i \text{ where } \chi_0 := |\varphi_2| \text{ and } \chi_{i+1} := \chi_i \cup B_{i+1} \text{ with } S^{\mathcal{R}}_{\chi_i}(|\varphi_1| \cup |\varphi_2|) \rightsquigarrow B_{i+1} \ . \tag{20}$$

$$|\varphi_1\, \forall\mathcal{U}\, \varphi_2| := \bigcup_{i \geq 0} \chi_i \text{ where } \chi_0 := |\varphi_2| \text{ and } \chi_{i+1} := |\chi_i \vee \neg z_{\varphi_1 \forall\mathcal{U} \varphi_2}.(\neg\chi_i\, \exists\mathcal{U}\, (\neg(\varphi_1 \vee \chi_i) \vee z_{\varphi_1 \forall\mathcal{U} \varphi_2} > 1))| \ . \tag{21}$$

$$|(z : I).\varphi| := |\varphi|[z/0] \text{ (i.e. replace all occurrences of } z \text{ in } |\varphi| \text{ by } 0) \ . \tag{22}$$

Proof. Equation (17) : The set of facts $|\psi|$ for the $\mathbf{z}$-extended state predicate $\psi = \bigcup_{l \in L} < l, \psi_l >$ consists of the set of admissible states that meet ψ. Equations (18)

[3] The prototype system CLP($\mathcal{R}$) [4] offers the operator *dump* to project constraints onto variables.

and (19) are clear. Equation (20) : The computation of the set of facts $|\varphi_1 \exists\mathcal{U} \varphi_2|$ for $\varphi_1 \exists\mathcal{U} \varphi_2$ mainly uses the bottom–up execution $\chi_0 \rightsquigarrow \chi_1 \rightsquigarrow \cdots \rightsquigarrow \chi_{i+1}$, which is another representation of:

$$|\varphi_2| \rightsquigarrow |\varphi_2| \cup B_1 \rightsquigarrow |\varphi_2| \cup B_1 \cup B_2 \rightsquigarrow \cdots \rightsquigarrow |\varphi_2| \cup B_1 \cup \cdots \cup B_{i+1} \ . \qquad (23)$$

If this execution terminates with the set of facts A, then $A = |\varphi_1 \exists\mathcal{U} \varphi_2|$. In every bottom–up step new sets of facts B_i are computed. Each fact in B_i is a concise description of states contained in χ_i that meet φ_1 or φ_2. Equation (21) : This proof is made in two steps. In the first step the operator $\forall\mathcal{U}$ is transformed into a timed operator $\forall\mathcal{U}_{\leq c}$ with $c \in \mathbb{Z}^+$. In the second step $\forall\mathcal{U}_{\leq c}$ is converted into the operator $\exists\mathcal{U}$, such that the proof of (21) is conducted back to the proof of (20). Equation (22) : It expresses the fact that the integrator z has to be set to zero since from that moment onwards its value plays an important role by the verification of φ. □

3.6 Delay of Constraints Strategy

The *Delay of Constraints Strategy (DCS)* is a small extension of the top–down and bottom–up semantics. Conceptually, the idea is as follows: After each top–down/bottom–up step a database of successful calls together with their generated answer constraint is mantained. If such call is made later in the computation, do not re–execute the call, but use the same answer constraint to update the current position as though the call had been executed and had returned that constraints. Moreover, if in each step the constraint is linear, it is solved directly. Otherwise, it is delayed and may get solved later, if it becomes linear during the computation. Therefore, DCS can be used in the verification methods mentioned above.

3.7 Subsumption and Indexing of Constraints

Top–down and bottom–up evaluation require in each iteration step a fixpoint check, i.e., a check if any new facts have been computed in the iteration step. Due to the nondeterministic features of hybrid systems there may be several answer constraints computed for an iteration step. A *subsumption check* enhances efficiency if subsumed facts can be discarded in each iteration step. Thus, we propose to combine fixpoint check and subsumption check in each iteration step. Srivastava [5] gave several algorithms to improve the efficiency of the subsumption check of CLP($\mathcal{R}_{lin}$) programs. The first one is a deterministic algorithm based on the divide and conquer strategy. The second one is an indexing algorithm, where only those polyhedra are indexed which do not intersect with a given polyhedron, such that they can be efficiently eliminated. The third algorithm is an incremental algorithm which interleaves the computation of a constraint with a subsumption check. For the application of these algorithms to linear hybrid systems we must refer to [8].

3.8 Intelligent Backtracking Strategy

Intuitively, intelligent backtracking is based on the idea: During the computation of the computation tree the failure in a branch is first explained as due to an inconsistent set of constraints, the *conflict set*, caused during the application of a discrete rule. A backtrack point has to be chosen in order to remove at least one element of the conflict. Conflicts are computed using a *Dynamic Intelligent Backtracking* algorithm (DIB). DIB needs a constraint solver able to detect a conflict when the constraint set is unsolvable. The unsolvability of a constraint set may be explained by several different conflicts. de Backer and Beringer showed in [2] that in CLP($\mathcal{R}_{lin}$) the computation of a minimal conflict has a polynomial complexity. The conflict set may give reasons for the unsatisfiability of a property (cf. [8]).

Acknowledgement. Thanks to Rajeev Alur and Tom Henzinger, who indirectly introduced me to hybrid systems and to Joxan Jaffar for making CLP($\mathcal{R}$) available. I am grateful to my advisors and thesis surveyors Günter Riedewald, Ernst–Rüdiger Olderog and Rolf Grützner. Finally, I wish to express my deepest gratitude to my wife, Katrin Breitschu, for the help and support.

References

1. R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138:3–34, 1995.
2. B. de Backer and H. Beringer. Intelligent Backtracking for CLP Languages: An Application to CLP($\mathcal{R}$). In V. Saraswat and K. Ueda, editors, *ILPS'91*, pages 405–419. MIT Press, 1991.
3. J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
4. J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) Language and System. *ACM : TOPLAS*, 14(3):339–395, 1992.
5. D. Srivastava. Subsumption and Indexing in Constraint Query Languages with Linear Arithmetic Constraints. *Annals of Math. and Artificial Intelligence*, 8:315–343, 1993.
6. L. Urbina. The Generalised Railroad Crossing: Its Analysis in CLP($\mathcal{R}$). This volume.
7. L. Urbina. Analysis of Robotics Applications in CLP($\mathcal{R}$). In A. Krall and U. Geske, editors, *11. Workshop Logische Programmierung*, number 270 in GMD–Studien, pages 39–48. GMD, 1995.
8. L. Urbina. *Analysis of Hybrid Systems in Constraint Logic Programming.* PhD thesis, Department of Computer Science, University of Rostock, 1996.
9. L. Urbina and G. Riedewald. A Framework for Symbolic Simulation of Hybrid Systems. In A. Krall and U. Geske, editors, *11. Workshop Logische Programmierung*, number 270 in GMD–Studien, pages 29–38. GMD, 1995.

On Query Languages for Linear Queries Definable with Polynomial Constraints

Luc Vandeurzen[1], Marc Gyssens[1], Dirk Van Gucht[2]

[1] Dept. WNI, University of Limburg, B-3590 Diepenbeek, Belgium.
lvdeurze@luc.ac.be, gyssens@charlie.luc.ac.be.
[2] Computer Science Dept., Indiana University, Bloomington, IN 47405-4101, USA.
vgucht@cs.indiana.edu.
WWW: http://www.luc.ac.be/theocomp.

Abstract. It has been argued that the linear database model, in which semi-linear sets are the only geometric objects, is very suitable for most spatial database applications. For querying linear databases, the language FO + linear has been proposed. We present both negative and positive results regarding the expressiveness of FO+linear. First, we show that the dimension query is definable in FO + linear, which allows us to solve several interesting queries. Next, we show the non-definability of a whole class of queries that are related to sets not definable in FO+linear. This result both sharpens and generalizes earlier results independently found by Afrati et al. and the present authors, and demonstrates the need for more expressive linear query languages if we want to sustain the desirability of the linear database model. In this paper, we show how FO + linear can be strictly extended within FO + poly in a safe way. Whether any of the proposed extensions is complete for the linear queries definable in FO + poly remains open. We do show, however, that it is undecidable whether an expression in FO + poly induces a linear query.

1 Introduction

Following the seminal work by Kuper, Kanellakis, and Revesz [20] on constraint query languages with polynomial constraints (FO + poly), various researchers have introduced geometric database models and query languages within this framework [18, 22]. These researchers have studied the desirability of their models for database applications involving geometric data objects, as well as the expressiveness of the proposed geometric query languages.

An important database model that has recently been studied in this context is the *linear spatial database model* [2, 3, 26], which we adopt in this paper. The linear model allows users to define relational databases, which may, besides conventional data, contain linear geometric data objects. Formally, these objects are so-called *semi-linear sets*, which can be defined in first-order logic over the reals with addition. The class of semi-linear sets suffices for the majority of applications encountered in GIS, geometric modeling, and spatial and

temporal databases [6, 23]. Furthermore, data structures and algorithms have been developed to efficiently implement a wide variety of operations on these sets [4, 11, 17, 24].

Associated with the linear model is the concept of *linear query*, which is a mapping from linear databases to linear databases. Because the linear database model is a sub-model of the polynomial database model, it is in principle possible to use the query language FO + poly to define natural linear queries, and, in fact, a vast number of important linear queries can indeed be so defined. Of course, not every query defined by an FO + poly formula induces a linear query, and, as is shown in Section 5, it is even undecidable whether an FO + poly formula induces a linear query.

Faced with this reality, several researchers [3, 26] have proposed the query language FO + linear as a natural query language to accompany the linear model. The FO + linear language is the sub-language of FO + poly wherein the polynomial constraints are restricted to *linear constraints*. Many important linear queries can be defined in FO + linear. Section 3 reviews some known results in this respect and presents some new ones. The most surprising of those is the definability of the *dimension query* which returns the topological dimension of a semi-linear set. This definability result allows us to solve some important practical queries. In particular, it follows that the *interval*-query, i.e., "Is the semi-linear set an interval?" and the *line*-query, i.e., "Is the semi-linear set a line?" are definable in FO + poly .

Unfortunately, FO + linear is *incomplete* for the linear queries definable in FO + poly as was recently shown by Afrati et al. [2]. The counter-example used by Afrati et al., however, is a technical one, and does not, in our view, adequately reveal the weaknesses of FO + linear as a language to define linear queries definable in FO + poly. In Section 4, building on the work of Afrati et al. [2] and on earlier work of the present authors [26], we show that natural FO + poly-definable linear queries, such as the query that yields the convex hull of a semi-linear set, are *not* FO + linear-definable. The conclusion we draw from these negative results is that, though FO + linear provides a good lower bound for the FO + poly-definable linear queries, FO + linear is too limited in expressiveness to be considered fully adequate to accompany the linear model.

This brings us to the last major topic of this paper. In Section 5, we introduce query languages that can only express FO + poly-definable linear queries, but that are strictly more expressive than FO + linear. These languages have some affinity with some operational languages that have been introduced in spatial database models, but that do not fall within the framework of Kuper, Kanellakis, and Revesz [20]. It is presently an open problem whether any of the query languages we propose in Section 5 is complete for the FO + poly-definable linear queries, though we conjecture this is not the case.

2 Preliminaries

In this paper we focus on the linear spatial database model as proposed in [26]. To put this paper into better perspective we briefly review some of the material in [26].

The linear model is extracted from the polynomial model, which is based on *real formulae*, i.e, formulae in first order logic over $(\mathbf{R}, \leq, +, *, 0, 1)$. Due to the work of Tarski [21], it is well known that this first order logic over the reals with inequality, addition and multiplication is a decidable theory. Every real formula $\varphi(x_1, \ldots, x_n)$ with free real variables $x_1, \ldots, x_n$ defines a geometrical figure $\{(x_1, \ldots, x_n) \mid (x_1, \ldots, x_n) \in \mathbf{R}^n \wedge \varphi(x_1, \ldots, x_n)\}$ in n-dimensional Euclidean space $\mathbf{R}^n$. Point sets defined in this way are called *semi-algebraic sets*.

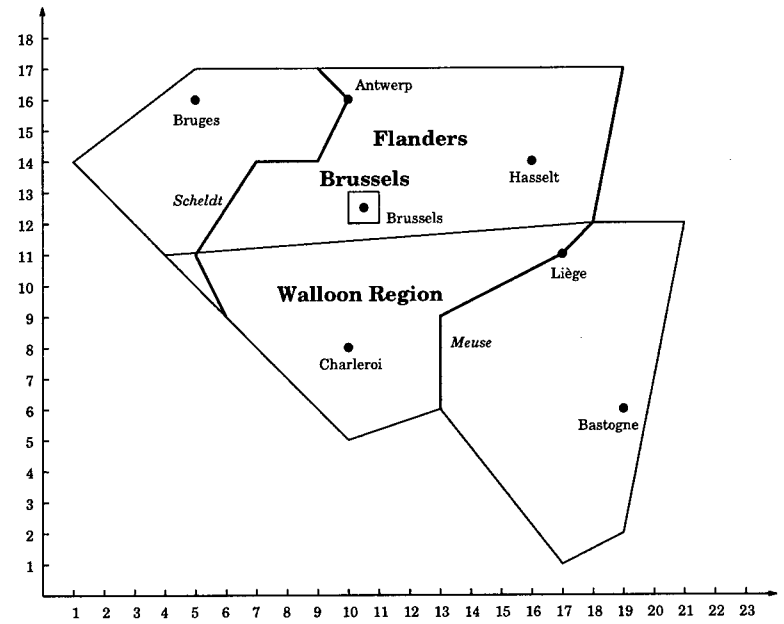
A *spatial database scheme*, $\mathcal{S}$, is a finite set of *relation names*. Each relation name, R, has a type which is a pair of natural numbers, $[m, n]$, where m denotes the number of non-spatial columns and n the dimension of the single spatial column of R. A database scheme has type $[m_1, n_1, \ldots, m_k, n_k]$ if the scheme consists of relation names, say $R_1, \ldots, R_k$, respectively of type $[m_1, n_1], \ldots, [m_k, n_k]$. A *syntactic database instance* is a mapping, $\mathcal{I}$, assigning to each relation name, R, of a scheme, $\mathcal{S}$, a syntactic relation $\mathcal{I}(R)$ of the same type. A *syntactic relation* of type $[m, n]$ is a finite set of tuples of the form $(v_1, \ldots, v_m; \varphi(x_1, \ldots, x_n))$, with $v_1, \ldots, v_m$ non-spatial values of some domain, U, and $\varphi(x_1, \ldots, x_n)$ a real formula with n free variables.

The semantics of a syntactic database instance, $\mathcal{I}$, over a database scheme, $\mathcal{S}$, is the mapping, I, assigning to each relation name, R, in $\mathcal{S}$ the semantic relation $I(\mathcal{I}(R))$. Given a syntactic relation, r, the semantic relation $I(r)$ is defined as $\bigcup_{t \in r} \{(t.v_1, \ldots, t.v_m)\} \times \{(x_1, \ldots, x_n) \mid t.\varphi(x_1, \ldots, x_n)\}$. This subset of $U^m \times \mathbf{R}^n$ can be interpreted as a possibly infinite $(m + n)$-ary relation, called *semantic relation*, the tuples of which are called *semantic tuples*.

Example 1. The example in Figure 1 shows a spatial database representing geographical information about Belgium. □

We consider a query of signature $[m_1, n_1, \ldots, m_k, n_k] \rightarrow [m, n]$ to be a mapping from database instances of a spatial database scheme of type $[m_1, n_1, \ldots, m_k, n_k]$ to database instances of a spatial database scheme of type $[m, n]$ that can be regarded in a consistent way both at the syntactic and semantic level, and is computable at the syntactic level.

In this context, we define the query language FO + poly as the language obtained by adding to the language of real formulae the following: (*i*) a totally ordered infinite set of variables called *non-spatial variables*, disjoint from the set of real variables, (*ii*) atomic formulae of the form $v_1 = v_2$, with v_1 and v_2 non-spatial variables, (*iii*) atomic formulae of the form $R(v_1, \ldots, v_m; x_1, \ldots, x_n)$, with $v_1, \ldots, v_m$ non-spatial variables, $x_1, \ldots, x_n$ real variables, and R a relation name of type $[m, n]$, and finally (*iv*) universal and existential quantification of non-spatial variables. A query of signature $[m_1, n_1, \ldots, m_k, n_k] \rightarrow [m, n]$ is definable in FO + poly if there exists an FO + poly formula φ with m free value variables



Regions

Name	*Geometry*
Brussels	$(y \leq 13) \wedge (x \leq 11) \wedge (y \geq 12) \wedge (x \geq 10)$
Flanders	$(y \leq 17) \wedge (5x - y \leq 78) \wedge (x - 14y \leq -150) \wedge (x + y \geq 45) \wedge$ $(3x - 4y \geq -53) \wedge (\neg((y \leq 13) \wedge (x \leq 11) \wedge (y \geq 12) \wedge (x \geq 10)))$
Walloon Region	$((x - 14y \geq -150) \wedge (y \leq 12) \wedge (19x + 7y \leq 375) \wedge (x - 2y \leq 15) \wedge$ $(5x + 4y \geq 89) \wedge (x \geq 13)) \vee ((-x + 3y \geq 5) \wedge (x + y \geq 45) \wedge$ $(x - 14y \geq -150) \wedge (x \geq 13))$

Cities

Name	*Geometry*
Antwerp	$(x = 10) \wedge (y = 16)$
Bastogne	$(x = 19) \wedge (y = 6)$
Bruges	$(x = 5) \wedge (y = 16)$
Brussels	$(x = 10.5) \wedge (y = 12.5)$
Charleroi	$(x = 10) \wedge (y = 8)$
Hasselt	$(x = 16) \wedge (y = 14)$
Liège	$(x = 17) \wedge (y = 11)$

Rivers

Name	*Geometry*
Meuse	$((y \leq 17) \wedge (5x - y \leq 78) \wedge (y \geq 12)) \vee$ $((y \leq 12) \wedge (x - y = 6) \wedge (y \geq 11)) \vee$ $((y \leq 11) \wedge (x - 2y = -5) \wedge (y \geq 9)) \vee$ $((y \leq 9) \wedge (x = 13) \wedge (y \geq 6))$
Scheldt	$((y \leq 17) \wedge (x + y = 26) \wedge (y \geq 16)) \vee$ $((y \leq 16) \wedge (2x - y = 4) \wedge (y \geq 14)) \vee$ $((x \leq 9) \wedge (x \geq 7) \wedge (y = 14)) \vee$ $((y \leq 14) \wedge (-3x + 2y = 7) \wedge (y \geq 11)) \vee$ $((y \leq 11) \wedge (2x + y = 21) \wedge (y \geq 9))$

Fig. 1. Example of a (linear) spatial database.

and n free real variables such that, for every input database instance of signature $[m_1, n_1, \ldots, m_k, n_k]$, $\{(v_1, \ldots, v_m, x_1, \ldots, x_n) \mid \varphi(v_1, \ldots, v_m, x_1, \ldots, x_n)\}$ evaluates to the corresponding output database, which is of type $[m, n]$.

Example 2. Assuming that S is a relation of type $[0, 2]$, i.e., a semi-algebraic set in the plane, the FO + poly-formula

$$(\exists x_1)(\exists y_1)(\exists x_2)(\exists y_2)(\exists x_3)(\exists y_3)(\exists \lambda)(\exists \mu)(\exists \nu)(S(x_1, y_1) \wedge S(x_2, y_2) \wedge S(x_3, y_3) \wedge \lambda \geq 0 \wedge \mu \geq 0 \wedge \nu \geq 0 \wedge \lambda + \mu + \nu = 1 \wedge x = \lambda x_1 + \mu x_2 + \nu x_3 \wedge y = \lambda y_1 + \mu y_2 + \nu y_3.$$

defines the *convex-hull*[3] query of signature $[0, 2] \rightarrow [0, 2]$ which associates with S its convex hull. □

Real formulae not containing non-linear polynomials are called *linear formulae.* Point sets defined by linear formulae are called *semi-linear sets.*

The linear spatial data model is defined in the same way as the general spatial data model above using linear formulae instead of real formulae. Similarly, linear queries can be defined. Notice that a general query induces a linear query if the query restricted to linear database instances is linear. Observe that the convex-hull query (Example 2) induces a linear query. Queries of signature $[m_1, n_1, \ldots, m_k, n_k] \rightarrow [0, 0]$ are called *Boolean* queries, because the sets $\{()\}$ and $\{\}$ can be seen as encoding the truth values *true* and *false*, respectively. Since both these sets are semi-linear, every Boolean query induces a linear query.

A very appealing linear query language for the linear spatial data model, called FO + linear, is obtained from FO + poly by only allowing linear formulae rather than real formulae.

Example 3. The following FO + linear formula defines a Boolean (and hence linear) query of signature $[0, 2] \rightarrow [0, 0]$ deciding whether S is *convex*:

$$(\forall x_1)(\forall y_1)(\forall x_2)(\forall y_2)(\forall x_3)(\forall y_3)(S(x_1, y_1) \wedge S(x_2, y_2) \wedge 2x_3 = x_1 + x_2 \wedge 2y_3 = y_1 + y_2 \Rightarrow S(x_3, y_3).$$ □

We prove in Section 4, however, that not every linear query definable in FO+poly is definable in FO+linear. (In particular, we will show that the convex hull query introduced in Example 2 is not definable in FO + linear.) Therefore, it makes sense to define FO+poly^{lin} as the set of FO + poly-definable queries inducing linear queries. Thus, the set of queries definable in FO+poly^{lin} is a strict subset of the set of queries definable in FO + poly.

Throughout the paper, we use vector notation to denote points. In this notation, formulae should be interpreted coordinate-wise. Hence, $\neg(\mathbf{x} = \mathbf{0})$ indicates that $\mathbf{x}$ is not the origin of the coordinate system, whereas $\mathbf{x} \neq \mathbf{0}$ denotes that *none* of the coordinates of $\mathbf{x}$ equals 0.

[3] Let $A \subseteq \mathbf{R}^n$. The convex hull of A is the smallest convex set of $\mathbf{R}^n$ containing A. In particular, the convex hull of a semi-linear set is a semi-linear set.

3 Expressiveness of FO + linear

In this section, we discuss the expressiveness of the query language FO + linear. To simplify the discussion, we focus on purely spatial queries, i.e., queries acting on linear databases consisting of relations of a type of the form $[0, n]$.

First, we briefly review some known results involving linear queries computable in FO + linear.

The following operations on semi-linear sets can be defined rather trivially in FO + linear: union, intersection, difference, complement, and projection. In general, any affine transformation of semi-linear sets can be defined in FO + linear. In [26], FO+linear expressions are given for the Boolean queries checking boundedness, convexity, and discreteness of semi-linear sets. The expressive power of FO+linear unfolds completely, however, when topological properties of geometrical objects are considered. The definitions of topological interior, boundary, and closure can indeed be translated almost straightforwardly into linear calculus expressions. Hence, for example, the regularization of a semi-linear set, defined as the closure of its interior, can be computed in FO + linear, which is of importance, since the regularized set operators union, intersection, and difference, turn out to be indispensable in most spatial database applications [10, 19, 12]. More generally, Egenhofer et al. showed in a series of papers [7, 8, 9] that a whole class of topological relationships such as *disjoint*, *in*, *contained*, *overlap*, *touch*, *equal*, and *covered* can be defined in terms of intersections between the boundary, interior, and complement of the geometrical objects.

Another property of geometrical objects often used in spatial database applications, is *dimension*. For instance, in [5], the dimension is used to further refine the class of topological relationships defined by Egenhofer et al. We now show that it can be decided in FO + linear whether a given semi-linear set has a given number as its dimension, which is the contribution of this section. Since there are only finitely many known possibilities for the dimension of a semi-linear set, it follows that the dimension can actually be *computed* in FO + linear.

Definition 1. The *dimension* of a semi-linear set S of $\mathbf{R}^n$ is the maximum value of d for which there exists an open d-dimensional cube fully contained in S. The dimension of the empty set is defined as -1.

Theorem 2. *The predicate* $\dim(S, d)$*, in which* S *is a semi-linear set of* $\mathbf{R}^n$ *and* d *is a number, and which evaluates to* true *if the dimension of* S *equals* d*, can be defined in* FO + linear.

The correctness of this theorem follows from two lemmas we present next. We will use the notation $\pi_i(S)$, with S a semi linear set of $\mathbf{R}^n$, to denote the semi-linear set $\{(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) \mid (\exists x_i) S(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n)\}$ of $\mathbf{R}^{n-1}$. Hence, $\pi_i(S)$ is the projection of S onto the i-th coordinate hyper-plane of $\mathbf{R}^n$.

Lemma 3. *The dimension of* $\pi_i(S)$*, with* S *a* d*-dimensional semi-linear set of* $\mathbf{R}^n$*, is at most* d *for* $1 \leq i \leq n$.

Lemma 4. *If S is a d-dimensional semi-linear set of $\mathbf{R}^n$, with $d < n$, then there exists i, $1 \leq i \leq n$, such that the dimension of $\pi_i(S)$* equals *d.*

The rather technical proof of Lemma 4 is omitted due to space limitations.

Now define empty(S) as the FO + linear formula $\neg(\exists \mathbf{x})S(\mathbf{x})$, maxdim($S$) as the FO + linear formula $(\exists \mathbf{x})(\exists \epsilon)(\epsilon \neq \mathbf{0} \wedge (\forall \mathbf{y})(\mathbf{x} - \epsilon < \mathbf{y} < \mathbf{x} + \epsilon \Rightarrow S(\mathbf{y})))$, and, $\max(d, d_1, \ldots, d_n)$ as the FO + linear formula (expression omitted) which evaluates to *true* if d is the maximum of $d_1, \ldots, d_n$. Then the FO + linear formula $(d = -1 \wedge \text{empty}(S)) \vee (d = 1 \wedge \text{maxdim}(S)) \vee (d = 0 \wedge \neg\text{empty}(S) \wedge \neg\text{maxdim}(S))$ clearly defines $\dim(S, d)$ in $\mathbf{R}$. In general, the FO + linear formula $(d = n \wedge \text{maxdim}(S)) \vee (\neg\text{maxdim}(S) \wedge \dim(\pi_1(S), d_1) \wedge \ldots \wedge \dim(\pi_n(S), d_n) \wedge \max(d, d_1, \ldots, d_n))$, inductively defined, by Lemma 3 and 4 defines $\dim(S, d)$ in $\mathbf{R}^n$.

Many interesting queries can be defined in a natural way using the dimension predicate, and are therefore also definable in FO + linear, as is illustrated by the following example.

Example 4. The Boolean query which decides whether a semi-linear set S is a line or a line segment, is definable in FO + linear, using the following expression:

$$\dim(S, 1) \wedge (\forall \mathbf{x})(\forall \mathbf{y})(S(\mathbf{x}) \wedge S(\mathbf{y}) \Rightarrow S((\mathbf{x} + \mathbf{y})/2)).$$

It should be noted that Afrati et al.[1] independently showed that the line query is definable FO + linear. Their solution does not use the dimension predicate. □

A precise characterization of the expressive power of FO + linear is still open, however.

4 Limitations of FO + linear

Recently, Afrati et al. [2] established that FO+linear can not define all FO + poly definable linear queries:

Proposition 5. *The Boolean query on semi-linear sets S of $\mathbf{R}$ which evaluates to true if there exist u and v of S with $u^2 + v^2 = 1$, is not definable in* FO + linear.

Even though the query in Proposition 5 involves a non-linear computation in order to evaluate it, it is nevertheless a linear query because it is boolean, and therefore suffices to establish the incompleteness of FO + linear for the FO + poly-definable linear queries. The query, however, is unsatisfactory because it provides little insight into whether more natural, non-boolean FO + poly-definable linear queries are FO + linear-definable. Two such queries are (1) the linear query from semi-linear sets of $\mathbf{R}^n$ to semi-linear sets of $\mathbf{R}^n$ computing the convex hull (discussed in Example 2 for $n = 2$); and (2) the Boolean query on semi-linear sets of $\mathbf{R}^n$ which evaluates to *true* if all points in the input are colinear. In this section, we show that the above queries are *not* definable in FO + linear if $n \geq 2$. To demonstrate this claim, we build on the following results established by Afrati et al. [2] and the present authors [26]:

Proposition 6. *Let $n \geq 2$ and $m \geq 3$. Then the following sets are* not *definable in* FO + linear*:*

1. $\{(\mathbf{u}_1, \ldots, \mathbf{u}_m) \in (\mathbf{R}^n)^m \mid \mathbf{u}_m \in \text{convex-hull}(\{\mathbf{u}_1, \ldots, \mathbf{u}_{m-1}\})\}$*; and*
2. $\{(\mathbf{u}_1, \ldots, \mathbf{u}_m) \in (\mathbf{R}^n)^m \mid \mathbf{u}_1, \ldots, \mathbf{u}_m \text{ are colinear}\}$.

Even though the undefinability in FO+linear of the *sets* defined in Proposition 6 may suggest that the related *queries* (1) and (2) mentioned earlier are also non-definable in FO + linear, this deduction is not obvious. To see the caveat, it suffices to notice that the sets defined in Proposition 6 are not even semi-linear, whereas the related queries are obviously linear. This technical gap appears to have been overlooked in both the work of Afrati et al. [2] and previous work of the present authors [26]. In what follows, however, we show that there exists a general technique to link results about the non-definability in FO + linear of sets to the non-definability of certain related linear queries.

Definition 7. Let P be a *semi-algebraic* subset of $(\mathbf{R}^n)^m$, $m, n \geq 1$. Let k be such that $0 \leq k \leq m$. Furthermore assume that P and k are such that, for each l, $1 \leq l \leq k$, for each sequence $\mathbf{u}_1, \ldots, \mathbf{u}_l$ in $\mathbf{R}^n$, and for all sequences $i_1, \ldots, i_k$ and $j_1, \ldots, j_k$ such that $1 \leq i_1, \ldots, i_k, j_1, \ldots, j_k \leq l$ and $\{\mathbf{u}_{i_1}, \ldots, \mathbf{u}_{i_k}\} = \{\mathbf{u}_{j_1}, \ldots, \mathbf{u}_{j_k}\} = \{\mathbf{u}_1, \ldots, \mathbf{u}_l\}$, the following permutation invariance property holds for all $\mathbf{u}_{k+1}, \ldots, \mathbf{u}_m$ in $\mathbf{R}^n$:

$$(\mathbf{u}_{i_1}, \ldots, \mathbf{u}_{i_k}, \mathbf{u}_{k+1}, \ldots, \mathbf{u}_m) \in P \Leftrightarrow (\mathbf{u}_{j_1}, \ldots, \mathbf{u}_{j_k}, \mathbf{u}_{k+1}, \ldots, \mathbf{u}_m) \in P.$$

The query $Q_{P,k}$ of signature $[0, n] \to [0, n(m-k)]$ is now defined as follows. If S consists of at most k points of $\mathbf{R}^n$, say $S = \{\mathbf{u}_1, \ldots, \mathbf{u}_k\}$ ($\mathbf{u}_1, \ldots, \mathbf{u}_k$ not necessarily all distinct), then

$$Q_{P,k}(S) = \{(\mathbf{u}_{k+1}, \ldots, \mathbf{u}_m) \mid (\mathbf{u}_1, \ldots, \mathbf{u}_k, \mathbf{u}_{k+1}, \ldots, \mathbf{u}_m) \in P\};$$

otherwise $Q_{P,k}(S)$ is empty.

Observe that the invariance property assumed for P and k guarantees that $Q_{P,k}$ is well-defined.

Example 5. 1. *Let P be the set*

$$\{(\mathbf{u}_1, \ldots, \mathbf{u}_m) \in (\mathbf{R}^n)^m \mid \mathbf{u}_m \in \text{convex-hull}(\{\mathbf{u}_1, \ldots, \mathbf{u}_{m-1}\})\},$$

with $n \geq 2$ and $m \geq 3$. Let $k = m - 1$. Then $Q_{P,k}$ is the linear query that associates with each set S consisting of $m - 1$ points, the convex hull of S, and with every other set S the empty set. Notice that, by Property 6, the set P is not FO + linear*-definable.*

2. *Let P be the set*

$$\{(\mathbf{u}_1, \ldots, \mathbf{u}_m) \in (\mathbf{R}^n)^m \mid \mathbf{u}_1, \ldots, \mathbf{u}_m \text{ are colinear}\},$$

with $n \geq 2$ and $m \geq 3$. Let $k = m$. Then $Q_{P,k}$ can be interpreted as the Boolean query which evaluates a semi linear set S to true *if and only if S consists of m colinear points. Notice that, by Property 6, the set P is* not FO + linear*-definable.* □

We must emphasize that the linear queries in Example 5 are closely related, but *not* identical, to the linear queries (1) and (2) in the beginning of this section. One can think of the queries in Example 5 as *restrictions* of the linear queries (1) and (2) to certain finite sets.

We now prove the following theorem:

Theorem 8. *Let P be a* semi-algebraic *subset of $(\mathbf{R}^n)^m$ m, $n \geq 1$, and let P and k satisfy the conditions of Definition 7. If P is undefinable in* FO + linear, *then the following holds:*

1. *The query $Q_{P,k}$ is undefinable in* FO + linear.
2. *If Q is a linear query from semi-linear sets of $\mathbf{R}^n$ to semi-linear sets of $(\mathbf{R}^n)^{m-k}$ such that, for every semi-linear set S of $\mathbf{R}^n$, $Q(S) = Q_{P,k}(S)$ if $Q_{P,k}(S)$ is not empty, then Q is undefinable in* FO + linear.

Proof. 1. Assume, to the contrary, that the query $Q_{P,k}$ is FO + linear-definable. Then there exists an FO + linear formula $\varphi_{P,k}(R; \mathbf{x}_{k+1}, \ldots, \mathbf{x}_m)$, with R an appropriate predicate name, such that, for each semi-linear set S of $\mathbf{R}^n$, $Q_{P,k}(S) = \{(\mathbf{u}_{k+1}, \ldots, \mathbf{u}_m) \mid \varphi_{P,k}(S; \mathbf{u}_{k+1}, \ldots, \mathbf{u}_m)\}$. We now argue that the predicate name R must effectively occur in $\varphi_{P,k}$. If this were not the case, then the query associated with $\varphi_{P,k}$ would be a constant function. This constant function cannot yield the empty set, for, otherwise, by the definition of $Q_{P,k}$, P would also be the empty set, which is obviously FO + linear-definable, contrary the hypothesis of the theorem. The constant function cannot yield a non-empty set, either, however, since again by the definition of $Q_{P,k}$, there is an infinite number of inputs for which $Q_{P,k}$ returns the empty set. Thus R must occur in $\varphi_{P,k}$.

Given the formula $\varphi_{P,k}$, we can construct the formula $\hat{\varphi}_{P,k}$ as follows. Let $\mathbf{x}_1, \ldots, \mathbf{x}_k$ be variables that do not occur in $\varphi_{P,k}$. Now replace every literal of the form $R(\mathbf{z})$ in $\varphi_{P,k}$ by the formula $\mathbf{z} = \mathbf{x}_1 \vee \cdots \vee \mathbf{z} = \mathbf{x}_k$. Observe that the formula $\hat{\varphi}_{P,k}$ is a linear formula with free variables $\mathbf{x}_1, \ldots, \mathbf{x}_m$. Our claim is that the formula $\hat{\varphi}_{P,k}$ defines the set P, a contradiction with the hypothesis of the theorem. Consider an m-tuple $(\mathbf{u}_1, \ldots, \mathbf{u}_m) \in (\mathbf{R}^n)^m$. From the definition of $Q_{P,k}$ and $\varphi_{P,k}$, we have $(\mathbf{u}_1, \ldots, \mathbf{u}_m) \in P \Leftrightarrow (\mathbf{u}_{k+1}, \ldots, \mathbf{u}_m) \in Q_{P,k}(\{\mathbf{u}_1, \ldots, \mathbf{u}_k\})$, whence $(\mathbf{u}_1, \ldots, \mathbf{u}_m) \in P \Leftrightarrow \varphi_{P,k}(\{\mathbf{u}_1, \ldots, \mathbf{u}_k\}; \mathbf{u}_{k+1}, \ldots, \mathbf{u}_m)$. It follows from the construction of $\hat{\varphi}_{P,k}$ from $\varphi_{P,k}$ that $(\mathbf{u}_1, \ldots, \mathbf{u}_m) \in P \Leftrightarrow \hat{\varphi}_{P,k}(\mathbf{u}_1, \ldots, \mathbf{u}_m)$.
2. Assume that Q is FO + linear-definable. Then there exists a formula

$$\varphi_Q(R; \mathbf{x}_{k+1}, \ldots, \mathbf{x}_m)$$

that defines Q. Given φ_Q, we can construct the formula $\hat{\varphi}_Q$:

$$\hat{\varphi}_Q(R; \mathbf{x}_{k+1}, \ldots, \mathbf{x}_m) \Leftrightarrow (|R| \leq k \wedge \varphi_Q(R; \mathbf{x}_{k+1}, \ldots, \mathbf{x}_m)) \vee (|R| > k \wedge \mathit{false}).$$

It is obvious that this expression for $\hat{\varphi}_Q$ can be translated into proper FO + linear syntax. It now follows from the properties of Q that the formula $\hat{\varphi}_Q$ defines the query $Q_{P,k}$. Hence, it would follow that $Q_{P,k}$ is FO + linear-definable, which is impossible by the first part of the theorem. □

Theorem 8 has the following corollary:

Corollary 9. *The convex hull query* (1) *and the colinearity query* (2) *are not definable in* FO + linear.

5 Extensions of FO + linear

Although, in Section 3, it is shown that a wide range of useful, complex linear queries can be defined in FO + linear, the language lacks the expressive power to define some important FO+poly^{lin} queries, as is clearly demonstrated in Section 4. Hence the search for languages that capture such queries is important. Without such languages, we would indeed be hard-pressed to substantiate the claim that the linear model is to be adopted as the fundamental model for applications involving linear geometric objects.

The obvious way to obtain a query language which is complete for the FO+poly^{lin} queries is to discover an algorithm that can decide which FO + poly formulae induce linear queries. Unfortunately, such an algorithm does not exist:

Theorem 10. *It is undecidable whether an arbitrary* FO + poly *formula induces a linear query.*

Proof. (Sketch.) The proof is a variation of a proof by Paredaens et al. [22] concerning undecidability of genericity in FO + poly (Theorem 1, pp. 285). The $\forall^*$-fragment of number theory is undecidable since Hilbert's 10th problem can be reduced to it. Encode a natural number n by the one-dimensional semi-algebraic set $enc(n) := \{0, \ldots, n\}$, and encode a vector of natural numbers $(n_1, \ldots, n_k)$ by $enc(n_1) \cup (enc(n_2) + n_1 + 2) \cup \ldots \cup (enc(n_k) + n_1 + 2 + \cdots + n_{k-1} + 2)$. The corresponding decoding is first-order. We then reduce a $\forall^*$-sentence $(\forall \mathbf{x})\varphi(\mathbf{x})$ of number theory to the following query of signature $[0,1] \to [0,0]$:

if R encodes a vector $\mathbf{x}$ **then if** $\varphi(\mathbf{x})$ **then** $\emptyset$ **else** $\{(u,v) \mid u^2 + v^2 = 1\}$ **else** $\emptyset$.

This query is definable in FO + poly and induces a linear query if and only if the $\forall^*$-sentence is valid. □

Theorem 10 shows that a top-down approach to discover a useful linear sub-query language is difficult. Observe that Theorem 10 still allows the isolation of a subset of the FO + poly formulae that define FO+poly^{lin} queries, in the same way that the undecidability of safeness in the relational calculus is not in contradiction with the existence of a sub-language of the relational calculus which has precisely the expressive power of the safe relational calculus queries. [25]

In this section, we therefore take a bottom-up approach to discover restrictions of FO+poly^{lin} that are strictly more expressive than FO + linear. The basic idea is to extend FO + linear with certain linear operators, such as the colinearity or the convex-hull query. It is important to observe in this respect how careful we have to be to avoid creating languages that are no longer linear.

A too liberal syntax can indeed lead to the definability of non-semi-linear sets associated to these operators, such as the sets exhibited in Proposition 6. This in turn can have as a consequence that the language obtains the full expressive power of FO + poly, as is shown by the following example [26].

Example 6. Extending FO + linear with the convex-hull predicate (or with the colinearity predicate which can be derived from the former) as defined in [26] leads to a language with the expressive power of FO+poly. [26], the reason being that the predicate product(x, y, z) defined by $z = xy$ is can be expressed as[4]

$$\neg(\exists!\mathbf{u})((\text{colinear}(\mathbf{x}, \mathbf{e}_2, \mathbf{u}) \wedge \text{colinear}(\mathbf{y}, \mathbf{z}, \mathbf{u}))),$$

where $\mathbf{x} = (x, 0)$, $\mathbf{y} = (0, y)$, $\mathbf{z} = (z, 0)$, $\mathbf{u} = (u_1, u_2)$, and $\mathbf{e}_2 = (0, 1)$. □

We now proceed with showing how FO + linear can be extended with operators in a *safe* way. The subtle point of our definition consists in disallowing free *real* variables in set terms. So, even though a set-term might have free value variables, it is disallowed to have free real variables.

An *operator* is defined to be an FO+poly^{lin} query. The signature of an operator is the signature of that query.

Let $\mathcal{O}$ be a set of operator names O typed with a signature, each of which represents an operator op(O) of the same signature.

The query language FO + linear + $\mathcal{O}$ is then defined as an extension of FO + linear, as follows. First, we extend the terms of FO + linear with *set terms*:

- If φ is an FO + linear + $\mathcal{O}$ formula with n free real variables $x_1, \ldots, x_n$ and m free value variables $v_1, \ldots, v_m$, and if $k \leq m$, then

  $$\{(v_1, \ldots, v_k, x_1, \ldots, x_n) \mid \varphi(v_1, \ldots, v_m, x_1, \ldots, x_n)\}$$

  is a *set term* of type $[k, n]$. Observe that of the value variables, $v_{k+1}, \ldots, v_m$ occur *free*, while *all* real variables, $x_1, \ldots, x_n$, occur *bounded* in the set term.[5]
- If O is an operator name in $\mathcal{O}$ of type $[m_1, n_1, \ldots, m_k, n_k] \to [m, n]$, and $S_1, \ldots, S_k$ are set terms of types $[m_1, n_1], \ldots, [m_k, n_k]$, respectively, then

  $$O(S_1, \ldots, S_k)$$

  is a *set term* of type $[m, n]$ with as free variables those in the union of all free variables in S_1 through S_k (which are all value variables).

Finally, we extend the atomic formulae of FO + linear:

- Let S be a set term of type $[m, n]$. Then $S(v_1, \ldots, v_m, x_1, \ldots, x_n)$, with $v_1, \ldots, v_m$ value variables and $x_1, \ldots, x_n$ real variables, is an *atomic formula* with free variables $v_1, \ldots, v_m$, $x_1, \ldots, x_n$ union the free (value) variables of S.

[4] The quantifier "$\exists!$" should be read as "there exists exactly one" and can be expressed in FO in a straightforward manner.

[5] Observe that this definition allows us to interpret a predicate name R of type $[k, n]$ as a set term of type $[k, n]$.

When actual values are substituted for the free variables, a set term of type $[m, n]$ represents a subset of $U^m \times \mathbf{R}^n$. Consider then an atomic formula of the form $S(v_1, \ldots, v_m, x_1, \ldots, x_n)$, this atomic formula evaluates to *true* if the evaluation of $(v_1, \ldots, v_m, x_1, \ldots, x_n)$ belongs to the set represented by the set term S. The full semantics of FO + linear + $\mathcal{O}$ is now straightforward to define.

If we constrain the operator in $\mathcal{O}$ to be FO + linear-definable, we can prove the following safety property by induction on the structure of FO + linear + $\mathcal{O}$-formulae:

Theorem 11. *The query language* FO + linear + $\mathcal{O}$ *only expresses* FO + poly^{lin}-*definable queries.*

The syntactic restriction that set terms contain only free *value* variables is essential for Theorem 11 to hold; otherwise, e.g., the formula in Example 6 could be expressed in FO + linear+colinear, whence FO + linear+colinear would have the full expressive power of FO + poly.

Without going into details, we mention that it is possible to define an algebraic query language equivalent to FO + linear + $\mathcal{O}$ by extending the linear algebra [26] with the operators represented by $\mathcal{O}$. This equivalence result forms a theoretical justification for the approach Güting has taken with the development of the ROSE-algebra. [13, 14, 15, 16], which is extending the relational algebra with a class of spatial operators.

Finally, we give an example of an FO + linear + $\mathcal{O}$ query language in which we can express the queries (1) and (2) in the beginning of Section 4. Thereto, define an infinite set of operator names seg^k of signature $[0, k] \to [0, k]$ and associate with each operator name seg^k the operator $\text{op}(\text{seg}^k)$ defined by $\text{op}(\text{seg})^k(S) = \{\mathbf{x} \in \mathbf{R}^k \mid (\exists \mathbf{y})(\exists \mathbf{z})(S(\mathbf{y}) \wedge S(\mathbf{z}) \wedge \mathbf{x} \in [\mathbf{y}, \mathbf{z}])\}$ for each semi-linear set S of $\mathbf{R}^k$. Let $\mathcal{S}$ be the set of all seg^k, $k \geq 0$. Now let R be a predicate representing a semi-linear set of $\mathbf{R}^k$. The FO + linear + $\mathcal{S}$ formula

$$\underbrace{\text{seg}^k(\text{seg}^k(\ldots \text{seg}^k(R)))(\mathbf{x})}_{k \text{ times}}.$$

computes the convex hull of the set represented by R. Using the convex-hull query as a macro, we can express co-linearity by the following FO + linear + $\mathcal{S}$ formula:

$$(\exists d)(\dim(\{\mathbf{x} \mid \text{convex-hull}(S)(\mathbf{x})\}, d) \wedge d \leq 1).$$

6 Conclusion

In this paper we studied languages that define FO+poly^{lin} queries. Amongst these languages, the most natural one is FO + linear. For this language, we showed that non-trivial FO+poly^{lin} queries, such as the dimension query, can be defined in it, but we also demonstrated that important FO+poly^{lin} queries, such as the convex hull, cannot be defined. These latter results led us to the introduction of extensions of FO + linear with FO+poly^{lin}-definable operators.

The crucial part of this construction was requiring that operators can only be applied to set terms *without* free real variables. As we showed with a counter-example, our construction can lead to unsafe query languages if that restriction is lifted.

We conclude by mentioning the two most prominent open problems raised by this paper: (*i*) does there exist a syntactic restriction on FO + poly formulae that yields a sublanguage of FO + poly which is sound and complete for the FO+poly^{lin}-definable queries; and (*ii*) does there exist an extension of FO + linear (or other sublanguages of FO + poly) with operators that yields soundness and completeness?

References

1. F. Afrati, T. Andronikos, T.G. Kavalieros, "On the Expressiveness of First-Order Constraint Languages," in Proceedings *ESPRIT WG CONTESSA Workshop*, (Friedrichshafen, Germany), G. Kuper and M. Wallace, eds., *Lecture Notes in Computer Science*, vol. 1034, Springer-Verlag, Berlin, 1996, pp. 22–39.
2. F. Afrati, S. Cosmadakis, S. Grumbach, and G. Kuper, "Linear Versus Polynomial Constraints in Database Query Languages," in Proceedings *2nd Int'l Workshop on Principles and Practice of Constraint Programming* (Rosario, WA), A. Borning, ed., *Lecture Notes in Computer Science*, vol. 874, Springer-Verlag, Berlin, 1994, pp. 181–192.
3. A. Brodsky and Y. Kornatzky, "The LyriC Language: Querying Constraint Objects," in Proceedings *Post-ILPS'94 Workshop on Constraints and Databases* (Ithaca, NY), 1994.
4. I. Carlbom, "An Algorithm for Geometric Set Operations Using Cellular Subdivision Techniques," *IEEE Computer Graphics and Applications*, 7:5, 1987, pp. 44–55.
5. E. Clementini, P. Di Felice, and P. van Oosterom, "A Small Set of Formal Topological Relationships Suitable for End-User Interaction," in Proceedings *3nd Symposium on Advances in Spatial Databases*, *Lecture Notes in Computer Science*, vol. 692. Springer-Verlag, Berlin, 1993, pp. 277–295.
6. J. Nievergelt and M. Freeston, eds., Special issue on spatial data, *Computer Journal*, 37:1, 1994.
7. M.J. Egenhofer, "A Formal Definition of Binary Topological Relationships," in Proceedings *Foundations of Data Organization and Algorithms*, W. Litwin and H.-J. Schek, eds., *Lecture Notes in Computer Science*, vol. 367, Springer-Verlag, Berlin, 1989, pp. 457–472.
8. M.J. Egenhofer, and J. Herring, "A Mathematical Framework for the Definition of Topological Relationships," in Proceedings *Fourth International Symposium on Spatial Data Handling* , K. Brassel and H. Kishimoto, eds., Zurich, Switzerland, 1990, pp. 803–813.
9. M.J. Egenhofer, "Reasoning about Binary Topological Relations," in Proceedings *Advances in Spatial Databases*, O. Günther and H.-J. Schek, eds., *Lecture Notes in Computer Science*, vol. 525, Springer-Verlag, Berlin, 1991, pp. 143–160.
10. M.J. Egenhofer, "What's Special about Spatial? Database Requirements for Vehicle Navigation in Geographic Space," *SIGMOD Records*, 22:2, 1993, pp. 398–402.
11. O. Günther, ed., *Efficient Structures for Geometric Data Management*, in *Lecture Notes in Computer Science*, vol. 337, Springer-Verlag, Berlin, 1988.

12. O. Günther, and A. Buchmann, "Research Issues in Spatial Databases," *SIGMOD Records*, 19:4, 1990, pp. 61–68.
13. R.H. Güting, "Geo-Relational Algebra: A Model and Query Language for Geometric Database Systems," in *Advances in Database Technology—EDBT '88*, Proceedings *Int'l Conf. on Extending Database Technology* (Venice, Italy), J.W. Schmidt, S. Ceri, and M. Missikoff, eds., *Lecture Notes in Computer Science*, vol. 303, Springer-Verlag, Berlin, 1988, pp. 506–527.
14. R.H. Güting, "Gral: An Extensible Relational Database System for Geometric Applications," in Proceedings *15th Int'l Conf. on Very Large Databases* (Amsterdam, the Netherlands), 1989, pp. 33–34.
15. R.H. Güting, "An Introduction to Spatial Database Systems," *VLDB-Journal*, 3:4, 1994, pp. 357–399.
16. R.H. Güting, "Implementations of the ROSE Algebra: Efficient Algorithms for Real-Based Spatial Data Types," in Proceedings *Advances in Spatial Databases*, M. Egenhofer and J. Herring, eds., *Lecture Notes in Computer Science*, vol. 951, Springer-Verlag, Berlin, 1995, pp. 216–239.
17. T. Huynh, C. Lassez, and J.-L. Lassez. Fourier Algorithm Revisited. In Proceedings *2nd Int'l Conf. on Algebraic an Logic Programming*, H. Kirchner and W. Wechler, eds. *Lecture Notes in Computer Science*, vol. 463. Springer Verlag, Berlin, 1990, pp. 117–131.
18. P.C. Kanellakis and D.Q. Goldin, "Constraint Programming and Database Query Languages," in Proceedings *2nd Conf. on Theoretical Aspects of Computer Software*, M. Hagiya and J.C. Mitchell, eds., *Lecture Notes in Computer Science*, vol. 789, Springer-Verlag, Berlin, 1994.
19. A. Kemper, and M. Wallrath, "An Analysis of Geometric Modeling in Database Systems," *Computing Surveys*, 19:1, 1987, pp. 47–91.
20. P.C. Kanellakis, G.M. Kuper and P.Z. Revesz, "Constraint Query Languages," *Journal of Computer and System Sciences*, to appear, also in Proceedings *9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Nashville, TN), 1990, pp. 299–313.
21. A. Tarski, "A Decision Method for Elementary Algebra and Geometry," University of California Press, Berkeley, California, 1951.
22. J. Paredaens, J. Van den Bussche, and D. Van Gucht, "Towards a Theory of Spatial Database Queries," in Proceedings *13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Minneapolis, MN), 1994. pp. 279–288.
23. N. Pissinou, R. Snodgrass, R. Elmasri, I. Mumick, T. Özsu, B. Pernici, A. Segef. B. Theodoulidis, and U. Dayal, "Towards an Infrastructure for Temporal Databases," *SIGMOD Records*, 23:1, 1994, pp. 35–51.
24. L.K. Putnam and P.A. Subrahmanyam, "Boolean Operations on n-Dimensional Objects," *IEEE Computer Graphics and Applications*, 6:6, 1986, pp. 43–51.
25. J. D. Ullman, "Principles of Database and Knowledge-base Systems," Computer Science Press, 1988.
26. L. Vandeurzen, M. Gyssens, and D. Van Gucht, "On the Desirability and Limitations of Linear Spatial Query Languages," in Proceedings *4th Symposium on Advances in Spatial Databases*, M. J. Egenhofer and J.R. Herring, eds. *Lecture Notes in Computer Science*, vol. 951, Springer Verlag, Berlin, 1995, pp. 14–28.

Analysis of Heuristic Methods for Partial Constraint Satisfaction Problems*

Richard J. Wallace

University of New Hampshire, Durham, NH 03824 USA

Abstract. Problems that do not have complete solutions occur in many areas of application of constraint solving. Heuristic repair methods that have been used successfully on complete CSPs can also be used on over-constrained problems. A difficulty in analyzing their performance is the uncertainty about the goodness of solutions returned in relation to the optimal (best possible) solutions. This difficulty can be overcome by testing these procedures on problems that can be solved by complete methods, which return certifiably optimal solutions. With this experimental strategy, comparative analyses of hill-climbing methods were carried out using anytime curves that could be compared with known optima. In addition, extensive analysis of parameter values for key strategies such as random walk and restarting could be done precisely and efficiently by allowing local search to run until a solution was discovered that was known to be optimal, based on earlier tests with complete methods. An important finding is that a version of min-conflicts that incorporates the random walk strategy, with a good value for the walk probability appears to be as efficient in this domain as several of the more elaborate methods for improving local search that have been proposed in recent years.

1 Introduction

Constraint satisfaction problems (CSPs) involve finding an assignment of values to variables that satisfies a set of constraints between these variables. In many important applications the problems may be overconstrained, so that no complete solution is possible. In these cases, 'partial' solutions (i.e., assignments that do not satisfy all constraints in the problem) may still be useful if a sufficient number of the most important constraints are satisfied.

An important class of partial constraint satisfaction problems (PCSPs) is the maximal constraint satisfaction problem (MAX-CSP), in which the goal is to find assignments of values to variables that satisfy the maximum number of constraints. Since this problem involves assigning (equal) penalties for each constraint, methods for solving it can be readily generalized to accomodate constraint preferences or weighted constraints.

* This material is based on work supported by the National Science Foundation under Grant Nos. IRI-9207633. and IRI-9504316. Some of this material was presented at the Workshop on Overconstrained Systems at CP95.

In recent years methods have been developed for solving CSPs which are based on local improvements of an initial assignment that violates an unspecified number of constraints, rather than on incremental extensions of a fully consistent partial solution. These procedures are hill-climbing methods that search a space of solutions, following local gradients based on the number of violated constraints. In some cases these local or heuristic repair methods have solved problems that are much larger than those that can be solved by complete methods that involve backtrack search [8, 11].

In view of these results, it seems likely that heuristic repair techniques can be applied successfully to overconstrained problems, including MAX-CSPs. Since these are optimization problems, the basic question is whether such local search methods will return optimal (here, maximal) or near-optimal solutions to these problems after a reasonable amount of time. One reason for expecting them to do well is that hill-climbing methods do not embody the concept of a complete, or fully consistent, solution; instead they rely on notions of better or worse in comparing different assignments of values to variables. Since this concept of relative goodness does not change when PCSPs are tested rather than CSPs, methods that rely on it should not be hampered in this new domain. In contrast, the different notions of goodness used by complete CSP algorithms (based on the Boolean AND function) and by algorithms for PCSPs such as branch and bound (based on additive penalty functions) make the latter much harder to solve with these methods (cf. [10]).

To evaluate local search methods, one must be able to assess the quality of solutions that they find. For problems with complete solutions this evaluation is easy, since an optimal solution is one that is complete, i.e., it must satisfy all the constraints in the problem. Quality can then be judged in terms of the difference between the solution found and a complete solution. In this case, the number of violated constraints or the sum of the weights of these constraints is a straightforward measure of quality.

In contrast, with overconstrained problems one cannot determine optimality *a priori*. However, rigorous assessment is possible if complete methods can be used that return guaranteed optimal solutions. Solutions found by heuristic repair methods can then be compared with those found by complete methods, and differences in quality can be assessed in the same way as for problems with complete solutions.

Unfortunately, the size of problems for which this is possible is restricted because of limits on the capacity of complete methods to solve large problems. However, complete methods are now available for solving all problems in some classes of MAX-CSPs with 30-60 variables. This allows unbiased sampling of problems that are large enough to be interesting in some applications, and may also allow some assessment of trends with increasing problem size (as well as other parameters such as density and constraint tightness). For larger problems of, say, 100 variables, it may be possible to solve a portion of the problems with these methods. But in this case the analysis is hampered by the possible introduction of bias, since search methods may perform differently on easy and

hard problems and here only the former can be properly evaluated.

Thus, in addition to its primary concern with effectiveness of heuristic repair techniques, this paper is also a study of how to evaluate heuristic methods when problems do not have complete solutions.

The next section gives some background pertaining to CSPs and describes the heuristic repair methods. Section 3 outlines the basic experimental methodology, and Section 4 gives experimental results with heuristic methods for MAX-CSPs. Section 5 gives conclusions.

2 Algorithms

A constraint satisfaction problem (CSP) involves assigning values to *variables* that satisfy a set of *constraints* among subsets of these variables. The set of values that can be assigned to one variable is called the *domain* of that variable. In the present work all constraints are binary, i.e., they are based on the Cartesian product of the domains of two variables. A binary CSP is associated with a constraint graph, where nodes represent variables and arcs represent constraints. If two values assigned to variables that share a constraint are not among the acceptable value-pairs of that constraint, this is an *inconsistency* or constraint violation.

For MAX-CSPs, the number of constraint violations in an assignment, termed *the distance* of a solution, is used as a measure of quality. Thus, better solutions are those with lower distances, and solutions with the lowest possible distance, within the set of possible assignments, are *optimal* solutions.

Heuristic repair procedures for CSPs begin with a complete assignment and try to improve it by choosing alternative assignments that reduce the number of constraint violations. An important example is the min-conflicts procedure, which was the first hill-climbing method to be tested on CSPs [7]. This procedure has two phases. The first is a greedy preprocessing step, in which assignments are made to successive variables so as to minimize the number of constraint violations with values already chosen. This is followed by a hill-climbing phase, in which, at each step, a variable is chosen whose assignment conflicts with one or more assignments, and a value is chosen for that variable that minimizes the number of conflicts. Normally both variable and value selection involve an element of randomness: variables are chosen at random from all those that have conflicts, and values are chosen at random from the set of min-conflict values for the selected variable.

Limitations of the basic min-conflicts procedure for many kinds of CSPs, including random CSPs and coloring problems have often been demonstrated (e.g., [14]). Here it will be shown that its major drawbacks, which are also found with MAX-CSPs, can be overcome by adding well-known strategies for escaping local minima. These strategies are: (i) a retry strategy, in which the procedure starts again with a new assignment after a certain number of changes in the original one, and (ii) a random walk strategy, in which, after choosing a variable with a conflicting value as before, a new value is chosen at random for assignment, with

probability, p, while the usual min-conflicts procedure is followed with probability, $1-p$. In the initial studies (Section 4.1), values for retry and walk are based on published sources; the number of assignments before restarting is five times the number of variables [3], while the probability of a random choice in the walk procedure is always 0.35 [11]. Since these values were originally used for SAT problems, there is no reason to think that they are optimal for CSPs. Hence, later tests include systematic variation of the values for assignments before restarting and for the walk probability.

The best versions of min-conflicts are also compared with other strategies originally proposed for complete CSPs: (i) GSAT, in which the next reassignment is one that yields a maximal improvement in the solution (i.e., a maximal decrease in the number of constraint violations), and which also incorporates a retry strategy [6], (ii) breakout, in which constraint violations associated with a current local minimum are penalized by weighting them more strongly, so that alternative assignments are then preferred [9], (iii) EFLOP, in which on encountering a local minimum, one variable in conflict is given another assignment at random and then adjacent variables are given assignments that reestablish consistency with those just reassigned (the present version also uses the "EFLOP heuristic", in which variables are chosen to minimize the number of new violations) [14], (iv) weak commitment search, which starts with an initial full assignment and then tries to extend a partial set of variables that is completely consistent, and which restarts this process whenever a variable is found that cannot be added without engendering an inconsistency [15]. In the original version, discarded partial solutions were added to the problem as nogoods; however, it was found here that the proliferation of nogoods slowed down processing considerably even with an efficient storage and lookup mechanism. Hence, this feature is not used in the present version, without apparent decline in effectiveness. In addition, all procedures begin with an assignment generated by the greedy preprocessing procedure used with min-conflicts in order to facilitate comparisons. Some limited observations have also been made with simulated annealing [5] and a version of min-conflicts that incorporates tabu search procedures [2].

In the present work branch and bound versions of CSP algorithms are used to determine the optimal number of constraint violations in the problems. These algorithms are described in detail in [1, 13].

3 Experimental Methods

Random CSPs were generated using a "probability of inclusion" (PI) model of generation (cf. [1]). In the present case the number of variables was fixed, as well as domain size. Each possible constraint and constraint value pair was then chosen with a specified probability. These problems had either 30 or 100 variables, and domains were fixed at either 5 or 10. All 30–variable problems could be run to completion with the branch and bound methods used, so that quality of solutions returned by heuristic methods could be evaluated by comparisons with known optimal distances. Further details on parameter values for specific problem sets are given in Section 4.

In addition, 'geometric' CSPs were generated, using the procedure described in [5]. Briefly, variables are given random coordinates within the unit square; when coordinate points are within some specified distance of each other, a constraint is added between the associated variables. In contrast to homogeneous random problems, these problems exhibit more clustering of constraints in the constraint graph, a feature that can increase the difficulty for hill-climbing methods [6].

For all classes of problems, the sample size was 25.

There were two types of experiment. In the first, each heuristic method in that experiment is run for a fixed period. Then, for a set of specified times the number of violations in the best solution found so far is averaged across the sample of problems. The resulting average distances are displayed as "anytime curves", that show the quality of solution obtained after successively longer durations of processing. In the second type of experiment, heuristic methods are run with a sufficient bound on the distance that is equal to the optimal distance, determined earlier with complete methods. This allows runs to be terminated as soon as an optimal solution is found; at this point the next run begins. In these experiments, the specified cutoff time is essentially set to 'infinity', so that all runs continue until an optimal solution is found.

Anytime curves reported in Section 4.1 are based on the conflicts (constraint violations) in the initial solution, as well as the number of conflicts in the best solution found after 0.05, 0.1, 0.5, 1, 10, 50 and 100 seconds. In the present experiments, these curves are based on means of five runs of 100 seconds with each of the 25 problems in a sample. Means reported in Section 4.2 for the time to find an optimal solution are based on ten runs per problem. (In preliminary tests, standard deviations for ten means of single runs on the 25 problems in a sample were found to be about 5% of the grand means of the runs for each time tested. This indicates that for samples of this size the mean is quite stable, even for single runs.)

All procedures were coded in Common Lisp using Lispworks by Harlequin. Testing was done on a DEC Alpha (DEC3000 M300LX).

4 Experimental Results

4.1 Behavior of Min-Conflicts Procedures

In the first experiments, four sets of 30–variable problems were used with expected densities of either 0.10 or 0.50 and average optimal distances close to 2 or to 8.5. (These distances were obtained at each expected density by varying the expected tightness of constraints; cf. Figures 1-2 and Section 4.2.) Figure 1 shows results based on one problem set. This figure shows anytime curves for an efficient branch and bound algorithm based on forward checking, with variable ordering by decreasing number of constraints (degree of its node in the constraint graph), and for min-conflicts hill-climbing. The latter finds markedly better solutions early in search. Since it normally begins with a greedy preprocessing step,

the number of conflicts in the initial assignment is much less than the number of conflicts in the first solution found by branch and bound. For this reason, a version of min-conflicts that starts with a random assignment is also included for comparison. In this case the initial number of conflicts is almost equal to that for branch and bound. Nonetheless, the anytime curve for hill-climbing descends much more rapidly than the one for branch and bound, and after the first second of runtime, it finds solutions that are as good as those found by the basic version of min-conflicts.

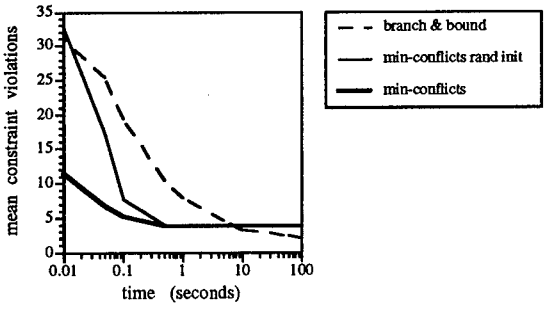


Fig. 1. Averaged anytime curves for forward checking branch and bound and min-conflicts (with and without greedy preprocessing). 30–variable problems, exp. density = 0.10, exp. tightness = 0.46, mean optimal distance = 2.08. Note the log scale on the abscissa.

Both versions of min-conflicts are, therefore, more efficient in finding good partial solutions than branch and bound at short intervals. However, as in the case of complete CSPs, min-conflicts quickly becomes 'stuck'; within five seconds it reaches a local minimum and remains at this level for the duration of the experiment. In fact, it does not seem to ever get 'unstuck' with these problems, as indicated by experiments with a cutoff time of 1000 seconds, in which no better solutions were found after the first few seconds.

Figure 2a shows anytime curves for the same problem set as in Figure 1, when strategies for escaping local minima are incorporated into min-conflicts. Although the basic min-conflicts strategy is superior at first, the curves for both the walk and retry strategies overtake the former after approximately one second, and within 10-50 seconds an optimal solution is found in the great majority of cases (specifically, in 105 of the 125 runs for walk and 119/125 for retry, versus 19/125 for the basic min-conflicts procedure).

Figure 2b shows corresponding curves for a set of problems having the same density but a greater number of violations in the optimal solution. The same pattern of results is observed as in the former experiment: all three variants show a rapid initial descent, but after approximately one second, the basic version reaches a plateau, while the other two continue to improve and eventually surpass the former.

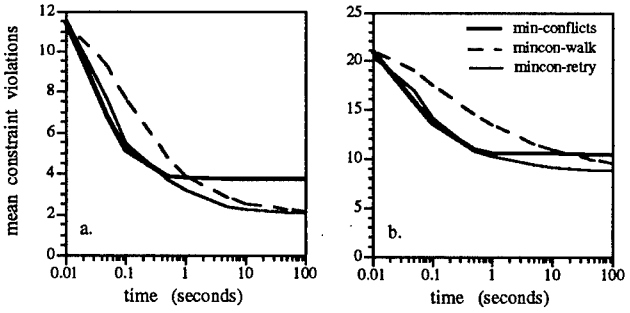


Fig. 2. Averaged anytime curves for basic min-conflicts versus min-conflicts with random walk and retry strategies. 30–variable problems, exp. density = 0.10. Problems in a. had tightness = 0.46, mean optimal distance = 2.08. Problems in b. had tightness = 0.60, mean optimal distance = 8.68.

A further experiment was run as a more direct test of the similarity of results with this heuristic procedure for problems with complete solutions and for PCSPs. A single set of 100 problems was generated near the phase transition [12], so that about 50% had solutions. (Problems had 30 variables, domain size of 5 and density = 0.10, as did the problems just discussed, and an expected tightness = 0.42.) The first 25 problems generated that either had a complete solution or did not formed the two groups in this case. These, of course, do not represent samples from a single population, but use of the same parameters for generation insures a fair degree of similarity. Tests with complete methods showed that the problems with complete solutions had more optimal solutions on average (788 thousand versus 36,000 for problems without complete solutions); on the other hand, the medians were much closer (29 and 15 thousand, respectively). Therefore, one might expect some differences in favor of the former set of problems, i.e., a more rapid descent to the mean optimal distance. But since this should pertain to a relatively small number of problems, the averaged curves should be similar.

In fact, anytime curves for both min-conflicts and min-conflicts plus walk (prob. = 0.35) closely parallel each other (Figure 3), and the average difference between procedures is almost identical for both sets of problems (Thus, after 100 sec the difference is 1.70 and 1.67 violations in favor of mincon-walk, for problems with and without solutions, respectively). In this experiment, min-conflicts with the walk strategy found optimal solutions on almost every run (124/125 runs after 50 seconds for problems with solutions and 121/125 runs for problems without solutions).

From these results it appears that min-conflicts behaves similarly on MAX-CSPs and on problems with complete solutions. Moreover, simple strategies for escaping local minima are quite effective, so that global minima can be obtained, often in a short period of time. In these experiments the retry strategy is su-

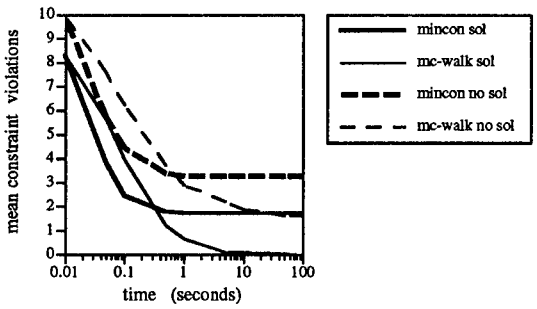


Fig. 3. Anytime curves for min-conflicts and mincon-walk for two problem sets based on a single set of parameters. Each problem in one set has a complete solution; problems in the other set do not have complete solutions (mean optimal distance = 1.56).

perior to the walk strategy for more difficult problems, especially those with greater average optimal distances. However, as shown in the next subsection, these results are highly dependent on the specific values used for the critical parameters: for retry, the number of reassignments before restarting, and for walk, the probability of making a random rather than a min-conflicts reassignment.

4.2 Parametric Analysis of Mincon-Walk and Mincon-Retry

The studies in this section were run to determine the best parameter values for the retry and walk strategies. The following procedure was used to obtain problems across the entire spectrum of densities while meeting the basic requirement of knowing the optimal distance for every problem tested. Values were chosen for expected density that covered most of the range, from 0.10 to 0.90. In each case, a value was chosen for the expected tightness of a constraint that gave problems with the same average optimal distance. This entailed decreasing the expected tightness with each increase in density. (If a single value is used for the tightness, optimal distance increases dramatically with increasing density, as in [13], and as shown there and elsewhere (e.g.,[10]), problems with greater optimal distances are much harder to solve with complete methods.) In addition, the effect of optimal distance was tested by generating sets of problems having the same densities but with different distances. In this way, the effect of each problem parameter could be separated.

All problems in this study had 30 variables. In the main series of problem sets, the domain size was 5. Five values were chosen for expected density: 0.1, 0.3, 0.5, 0.7 and 0.9. In addition there were two limited and widely separated ranges of average optimal distance: 1.52-2.12 and 8.64-8.68. For brevity, these are referred to as problems with distance 2 and distance 8. To obtain problems with a particular average optimal distance, expected tightness was varied, and 25 problems were generated for each parameter value, all of which were solved to completion. This process continued until a sample was obtained for a

given density with the desired average distance. For the lower optimal distance, problems were generated at all five densities. For the higher optimal distance, problems were generated at the three lower densities, since at higher densities problems were too difficult to solve with complete methods. (Note. These eight problem sets included the four used in the first experiments described in the last subsection.)

Table 1. Average time (seconds) required to find an optimal solution for different random problem classes and walk probabilities. Problems with distance 2.

	walk probability				
	0.05	0.10	0.15	0.25	0.35
			density = 0.10		
M	4	2	2	5	26
SD	8	3	3	8	62
			density = 0.30		
M	7	4	5	17	157
SD	10	6	7	40	378
			density = 0.50		
M	15	7	7	19	
SD	31	11	9	27	
			density = 0.70		
M	11	5	6	23	
SD	25	8	7	37	
			density = 0.90		
M	28	11	9	30	
SD	87	19	12	46	

Notes. M is mean, SD is standard deviation. 30–variable problems with mean optimal distance 1.5-2.

After the requisite problems had been collected and their optimal distances verified, this information was used in more systematic experiments with heuristic methods, to determine average time and number of constraint checks needed to find an optimal solution. In these experiments five values for the walk probability were tested: 0.05, 0.10, 0.15, 0.25, and 0.35. Eight different values were tested for the retry parameter (number of new assignments before restarting): 50, 100, 150, 250, 500, 1000, 2500, and 5000. Each set of 25 problems was run ten times with each parameter value. The mean of these 250 runs is the statistic shown in the tables in this section.

Tables 1 and 2 show results for walk probabilities, the first table for problems with distance 2, the second for problems with distance 8. Table 3 shows a more limited set of results for the retry parameter, for problems with distance 2. For either strategy and for each set of problems there is a curvilinear relation between parameter value and efficiency. For the walk strategy, the best probability values are 0.10-0.15 when the average optimal distance is 2. For problems with greater

Table 2. Average time (seconds) required to find an optimal solution for different classes of random problems and walk probabilities. Problems with distance 8.

	walk probability			
	0.05	0.10	0.15	0.25
	density = 0.10			
M	3	3	7	76
SD	5	4	11	232
	density = 0.30			
M	7	6	13	243
SD	8	8	18	468
	density = 0.50			
M	20	12	22	367
SD	40	16	33	616

optimal distances, the performance curve is shifted downward, with best results for probability 0.10 and next-best at 0.05. In both cases the performance ranking is the same throughout the range of densities tested. However, with increasing density, differences in performance are accentuated, since effort increases more rapidly with poorer parameter values.

The likely explanation for poorer results with high walk probabilities or low restart values is that search is cut off before a promising part of the solution space has been explored sufficiently. On the other hand, when the strategy is too conservative (low walk probability or high restart value), more time than necessary is spent in a particular part of the search space, so efficiency is diminished. It is not clear why the best walk probability is lower when the optimal distance is greater; this may be due the greater number of combinations of constraint violations with tighter constraints. In addition, problem difficulty increases for all settings with increased density, even with average optimal distance controlled; this may simply be due to the number of constraints that must be checked at each step.

Table 3. Average time (seconds) required to find an optimal solution for different classes of random problems and retry parameter values. Problems with distance 2.

	tries before restarting						
	50	100	150	250	500	1000	2500
	density = 0.10						
M	62	15	12	12	17	30	65
SD	238	35	34	33	62	98	222
	density = 0.50						
M		79	60	40	40	54	137
SD		212	164	85	78	102	255

In this extended study of walk and retry strategies, the best walk probability gave results that were clearly better than the best retry strategy, by a factor of 5-6. From these results we can see that the original walk probability of 0.35 was much too high for these problems, while the retry parameter value of 150 was close to the best. In addition, there is some evidence that the best value for walk probability varies less as a function of problem characteristics than the best value for assignments before restarting.

Table 4. Average number of optimal solutions per problem for problems with different densities and average optimal distance.

	dist $\approx$ 2	dist $\approx$ 8
	density = 0.10	
M	32,954	733
SD	139,080	1385
	density = 0.30	
M	203	52
SD	225	59
	density = 0.50	
M	375	
SD	762	
	density = 0.70	
M	375	
SD	762	
	density = 0.90	
M	64	
SD	128	

When the optimal distance is used as the initial upper bound, branch and bound can find all optimal solutions to problems like these with considerable efficiency. This allows one to count the number of optimal solutions for these problems. Table 4 shows the average number of optimal solutions for problems with different expected density and average optimal distance. These results can be compared with the performance of min-conflicts on these problems. The most important finding is that, for walk probabilities at least, a large decrease in the number of optimal solutions (from density = 0.1 to density = 0.3, and from distance 2 to distance 8) has only small effects on performance when the best parameter values are used.

Geometric problems also had 30 variables, a density of about 0.10, and expected tightnesses that gave distances similar to those of the random problems. The analysis of walk probabilites gave results that were similar to those with random problems (Table 5). The slightly shorter mean time to find an optimal solution for distance 2 problems may be due to the large number of optimal solutions in these problems (a mean of 728 thousand). Again, the marked reduc-

Table 5. Average time (seconds) required to find an optimal solution for different classes of geometric problems and walk probabilities.

	walk probability			
	0.05	0.10	0.15	0.25
	mn. distance = 2.56			
M	3	2	2	3
SD	8	4	5	7
	mn. distance = 8.28			
M	7	5	11	50
SD	19	16	47	216

tion in number of optimal solutions for problems with higher optimal distance (a mean of 6811 optimal solutions for problems with distance 8) is accompanied by only a small increase in run time for the best walk probabilites, while there is a much greater increase for probabilites elsewhere in the range.

4.3 Comparisons among hill-climbing procedures

In recent years a number of improvements have been proposed for hill-climbing search for CSPs, including the procedures outlined in Section 2. In evaluating these new procedures, the authors have usually compared them with min-conflicts in its basic form. In the present subsection, comparisons are made both with the basic min-conflicts and with mincon-walk, using the best value for the walk probability found in the studies recounted above. These tests were done using the set of 30–variable PI problems having an expected density of 0.50 and an average distance of 8.64. These were chosen as somewhat more difficult problems among the 30–variable series tested (cf. Table 2). Data were also collected for the problems with distance 2 and expected density = 0.10, which corroborate the findings presented here.

Comparisons of both versions of min-conflicts with GSAT for CSPs, breakout, EFLOP and weak commitment search are shown in Figure 4. (The curves for min-conflicts are the same in both 4a and 4b). Consistent with earlier demonstrations (see papers cited in Section 2), each of these four hill-climbing procedures is superior to the basic version of min-conflicts, which indicates that they are effective in escaping from local minima, and optimal solutions are found on most runs before the 100-second cutoff. GSAT, however, lags behind the other three, presumably because of the amount of checking involved at each step. (The failure of GSAT to outperform procedures that do not try to find the best move at each step corroborates results with SAT [4].) However, when the random walk strategy is added to min-conflicts, the resulting procedure does about as well as any of the others. In fact, it is the only one to find optimal solutions on all runs within 100 seconds.

Results with simulated annealing and tabu search, using parameter values suggested from the literature (e.g., a tabu list of length 7 and and candidate

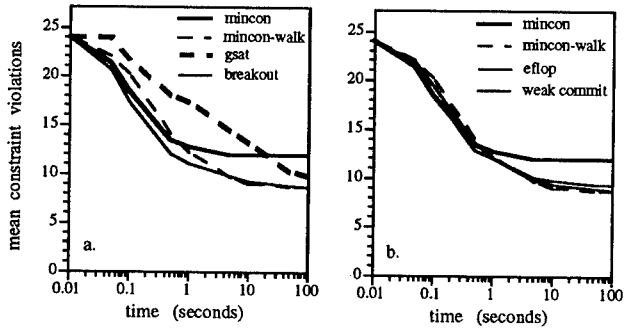


Fig. 4. Averaged anytime curves for heuristic repair algorithms. 30–variable problems, exp. density = 0.50, exp. tightness = 0.25, domain size = 5. In figure b. the curve for EFLOP is slightly higher than that for weak commitment at the end of the run.

list of length 10 or 20) did not improve on min-conflicts with walk for these problems. However, since these procedures can be parameterized in a variety of ways (and since the importance of good parameter values is well-demonstrated in the present work), these results must be considered tentative.

4.4 Results for larger problems

Further experiments were done with 100–variable PI problems with expected density = 0.10, domain size fixed at 5, and expected tightness = 0.25. These problems have *not* been solved to optimality with complete methods, so the average optimal distance is not known. It is, of course, still possible to make comparisons between procedures. For min-conflicts with walk probability = 0.05, the average best distance after 500 seconds was 9.56. Based on results with smaller problems generated in the same fashion, this is probably close to the optimum.

Figure 5a shows performance over time for three versions of min-conflicts. The walk and retry results are each part of a series: walk probabilities of 0.10, 0.15 and 0.25 and retries after every 250, 500, 1000, 5000 and 25,000 assignments were also tested. The parameter values shown in the figure were the best that were found in each series. (Note that the best value for walk probability is close to the best values found with 30–variable problems, while the best value for the retry parameter is much greater than the best value found with the smaller problems.) In a further test, the best walk and retry values were combined, but this did not improve on the walk probability shown here.

Figure 5b shows a comparison of mincon-walk with the two procedures that gave best results on the 30–variable problems. For these larger problems min-conflicts augmented with the simple walk strategy does as well or better than the other procedures that were designed to enhance hill-climbing search.

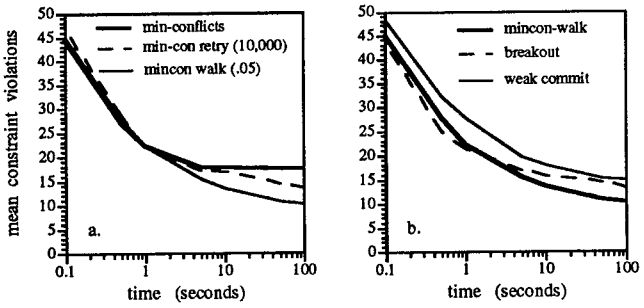


Fig. 5. Averaged anytime curves for heuristic repair algorithms. 100–variable problems, exp. density = 0.10, exp. tightness = 0.25, domain size = 5.

5 Conclusions

By combining branch and bound and heuristic methods within a single experimental paradigm, it has been possible to analyze the effectiveness of local search techniques for PCSPs with some precision. Although most of the work was done on random problems, the results appear to have some degree of generality, as indicated by findings with geometric problems. Obviously, this approach can be extended to other types of problems that are small enough to be solved by complete methods. The results of the last section also show that extensions to larger problems can be partly supported by more precise analysis of similar smaller problems.

This work provides a partial answer to the question posed in the Introduction. While the basic min-conflicts procedure does not usually find an optimal solution to a MAX-CSP, variants which introduce an element of randomization can find optimal solutions with great efficiency, at least for small to medium-sized problems. The effectiveness of these methods does depend on proper settings of parameter values in the procedure. However, for the most effective strategy the best settings were always within a fairly restricted range. These results also indicate that the random walk strategy is superior to a repeated restart strategy. In addition, there was no evidence of improvement when restarting was combined with the walk strategy.

The comparisons of Sections 4.3 and 4.4 raise some interesting questions about the various enhancements of hill-climbing that have been proposed in the past few years. All of them are successful in escaping from local minima. But it is not clear whether they offer any benefits other than those gained by using a simple random walk strategy, which is probably less expensive. In some cases, such as EFLOP, this strategy may be the critical feature in the procedure. (EFLOPing always begins with a random assignment to a variable in conflict.) Other procedures, such as GSAT and weak commitment, use the restarting strategy, which appears to be inferior, at least for PCSPs. GSAT also uses a strategy of finding the best move to make, which is very expensive and which does not

appear to pay off with appreciably better moves. Together, these results show how important it is to critically examine the features of a procedure, as done here and in the earlier work of [4].

References

1. E. C. Freuder and R. J. Wallace. Partial constraint satisfaction. *Artificial Intelligence*, 58:21–70, 1992.
2. F. Glover. Tabu search: a tutorial. *Interfaces*, 20:74–94, 1990.
3. I. Gent and T. Walsh. *The enigma of SAT hill-climbing procedures.* Research Paper No. 605, University of Edinburgh, 1992.
4. I. Gent and T. Walsh. Towards an understanding off hill-climbing procedures for sat. In *Proceedings AAAI-93*, pages 28–33, 1993.
5. D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Shevon. Optimization by simulated annealing: An experimental evaluation; part ii, graph coloring and number partitioning. *Operations Research*, 39:378–406, 1991.
6. K. Kask and R. Dechter. GSAT and local consistency. In *Proceedings IJCAI-95*, pages 6i6–622, 1995.
7. S. Minton, M. Johnston, A. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings AAAI-90*, pages 17–24, 1990.
8. S. Minton, M. D. Johnston, A. B. Philips, and P. Laird. Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58:161–205, 1992.
9. P. Morris. The breakout method for escaping from local minima. In *Proceedings AAAI-93*, pages 40–45, 1993.
10. T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings IJCAI-95*, pages 631–637, 1995.
11. B. Selman and H. A. Kautz. An empirical study of greedy local search for satisfiability testing. In *Proceedings AAAI-93*, pages 46–51, 1993.
12. B. Smith. Phase transition and the mushy region in constraint satisfaction problems. In *Proceedings ECAI-94*, pages 100–104, 1994.
13. R. J. Wallace. Directed arc consistency preprocessing as a strategy for maximal constraint satisfaction. In M. Meyer, editor, *Constraint Processing*, volume 923 of *Lecture Notes in Computer Science*, pages 121–138. Springer-Verlag, Heidelberg, 1995.
14. N. Yugami, Y. Ohta, and H. Hara. Improving repair-based constraint satisfaction methods by value propagation. In *Proceedings AAAI-94*, pages 344–349, 1994.
15. M. Yokoo. Weak-commitment search for solving constraint satisfaction problems. In *Proceedings AAAI-94*, pages 313–318, 1994.

Solving Satisfiability Problems Using Field Programmable Gate Arrays: First Results

Makoto Yokoo, Takayuki Suyama and Hiroshi Sawada

NTT Communication Science Laboratories
2-2 Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02, Japan
e-mail: yokoo/suyama/sawada@cslab.kecl.ntt.jp

Abstract. This paper presents an initial report on an innovative approach for solving satisfiability problems (SAT), i.e., creating a logic circuit that is specialized to solve each problem instance on Field Programmable Gate Arrays (FPGAs). Until quite recently, this approach was unrealistic since creating special-purpose hardware was very expensive and time consuming. However, recent advances in FPGA technologies and automatic logic synthesis technologies have enabled users to rapidly create special-purpose hardware by themselves.
This approach brings a new dimension to SAT algorithms, since all constraints (clauses) can be checked simultaneously using a logic circuit. We develop a new algorithm called *parallel-checking*, which assigns all variable values simultaneously, and checks all constraints concurrently. Simulation results show that the order of the search tree size in this algorithm is approximately the same as that in the Davis-Putnam procedure. Then, we show how the parallel-checking algorithm can be implemented on FPGAs. Currently, actual implementation is under way. We get promising initial results which indicate that we can implement a hard random 3-SAT problem with 300 variables, and run the logic circuit at clock rates of about 1MHz, i.e., it can check one million states per second.

1 Introduction

A constraint satisfaction problem (CSP) is a general framework that can formalize various problems in Artificial Intelligence, and many theoretical and experimental studies have been performed [9]. In particular, a satisfiability problem for propositional formulas in conjunctive normal form (SAT) is an important subclass of CSP. This problem was the first computational task shown to be NP-hard [3].

Virtually all existing SAT algorithms are intended to be executed on general-purpose sequential/parallel computers. As far as the authors know, there has been no study on solving SAT problems by creating a logic circuit specialized to solve each problem instance. This is because until quite recently, creating special-purpose hardware was very expensive and time consuming. Therefore, making a logic circuit for each problem instance was not realistic at all. However, due to recent advances in Field Programmable Gate Array (FPGA) technologies [1],

users can now create logic circuits by themselves, and reconfigure them electronically, without any help from LSI vendors. Furthermore, by using current automatic logic synthesis technologies [2], [11], users are able to design logic circuits automatically using a high level hardware description language (HDL). These recent hardware technologies have enabled users to rapidly create logic circuits specialized to solve each problem instance.

In this paper, we present an initial report on an innovative approach for solving SAT, i.e., creating a logic circuit that is specialized to solve each problem instance using FPGAs. This approach brings a new dimension to SAT algorithms, since all constraints (clauses) can be checked simultaneously using a logic circuit.

We develop a new algorithm called the *parallel-checking* algorithm. This algorithm has the following characteristics.

- Instead of determining variable values sequentially, all variable values are determined simultaneously, and all constraints are checked concurrently. Multiple variable values can be changed simultaneously when some constraints are not satisfied.
- In order to prune the search space, this algorithm introduces a technique similar to *forward checking* [8].

Simulation results show that the order of the search tree size in this algorithm is approximately the same as that in the Davis-Putnam procedure [5], which is widely used as a complete search algorithm for solving SAT problems.

Then, we show how the parallel-checking algorithm can be implemented on FPGAs by using recent hardware technologies. Currently, actual implementation is under way. We get promising initial results which indicate that we can implement a hard random 3-SAT problem with 300 variables, and run the logic circuit at clock rates of about 1MHz, i.e., it can check one million states per second.

In the remainder of this paper, we briefly describe the problem definition (Section 2), and describe the parallel-checking algorithm in detail (Section 3). Then, we show simulation results for evaluating the search tree size of this algorithm (Section 4). Furthermore, we show the way for implementing this algorithm on FPGAs and describe the status of the current implementation (Section 5). Finally, we discuss the relation of this algorithm with recently developed algorithms [4], [7], [6] that improve the Davis-Putnam procedure (Section 6).

2 Problem Definition

A satisfiability problem for propositional formulas in conjunctive normal form (SAT) can be defined as follows. A boolean *variable* x_i is a variable that takes the value true or false (represented as 1 or 0, respectively). In this paper, in order to simplify the algorithm description, we represent the fact that x_i is true as $(x_i, 1)$. We call the value assignment of one variable a *literal*. A *clause* is a disjunction of literals, e.g., $(x_1, 1) \vee (x_2, 0) \vee (x_4, 1)$, which represents a

logical formula $x_1 \vee \overline{x_2} \vee x_4$. Given a set of clauses $C_1, C_2, \ldots, C_m$ and variables $x_1, x_2, \ldots, x_n$, the satisfiability problem is to determine if the formula

$$C_1 \wedge C_2 \wedge \ldots \wedge C_m$$

is satisfiable. That is, is there an assignment of values to the variables so that the above formula is true.

In this paper, if the formula is satisfiable, we assume that we need to find all or a fixed number of solutions, i.e., the combinations of variable values that satisfy the formula. Most of the existing algorithms for solving SAT aim to find only one solution. Although this setting corresponds to the original problem definition, some application problems, such as visual interpretation tasks [14], and diagnosis tasks [13], require finding all or multiple solutions. Furthermore, since finding all or multiple solutions is usually much more difficult than finding only one solution, solving the problem by special-purpose hardware will be worthwhile. Therefore, in this paper, we assume that the goal is to find all or multiple solutions.

In the following, for simplicity, we restrict our attention to 3-SAT problems, i.e., the number of literals in each clause is 3. Relaxing this assumption is rather straightforward.

3 Algorithm

3.1 Basic Ideas

We are going to describe the basic ideas of the parallel-checking algorithm. This algorithm is obtained by gradually improving a simple enumeration algorithm.

Simple Enumeration Algorithm: We represent one combination of value assignments of all variables as n-digit binary value. Assuming that variable x_i's value is v_i, a combination of value assignments can be represented by an n-digit binary value $\Sigma_{i=1}^{n} 2^{i-1} v_i$, in which the value of i's digit (counted from the lowest digit) represents the value of x_i. We call one combination of all variable values one *state*. In this algorithm, the state is incremented from 0 to $2^n - 1$. For each state, the algorithm checks whether clauses are satisfied. If all clauses are satisfied, the state is recorded as a solution. Obviously, this algorithm is very inefficient since it must check all 2^n states.

Introducing Backtracking: When some clauses are not satisfied, instead of incrementing x_1's digit, we can increment the lowest digit that is included in these unsatisfied clauses; thus the number of searched states can be reduced. The algorithm obtained after this improvement is very similar to the backtracking algorithm where the order of the variable/value selection is fixed.

Introducing Forward Checking (i): Furthermore, instead of checking the current value (0 or 1) only, if we check another value concurrently, we can reduce the number of searched states. For example, assume that there exist variables x_1, x_2, x_3, x_4, x_5 and three clauses:

C_1: $(x_1, 1) \vee (x_4, 1) \vee (x_5, 1)$,
C_2: $(x_1, 0) \vee (x_3, 1) \vee (x_4, 1)$,
C_3: $(x_1, 0) \vee (x_2, 1) \vee (x_5, 1)$.

The initial state $\{(x_1, 0), (x_2, 0), (x_3, 0), (x_4, 0), (x_5, 0)\}$ does not satisfy C_1. If we increment x_1's digit and change x_1's value to 1, then C_2 and C_3 are not satisfied. If we perform the check for the case that x_1's value is 1, we can confirm that incrementing x_1's digit is useless.

In this case, which digit should be incremented? If x_1 is 0, C_1 is not satisfied, and the second lowest digit in C_1 is x_4. If x_1 is 1, C_2 and C_3 are not satisfied, and the second lowest digit in C_2 is x_3, while the second lowest digit in C_3 is x_2. Therefore, we can conclude that at least x_3's digit must be changed to satisfy all clauses; changing digits lower than x_3 is useless.

This procedure is similar to the backtracking algorithm that introduces *forward checking* [8], where backtracking is performed immediately after some variable has no consistent value with the variables that have already assigned their values.

Introducing Forward Checking (ii): Another procedure that greatly contributes to the efficiency of forward checking is to assign the variable value immediately if the variable has only one value consistent with the variables that have already assigned their values. This procedure is called *unit resolution* in SAT.

In order to perform a similar procedure in this algorithm, for each variable x_i, we define a value called unit(x_i). If unit(x_i)=j, there exists only one possible value for x_i, which is consistent with the upper digit variables[1], and the second lowest digit in the clause that is constraining x_i is x_j's digit.

For example, in the initial state $\{(x_1, 0), (x_2, 0), (x_3, 0), (x_4, 0), (x_5, 0)\}$ of the problem described in 3.1, x_1 has only one consistent value 1 by C_1. Therefore, we set unit(x_1) to 4 (since x_4 is the second lowest digit in C_1), and change x_1's value to 1. The value of unit(x_1) represents the fact that unless at least x_4's digit is changed, x_1 has only one possible value. The next state will be $\{(x_1, 1), (x_2, 0), (x_3, 1), (x_4, 0), (x_5, 0)\}$. This state does not satisfy C_3. Since the lowest digit in C_3 is x_1, x_1's value is changed in the original procedure. However, the value of unit(x_1) is 4 and the second lowest digit in C_3 is x_2, where x_2 is lower than x_4. Therefore, x_2's value is changed. The next state will be $\{(x_1, 1), (x_2, 1), (x_3, 1), (x_4, 0), (x_5, 0)\}$. This state satisfies all of the three clauses.

[1] When there exist multiple possible values, unit(x_i)=i.

3.2 Details of the Algorithm

Term Definitions: In the following, we define concepts and terms used in the algorithm.

- Each clause $(x_i, v_i) \vee (x_j, v_j) \vee (x_k, v_k)$ is converted to the following rule (where flip(0)=1, flip(1)=0):

$$\begin{aligned} &\text{if } (x_j, \text{flip}(v_j)) \wedge (\text{unit}(x_j) > i) \wedge (x_k, \text{flip}(v_k)) \wedge (\text{unit}(x_k) > i) \\ &\text{then} \qquad\qquad (x_i, \text{flip}(v_i)) \text{ is prohibited.} \end{aligned}$$

  It must be noted that one clause is converted to three different rules, since there are three possibilities for choosing the variable in the consequence part.
- Each rule is associated with the value flip(v_i) of variable x_i in the consequence part.
- We call a rule is *active* if the condition is satisfied in the current state.
- For value v_i of variable x_i, if some rule associated with v_i is active, we call value v_i is *prohibited.*
- For each active rule, if the variables in the condition part are x_j, x_k, we call min(unit(x_j), unit(x_k)) the *backtrack-position* of the rule.
- If value v_i of variable x_i is prohibited, the maximum of the backtrack-positions of active rules associated with v_i is called v_i's backtrack-position.
- When a state is given, we classify the condition of variable x_i in the following four cases:

  satisfied/free: both values are not prohibited.

  satisfied/constrained: the current value v_i is not prohibited, but flip(v_i) is prohibited.

  not-satisfied/possible: the current value v_i is prohibited, but flip(v_i) is not prohibited.

  not-satisfied/no-way: both values are prohibited.
- For variable x_i which is not-satisfied, we define the backtrack-position of x_i as follows:

  when x_i is not-satisfied/no-way: the minimum (the lower digit) of the backtrack-positions of values 0 and 1.

  when x_i is not-satisfied/possible: i (its own digit).

Parallel-Checking Algorithm: In the initial state, all variable values are 0, and the value of unit(x_i) is i.

1. For each value of each variable, concurrently check whether the associated rules are active. For a prohibited value, calculate the backtrack-position of the value. Calculate the condition and backtrack-position of each variable.
2. For each variable x_i, if its condition is satisfied/constrained, set the value of unit(x_i) to the backtrack-position of flip(v_i), where v_i is x_i's current value. Otherwise, set unit(x_i) to i.

3. Calculate m, which is the maximum of the backtrack-positions of not-satisfied variables. If all variables are satisfied, record the current state as a solution, and set m to 1.
4. Calculate max, which is the lowest digit that satisfies the following conditions: $max \geq m$, $v_{max} = 0$, and x_{max} is satisfied/free, where v_{max} is x_{max}'s current value. If there exists no value that satisfies these conditions, terminate the algorithm.
5. Change the value of x_{max} from 0 to 1. For each variable x_i which is lower than x_{max}, execute the following procedure:
   when x_i is satisfied/constrained, and unit(x_i) is larger than max: do not change x_i nor unit(x_i).
   when x_i is not-satisfied and for one of x_i's values v_i, v_i's backtrack-position is larger[2] than max: set x_i's value to flip(v_i), and set unit(x_i) to v_i's backtrack-position.
   otherwise: set x_i's value to 0, and set unit(x_i) to i.
6. Return to 1.

4 Simulation Results

In this section, we evaluate the efficiency of the parallel-checking algorithm by software simulation. We measured the number of searched states in the algorithm. For comparison, we used the Davis-Putnam procedure [5], which is widely used as a complete algorithm for solving SAT problems. The Davis-Putnam procedure is essentially a resolution procedure. It performs backtracking search by assigning the variable values and simplifying clauses. We call a clause that is simplified, such that it contains only one literal, a *unit clause*. When a unit clause is generated, the value of the variable that is contained in the unit clause is assigned immediately so that the unit clause is satisfied. This procedure is called *unit resolution*.

We use hard random 3-SAT problems as example problems. Each clause is generated by randomly selecting three variables, and each of the variables is given the value 0 or 1 (false or true) with a 50% probability. The number of clauses divided by the number of variables is called the *clause density*, and the value 4.3 has been identified as the critical value that produces particularly difficult problems [10].

In Fig. 1, we show the log-scale plot of the average number of visited states over 100 example problems, by varying the number of variables n, where the clause density is fixed to 4.3. The number of visited states in the Davis-Putnam procedure is the number of binary choices made during the search; it does not include the number of unit resolutions. Since a randomly generated 3-SAT problem tends to have a very large number of solutions when it is solvable, in order

[2] Since m is the maximum of the backtrack-positions of all not-satisfied variables, the backtrack-positions of both values can not be larger than max, which is larger than m.

to finish the simulation within a reasonable amount of time, we terminate each execution after the first 100 solutions are found.

We perform a simple variable rearrangement before executing these algorithms, i.e., the variables are rearranged so that strongly constrained variables (variables included in many clauses) are placed in higher digits, and the variables that are related by constraints (the variables included in the same clause) are placed as close as possible. The Davis-Putnam procedure utilizes this rearrangement by selecting a variable in the order from x_n to x_1 (except for unit resolutions).

From Fig. 1, we can see the following facts.

- The number of visited states in the parallel-checking algorithm is three to eight times larger than that in the Davis-Putnam procedure. This result is reasonable since the number of states in the Davis-Putnam procedure does not include the number of unit resolutions, while the number of states in the parallel-checking includes state transitions that are caused by unit resolutions.
- The order of visited states (the order of the search tree size) in the parallel-checking algorithm is approximately the same as that in the Davis-Putnam procedure, i.e., for each algorithm, the number of visited states grows at the same rate as the number of variables increases.

The computation executed for each state in the Davis-Putnam procedure is in the order of $O(n)$ (which includes repeated applications of unit-resolutions). On the other hand, the computation executed for each state in the parallel-checking algorithm can be finished in one clock[3] when the algorithm is implemented on FPGAs.

These results indicate that the parallel-checking algorithm implemented on FPGAs will be much more efficient than the Davis-Putnam procedure implemented on a general-purpose computer.

5 Implementation

In this section, we give a brief description of FPGAs. Then, we show how the parallel-checking algorithm can be implemented on FPGAs, and report the current status of our implementation.

5.1 Field Programmable Gate Arrays

An example of FPGA architecture is shown in Fig. 2. It consists of a two-dimensional array of programmable logic blocks, with routing channels between

[3] The required time for one clock is not constant, since the possible clock rate of a logic circuit is determined by the delay of the logic circuit, and the delay is certainly affected by the problem size n. However, the order would be much smaller than $O(n)$, i.e., at most $O(log(n))$.

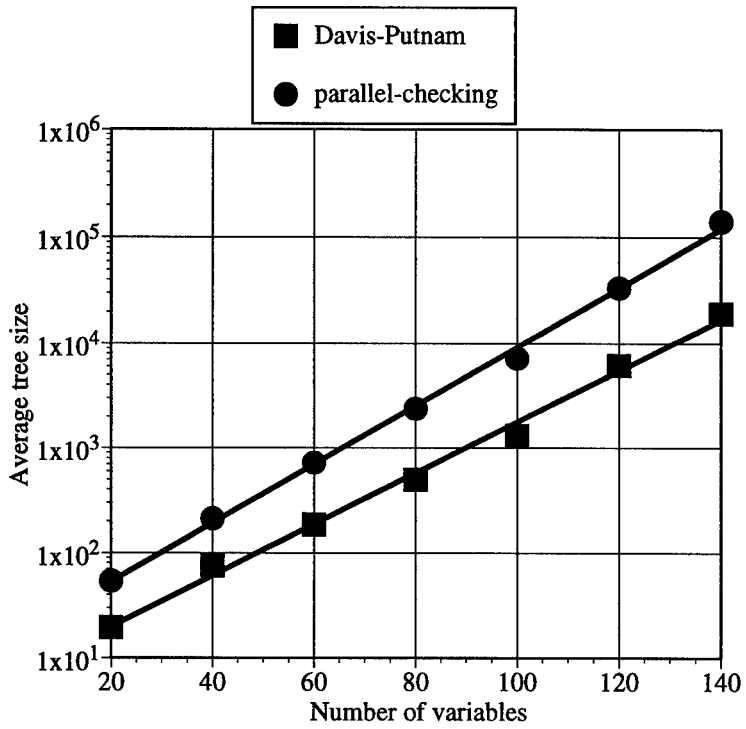


Fig. 1. Results for parallel-checking and Davis-Putnam procedure on hard random 3-SAT problems

these blocks. These logic blocks and interconnections are user-programmable by rewriting static RAM cells. We use an FPGA hardware system called ZyCAD RP2000 [16]. This system has 32 FPGA chips (each chip is a Xilinx XC4010), and can implement a large-scale logic circuit by dividing it into multiple FPGA chips. The equivalent gate count of a Xilinx XC4010 is about 8.0k to 10.0k.

5.2 Logic Circuit Configuration

We show the configuration of the logic circuit that implements the parallel-checking algorithm in Fig. 3. The logic circuit consists of the following three functional units.

1. Rule Checker
2. Next State Generator
3. Next Unit Generator

In the Rule Checker, the condition and the backtrack-position of each digit are calculated from the current state and the unit values in parallel. The Next State

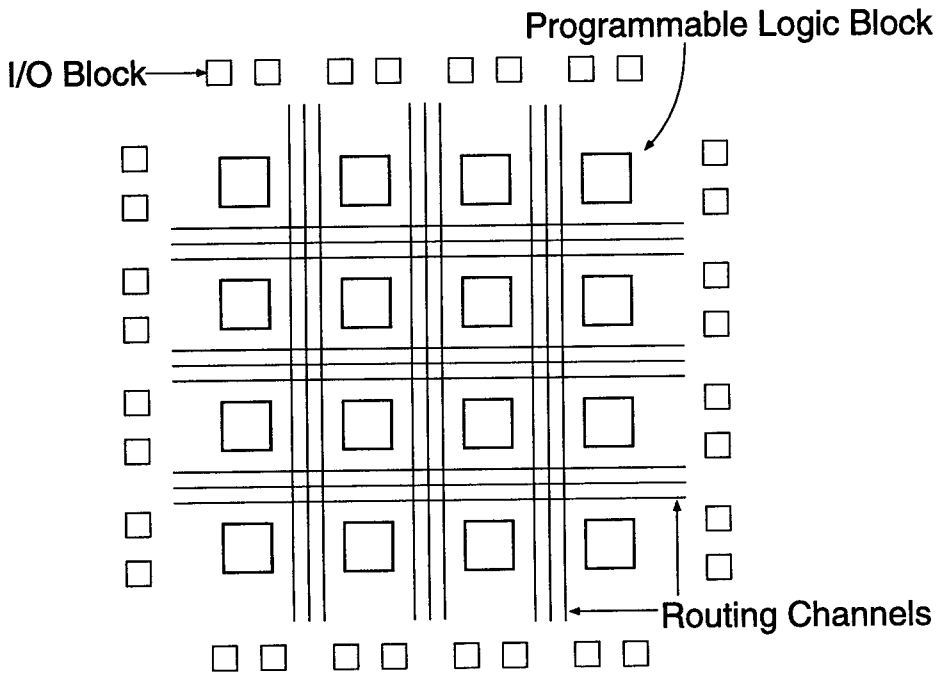


Fig. 2. General architecture of FPGA

Generator first calculates max, i.e., the digit that must be incremented, using the outputs of the Rule Checker. Then, the Next State Generator calculates the next state by incrementing the digit of max. The value of each lower digit is determined by its condition and backtrack-position. The Next Unit Generator calculates the unit values in the next state, using the outputs of the Rule Checker and max. These calculated values (the next state and next unit values) are used as feedback and stored in registers.

5.3 Logic Circuit Synthesis

A logic circuit that solves a specific SAT problem is synthesized by the following procedure (Fig. 4). First, a text file that describes a SAT problem is analyzed by an SFL generator written in the C language. This program generates a behavioral description specific to the given problem with an HDL called SFL. Then, a CAD system analyzes the description and synthesizes a netlist, which describes the logic circuit structure. We use a system called PARTHENON [2], [11], which was developed at NTT. PARTHENON is a highly practical system that integrates a description language, simulator, and logic synthesizer. Furthermore, the FPGA Mapper of the Zycad system generates FPGA mapping data for RP2000 from the netlist.

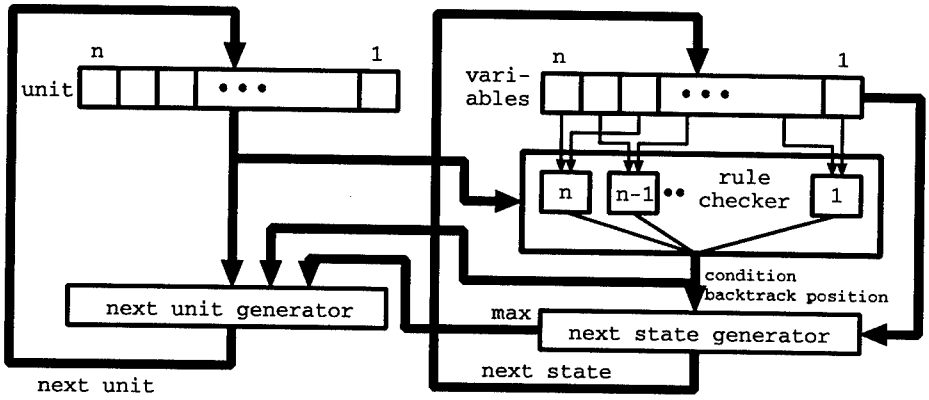


Fig. 3. Logic circuit configuration

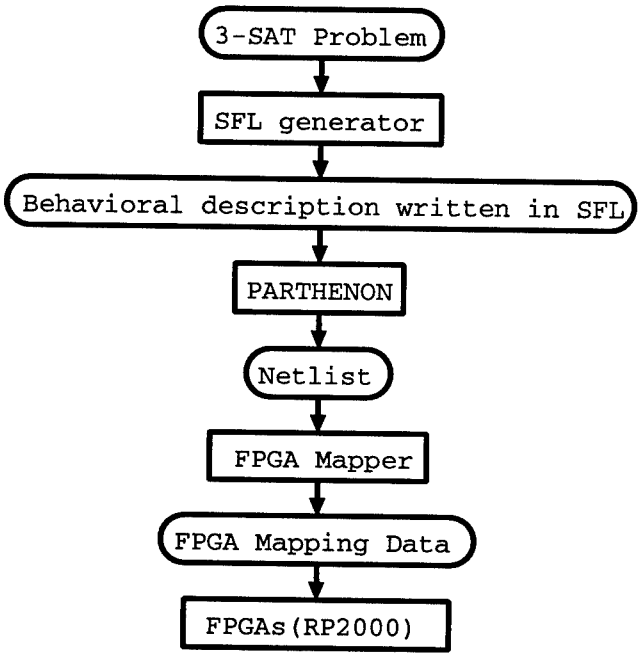


Fig. 4. Flow of logic circuit synthesis

5.4 Current Implementation Status

We have developed an SFL generator which generates a description of the logic circuit that implements the parallel-checking algorithm. By using this program, we have successfully implemented a hard random 3-SAT problem with 128 variables and 550 clauses using our current hardware resources. This logic circuit is capable of running at the clock rates of at least 1MHz, which is the maximal setting of the clock generator we are currently using. Therefore, we can assume that it is capable of running at higher clock rates. The logic circuit for a 128-variable problem fits in the current hardware resources. Since the number of FPGA chips in this system can be increased up to 64 (twice as many as the current configuration) and the mapping quality can stand further improvement, we could possibly implement much larger problems, e.g., a problem with 300 variables, without any trouble[4]. We are increasing our hardware resources and trying to implement much larger problems.

Currently, generating a logic circuit from a problem description takes a few hours. This is because we are using general-purpose synthesis routines. These routines can be highly optimized for SAT problems since many parts in a logic circuit are common in all problem instances. The required time for generating a logic circuit could be reduced to at most several ten minutes.

6 Discussions

Recently, several improved versions of the Davis-Putnam procedure have been developed [4], [6], [7]. These algorithms use various sophisticated variable/value ordering heuristics. How good is the parallel-checking algorithm compared with these algorithms? Unfortunately, these algorithms aim to find only one solution, and various procedures for simplifying formulas, such as removing variable values that do not affect the satisfiability of the problem, are introduced. Therefore, the evaluation results of these algorithms can not be compared directly with the results of the parallel-checking algorithm. It is not very straightforward to modify these algorithms so that they can find all solutions. In our future works, we are going to examine these algorithms carefully, modify them so that they can find all solutions, and compare the modified algorithms with the parallel-checking algorithm. Furthermore, we are going to examine the possibility of introducing the heuristics used in these algorithms into the parallel-checking algorithm.

7 Conclusions and Future Works

This paper presented an initial report on solving SAT using FPGAs. In this approach, a logic circuit specific to each problem instance is created on FPGAs. This approach brings a new dimension to SAT algorithms since all constraints

[4] Of course, the efficiency of the parallel-checking algorithm must be improved in order to solve such a large-scale problem within a reasonable amount of time.

can be checked in parallel using a logic circuit. We developed a new algorithm called *parallel-checking*, which assigns all variable values simultaneously, and checks all constraints concurrently. Simulation results showed that the order of the search tree size in the parallel-checking algorithm is approximately the same as that in the Davis-Putnam procedure, which is widely used as a complete algorithm for solving SAT problems. We have implemented a hard random 3-SAT problem with 128 variables, and run the logic circuit at clock rates of about 1MHz, i.e., it can check one million states per second. Currently, we are increasing our hardware resources so that much larger problems can be implemented. We are going to perform various evaluations on implemented logic circuits.

Our future works include comparing this approach to recently developed algorithms [4], [6], [7] that improve the Davis-Putnam procedure, and introducing the heuristics used in these algorithms into the parallel-checking algorithm. Furthermore, we are going to implement iterative improvement algorithms for solving SAT [12], [15] on FPGAs.

Acknowledgments

The authors wish to thank K. Matsuda for supporting this research. We also appreciate helpful discussions with N. Osato and A. Nagoya.

References

1. Brown, S. D., Francis, R. J., Rose, J., and Vranesic, Z. G.: *Field-Programmable Gate Arrays*, Kluwer Academic Publishers (1992)
2. Camposano, R. and Wolf, W.: *High-level VLSI synthesis*, Kluwer Academic Publishers (1991).
3. Cook, S.: The complexity of theorem proving procedures, *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computation* (1971) 151–158
4. Crawford, J. M. and Auton, L. D.: Experimental results on the crossover point in satisfiability problems, *Proceedings of the Eleventh National Conference on Artificial Intelligence* (1993) 21–27
5. Davis, M. and Putnam, H.: A computing procedure for quantification theory, *Journal of the ACM*, Vol. 7, (1960) 201–215
6. Dubois, O., Andre, P., Boufkhad, Y., and Carlier, J.: SAT versus UNSAT, in Johnson, D. S. and Trick, M. A. eds., *Proceedings of the Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science (1993)
7. Freeman, J. W.: *Improvements to propositional satisfiability search algorithms*, PhD thesis, the University of Pennsylvania (1995)
8. Haralick, R. and Elliot, G. L.: Increasing tree search efficiency for constraint satisfaction problems, *Artificial Intelligence*, Vol. 14, (1980) 263–313
9. Mackworth, A. K.: Constraint satisfaction, in Shapiro, S. C. ed., *Encyclopedia of Artificial Intelligence*, Wiley-Interscience Publication, New York (1992) 285–293, second edition
10. Mitchell, D., Selman, B., and Levesque, H.: Hard and easy distributions of SAT problem, *Proceedings of the Tenth National Conference on Artificial Intelligence* (1992) 459–465

11. Nakamura, Y., Oguri, K., Nagoya A., Yukishita M., and Nomura R.: High-level synthesis design at NTT Systems Labs, *IEICE Trans. Inf & Syst.*, Vol. E76-D, No.9, pp. 1047–1054 (1993).
12. Morris, P.: The breakout method for escaping from local minima, *Proceedings of the Eleventh National Conference on Artificial Intelligence* (1993) 40–45
13. Reiter, R.: A theory of diagnosis from first principles, *Artificial Intelligence*, Vol. 32, No. 1, (1987) 57–95
14. Reiter, R. and Mackworth, A.: A logical framework for depiction and image interpretation, *Artificial Intelligence*, Vol. 41, No. 2, (1989) 125–155
15. Selman, B., Levesque, H., and Mitchell, D.: A new method for solving hard satisfiability problems, *Proceedings of the Tenth National Conference on Artificial Intelligence* (1992) 440–446
16. Zycad Corp.: *Paradigm RP Concept Silicon User's Guide, Hardware Reference Manual, Software Reference Manual* (1994).

A Constraint Program for Solving the Job-Shop Problem

Jianyang ZHOU[†]

Laboratoire d'Informatique de Marseille

Abstract

In this paper, a method within the framework of propagation of interval constraints and based on the branch-and-bound optimization scheme for solving the job-shop scheduling problem will be presented. The goal is to provide a constraint program which is clean, flexible and robust. The design of the constraint program is based on an idea of sorting the release and due dates of tasks, which is a successful application of a previous but not yet published work on a *distinct integers* constraint. Based on the sorting constraint, by assembling redundant constraints and applying an efficient search strategy, the current program for the job-shop problem can solve the ten 10×10 instances in the paper of Applegate and Cook (1991) in satisfactory computational time. Moreover, good results have been achieved on some harder instances.

Keywords: *Constraint Programming, Interval Constraints, Constraint of Distinct Integers, Permutation, Sorting, Job-shop Scheduling*

1 Introduction

Given n jobs each consisting of m tasks that have to be processed on m machines, the job-shop problem requires to schedule the jobs on the machines so as to minimize the maximum of the completion times of all jobs, subject to:

- the order $(\tau_{1j}, \ldots, \tau_{mj})$ of the m machines to process any job j is known (the precedence constraint);
- the duration d_{ij} of job j processed on machine i is known (the duration constraint);

*This work has been supported by the Esprit Project Acclaim n° 7195

[†]Corresponding author (zhou@lim.univ-mrs.fr), Laboratoire d'Informatique de Marseille, 163 avenue de Luminy, 13288 Marseille Cedex 9, France

- on any machine, at any moment, there can be at most one task processed (the disjunctive constraint).

This problem is NP-hard in the strong sense [11]. A famous instance for it is the 10×10 `mt10` posed in 1963 by Muth and Thompson [13], which had resisted efforts of many researchers for over 20 years before it was solved by Carlier and Pinson in 1989 [4]. The approach of Carlier and Pinson is to settle the disjunctive constraints by ordering task pairs with edge-finding search strategy which concentrates on possible first or last tasks of a certain set of tasks on a certain machine to establish a search tree [4, 1, 2, 7]. To our knowledge, many successful algorithms for solving the job-shop problem are based on such a resolution scheme. In this paper, we propose a new way based on the idea of sorting the release and due dates of jobs to solve the job-shop problem. The principal distinction from the method of Carlier and Pinson is: with the distinct integers constraints used, the disjunctive constraints of the job-shop problem will be settled more globally than concentrating only on the orders of task pairs and a more global search strategy oriented by the distinct integers constraints will be applied. The technique used is the system of interval constraints [3, 8, 10, 15]. The goal is to give a constraint program which is clean, flexible and robust.

2 The constraint solving paradigm

2.1 Introduction

In 1987, Cleary introduced the interval method into logic programming for logical arithmetic [8]. This technique was then developped for constraint systems at Bell-Northern Research (BNR) in BNR Prolog [15]. As a new approach for solving constraints, though the interval method possibly has its drawback: relatively poor pruning efficiency due to the only-on-the-two-bounds domain reduction, this method does not lack advantages: it provides a simple and clean technique for constraint solving which makes it promising to solve constraint systems of large scale; it is a good candidate for dealing with continuous problems, and a good candidate for systems of mixed constraints, which brings about elegant operational symantics. In this paper, to pursue simplicity, we deal with only intervals of integers.

2.2 Notations and definitions

Let $\mathbf{Z}$ be the set of all integers. An *interval* I is a subset of $\mathbf{Z}$ which is of the form $\{k \in \mathbf{Z} \mid i \leq k \leq j\}$, where i, j are elements of $\mathbf{Z} \cup \{\pm\infty\}$, the order relation $\leq$ being extended in the way that $-\infty \leq k \leq +\infty$ for all integer k. Such an interval I will be denoted by $[i, j]$.

Let $\mathbf{V}$ be a set of variables. We will assume that to each variable x of $\mathbf{V}$ an interval $\mathtt{dom}(x)$, called the *domain* of x, is associated. To simplify our notations the *upper bound* and *lower bound* of $\mathtt{dom}(x)$ are denoted respectively by $\overline{x}$ and $\underline{x}$. The *cardinality* of $\mathtt{dom}(x)$ is denoted by $|x|$, it is either an integer or ∞.

A *constraint* over $\mathbf{V}$ is an expression of the form $r(x_1,\ldots,x_n)$, where the x_i's are distinct variables taken from $\mathbf{V}$ and where r is an n-ary relation over $\mathbf{Z}$, that is to say a subset of $\mathbf{Z}^n$. In this paper, we are interested in dealing with 16 kinds of constraints which are displayed in table 1. They will be used in the constraint program for solving the job-shop problem.

A *system* Φ of constraints is a conjunction of constraints over $\mathbf{V}$. A mapping σ from $\mathbf{V}$ to $\mathbf{Z}$ is a *solution* of Φ iff for all constraint $r(x_1,\ldots,x_n)$ of Φ, $(\sigma(x_1),\ldots,\sigma(x_n)) \in r$. A mapping σ' from subset V' of V to $\mathbf{Z}$ is called a *partial solution* of Φ iff there exists a solution σ of Φ such that $\sigma(x) = \sigma'(x)$ for all $x \in V'$.

For any subset q of $\mathbf{Z}^n$, $\texttt{hull}(q)$ denotes the smallest cartesian product of intervals containing q and $\texttt{proj}_i(q)$ denotes the i-th projection of q. For almost all relations r of table 1 we have developped a *narrowing* algorithm that, given an n-uple $(I_1,\ldots,I_n)$ of intervals, computes perfectly $\texttt{hull}(r \cap I_1 \times \cdots \times I_n)$. Special attention has been devoted to the *distinct integers* relation involved in constraint $x_1 \neq \cdots \neq x_n$. The narrowing algorithms are used to remove inconsistent values from variable domains by computing $\texttt{hull}(r \cap \texttt{dom}(x_1) \times \cdots \times \texttt{dom}(x_n))$ for constraint $r(x_1,\ldots,x_n)$.

Table 1: 16 primitive constraints

Notation for $r(x_1,\ldots,x_n)$	Relation r, viewed as a set of n-uples of integers
$x_1 \leq x_2$	$\{(p_1,p_2) \mid p_1$ is less than or equal to $p_2\}$
$x_3 = x_1 + x_2$	$\{(p_1,p_2,p_3) \mid p_3$ is the sum of p_1 and $p_2\}$
$x_1 \in D$	$\{p_1 \mid p_1$ is element of $D\}$
$x_2 \equiv \neg x_1$	$\{(p_1,p_2) \mid (p_2 = 0 \wedge p_1 = 1) \vee (p_2 = 1 \wedge p_1 = 0)\}$
$x_1 \Rightarrow x_2$	$\{(p_1,p_2) \mid p_1 = 0 \vee (p_1 = 1 \wedge p_2 = 1)\}$
$x_2 \equiv (x_1 \in D)$	$\{(p_1,p_2) \mid (p_2 = 0 \wedge p_1 \notin D) \vee (p_2 = 1 \wedge x_1 \in D)\}$
$x_3 \equiv (x_1 = x_2)$	$\{(p_1,p_2,p_3) \mid (p_3 = 0 \wedge p_1 \neq p_2) \vee (p_3 = 1 \wedge p_1 = p_2)\}$
$x_3 \equiv (x_1 < x_2)$	$\{(p_1,p_2,p_3) \mid (p_3 = 0 \wedge p_1 \geq p_2) \vee (p_3 = 1 \wedge p_1 < p_2)\}$
$x_3 \equiv (x_1 \leq x_2)$	$\{(p_1,p_2,p_3) \mid (p_3 = 0 \wedge p_1 > P_2) \vee (p_3 = 1 \wedge p_1 \leq p_2)\}$
$x_4 =$ if x_1 then x_2 else x_3	$\{(p_1,p_2,p_3,p_4) \mid (p_1 = 0 \wedge p_4 = p_3) \vee (p_1 = 1 \wedge p_4 = p_2)\}$
$x_n = \min_{i=1}^{n-1} x_i$	$\{(p_1,\ldots,p_n) \mid p_n$ is the minimum of $p_1,\ldots,p_{n-1}\}$
$x_n = \max_{i=1}^{n-1} x_i$	$\{(p_1,\ldots,p_n) \mid p_n$ is the maximum of $p_1,\ldots,p_{n-1}\}$
$x_n = \sum_{i=1}^{n-1} x_i$	$\{(p_1,\ldots,p_n) \mid p_n$ is the summation of $p_1,\ldots,p_{n-1}\}$
$x_1 \neq \cdots \neq x_n$	$\{(p_1,\ldots,p_n) \mid (i \neq j) \Rightarrow (x_i \neq x_j)$, for all $i,j\}$
$x_n = \texttt{sort}_k(x_1,\ldots,x_{n-1})$	$\{(p_1,\ldots,p_n) \mid p_n = p'_k$ where $(p'_1,\ldots,p'_{n-1})$ is sorting of $(p_1,\ldots,p_{n-1})\}$
$x_n \in \texttt{sigma}_k(x_1,\ldots,x_{n-1})$	$\{(p_1,\ldots,p_n) \mid$ there exist k distinct indices $i_1,\ldots,i_k$ from 1 to $n-1$ such that $p_n = p_{i_1} + \cdots + p_{i_k}\}$

Here D denotes a finite subset of $\mathbf{Z}$ and k a positive integer.

2.3 The propagation of constraints

Consistency techniques are widely used in artificial intelligence for solving constraint satisfaction problems. In the case of interval constraints, the propagation

technique as specified in [3], which can be traced back to the work of Mackworth (arc-consistency [12]), will be applied to reduce the domains of the variables. Given a system of constraints, with a first-in-first-out queue for constraints which is in the beginning filled with all constraints of the system, the following propagation algorithm is used.[1]

while $\exists$ $\mathtt{r}(x_1, \ldots, x_n)$ *in* queue *do*
  $P := \mathtt{hull}(\mathtt{dom}(x_1) \times \cdots \times \mathtt{dom}(x_n) \cap \mathtt{r})$
  if $P = \emptyset$ *then*
    stop (no solution for the system)
  else
    for $i := 1$ *to* n *do*
      if $\mathtt{proj}_i(P) \neq \mathtt{dom}(x_i)$ *then*
        queue := queue $\cup$ {constraints over x_i}
        $\mathtt{dom}(x_i) := \mathtt{proj}_i(P)$
      endif
    endfor
  endif
  queue := queue $\setminus$ $\mathtt{r}(x_1, \ldots, x_n)$
endwhile

2.4 The nondeterministic search

The constraint solving paradigm adopted by us is based on the hybrid algorithm of propagating constraints and a simple search scheme: each time the propagation of constraints terminates, if solution (or partial solution) is not reached, a simple enumeration scheme will be used to search for solutions. The idea is to split the domain of some variable into two sub-intervals and deal with the subproblems corresponding to these two parts respectively. As the constraint propagation (narrowing of intervals) and the branching (splitting of intervals) go on, the system will either reach some solution or prove *no solution* to the sub-problem (due to the finiteness of variable domains). In the latter case, the system backtracks to search other branches for solutions.

2.5 The optimization

For optimization, the depth-first branch-and-bound strategy is used: each time a solution is found, a new upper bound (for minimization) or lower bound (for maximization) on the solutions which is better (to the extent as the case should be) will be imposed. In the following, only better solutions are searched and thus the solution finally gotten is optimal.

[1] As pointed out in [3], the algorithm trivially terminates

3 The sorting constraint

In conventional procedural programming, sorting is an important issue. So is it in constraint programming: as will be seen in the resolution of the job-shop problem, the sorting constraint, which requires that n integers $y_1, \ldots, y_n$ are sorting of $x_1, \ldots, x_n$ in ascending order, plays a main role.

Based on the same principle, the authors of [14] discussed in general terms this constraint (where they call it sortedness constraint) and its potential applications in scheduling problems. In a different spirit, we deal with this issue by giving a set of primitive constraints for it and use this constraint as basis to formulate the resolution of the job-shop problem.

Consider a permutation constraint $\texttt{permutation}(x_1, \ldots, x_n)$ which expresses the fact that the n-uple $(x_1, \ldots, x_n)$ is a permutauion of the n-uple $(1, \ldots, n)$. Based on the *distinct integers* constraint which requires that n integers $x_1, \ldots, x_n$ are distinct, the permutation constraint can be stated equivalently as:

$$o_i \in [1, n], \quad \text{for } 1 \leq i \leq n$$

$$o_1 \neq \cdots \neq o_n$$

In the light of the permutation constraint, the sorting can be brought into the scope of constraint programming: that $(y_1, \ldots, y_n)$ is the ascending sorting of $(x_1, \ldots, x_n)$ is equivalent to that there exists permutation $(o_1, \ldots, o_n)$ of $\{1, \ldots, n\}$ such that $y_1 \leq \cdots \leq y_n$ and $x_i = y_{o_i}$ for $1 \leq i \leq n$. So with occurrence of the permutation variables $o_1, \ldots, o_n$, the sorting relation shall be defined as:

$$\texttt{sorting} = \left\{ \begin{array}{l} (x_1, \ldots, x_n, y_1, \ldots, y_n, o_1, \ldots, o_n) : \\ \quad (o_1, \ldots, o_n) \texttt{ is permutation of } (1, \ldots, n), \\ \quad y_1 \leq \cdots \leq y_n, \\ \quad x_1 = y_{o_1}, \ldots, x_n = y_{o_n} \end{array} \right\}.$$

The goal is then to solve the constraint $\texttt{sorting}(x_1, \ldots, x_n, y_1, \ldots, y_n, o_1, \ldots, o_n)$. Inspired from the definition of the sorting relation, a basic constraint program for it is as follows: for $1 \leq i, j \leq n$,

$$\texttt{permutation}(o_1, \ldots, o_n)$$

$$y_1 \leq y_2, \ldots, y_{n-1} \leq y_n$$

$$(o_i \leq j) \Rightarrow (x_i \leq y_j)$$

$$(x_i < y_j) \Rightarrow (o_i < j)$$

where the third and the fourth constraints are a more effective version of the $x_i = y_{o_i}$ constraint (usually called constraint of i-th element) by making use of the property that $y_1, \ldots, y_n$ are in ascending sorting. They are respectively decomposed into:

$$a_{ij} \equiv (o_i \leq j), \quad b_{ij} \equiv (x_i \leq y_j), \quad a_{ij} \Rightarrow b_{ij}$$

$$a'_{ij} \equiv (x_i < y_j), \quad b'_{ij} \equiv (o_i \leq j), \quad a'_{ij} \Rightarrow b'_{ij}$$

where $a_{ij}, a'_{ij}, b_{ij}, b'_{ij}$ are intermediate boolean variables taking value 0 (`false`) and 1 (`true`).

We use the following example to illustrate the effect of our constraint program for sorting:

$$\begin{pmatrix} o_1 & o_2 & o_3 & o_4 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \end{pmatrix} \quad : \quad \begin{pmatrix} [3,4] & [1,2] & [2,3] & [2,3] \\ [4,4] & [1,2] & [3,4] & [2,4] \\ [1,1] & [1,2] & [2,3] & [3,4] \end{pmatrix}$$

can be maximally narrowed to:

$$\begin{pmatrix} o_1 & o_2 & o_3 & o_4 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \end{pmatrix} \quad : \quad \begin{pmatrix} [4,4] & [1,1] & [3,3] & [2,2] \\ [4,4] & [1,1] & [3,3] & [2,2] \\ [1,1] & [2,2] & [3,3] & [4,4] \end{pmatrix}$$

The basic program is sound and complete for the sorting constraint. But its procedural performance is not quite satisfactory. For instance, due to locality of the constraint propagation, the basic set of constraints can reduce no interval in the following case:

$$\begin{pmatrix} o_1 & o_2 & o_3 & o_4 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \end{pmatrix} \quad : \quad \begin{pmatrix} [3,4] & [1,2] & [2,4] & [1,3] \\ [3,4] & [1,2] & [3,3] & [2,2] \\ [1,2] & [1,3] & [2,4] & [3,4] \end{pmatrix}$$

A trivial redundant constraint to the sorting constraint is: the number of x_i's that are less than or equal to y_i must be at least i and the number of x_i's that are greater than or equal to y_i must be at least $n - i + 1$. In meta-level, it can be expressed as follows:

$$d_i \leq y_i \leq d'_i, \qquad \text{where} \begin{cases} (d_1, \ldots, d_n) \text{ is ascending sorting of } (\underline{x_1}, \ldots, \underline{x_n}), \\ (d'_1, \ldots, d'_n) \text{ is ascending sorting of } (\overline{x_1}, \cdots, \overline{x_n}). \end{cases}$$

To remain in the framework of constraint solving, we encapsulate the above meta-level constraint into the following constraint:

$$y_i = \mathtt{sort}_i(x_1, \ldots, x_n)$$

understanding that y_i ranks i-th in the ascending sorting of $x_1, \ldots, x_n$. See Table 1 for the definition of the $\mathtt{sort}_k$ constraint.

The $\mathtt{sort}_k$ constraint turns out to be quite useful for narrowing the intervals. The basic program for sorting plus the $\mathtt{sort}_k$ constraints can maximally narrow the intervals of the above example to be:

$$\begin{pmatrix} o_1 & o_2 & o_3 & o_4 \\ x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \end{pmatrix} \quad : \quad \begin{pmatrix} [3,4] & [1,2] & [3,4] & [1,2] \\ [3,4] & [1,2] & [3,3] & [2,2] \\ [1,2] & [2,2] & [3,3] & [3,4] \end{pmatrix}$$

4 Solving the job-shop problem

Now we come to the resolution of the job-shop problem. If we introduce the following variables for the unknown: for $1 \leq i \leq m, 1 \leq j \leq n$,

- o_{ij} : the order of job j processed on machine i.
- x_{ij} : the release date of job j on machine i.
- y_{ij} : the due date of job j on machine i.
- u_{ij} : the release date of the job scheduled j-th on machine i.
- v_{ij} : the due date of the job scheduled j-th on machine i.
- w : the maximum of the completion times of all jobs.

then a basic constraint program can be given as follows: for $1 \leq i \leq m, 1 \leq j \leq n$,

- $y_{\tau_{ij},j} \leq x_{\tau_{i+1,j},j}$ precedence constraint
- $y_{ij} = x_{ij} + d_{ij}$ duration constraint
- $\texttt{sorting}(x_{i1}, \ldots, x_{in}, u_{i1}, \ldots, u_{in}, o_{i1}, \ldots, o_{in})$ disjunctive constraint

  $\texttt{sorting}(y_{i1}, \ldots, y_{in}, v_{i1}, \ldots, v_{in}, o_{i1}, \ldots, o_{in})$

  $v_{ij} \leq u_{i,j+1}$
- $v_{in} \leq w$ for minimization

The sorting constraints require that the release date x_{ij} (due date y_{ij}) is sorted o_{ij}-th in the scheduled release dates $u_{i1}, \ldots, u_{in}$ (scheduled due dates $v_{i1}, \ldots, v_{in}$). Then by imposing the precedence constraints $v_{ij} \leq u_{i,j+1}$ over the sorted dates to make disjoint the scheduled tasks, the disjunctive constraints of the job-shop problem can be settled. The feature of such an approach is that it takes into account the ordering of tasks on a machine globally rather than pairwisely.

The goal is to instantiate all of the ordering variables o_{ij}'s which make up a partial solution of the job-shop problem. In the following subsections, we discuss the branch-and-bound method for instantiating o_{ij}'s and give redundant constraints to promote the efficiency of the constraint program.

4.1 The search strategy

Optimization is carried out by the depth-first branch and bound search: 1. each time the propagation of constraints terminates, if not all ordering variables are instantiated, the system will split the domain of some ordering variable not instantiated and enumerate the split parts which each constitute a subproblem; 2. each time a

solution is found, a new upper bound ($\underline{w} - 1$) for the completion times of all jobs will be imposed.

The search strategy is based on the first-fail principle. Firstly, we select critical machine i such that $\sum_{j=1}^{n} |u_{ij}| + |v_{ij}|$ is minimum. Thus the machine, on which the total slack of sorted dates is minimum and thus the tasks are the most constrained, is selected. This is justified by the aim of forcing bottleneck of failures (backtracks) as early as possible so that smaller space be searched.

Empirical results tell that fixing the first critical machine until all jobs on it are completely scheduled yields better results. So each time the first critial machine is selected, it will be fixed until the schedule on it is done. On the critical machine i, the selection of critical job j is carried out in accordance with the following criteria:

1. $\frac{\sum_{k \in dom(o_{ij})} (|u_{ik}| + |v_{ik}|)}{|o_{ij}|}$ is minimum.
2. $\sum_{k=1}^{m} |o_{kj}|$ is maximum.
3. $|\alpha_1 - \alpha_2|$ is maximum.

where

$$\alpha_1 = \left| (\overline{x_{ij}} + \underline{x_{ij}} + \overline{y_{ij}} + \underline{y_{ij}}) - \frac{\sum_{k \in \beta_1} (\overline{u_{ik}} + \underline{u_{ik}} + \overline{v_{ik}} + \underline{v_{ik}})}{|\beta_1|} \right|$$

$$\alpha_2 = \left| \frac{\sum_{k \in \beta_2} (\overline{u_{ik}} + \underline{u_{ik}} + \overline{v_{ik}} + \underline{v_{ik}})}{|\beta_2|} - (\overline{x_{ij}} + \underline{x_{ij}} + \overline{y_{ij}} + \underline{y_{ij}}) \right|$$

where β_1 and β_2 are the lower half and the upper half of the domain of o_{ij}.

The goal is to select the job for which the total slack of release and due dates of tasks scheduled in the interval $\mathtt{dom}(o_{ij})$ is minimum (like the criterion for selecting critical machine, this criterion is also in the hope of forcing the bottleneck as early as possible). In case of tie, select the job for which the total slack of its ordering on all machines is maximum. To further break tie if necessary, select the job with the maximal difference between the biases to the date averages of the jobs scheduled in the interval β_1 and β_2. Both latter are in the hope of resulting in greatest change to the constraint system when splitting the interval.

For branching, we split the domain of the ordering variable o_{ij} in two and enumerate the two branches with the following heuristic:

if $\alpha_1 < \alpha_2$ then to deal with the lower half β_1 first
else to deal with the upper half β_2 first

The goal of this heuristic is to reach the solutions as rapidly as possible.

4.2 The redundant constraints

The basic constraint program gives a clear idea for solving the job-shop problem. However, the hardness of the problem requires us to add into the constraint system redundant constraints to prune the search tree as much and as early as possible. The idea is to tighten the constraints over orders of the tasks, release and due dates, and the scheduled dates.

For task orders, consider the constraints between o_{ij} and o_{ik} for $1 \leq i \leq m$ and $1 \leq j < k \leq n$. Based on the fact that all tasks on a machine must be disjunctive, the following equivalences hold:

$$(o_{ij} < o_{ik}) \equiv \neg(o_{ik} < o_{ij}) \equiv (x_{ij} < y_{ik}) \equiv (y_{ij} \leq x_{ik}) \tag{1}$$

The essence of these constraints is to tap the fact that due to disjunction, that release date x_{ij} is less than due date y_{ik} is equivalent to that task (i,j) precedes task (i,k). This is very useful in pruning the search tree.

Moreover, the order of job j on machine i must be equal to the sum of (difference between) minimal order (maximal order) and the number of tasks preceding (succeeding) it, which is trivial:

$$o_{ij} = \min_{1 \leq k \leq n} (\text{if } o_{ik} \leq o_{ij} \text{ then } o_{ik} \text{ else} + \infty) + \sum_{1 \leq k \leq n, k \neq j} (o_{ik} < o_{ij}) \tag{2}$$

$$o_{ij} = \max_{1 \leq k \leq n} (\text{if } o_{ij} \leq o_{ik} \text{ then } o_{ik} \text{ else} - \infty) - \sum_{1 \leq k \leq n, k \neq j} (o_{ij} < o_{ik}) \tag{3}$$

Similarly, for release and due dates, consider the constraints between x_{ij} and x_{ik} (y_{ij} and y_{ik}) for $1 \leq i \leq m$ and $1 \leq j \leq n$. The release dates (due dates) of job j on machine i must be greater than (less than) or equal to the sum of (difference between) the minimal release date (maximal due date) and the total durations of tasks preceding (succeeding) job j.

$$x_{ij} \geq \min_{1 \leq k \leq n} (\text{if } o_{ik} \leq o_{ij} \text{ then } x_{ik} \text{ else} + \infty) + \sum_{1 \leq k \leq n, k \neq j} \text{if } o_{ik} < o_{ij} \text{ then } d_{ik} \text{ else } 0 \tag{4}$$

$$y_{ij} \leq \max_{1 \leq k \leq n} (\text{if } o_{ij} \leq o_{ik} \text{ then } y_{ik} \text{ else} - \infty) - \sum_{1 \leq k \leq n, k \neq j} \text{if } o_{ij} < o_{ik} \text{ then } d_{ik} \text{ else } 0 \tag{5}$$

For constraints (4) and (5), meta-level constraints can be given (similar case for constraints (2) and (3)):

$$x_{ij} \geq \min_{1 \leq k \leq n \ \wedge \ \overline{(o_{ik} \leq o_{ij})} = 1} x_{ik} + \sum_{1 \leq k \leq n \ \wedge \ k \neq j \ \wedge \ \overline{(o_{ik} < o_{ij})} = 1} d_{ik}$$

$$y_{ij} \leq \max_{1 \leq k \leq n \ \wedge \ \overline{(o_{ij} \leq o_{ik})} = 1} y_{ik} + \sum_{1 \leq k \leq n \ \wedge \ k \neq j \ \wedge \ \overline{(o_{ij} < o_{ik})} = 1} d_{ik}$$

The basic idea of the constraints (1), (4) and (5) can be traced back to the work of Carlier and Pinson and that of Applegate and Cook. While constraints (2) and (3) are extentions of (4) and (5) to ordering variables. These types of constraints are widely used in solving the job-shop problem. So they can be viewed as classical.

Peculiar to our approach, powerful redundant constraints can be used over sorted dates. Let's consider the gaps between v_{ik} and u_{ij} for $1 \leq i \leq m$ and $1 \leq j \leq k \leq n$. Firstly, the gap between due and release dates for each sorted task must be equal to some duration of the tasks (this is trivial):

$$v_{ij} - u_{ij} \in \{d_{i1}, \ldots, d_{in}\} \tag{6}$$

Furthermore, they must satisfy the following constraints:

$$v_{ij} - u_{ij} \geq \min_{1 \leq k \leq n} \text{if } o_{ik} = j \text{ then } d_{ik} \text{ else } +\infty \tag{7}$$

$$v_{ij} - u_{ij} \leq \max_{1 \leq k \leq n} \text{if } o_{ik} = j \text{ then } d_{ik} \text{ else } -\infty \tag{8}$$

Finally we wish to constrain the gaps with this idea: at any moment during the constraint propagation, regarding the interval $[j, k]$ of orders, we can evaluate a lower bound for the gap between the jobs scheduled in this interval by

- adding up the durations of the jobs whose execution orders fall in $[j, k]$.
- adding up the $k - j + 1$ minimal durations of the jobs whose execution orders do not surely fall out of $[j, k]$.

The constraints specified by us are as follows:

$$v_{ik} - u_{ij} \geq \sum_{1 \leq l \leq n} \text{if } o_{il} \in [j, k] \text{ then } d_{il} \text{ else } 0 \tag{9}$$

$$v_{ik} - u_{ij} \geq \sum_{1 \leq p \leq k-j+1} \text{if } o_{il_p} \in [j, k] \text{ then } d_{il_p} \text{ else } +\infty \tag{10}$$

for some distinct indices $l_1, \ldots, l_{k-j+1}$

where the constraint of partial sum $x_0 \in \texttt{sigma}_p(x_1, \ldots, x_n)$ states that x_0 is the summation of some p elements of $x_1, \ldots, x_n$. For this constraint, a meta-level necessary constraint (thus an incomplete algorithm for the narrowing operator) is:

$$d_1 + \cdots + d_p \leq y \leq d'_{n-p+1} + \cdots + d'_n,$$

$$\text{where } \begin{cases} (d_1, \ldots, d_n) \text{ is ascending sorting of } (\underline{x_1}, \ldots, \underline{x_n}), \\ (d'_1, \ldots, d'_n) \text{ is ascending sorting of } (\overline{x_1}, \ldots, \overline{x_n}). \end{cases}$$

Constraints (9) and (10) are very useful in pruning search tree. They are in fact just a simplification of the following more efficient meta-level constraints:

$$v_{ik} - u_{ij} \geq \min_{S \subseteq \{d_{i1}, \ldots, d_{in}\} \wedge |S| = k-j+1} \Sigma S$$

subject to:

1. if $\mathtt{dom}(o_{il}) \subseteq [j,k]$ then $d_{il} \in S$
2. if $\mathtt{dom}(o_{il}) \cap [j,k] = \emptyset$ then $d_{il} \notin S$

Remark: All complex constraints are decomposed into primitive constraints in our implementation in the spirit of emulating a pure constraint language which supports the constraint solving paradigm as described in section 2. Besides the basic primitive constraints, the redundant constraints reduce to the following 5 specific primitive constraints (See Table 1 for definition):

$x_0 =$ if x_1 then x_2 else x_3	conditional constraint
$x_0 = \min_{i=1}^{n} x_i$	constraint of minimum
$x_0 = \max_{i=1}^{n} x_i$	constraint of maximum
$x_0 = \sum_{i=1}^{n} x_i$	constraint of summation
$x_0 \in \mathtt{sigma}_k(x_1, \ldots, x_n)$	constraint of partial sum

For example, Constraint (10) will be docomposed into:

$z1_{ijk} = v_{ik} - u_{ij}$

$z2_{ijkl} \equiv (o_{il} \in [j,k])$

$z3_{ijkl} =$ if $z2_{ijkl}$ then d_{il} else $+\infty$

$z4_{ijk} \in \mathtt{sigma}_{k-j+1}(z3_{ijk1}, \ldots, z3_{ijkn})$

$z1_{ijk} \geq z4_{ijk}$

4.3 The empirical results

A prototype of the interval constraint system which involves all the primitive constraints of Table 1 has been developped by us to support the constraint program for the job-shop problem. The results of our constraint program on the 10×10 instances of [1] is given in Table 2. They are obtained on a Sun Sparcstation 10. The fifth column gives the numbers of backtracks needed for the first solution while the seventh column for the numbers of choice points of search trees. As far as the number of choice points examined by the constraint system is concerned, the experimental result of the approach is satisfactory compared to specific operations research algorithms (e.g, [1, 5]). An interesting result is that the ratio $\frac{\text{number of nodes}}{\text{time}}$ is nearly constant. This is because of the propagation of primitive constraints. The maximal depths of search trees are small mainly due to the effective control of the distinct integers constraints. All these permit us to better predict the size of the search space and the execution time. Regarding the results with different initial upper bounds, the convergency of the constraint program on the instance `mt10` is measured in Table

3. The convergence speed is satisfactory (compared to [1]). Experiments on other 10×10 instances give similar results.

For the 10×10 instances and for our constraint system, the current constraint program produces about 36,000 variables (basic and intermediate) and 40,000 primitive constraints. In general, for m machines and n jobs problem, the number of variables and the number of constraints are both polynomial ($O(mn^3)$). In fact, we sacrificed remarkably the efficiency of space pruning as well as the speed due to restricting our algorithm to be within the framework of propagating primitive constraints and the naive enumeration scheme. In practice, a solution would be to apply meta-level constraints so as to enforce the pruning of a search tree. For this, the last column of Table 1 gives the numbers of choice points of search trees if the constraints (2)-(5) and (9)-(10) are replaced by their corresponding meta-level constraints. Since the redundant constraints produce most of the intermediate variables and primitive constraints, the number of variables and constraints will greatly decrease if meta level implementation of the redundant constraints is allowed. And in practice, some of the redundant constraints (e.g, constraints (2), (3), (6), (7), (8)) which are relatively weaker in pruning search trees can be removed at user's option. On such considerations, in meta-level our redundant constraints are even easier to implement than those of some other efficient approaches (e.g, [2, 7]).

Besides the 10×10 instances, Table 4 gives some results for several instances of larger size. For these results, Constraints (2)-(5) and (9)-(10) are all replaced by the meta-level constraints and are implemented wholely as large constraints (not decomposed). The main aim is to save memory and promote the space pruning power. Interestingly, the ratio $\frac{\text{number of nodes}}{\text{time}}$ remains constant for instances of same size and we achieved good results: the program found the optimal solution of the 10 machines $\times$ 15 jobs instance `la24` and proved its optimality by examining only 616,298 nodes, better than the result (16,115,842 nodes) given in [1]; for the `la40` which is considered hard in [1], our program only needs to examine 55,571 nodes for the optimal solution and the proof of optimality (to our knowledge, no detailed solution result was given for this instance in the literature). Considering the global search controlled by the distinct integers constraint and the good performance on harder instances, our constraint program is expected to work better for solving job-shop problems of larger size. Indeed an intuition is that, our constraint program costs constraint processing overhead on "small" instances like `mt10` but pays off in larger instances. From the angle of these considerations, the performance of our constraint program compares well with specific operations research algorithms (e.g, [1, 5]) and other efficient approaches for which highly practical implementation was pursued (e.g, [2, 7]).

Table 2: solutions of ten 10 × 10 instances

Instance	Initial upper bound	Opt. upper bound	First solution found: Sol. bound	Backtracks	Time (s)	Opt. proven: Total nodes	Time (s)	Max depth	Nodes (meta level)
mt10	930	930	930	3339	905	11591	3323	23	9209
orb1	1070	1059	1068	4734	1183	73189	18463	34	55828
orb2	890	888	890	201	52	38826	8126	47	27199
orb3	1021	1005	1020	7840	2063	145304	36290	44	98099
orb4	1019	1005	1019	915	228	7346	1862	26	2736
orb5	896	887	896	239	63	3883	900	31	3156
abz5	1245	1234	1242	3404	860	8218	2138	38	6619
abz6	943	943	943	22	9	4892	1227	28	560
la19	848	842	846	1246	367	9130	2339	28	7915
la20	911	902	910	1533	319	22519	5564	47	24163

In particular, the first critical machine is fixed to be 5th for orb3.

Table 3: convergency analysis on mt10

Initial bound	First solution found: Sol. bound	Backtracks	Time(s)	Optimality proven: Nodes	Time(s)	Max depth
$+\infty$	1488	0	5	69665	12027	229
30000000	1488	0	5	69665	12004	229
10000	1488	0	5	69665	11985	229
1000	996	43	10	52050	11175	55
950	949	1687	315	46158	10328	42
940	939	10460	2726	22067	5996	37
929	—-	—-	—-	10954	3220	22

Table 4: Solutions of some instances of larger size

Instance	Init. bound	Opt. bound	Sol. found: Backtracks	Opt. proven: Total nodes	Max dep	Nodes/s
la24 (10×15)	935	935	355889	616298	49	4
la36 (15×15)	1268	1268	1694	3370	58	3
la39 (15×15)	1233	1233	2443	3324	53	3
la40 (15×15)	1222	1222	541	55571	47	3

5 Concluding remarks

The goal of this paper is to propose a new approach which is clean, flexible and robust for solving the job-shop problem. The spirit (different from that of usual approaches) is to simulate the resolution of a hard problem by a pure constraint language. So by clean we mean, the constraint program makes use of pure constraint programming techniques. In other words, one feature of our constraint program is its indenpendency on algorithmic detail. In fact all the primitive constraints of Table 1

except $\texttt{sort}_k$ and $\texttt{sigma}_k$ exist in the literature of constraint systems or are easy to implement. For the last two constraints introduced by us, we can only narrow the domains of part of the variables. However, we isolated these two constraints for the sake of better stating the problem. In the resolution of the job-shop problem, they turn out to be very useful in pruning a search tree. These two constraints integrate the conventional sorting algorithm into the resolution of the job-shop problem, and thus completing the idea of sorting the release and due dates of jobs.

The flexibility of our approach is inherent in constraint programming methods, and due to the use of the distinct integers constraint, more global control over the task orders is obtained. The flexibility and the good performance exhibited in solving hard instances are proofs that our constraint program is robust for solving the job-shop problem.

The gains of formalizing the job-shop problem by sorting the release and due dates of the jobs on the machines are: a pure constraint programming foumulation is obtained; the ordering of the tasks and the sorted dates can be constrained as wished, which exhibits flexibility and will be helpful for solving realistic scheduling problems; a simple search scheme (enumeration based on splitting of the intervals) oriented by the distinct integers constraint makes the enumeration more global (and thus a good distinction from other classical search strategy, e.g, edge-finding).

For future work, on one hand we will try to promote the inference power of our prototype for the constraint system (e.g, the constraint propagation speed), and on the other hand, to solve harder instances of the job-shop problem or realistic scheduling problems. For the latter, more powerful constraints or more efficient search strategy may be needed.

Acknowledgements

The author is grateful to Alain Colmerauer for his guide and help in this research work. The author thank Frédéric Benhamou and William J. Older for their help and encouragement. We also thank Jean François Maurras and Michel Van Caneghem for their help on the issue of distinct integers constraint. Especially, the author would like to thank Yves Colombani for kindly providing us with some references and data sets of job-shop test problems. Frequent exchange of experimental results with him was helpful. The discussions with Cristophe Aillaud and Philippe Refalo on techniques of constraint systems were also helpful.

References

[1] Applegate, D. and Cook, B. *A Computational Study of the Job Shop Scheduling Problem*, Operations Research Society of America, vol 3, no 2, 1991.

[2] Baptiste, P., Le Pape, C. and Nuijten, W. *Constraint-based Optimization and Approximation for Job Shop Scheduling*, Proc. of the IJCAI 95 workshop on Intelligent Manufacturing Systems, Montréal, 1995.

[3] Benhamou, F. and Older, W.J. *Applying Interval Arithmetic to Real, Integer and Boolean Constraints*, to appear in the Journal of Logic Programming.

[4] Carlier, J. and Pinson, E. *An algorithm for solving the job shop problem*, Management Science, vol 35, no 2, February 1989.

[5] Carlier, J. and Pinson, E. *Adjustment of heads and tails for the Job-shop problem*, European Journal of Operational Research, 78:146-161, 1994.

[6] Caseau, Y. and Laburthe, F. *Improved CLP scheduling with task intervals*, Proc. of the 11th International Conference on Logic Programming, 1994.

[7] Caseau, Y. and Laburthe, F. *Disjuctive scheduling with task intervals*, LIENS Technical Report $n°$ 95-25, 1995.

[8] Cleary, J. *Logical Arithmetic*, Future Computing Systems, Vol 2, No 2, 1987.

[9] Colmerauer, A. *An introduction to Prolog III*, in Communications of the ACM, 33(7):69, July 1990.

[10] Lee J. and van Emden, M. *Adapting CLP(R) to Floating point Arithmetic*, in Proc. of the Fifth Generation Computer Systems Conference, Tokyo, Japan, 1992.

[11] Lenstra, J.K. and Rinnooy Kan, A.H.G. *Computational Complexity of Discrete Optimization Problems*, Annals of Discrete Mathematics 4, 121-140, 1979.

[12] Mackworth, A.K. *Consistency in Networks of Relations*, in Artificial Intelligence 8, p 99-118, 1977.

[13] Muth, J.F. and Thompson, G.L. *Industrial Scheduling*, Prentice Hall, Engleclood Cliffs, N.J, 1963.

[14] Older, W.J., Van Emden, M. and Swinkels, F. *Getting to the Real Problem:experience with BNR Prolog in OR*, in Proceedings of the Third International Conference on the Practical Applications of Prolog,(PAP'95), Paris, April, 1995, Alinmead Software Ltd., ISBN 0 9 525554 0 9.

[15] Older, W.J. and Vellino, A. *Constraint Arithmetic on Real Intervals*, Constraint Logic Programming: Selected Research, F. Benhamou and A. Colmerauer (eds), MIT Press, 1992.

[16] Van Hentenryck, P., Simonis, H. and Dincbas, D. *Constraint satisfaction using constraint logic programming*, Artificial Intelligence, vol 58, no.1-3, p113-157, December 1992.

[17] Zhou, J. *Solving the job-shop problem by constraint programming*, Proc. of the post-ILPS'95 workshop on Interval Constraints, Portland Oregon, December 1995.

PSAP - A Planning System for Aircraft Production (Extended Abstract)

Patrick Albers[1] and Jacques Bellone[2]

[1] LAAS-CNRS, 7 av. Colonel Roche, 31077 Toulouse
email: patrick@laas.fr
[2] Centre Spacial Dassault, 17 av. Didier Daurat, 31701 Blagnac ***

PSAP[4] is an aircraft production planning decision support system, whose aim is to schedule the aircraft production over the next years (more than five years) at Dassault factories. This long term production consists in scheduling the assembly lines where large *sections* of aircraft are manufactured. A section is a major aircraft part (i.e. cockpit, wing, rear fuselage, final assembly, ...). Up to 400 aircraft and 30 assembly lines are concerned.

The objective is to find schedules respecting the *delivery dates* for all aircraft and being a satisfactory compromise between *two production criteria*. On the one hand, the *storage* between the different assembly lines of the factory must be minimized. On the other hand, the production uses highly skilled labour, and a change of *production rate* involves a work *overload*. The aim is then to minimize the storage time for a given maximum number of production rate changes.

Production planning and scheduling are complex operations involving a great number of constraint s, both numerical and symbolic. Unfortunately, the difficulty to express these constraints doesn't allow the use of the current scheduling or planning tools. A first implementation has been done in the CLP language CHIP[5]. The constraints used in PSAP are in finite domains. Good results was found[6] with appropriate heuristics; nevertheless, considering the very large size of the search tree, we have to prune it.

Finding an optimal solution means exploring the whole search tree. The search tree size depends on the fact of choosing Y changes out of an X-element set (let $\binom{Y}{X}$ branches) and on the fact that, for each change there are P possible values (P^Y branches); with P the number of possible values of production rate possible values on the same assembly line (generally the P average value after the constraint propagation is 20), X the number of aircraft where a production rate change can occur and Y the number of rate changes given by the end-user (Y average value is below 10).

The need for parallelism has also been felt necessary since the sequential optimization could run for small-size data sets (i.e. 70 aircraft), but not for actual large-size data sets (i.e. 250 aircraft). A parallel implementation has been

*** J. Bellone works now at CR2a-DI, 32 rue des cosmonautes 31400 Toulouse

[4] This work was partially funded by ESPRIT project number 6708, APPLAUSE.

[5] COSYTEC: CHIP user manual, v.4.0, COSY/ref/001, june 93

[6] see [J. Bellone, A. Chamard and C. Pradelles] An evolutive planning system for aircraft production - PAP'92

realized with the languages ElipSys [7] and ECL^{i}PS^e [8]; (ECL^{i}PS^e is the successor of ElipSys). At certain nodes of the search tree, instead of trying each possibility in sequence by backtracking, we try them in parallel on different processors. The parallelism deals with predicates, either built-in or defined by user. In PSAP, the parallel predicate is the "minimize" procedure, which uses branch and bound method in order to explore the search tree.

Numerous benchmarks were conducted and concerned only the schedule of the last assembly line, which is the most difficult to schedule. Two cases may be identified: the case where the whole search tree is taken into account, and the case where the search is halted when the solution cost is lower than the minimum bound. On the one hand, the solution found is the optimal one, on the other hand is a sub-optimal one.

In order to illustrate these cases, two benchmarks are presented; they are performed on a drs6000 machine with 4 processors. To the left, an optimal solution search where 120 subsets of possible production rate changes are explored in parallel (data set of 70 mirage 2000). To the right, a sub-optimal solution search where 84 subsets are explored indeed of the 364 possible subsets (date set of 250 mirage 2000).

number of workers	1	2	3	4
time in sec.	534	280	198	160
speed-up		1.9	2.69	3.32

number of workers	1	2	3	4
time in mn.	212	52	22	18
speed-up		4.02	9.7	11.51

In this kind of benchmarks for optimal solution, the speed-up gain is quasi-linear. Moreover, another problem arises: the possible gain of time with parallelism considering the constraint propagation. Indeed when the optimal solution is in the left side of the tree (depth first search), it will be found very quickly whether using the parallel method or the sequential one. The problem here is to know a priori the parallel grain size. In the PSAP problem, this size depends on the number of rate changes, on the data set size and on the first solution found. In particular in sequential execution, if the first solution found is the optimal solution, the parallel grain size may be too small to obtain speed-ups while searching for the solution optimality proof if there is good constraint propagation.

For the case of sub-optimal search, the execution may be *super linear*. These results confirm the idea that the position of the required solution is important in order to obtain good results of speed-ups. In the above example, the solution is in the right side of the tree, therefore the more workers there are, the faster the solution will be found. Nevertheless, there may exist a threshold such as using n workers will be the same as using $n-1$, whereas using $n+1$ will yield a significant gain of time. This was confirm with tests on a 12-processor SGI computer. These levels prove that the parallel gain is due, for a big part, to the positions of optimal or suboptimal solutions in the search tree.

[7] ECRC: user manual ElipSys 0.7 - dec 93
[8] ECRC: user manual ECL^{i}PS^e 3.5 - dec 94

Using Partial Arc Consistency in a Database Environment

Steven A. Battle

University of the West of England, Bristol, UK.
email: steve.battle@ics.uwe.ac.uk

1 Introduction

The establishment of local consistency in a database can be shown to have a positive effect in reducing the complexity of search. The lower complexity of achieving directed arc-consistency (DAC), based on a total ordering of the variables, relative to full arc-consistency (AC), makes it the more viable technique in practical database systems. In many high volume transaction based systems it is desirable to optimise for a number of fixed variable orderings in advance, rather than rely upon dynamic, run-time techniques. Achieving DAC across a set of orderings defines the new problem of achieving consistency with respect to a partial ordering of the variables. This problem may be defined as partial arc-consistency (PAC).

2 The Application Domain

Computerised Reservation Systems (CRS) are used to control almost every aspect of the travel business from bookings and the issuing of tickets, to inventory management and price planning. This work is concerned with just one part of this process, the initial *availability search* which takes place prior to booking (Battle et al. 1995b).

Conventional reservation systems typically use holiday 'packages' stored as low-level 'templates' of which there may be many hundreds of thousands of variations. Because there is so little shared structure, the search for matching holidays is an essentially brute-force process, a fact that makes search harder than it need be. The business aims are to improve the flexibility of the database using relational technology, but to maintain present performance.

At any one time, many hundreds of travel agent sessions may be in progress, and the response time must be on the order of a few seconds. In such high volume transaction-based systems, the database typically receives only a narrow range of queries. This high transaction rate favours off-line pre-processing techniques in preference to dynamic, run-time optimisation. The run-time process is therefore fairly minimal, with much of the query processing being performed in advance (Battle 1995a), so the results can be embedded within a conventional programming language.

3 Partial Arc Consistency

Arc-consistency is a desirable property but is expensive to achieve. A more pragmatic approach to consistency is *directed* arc-consistency, which is able to exploit a known variable ordering. The effect of DAC is to reduce the thrashing associated with backtracking. Savings due to DAC in the availability search averaged out at over 52 consistency checks (database accesses) per run.

The definition of DAC assumes that only a single ordering is used to search the graph. Given a number of variable orderings, establishing DAC for each ordering in turn is not the most efficient approach, nor is DAC guaranteed if established only once for each ordering as subsequent runs can interfere with the consistency of earlier orderings.

PAC can be established using an arc-consistency algorithm (AC3), initialised with a graph derived from the set of orderings by taking the union of the ordered graphs (The directed graph for a sequence is formed by aligning each edge along the direction of the ordering). An algorithm such as AC3 will then establish DAC for every ordering corresponding to a path through this directed graph that visits each variable exactly once.

This adaptation of AC3 can be improved by exploiting the existence of cycles within this directed graph, allowing it to be partitioned into a number of disjoint subgraphs. The dependencies between these groups are noted. Redundant consistency checks may be avoided by ensuring that constraints within a partition are checked before those leading out of the partition. To avoid any partition being checked twice they must be arranged in order such that no group depends upon a later group. These partitions are processed in a single directed pass, with AC being achieved within each partition.

Directed arc-consistency is achieved in a given total ordering if every constraint, $i \rightarrow j$, is checked after every revision of variable j. Where the relationship, $i \rightarrow j$, does not belong to a cycle, i and j are placed in separate partitions, so that all revisions to j are made before the constraint, $i \rightarrow j$, is ever checked, ensuring DAC on that constraint. If the constraint, $i \rightarrow j$, is part of a cycle, PAC establishes AC which again implies DAC on that constraint. PAC therefore guarantees DAC for every totally ordered graph that is a subset of the input graph.

In the travel database, AC3 required 67 consistency checks to enforce PAC, as opposed to only 58 for the algorithm outlined above. This 13% reduction is within the maximum 30% saving predicted in tests on randomly generated constraint graphs.

References

Battle, S. A.: Generating database queries from a constraint network representation. In *5th Scandanavian Conference on AI* (1995a) 343–347.

Battle, S. A., McClatchey, R. H.: A computerised reservation system using a relational database augmented by constraint based techniques. In *Databases and Expert Systems* (1995b).

Functional Constraint Hierarchies in CLP

Mouhssine Bouzoubaa
INRIA-CERMICS, B.P.93, 06902 Sophia-Antipolis Cedex, France
mbouzou@sophia.inria.fr

Abstract. HCLP extend CLP to include constraint hierarchies. We present an algorithm based on our previous work and on the extended notion of comparators for comparisons between the hierarchies that arise from alternate rule choices in the program.
Keywords: Constraints, Constraint hierarchy, CLP, HCLP

1 Introduction, Extended Theory and Houria Review

A prototype implementation of HCLP(Real, Locally-Predicate-Better) is described in [1], experience with writing programs in HCLP(R,LPB) has provided many examples where the LPB comparator fails, ruling out, non-intuitive solutions. This results from HCLP's inability to discriminate between solutions arising from different rule choices. The need for inter-hierarchical comparisons is necessary. To achieve that, the extension of the original theory in [2] consists of defining the set of solutions to many constraint hierarchies. This lays the theoretical foundation for inter-hierarchy comparators and will allow to eliminate the undesirable solutions. $S_0 = \{\theta_h : \forall c \in h_0 \ \mathrm{Sat}(\theta, c)\}$ (i.e. the set of valuations that satisfy the hard constraints in the hierarchies) and $S = \{\theta_h : \theta_h \in S_0 \wedge \forall \eta_{h'} \in S_0 \neg$ better$(\eta_{h'}, \theta_h, H)\}$. Two types of comparators are distinguished: t! he $Locally - Better$ and the $Globally - Better$ comparators. Only the $Globally - Better$ comparators are extended to compare valuations arising from different hierarchies since they take some aggregate measure to combine the errors obtained in each level of the hierarchy. The original version of the Houria solver implements the global comparator $Unsatisfied - Count - Better$ [3]. Houria II and Houria III are extensions of Houria. They handle different classes of labeled soft constraints where each class may contain weighted constraints. The global criteria used respectively are the $Unsatisfied - Count - Best - Case - Better$ and the $Weighted - Sum - Better$ comparators [4].

2 Algorithm for Inter-Hierarchy Comparisons

We describe our approach with the following steps :

- Form the set H of the constraint hierarchies resulting from the alternate rule choices in the HCLP program.
- For each hierarchy in H

  - Compute the agregate weight of the satisfied constraints depending on the criterion, that contain only the bounded variables (denoted by $w - bounded$).
  - Compute the weight of the constraints that contain at least one free variable (denoted by $w - free$).
- Eliminate from H the set of the hierarchies of which the sum of $w - bounded$ and $w - free$ is strictly less than the sum of another hierarchy in H.
- Order the resulting set H by the criterion sum of $w - bounded$ and $w - free$ decreasing.

```
Repeat
        h ← pop(H)
        call Houria solver in order to extract from h
            the maximum subset of the constraints in h that
            contains free variables and can be solved
-       H' ← push(h)
Until ((H is empty) or (the sum of the weight of the
      maximum subset and w − bounded is not less than
      the sum of w − bounded and w − free of the hierarchy
      in the head of H)).
```

- Keep in H' only the hierarchies that contain the maximal weight.
- The free variables of each hierarchy in H' are computed and returned.

3 Conclusion

Our algorithm can be included in the HCLP languages. It's promising when the alternate rule choices number of each predicate is not large, in the opposit, Houria system can be used in a B&B algorithm to obtain the inter-hierarchy comparisons.

References

[1] Borning, A. , Wilson, M. Extending Hierarchical Constraint Logic Programming: Nonmonotonicity and Inter-Hierarchy Comparison, in Proceedings of the North American Conference on Logic Programming, Cleveland, Oct. 1989.

[2] Freeman-Benson, B.N. Multiple solutions from constraint hierarchies. Tech. Rep., 88-04-02, University of Washington., Apr. 1988.

[3] Bouzoubaa, M., The Houria Constraint Solver, in Proccedings of the Fifth Scandinavian Conference On A.I., IOS press, Tron., Norway, May, 1995.

[4] Bouzoubaa, M., Neveu, B., Hasle, G. Houria III: Solver For Hierarchical System, Planning of LWSBG for Equational Constraints, in book of the Fifth INFORMS Computer Science, Dallas, Texas, Jan. 8-10, 1996.

Towards an Open Finite Domain Constraint Solver

Mats Carlsson[1,2], Björn Carlson[3], Greger Ottosson[1]

[1] Computing Science Dept., Uppsala University
PO Box 311, S-751 05 UPPSALA, Sweden
[2] SICS, PO Box 1263, S-164 28 KISTA, Sweden
[3] Xerox PARC, 3333 Coyote Hill Road, Palo Alto, CA, 94304-9960

1 Summary

We describe the design and implementation of a finite domain constraint solver embedded in SICStus Prolog system using an extended unification mechanism via attributed variables as a generic constraint interface. The solver is based on *indexicals*, i.e. reactive functional rules performing incremental constraint solving or entailment checking. Propagation is done using the arc-consistency algorithm AC-3, adapted for non-binary constraints. At the heart of the algorithm is an evaluator for indexicals.

The solver provides the usual predefined search strategies (fixed order, fail-first principle, branch and bound for minimization or maximization). Access predicates for the relevant variable attributes (domain value, bounds and size etc.) are also provided, making customized search strategies easily programmable.

A design goal has been to keep the solver open-ended and extendible as well as to keep a substantial part written in Prolog, partly contradicting conventional wisdom in implementing constraint solvers.

2 Design Overview

The solver has an open-ended design in several senses: (1) The user can define new elementary constraints in terms of indexicals. (2) Non-local constraint propagation is available via *global* constraints defined by the user via a programming interface. Such constraints may use specialized algorithms for application specific constraint solving. (3) An elementary constraint can be linked to a 0/1 variable denoting its truth value.

Indexicals are used to define both the constraint solving and the entailment checking aspects of a constraint. Arithmetic expressions and symbolic constraints are compiled to elementary ones or to indexicals (see e.g. [2]). Constraints can also be arbitrarily combined using the propositional connectives.

The interface between the SICStus Prolog engine and the solver is provided in part by the attributed variables mechanism [3], which has been used previously to add several constraint solvers to SICStus and ECL^{i}PS^e Prologs. This

mechanism associates solver-specific information with variables, and provides hooks for extended unification and projection of answer constraints.

The generality and flexibility of the indexical approach comes with a cost. Indexicals have small grain size; the ratio of useful invocations is low for many problems; they prune only one variable at a time, and incur high scheduling overhead since they produce many suspensions ($\mathcal{O}(n^2)$ for a n-ary constraint).

Global constraints, on the other hand, maintain consistency over an *arbitrary* amount of variables, are *resumed* when needed (under certain constraint specified conditions), and use *specific*, *incremental* algorithms for each constraint, e.g. graph algorithms and OR techniques.

In our framework, we mix indexicals and global constraints by having separate scheduling queues for the two; a global constraint is only resumed when no indexicals are scheduled for execution. Thus, global constraints can be seen as having lesser priority than indexicals. This is reasonable, since indexicals (per invocation) are cheap, while specialized algorithms for global consistency are often expensive.

Global constraints are defined via a programming interface, making it possible to incorporate problem specific algorithms to enhance propagation power. For an occurrence of a global constraint c, the interface provides services to suspend and resume c on a collection of variables, to maintain a private state containing information used by incremental algorithms, and to prune variables and propagate the effects. The constraint specific algorithm is responsible for determining (dis)entailment, and for computing the new domains of any variables to be pruned.

References

1. B. Carlson, M. Carlsson, and D. Diaz. Entailment of finite domain constraints. In *Proceedings of the Eleventh International Conference on Logic Programming.* MIT Press, 1994.
2. D. Diaz and P. Codognet. A Minimal Extension of the WAM for CLP(FD). In *Proceedings of the International Conference on Logic Programming*, pages 774–790, Budapest, Hungary, 1993. MIT Press.
3. C. Holzbaur. *Specification of Constraint Based Inference Mechanism through Extended Unification.* PhD thesis, Dept. of Medical Cybernetics and AI, University of Vienna, 1990.
4. Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Design, implementation and evaluation of the constraint language cc(FD). In A. Podelski, editor, *Constraints: Basics and Trends*, volume 910 of *Lecture Notes in Computer Science.* Springer-Verlag, 1995.

Efficient Constraint Propagation With Good Space Complexity

Assef Chmeiss Philippe Jégou
LIM- URA CNRS 1787
CMI- Technopole de Chateau Gombert
39, rue Joliot Curie - 13453 Marseille Cedex 13 - FRANCE
{jegou,chmeiss}@lim.univ-mrs.fr

A constraint network is defined by (X, D, C, R), where X is a set of n variables $\{x_1, \ldots x_n\}$ and D is a set of n domains $\{D_1, \ldots D_n\}$ such that D_i is the set of the possible values for variable x_i. C is a set of e binary constraints where each $C_{ij} \in C$ is a constraint between the variables x_i and x_j defined by its associated relation R_{ij} defining the allowed pairs of values for x_i and x_j (i.e $R_{ij} \subset D_i \times D_j$). Moreover $(b, a) \in R_{ji} \Leftrightarrow (a, b) \in R_{ij}$. The fact that $(a, b) \in R_{ij}$ will be denoted by $R_{ij}(a, b)$ is true. Recently, efficient algorithms have been proposed to achieve arc- and path-consistency in constraint networks (for all definitions and references, see [ChmJeg96b]). The best path-consistency algorithm proposed is PC-6 which is a natural generalization of AC-6 [Bessi94] to path-consistency [ChmJeg95]. Its time complexity is $O(n^3d^3)$ and its space complexity is $O(n^3d^2)$. Unfortunatly, we have remarked that PC-6, though it is widely better than PC-4, was not very efficient in practice, specialy for those classes of problems that require an important space to be run. Therefore, we propose here a new path-consistency algorithm called PC-8, the principle of which is also used to define a new algorithm to achieve arc-consistency, called AC-8. For details on algorithms AC-8 and PC-8 see [ChmJeg96b].

AC-8 and PC-8 are based on the notion of support. For arc-consistency, a *support* for a value $a \in D_i$ w.r.t. a constraint C_{ij} is a value $b \in D_j$ compatible with a, i.e. such that $R_{ij}(a, b)$ is true. If a value $a \in D_i$ does not possess a support w.r.t. a constraint C_{ij}, this value cannot satisfy arc-consistency and then must be removed from its domain. For path-consistency, a support is then a value that allows a pair of values to be compatible: a pair of values $(a, b) \in R_{ij}$ is supported by the value $c \in D_k$ if the relations $R_{ik}(a, c)$ and $R_{jk}(b, c)$ hold. AC-8 is based on supports but without recording any of them. When a value (i, a) is removed from its domain D_i, AC-8 records i, the reference of the variable x_i in a list denoted *List-AC*. Propagations will be realized w.r.t. variables in this list. Suppose that i is removed from the list, then all neighbouring variables will be considered, i.e. for all $j \in X$ such that $C_{ji} \in C$, and for each value $b \in D_j$, AC-8 will ensure that there is a value $a \in D_i$ such that $R_{ij}(a, b)$ holds. Unlike AC-6, AC-8 has to start again the search from the first value of the domains. If no support a of b is found in D_i, then b must be deleted, and j must be inserted in *List-AC*. The inititalization phase consists in checking if, for each $j \in X$ such that $C_{ij} \in C$, there exists at least one support per value $a \in D_i$. So, if for some variable x_j, a has no support in D_j, it must be deleted and i must be added to the list. Concerning the propagation phase, AC-8 restarts looking for a new support from

the first value of the domain. Finally, the scheme of AC-8 is a classical scheme of propagation: the algorithm stops when the list of propagation becomes empty, each step corresponding to the propagation of a deletion. The space complexity of AC-8 is bounded by the size of the list, so it is $O(n)$ while the time complexity of AC-8 is $O(ed^3)$.

PC-8 is based on the same principles as AC-8 and appears to be an optimization of PC-7 [ChmJeg96] with better space complexity. When a pair of values (a, c) is removed from a relation R_{ik}, two 3-tuples will be recorded in a list called *List-PC*: (i, a, k) and (k, c, i). Propagations will be realized w.r.t. these 3-tuples: if (i, a, k) is propagated, then for all values $b \in D_j$ such that $R_{ij}(a, b)$ holds, the propagation must verify that there exists $c \in D_k$ which supports $(a, b) \in R_{ij}$. If no support c of (a, b) is founded in D_k, then (a, b) must be deleted, and two 3-tuples, namely (i, a, j) and (j, b, i) have to be inserted in *List-PC*. The initialization phase consists in checking if there exists at least one support $c \in D_k, \forall k \in X$ per pair of values $(a, b) \in R_{ij}$. So, any pair with no support must be deleted and two 3-tuples, (i, a, j) and (j, b, i), must be added to the list. Finally, the scheme of PC-8 is slightly different from the scheme of PC-4 or PC-6, but it is a classical scheme of propagation since propagations stop when the *List-PC* becomes empty. The space complexity of PC-8 is $O(n^2d)$ while the time complexity of PC-8 is $O(n^3d^4)$.

In [ChmJeg96b], experiments were performed over randomly generated CSPs. AC-8 was compared with AC-3, AC-4 and AC-6. We have found that for the number of consistency checks, AC-6 is the best one. By contrast, concerning CPU time, AC-3 and AC-8 are similar and both outperform AC-6. Concerning path-consistency algorithms, PC-8 was compared with PC-2, PC-6 and PC-7. From our experiments, it is clair that PC-6 realizes the smallest number of consistency checks while PC-7 and PC-8 that are similar, outperform PC-2. For CPU time as a measure of performance, PC-7 and PC-8 are clearly the best algorithms. The fact that for CPU time, PC-8 (resp. AC-8) outperforms PC-6 (resp. AC-6) can be explained naturally by considerations lied to the theoretical complexity in time and space. As AC-6, the data structures of PC-6 leads to an optimal theoretical time complexity, but they increase the CPU time because of the required number of operations for each propagation, which is widely more important than the one of PC-8 (resp. AC-8). So, the multiplicative hidden constant of PC-6 (resp. AC-6) is widely greater than for PC-8 (resp. AC-8).

References

[Bessi94] C. Bessière, Arc-Consistency and Arc-Consistency Again, *Artif. Intell.* 65 (1994) 179-190.

[ChmJeg95] A. Chmeiss and Ph. Jégou, Partial and global path Consistency revisited, *Rapport Interne 1995.120, L.I.M., Marseille* (1995).

[ChmJeg96] A. Chmeiss and Ph. Jégou, Path Consistency: When Space Misses Time, In *Proc. of AAAI-NCAI* (1996) *to appear.*

[ChmJeg96b] A. Chmeiss and Ph. Jégou, Efficient Constraint Propagation With Good Space Complexity, *Rapport Interne 1996, L.I.M., Marseille* (1996).

Anytime Temporal Reasoning : Preliminary Report (Extended Abstract)*

Mukesh Dalal and Yong Feng

Columbia University
Department of Computer Science
New York, New York, 10027, USA.

We present a new approach for determining consistency of temporal constraint networks based on Allen's interval-based framework [All83]. The temporal network is first translated into a clausal theory in propositional logic [Men64], whose satisfiability is then determined using an anytime family of tractable reasoners [Dal96].

Anytime reasoners are complete reasoners that provide partial answers even if stopped prematurely; the completeness of the answer improves with the time used in computing the answer. Our anytime family $\vdash_0, \vdash_1, \ldots$ is built upon clausal boolean constraint propagation (BCP) [McA90], a variant of unit resolution. It is known that each $\vdash_i$ is sound and tractable, each $\vdash_{i+1}$ is at least as complete as $\vdash_i$, and each propositional theory has a complete reasoner $\vdash_i$ for reasoning with it.

A straight-forward translation of a network into a propositional theory can be obtained by directly instantiating each entry of Allen's transitivity table by relations among each triple of intervals in the network. We improve upon this by first embedding a tree in the network and then using the edges of this tree to restrict the number of formulas.

A temporal network is a directed graph whose edges are labeled by subsets of 13 basic temporal relations defined by Allen. Consider any maximal tree M embedded in the given connected network N (our approach extends easily to non-connected networks). The edges in M are called *tree edges*, and the rest of the edges in N are called *back edges*. The notions of *parent*, *ancestor*, etc. are defined as usual, with respect to the tree M. For each tree edge (x, y), $Rel(x, y)$ is defined to be the label of edge (x, y), and for sub-branch $x_1, \ldots, x_k$ of nodes in M, where $k > 2$, $Rel(x_1, x_k)$ is defined to be the set $Rel(x_1, x_2) \circ \ldots \circ Rel(x_{k-1}, x_k)$ of relations. Note that $\circ$ is the composition relation defined by Allen. The translated theory $\mathrm{Tr}(N)$ with respect to tree M consists of exactly the following formulas:

I: $xr_1y \vee \ldots \vee xr_ky$ for each edge (x, y) with label $\{r_1, \ldots, r_k\}$ in N;

II: $\neg xry \vee \neg ytz \vee xr_1z \vee \ldots \vee xr_kz$ for each triple x, y, z of nodes such that x is a proper ancestor of y and y is a proper ancestor of z, for each relation r in $Rel(x, y)$, and for each relation t in $Rel(y, z)$, where $r \circ t = \{r_1, \ldots, r_k\}$;

III: similar to formulas II above, except that $z = x$, x is an ancestor of y, and there is a back edge from y to x; and

* This work is partially supported by NSF Grant No. IRI-94-10117 and ARPA/ARL Contract No. DAAL01-94-K-0119. Email: {dalal,yfeng}@cs.columbia.edu.

IV: zez and $\neg zuz$ for each node z with an incoming back edge and each relation u different from e (equal) such that the atom zuz occurs in some formula III above.

Theorem 1 shows that the above translation is sound and complete:

Theorem 1. *A network N is consistent iff the theory* $\mathrm{Tr}(N)$ *is satisfiable.*

We present an algorithm that obtains a translated theory for any given temporal network N, and prove that the time taken by the algorithm and the size of the translated theory $\mathrm{Tr}(N)$ are both bounded by $O(nm^2)$, where n is the number of nodes in N and m is the length of the longest branch in the embedded tree M. Since m can grow as large as n, the worst-case size of a translated theory is cubic in the size of the network.

Since we are interested only in determining consistency of temporal networks, we use the anytime family $\vdash_0, \vdash_1, \ldots$ to determine the satisfiability of translated theories. The least k needed for obtaining $\mathbf{f}$ from the translated theory using $\vdash_k$ is called the *intricacy* of the network. Since each $\vdash_k$ can be determined in time exponential in k, the intricacy of a network captures the difficulty of detecting its inconsistency using our approach.

We randomly generated thousands of small networks for comparing our approach with three other incomplete reasoners. We found that intricacy is 1 for all networks that were found inconsistent by Allen's path consistency algorithm. However, the other two algorithms, that first translate interval networks into point algebra networks (with and without $\neq$, respectively), could not detect inconsistency in most networks with intricacy 1. We have not yet found any inconsistent network with intricacy greater than 2.

Since the translated theories are quite large, our current approach can be used only for networks with a small number of intervals. We are currently working on further reducing the size of the translated theory, by translating fewer transitivity rules. We are also extending our approach to handle qualitative constraints like "A before B or B before C" that can not be expressed in temporal networks. Our current work also involves inferring implicit temporal constraints from temporal networks, rather than just determining whether a network is consistent.

References

[All83] J.F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

[Dal96] M. Dalal. Anytime families of tractable propositional reasoners. In *Fourth International Symposium on Artificial Intelligence and Mathematics (AI/MATH-96)*, pages 42–45, Florida, 1996.

[McA90] D. McAllester. Truth maintenance. In *Proceedings Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 1109–1116, 1990.

[Men64] E. Mendelson. *Introduction to Mathematical Logic.* Van Nostrand, Princeton, N.J., 1964.

From Constraint Minimization to Goal Optimization in CLP Languages

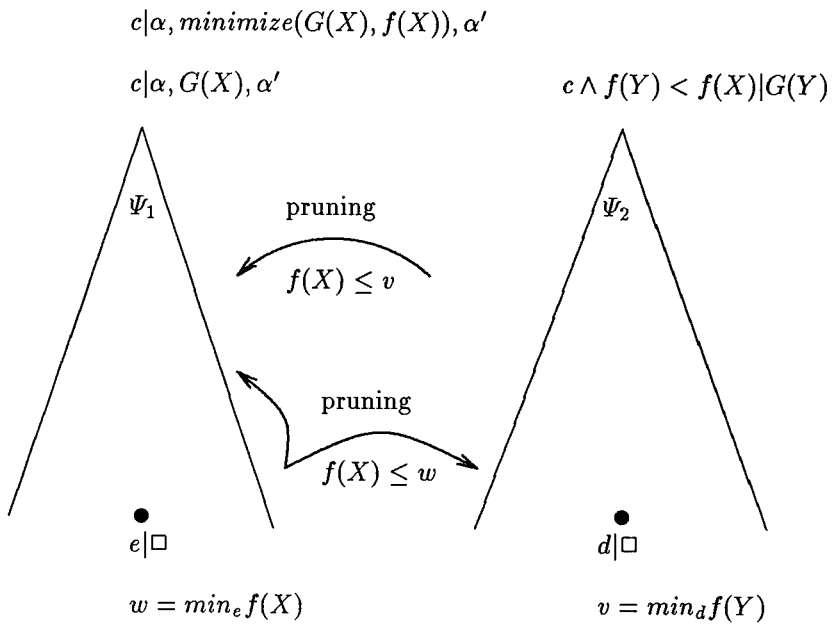
François Fages

LIENS CNRS, Ecole Normale Supérieure,
45 rue d'Ulm,
75005 Paris, France
e-mail:fages@dmi.ens.fr http://www.ens.fr

Constraint logic programming and concurrent constraint programming are simple and powerful models of computation that have been implemented in several systems over the last decade, and proved successful in a variety of applications ranging from combinatorial optimization problems to complex system modeling. In particular the CLP approach is increasingly used to solve hard scheduling and planning problems. Of course in these applications the ability of the system to generate not all solutions but only best solutions is a fundamental property. The basic optimization procedure currently used in CLP systems is a variant of the branch and bound procedure, where constraints are used to prune the search space. That procedure can be used to find optimal solutions of the top-level query w.r.t. an objective function, but it becomes unsound when applied to subgoals of the program. The extension of CLP languages with optimization predicates is an important issue to solve multi-criteria optimization problems and modelize multi-component systems for which several optimization goals have to be combined in the query and/or the program. The problem is to reconciliate the evaluation procedures for optimization goals with the declarative semantics of CLP, and its properties of compositionality.

In this poster we review several forms of optimization within CLP languages, and study different execution models which are *complete* w.r.t. Kunen's three-valued logical semantics of the program's completion. First we define the constraint minimization problem that the constraint solver is assumed to solve, and the basic branch and bound procedure used for query optimization w.r.t. an objective function. Then we show how optimization subproblems can be encapsulated in CLP programs with an optimization higher-order predicate, which is interpreted declaratively under Kunen's three-valued logical semantics of general CLP programs, and for which we give a *complete top-down evaluation procedure*. In this approach optimization predicates can be combined arbitrarily in the program, recursion through optimization predicates is supported without any restriction. The top-down procedure is based on a *concurrent pruning mechanism between standard derivation trees*, the successful derivations to the minimization goal are the successful derivations in the main tree ψ_1 whenever the auxiliary tree ψ_2 gets finitely failed after pruning (see figure below).

More general local optimization predicates with set of protected variables have the power of general CLP programs with negation. We derive a more complex complete top-down procedure from the scheme of constructive negation by



pruning [1] in this context. and propose an alternative bottom-up evaluation procedure based on a finitary version of Fitting's operator [1]. Our claim is not that the complete procedures described for local optimization predicates can be used directly to solve efficiently complex optimization problems but that they can serve as a basis for designing more efficient procedures in particular cases, e.g. under termination or groundness assumptions, and for analyzing their completeness w.r.t. the declarative semantics. For instance if the local optimization goals are delayed until the protected variables get instanciated then one can clearly rely on the previous simpler procedure.

Constructive negation by pruning can be used also to interpret directly preferences among solutions expressed by CLP programs (instead of objective functions) [2]. In this more general setting, an incremental execution model based on dynamic constraint solvers and on the set of operators for transforming derivations described in [3] is discussed w.r.t. interactive and multi-criteria optimization problems.

References

1. F. Fages, "Constructive negation by pruning", LIENS tech report 94-14 to appear in the Journal of Logic Programming.
2. F. Fages, J. Fowler, T. Sola, "Handling preferences in constraint logic programming with relational optimization", Proc of PLILP'94, Madrid, Springer Verlag, LNCS 844, pp.261-276, 1994.
3. F. Fages, J. Fowler, T. Sola, "A reactive constraint logic programming scheme", Proc of ICLP'95, Kanagawa, MIT Press pp.149-163, 1995.

Looking at Full Looking Ahead *

Daniel Frost and Rina Dechter

Dept. of Information and Computer Science
University of California, Irvine, CA 92717-3425 U.S.A.
{dfrost, dechter}@ics.uci.edu

In 1980, Haralick and Elliott [1] introduced the forward checking (FC) algorithm, as well as two variants which they called partial looking ahead (PLA) and full looking ahead (FLA). Forward checking is a modification to backtracking search: after each variable is instantiated, values from the domains of future variables are filtered out if they are inconsistent with the present partial assignment. The extra work required to do this filtering almost always pays off in the reduced size of the search space. Full looking ahead is a further modification which does more extensive processing of future variables by comparing each future variable with each other, in effect performing a single iteration of the arc-consistency "revise" procedure described in [2]. Partial looking ahead is an abbreviated form of full looking ahead, where future variables are only compared with those after them in the ordering.

Over the last 15 years, forward checking has become one of the primary algorithms in the CSP-solver's arsenal, while partial and full looking ahead have received little attention. This neglect is due, no doubt, in large part to the conclusions reached in [1]: "The checks of future with future units do not discover inconsistencies often enough to justify the large number of tests required."

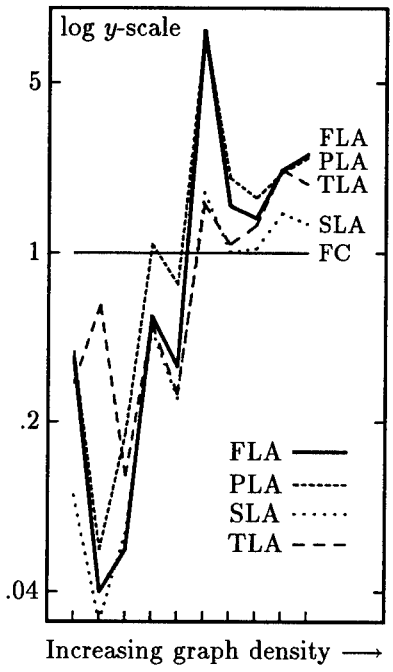
Our experiments demonstrate that when combined with the dynamic variable ordering heuristic described in [1], the full looking ahead algorithm is more useful than often supposed, and in fact substantially outperforms forward checking on problems with relatively tight constraints and relatively sparse constraint graphs. Because we find experimentally that each algorithm is superior to the other on certain types of problems, we are interested in the possibility of automatically invoking the superior heuristic on any individual problem. Another approach is to vary the amount of look ahead within an individual problem, adopting either the forward checking level or the full looking ahead level on different sub-trees, according to some heuristic. We present below three new variants of looking ahead which take this approach, using three different heuristics.

Experiments reported in the full paper (available through `http://www.ics.uci.edu/~dechter`) indicate that the extra work performed by full looking ahead is most effective higher up in the search tree. We therefore developed a version of full looking ahead which we call truncated looking ahead (TLA). The modification is simple: the extra processing associated with full looking ahead is done

* This work was partially supported by NSF grant IRI-9157636, by the Electrical Power Research Institute (EPRI) grant RP8014-06, and by Rockwell MICRO grants UCM-20775 and 95-043.

only when the newly instantiated variable is at depth 10 or less in the search tree. At greater depths, truncated looking ahead is identical to forward checking.

The self-adjusting looking ahead algorithm (SALA) starts the full looking ahead process after each instantiation at every level in the search tree, but stops (for that one instantiation) if sufficient progress is not being made. Progress is defined as removing a sufficient fraction of the values in the domains of future variables, and is controlled by a parameter.



One advantage of performing full looking ahead is that many values in the domains of future variables are removed. This leads to another advantage: a dynamic variable ordering heuristic based on domain size is more likely to find a future variable which has a very small domain. In the absence of a dead-end, we hope to find a future variable with a domain size of 1, since instantiating this single value represents a forced choice that will have to be made eventually. The smart looking ahead (SLA) algorithm performs the full looking ahead level of consistency enforcing at each instantiation, but stops when the remaining domain size of some future variable becomes 0 or 1. The goal is to do enough looking ahead to guide effectively the variable ordering heuristic.

The chart shows the relative performance of five algorithms (SALA has been omitted for legibility), each run with the dynamic variable ordering scheme proposed in [1]. The x-axis indexes 10 sets of parameters (number of variables, number of values per variable, number of binary constraints, tightness of constraints), all at the 50% solvable crossover point. For each set of parameters we generated 500 random problems. The y-axis displays the ratio of each algorithm's mean CPU time to that of forward checking, using a logarithmic scale. Our conclusion is that smart looking ahead is the most promising: its performance is between forward checking's and full looking ahead's in six out of ten cases, and is better than both in the other four.

References

1. R. M. Haralick and G. L. Elliott. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.
2. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

The Arc and Path Consistency Phase Transitions

Stuart A. Grant* and Barbara M. Smith

Division of Artificial Intelligence, School of Computer Studies,
University of Leeds, Leeds LS2 9JT, United Kingdom
e-mail: {stuartg,bms}@scs.leeds.ac.uk
http://www.scs.leeds.ac.uk/{stuartg,bms}/

The phenomenon of phase transitions occurring in many classes of NP-complete problem as a control parameter is varied has been recognised and studied extensively in recent years. Cheeseman, Kanefsky and Taylor [1] first reported the phase transition as the interface between a region where almost all problems have many solutions and are relatively easy to solve, and a region where almost all problems have no solution and their insolubility is relatively easy to prove. In this intervening region, the probability of problem solubility falls from close to 1 to close to 0, and the median cost of searching these problems is highest, reaching a peak at the *crossover point* [2] where 50% of problems are soluble. In binary constraint satisfaction problems (CSPs) with a fixed size and constraint density, the phase transition occurs as the tightness of the constraints increases, from where problems are under-constrained to where they are over-constrained.

In looking at establishing levels of *consistency* in CSPs, reported fully in [3], we observe what appears to be phase transition behaviour exactly analogous to that found for full search. Arc consistency (AC) and path consistency (PC) were established in samples of randomly generated CSPs of fixed size and varying constraint density and tightness, using the AC3 and PC2 algorithms [4] respectively. For problems of fixed size and constraint density, a peak in the average cost of establishing consistency occurs between a region of constraint tightness where the particular level of consistency can be established in all problems, and achieving this is easy, and a region where the particular level of consistency cannot be established in any problem (showing each to be insoluble), and achieving this is easy. In the intervening region, a proportion of problems can be made consistent, and it is observed that the peak in cost approximately coincides with the point where this is true for 50% of problems.

The average costs of AC3 and PC2 were investigated in terms of both consistency checking effort and the number of arc- or path-inconsistent values, or "nogoods", pruned from variable domains. Both algorithms were set to terminate upon the first wipe-out of an entire variable domain, when it is clear that a problem is insoluble. In terms of domain pruning, as we move into the AC or PC phase transitions, AC3/PC2 starts to find a number of nogoods in average problems, although not enough to cause the complete wipe-out of any variable domain. At the peak in cost, domain wipe-out occurs for about 50% of problems, allowing the algorithm to terminate early. Moving into the inconsistent

* Stuart Grant is partly supported by a studentship from British Telecom plc.

region, the number of nogood values in problems is still increasing, but causes domain wipe-out to occur more quickly. This results in earlier termination of the algorithm, and thus a fall in the average domain pruning cost. The pattern of behaviour for consistency checking through the AC and PC phase transitions matches that of domain pruning, with the peak in effort coinciding with the peak in nogoods pruned by the algorithm.

In reporting the phase transition behaviour associated with establishing AC and PC in CSPs, we make an analogy with the phase transition behaviour observed when finding a *single* solution to the same problems. At first glance, it appears that the analogy should in fact be made with respect to finding *all* solutions to problems. However, if we consider establishing a certain level of consistency in a problem as *performing the minimal amount of work necessary to prove that the problem can possess such consistency*, then the validity of the analogy becomes clear: when attempting to establish k-consistency in an n-variable problem, where $k < n$, all paths of length k must be made consistent and the effects of the removal of inconsistent assignments must be propagated around the other paths; but when attempting to establish n-consistency, only one path of length n exists (the variable ordering is immaterial), and the discovery of one consistent set of labels for the variables in the form of a solution is sufficient to *prove* that the problem can be made n-consistent. The existence of phase transition behaviour in establishing AC and PC suggests that similar behaviour will be found when establishing the existence of higher levels of consistency in CSPs, up to that of establishing full consistency by finding a solution.

A practical application of the AC and PC phase transitions is that they clearly indicate the regions where establishing consistency has a domain pruning effect. While there is very little effect on average in the under-constrained region, establishing consistency in the over-constrained region proves the insolubility of many problems without the need for further search. Thus, if the location of consistency phase transitions can be accurately predicted for CSPs, as that for full consistency can [5], then we can easily determine the suitability and likely cost of establishing consistency as a preprocessing step to full search.

References

1. P. Cheeseman, B. Kanefsky, and W. Taylor. Where the Really Hard Problems are. In *Proceedings IJCAI-91*, volume 1, pages 331–337, 1991.
2. J. M. Crawford and L. D. Auton. Experimental Results on the Crossover Point in Satisfiability Problems. *Artificial Intelligence*, 81:31–57, Apr. 1996. Special Issue on Frontiers in Problem Solving: Phase Transitions and Complexity.
3. S. A. Grant and B. M. Smith. The Arc and Path Consistency Phase Transitions. Research Report 96.09, School of Computer Studies, University of Leeds, Mar. 1996.
4. A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.
5. B. M. Smith and M. E. Dyer. Locating the Phase Transition in Constraint Satisfaction Problems. *Artificial Intelligence*, 81:155–181, Apr. 1996. Special issue on Frontiers in Problem Solving: Phase Transitions and Complexity.

Experiences with Combining Constraint Programming and Discrete Event Simulation

ir. Wim Hellinck
SimCon n.v.
Plantin & Moretuslei 216, B-2018 Antwerpen, Belgium
Tel: +32 3 2350805 / Fax: +32 3 2356469 / e-mail: simcon@simcon.be

1 Introduction

At SimCon we developed the SimCon Constraint Object Oriented Programming library (SCOOP) to be used for a large planning application and to be integrated with AutoMod, a discrete event simulation tool. We outline SCOOP's main features and we describe how we combined constraint programming with discrete event simulation. We conclude with some interesting experiences.

2 SCOOP

SCOOP includes several classes of constrained variables, built-in constraints, control structures and search strategies, and it provides specific support for debugging. The library is easily extendible and adaptable to fit application specific needs.

All constrained variables in SCOOP are modeled as C++ objects and we support finite domains containing integers, strings, booleans or pointers (to objects). All constrained variable objects contain an info-object. This allows to add semantic information to the constrained variable, which may be consulted during search or propagation. This has proven to be very useful for writing problem specific constraints or search strategies.

Constraints between constrained variables are objects as well in our library. How a constraint should react upon changes of the domain of one of its constrained variables is defined by demons. A demon is triggered in reaction to a propagation event (e.g. minimum of the domain has increased). The user can write its own constraints by combining built-in constraints or by defining a new constraint class (deriving from the C_Constr virtual base class) for which propagation demons have to be defined. Each constraint can be put to sleep. This implies that the constraint will no longer be used in the program and it is completely ignored until you explicitly wake it up. The ability to deactivate constraints is very powerful to answer what-if questions.

By default all constraints are treated equally important. This means that all actions resulting upon propagation events are handled on a first come first served basis. However, you can attach priorities to constraints to change this behavior. In that case actions with the highest priority are treated first.

SCOOP incorporates a number of general tree search strategies which are easily extendible. The backtracking mechanism is open, allowing the user to introduce its own datastructures that should be backtracked upon. Currently we support chronological backtracking and backtracking to a 'labeled' choicepoint.

In order to increase the usefulness of SCOOP we have introduced several user exits in the library. These are places where the user may introduce its own functions to enhance or modify the built-in behavior of the library.

Another feature of SCOOP is the ability to reset the domain of a constrained variable. This is useful for an optimization method that modifies a solution by perturbing it.

SCOOP includes classes implementing time-intervals and constraints between them (e.g. precedes, meets with, ...). To efficiently tackle disjunctive constraints on intervals we apply an edge finding technique.

3 Combining Constraint Programming with Simulation

Discrete event simulation is mostly used to help determine operational control rules and to accurately answer system design questions. Simulation based scheduling is especially useful in large discrete manufacturing environments where other technologies are not applicable. Simulation is fast but it does not allow backtracking. This makes addressing all constraints difficult in a complex environment.

We tried several mechanisms to combine constraint programming with simulation. In a sequential approach we first apply simulation followed by constraint programming. In fact what we do is we reduce the complexity of the problem by using the results of the simulation to turn preferences into hard constraints, derive additional constraints, derive values for some constrained variables, or parametrise the search strategy.

We have already applied discrete event simulation after we have found a solution with constraint programming, but the tasks of simulation were limited to gather some statistical data about the solution and to offer the user a 3D graphical animation.

Another approach is that both technologies could be used on a different level of abstraction. You can e.g. solve the problem on a higher level of abstraction using constraint programming and detail the solution on a lower level using simulation.

Another general integration approach we have tried is to decompose a problem and apply the most appropriate technology to each subproblem.

4 Experiences

In this section we will present some of our experiences with solving problems with constraint programming. A typical difficulty are soft constraints representing preferences. A way of tackling such a problem is by modeling it at as a Minimal Violation Problem. If you attach costs to violations of preferences, you can reformulate your initial problem as a minimization problem of this cost function. Unfortunately, complex cost constraints usually lead to insufficient propagation for use in a branch & bound optimization. Another way of dealing with soft constraints is by writing specific variable and value ordering heuristics that take these preferences into account. This method is only useful if the number of soft constraints is limited and if the hard constraints of the problem propagate well enough to quickly reduce the search space. We have also dealt with soft constraints in a problem by partially converting them into hard constraints. We verified whether this was justified by performing some preprocessing simulations.

A major difficulty we encountered with constraint programming is to explain why no solution for a problem can be found. The user wants some feedback to identify how he should adapt the input data or which constraints should be relaxed. Depending on the problem you may add more preprocessing of data to eliminate the more obvious cases that would lead to inconsistencies or to indicate potential problem constraints. Another approach is to let the user interactively set or relax some constraints. This way you give the user some control over the construction of a solution. Usually the order in which he sets constraints corresponds to their relative importance. If a definitive failure occurs it relates to the constraint that is last introduced.

Hill-Climbing with Local Consistency for Solving Distributed CSPs

Katsutoshi Hirayama

TISE, Kobe University of Mercantile Marine
5-1-1 Fukae-minami-machi, Higashinada-ku, Kobe 658, JAPAN
E-mail: hirayama@ti.kshosen.ac.jp
WWW: http://jos3.ti.kshosen.ac.jp:8080/~hirayama/

1 Introduction

Distributed CSPs have recently drawn the attention of the researchers in Multi-Agent Systems. Those problems can be considered to be CSPs where variables and constraints are distributed among multiple agents.

We have presented the hill-climbing based distributed constraint satisfaction algorithm[1], in which agents perform hill-climbing mutually excluding each other, and form *coalitions* to make violated constraints consistent when they meet dead-ends. This algorithm, called *Hill-Climbing with Local Consistency* (HCLC), is characterized by the local consistency procedure which is invoked by an agent at a dead-end, and the negotiation procedure for agents to mutually exclude their hill-climbing. In the previous paper, we mainly discussed the local consistency procedure in detail, but not fully discussed the property and effect of the negotiation procedure.

We discuss two things about the negotiation procedure in this paper. One is the property that the action of neighboring agents is actually suppressed through the procedure. The other is experimental results which shows the effect of negotiation. Added to these, we propose a simple strategy, called *weight adjusting*, for improving the performance of the algorithm.

2 Framework

A Distributed CSP is defined as a problem where each agent has variables, domains, and constraints. The goal of each agent is to find one set of assignments to its variables that satisfies all its constraints. Some of the constraints, however, are defined over sets of variables including other agents'. Accordingly, agents must communicate each other to achieve their goals.

We have developed a hill-climbing method for solving Distributed CSPs. In this method, all agents try to change their assignments in order to reduce *costs*, which is the numbers of violated constraints of their own. However a change of assignments by one agent affects the *state spaces* of its neighbors. This means that if we permitted all agents to change their assignments as they like, there might be a situation where an agent repeatedly fails to reduce a cost, and not really reaching a solution as a result. To eliminate such a situation, we have

developed a coordination mechanism that enables an agent to explicitly suppress the neighbors' action of hill-climbing. We refer to the procedure as *negotiation.* For lack of space we omit the description of the negotiation procedure, and just give the following theorem without proof.

Theorem 1. *Given finite delays in delivering messages and FIFO(First-In-First-Out) message passing, the negotiation procedure realizes mutual exclusion of hill-climbing among neighboring agents.*

3 Experimental Results

To see an effect of the negotiation procedure, we made experiments to compare HCLC with *Asynchronous Weak-Commitment search* (AWC)[2], which does not involve a mechanism for explicit coordination. Two classes of instances, called *sparsely-connected graphs* and *densely-connected graphs*, of distributed 3-coloring problems were generated and solved with both algorithms on a simulator. The number of variables is ranged over 30, 50, and 70 for each class of instances.

Results of our experiments are summarized as follows: (1) AWC works better in terms of the estimated time complexity in all cases; (2) for loads of constraint checks in agents, HCLC is better on densely-connected graphs. It's also better on small-sized sparsely-connected graphs; (3) HCLC uses less messages in all cases; and (4) a smaller amount of repairing assignments is required for HCLC.

Our next concern is to reduce the amount of constraint checks which is done by HCLC for large-sized sparsely-connected graphs. To do this, we define a *weight*, a positive integer, for each constraint in one agent such that its value corresponds to the number of variables in another agent who shares the constraint; and measure a cost as the sum of weights of violated constraints. We call this strategy *weight adjusting.* Results of our experiments are encouraging, because we got at least four times improvement.

4 Conclusions

We mainly discussed the property and effect of the negotiation procedure used in HCLC, and proposed a new strategy for improving its performance on a certain class of distributed 3-coloring problems. Our future work is testing the algorithm on other classes of the problem and other kinds of problems.

References

1. Hirayama, K. and Toyoda, J.: Forming Coalitions for Breaking Deadlocks. *Proceedings of First International Conference on Multiagent Systems (ICMAS-95)* (1995) 155–162
2. Yokoo, M.: Asynchronous Weak-commitment Search for Solving Distributed Constraint Satisfaction Problems. *Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP-95)* (1995) 88–102

Approximate Algorithms for Maximum Utility Problems

F.J. Jüngen and W. Kowalczyk

Vrije Universiteit Amsterdam, Artificial Intelligence Group, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands, email: {fjjunge,wojtek}@cs.vu.nl

Abstract. Many practical problems are overconstrained, i.e., it is impossible to find a solution that would satisfy all constraints. In such situations one may try to find solutions that satisfy as many constraints as possible, see [1]. In a more general approach one can assign some weights (utilities) to constraints and then look for solutions that maximize the sum of weights of satisfied constraints. The problem of finding such solutions, the Maximum Utility Problem, MUP, is the subject of this paper. We present a number of approximate algorithms for solving this problem. Numerous tests have demonstrated high performance of these algorithms: approximated solutions are, on average, about 2% worse than optimal ones. Our algorithms are local search procedures that iteratively improve the current solution by modifying it. As a heuristic that guides the whole process we use the expected utility value of a solution that can be obtained from the current (partial) solution by extending it to a complete one at random. This heuristic has been motivated by an old algorithm for solving the k-MAXGSAT problem, which was proposed by Johnson, [2].

1 Main Result

We will start with some definitions. *A Maximum Utility Problem, MUP*, is a tuple $P = < V, D, C, U >$ given by: a set of variables, $V = \{x_1, x_2, \cdots, x_n\}$, a collection of finite domains for these variables $D = \{D_1, \cdots, D_n\}$, a set of constraints $C = \{C_1, \cdots, C_m\}$ and a set of corresponding utilities $U = \{u_1, \cdots, u_m\}$, where each utility is a real number.

A solution of P is an arbitrary (complete) valuation v of involved variables that maximizes the total utility of constraints: $u(v) = \sum_{i=1}^{m} u_i \chi(C_i, v)$ where $\chi(C_i, v) = 1$ if C_i is satisfied by v; 0 otherwise.

Let v be a partial valuation and let $p(v, C_i)$ denote the probability of satisfying C_i by a randomly selected complete extension of v (all extensions have the same chance). Then the *expected utility of* v, $e(v)$, is defined as $e(v) = \sum_{i=1}^{m} u_i p(v, C_i)$.

The expected utility has the following, very useful, property:

Theorem 1. *Let v be a partial valuation and x an arbitrary variable which is not instantiated by v. Then there exists a value d for x such that $e(v[x/d]) \geq e(v)$. Consequently, v can be extended to a complete valuation w such that $u(w) \geq e(v)$.*

2 Experiments

We developed a number of algorithms based on the results of section 1. $SAT1$ chooses the best variable/value pair to instantiate in each step (a variable/value pair which maximizes the expected utility). $SAT2$ and $SAT3$ do the same but they instantiate 2 resp. 3 variable/value pairs in each step. $SAT1+$, $ESAT1$ and $LSAT1$ are all based on $SAT1$. $SAT1+$ tries to improve a solution found by $SAT1$ by revising old assignments. $ESAT1$ works similarly to $SAT1$, but instead of focusing only on uninstantiated variables, all variables are taken into account. Finally, $LSAT1$ extends $SAT1$ by constantly revising the generated fragment of a solution. See [3] for a more detailed description.

The algorithms were tested on 4725 problems which were optimally solved by branch and bound. The problem parameters were chosen as follows: number of variables (10, 15, 20), maximum utility of a constraint (1, 10, 20), density of the constraint graph (0.2, 0.3, $\cdots$, 0.8) and the tightness (0.2, 0.5, 0.8). The domain size remained constant (4). For each combination of above parameters 25 problems were generated and solved. The results of the tests can be found in table 1.

Table 1. Compressed results of the approximate algorithms

Algorithm	mean error (%)	std. deviation	no. constraint checks	no. optimal
SAT1	2.72	3.41	2.38e+08	1539
SAT2	2.00	2.67	2.87e+09	1830
SAT3	1.65	2.33	3.61e+10	2004
SAT1+	2.10	3.03	2.61e+08	1905
ESAT1	2.08	3.02	5.06e+08	1914
LSAT1	1.86	2.86	5.15e+08	2051

3 Conclusions and Further Research

All algorithms perform surprisingly well: average approximation error varies from 2.72% for $SAT1$ to 1.86% for $LSAT1$, the best algorithm with complexity comparable to that of $SAT1$. Further research will include testing these algorithms on large problems and comparing them with algorithms like simulated annealing and genetic algorithms.

References

1. Freuder, E.C., Wallace, R.J.: Partial constraint satisfaction. Artificial Intelligence **58** (1992) 21–70
2. Johnson, D.S.: Approximate algorithms for combinatorial problems. Journal of Computer and Systems Sciences **9** (1974) 256–278
3. Jüngen, F.J., Kowalczyk, W.: Solving Over-constrained Problems with an Expected Utility Heuristic. IR 405, Faculty of Mathematics and Computer Science, Vrije Universiteit Amsterdam (1992)

A Meta Constraint Logic Programming Architecture (Extended Abstract)

E. Lamma[1], P. Mello[2], M. Milano[1]

[1] DEIS, Univ. Bologna, Viale Risorgimento 2, 40136 Bologna, Italy
email: {mmilano, elamma}@deis.unibo.it
[2] Dip. Ingegneria, Univ. Ferrara, Via Saragat, 41100 Ferrara, Italy
email pmello@ing.unife.it

Abstract. This work presents a general meta CLP architecture which can be built by adding CLP solvers as meta levels reasoning on the constraints of the underlying object system. We propose two specializations on finite domains: the first concerns the possibility of embedding the capability of performing qualitative reasoning in a CLP framework, while the second concerns a multi-level architecture for obtaining different degrees of consistency by using weaker algorithms.

Constraint Logic Programming (CLP) [1, 2] is a class of programming languages combining the LP declarative semantics with the efficiency of constraint solving. These features have lead CLP to receive increasing attention in recent years from both a theoretical and a practical point of view.

In this work, we focus our attention on CLP on finite domains, CLP(FD), an expressive and flexible language for solving the so called Constraint Satisfaction Problems (CSP). However, some applications in CSP field need a greater flexibility than that provided by usual CLP(FD) solvers. For example, in the field of temporal reasoning, some applications require a more powerful propagation algorithm than arc-consistency, like path or 4-consistency. CLP(FD) constraint solvers have an embedded propagation algorithm (namely arc-consistency) which cannot be changed by the user in accordance with the application to be solved.

Furthermore, temporal reasoning problems usually require to reason also on the so called *qualitative constraints* [4]. One of the problems faced in a temporal framework is to find the minimal network, i.e., a constraint graph where all constraints are the tightest. For example, if $X < Y$, $Y < Z$ and $X \leq Z$ then we want to infer the tightest constraint $X < Z$. Moreover, in usual constraint solvers variable domains are reduced in accordance with constraints, but it never happens that constraints are reduced according to variable values. For example, if X ranges on [1..5] and Y on [7..16], and the constraint linking the two variables is $X \leq Y$, the constraint $=$ is no longer entailed by variables values. We would like to propagate the constraint that become $X < Y$. This propagation is not provided by usual constraint solvers because they do not reason on *passive constraints* (e.g., binary constraints) in the sense that these constraints are used for the propagation but are never changed during the computation.

In this paper, we propose a solution to both the above mentioned problems via a meta CLP architecture which can be specialized on particular domains (a longer version of this paper can be found in [3]). In the meta architecture, each level contains a representation of the object-level constraint store in its domain. For example, a CLP(FD) solver works on constraints like $X :: [1..10]$, $Y :: [5..10]$, $X \leq Y$. Unary constraints (active constraints) are modified by the solver (by reducing domain values through constraint propagation) according to the binary constraint. The binary relation (passive constraint), instead, is never changed during the computation. If we want to reason on this relation, we can represent it explicitly or implicitly in the meta constraint store.

We propose two specializations of a meta CLP(FD) architecture: the first concerns the possibility of embedding in a CLP framework the capability of performing qualitative reasoning. We implicitly represent the relation $X \leq Y$ via a meta variable whose domain contains the constraint symbols linking the object level variables X and Y. We can define meta-operations on constraint symbols (like union, composition and intersection) and meta-constraints (like the tighter relation). In this way, we can find the minimal network and reason on constraints of the underlying system thus performing qualitative reasoning.

The second specialization is a multi-level architecture which reaches whatever degree of consistency by composing weaker algorithms. Each level is a CLP(FD) solver adopting an arc-consistency propagation algorithm. By adding further levels, we are able to incrementally increase the consistency degree achieved in the whole architecture by using weaker algorithms without changing the structure of each constraint solver. We explicitly represent in the meta-level the relation $X \leq Y$ in terms of consistent couples allowed for the object level variables X and Y. Therefore, the meta-level variables range on the consistent couples. By performing an arc-consistency on couples of values we achieve path consistency on the underlying level solver. In general, by adding a meta-level solver performing a k-consistency, we incrementally increase the consistency achieved by the overall architecture by a factor $k - 1$.

In our architecture, we combine the efficiency and declarativeness of CLP with the flexibility and modularity of meta architectures. We are currently investigating the possibility of performing the *amalgamation* of different levels in a single constraint solver thus also increasing the expressive power of CLP(FD).

References

1. J.Jaffar , J.L.Lassez, "Constraint Logic Programming", in *Proceedings of the Conference on Principle of Programming Languages*, Munich 1987.
2. J.Jaffar, M.J.Maher, "Constraint Logic Programming: a Survey", in *Journal of Logic Programming on 10 years of Logic Programming*, 1994.
3. E.Lamma, P.Mello, M.Milano, "A Meta Constraint Logic Programming Scheme", Tech. Report DEIS-LIA-95-005, Bologna University, 1995.
4. P.Van Beek, "Reasoning about Qualitative Temporal Information", in *Artificial Intelligence*, Vol. 58, 1992, pp. 297–326.

N-Ary Consistencies and Constraint-Based Backtracking

Pierre-Paul Mérel, Zineb Habbas, Francine Herrmann and Daniel Singer

Laboratoire de Recherche en Informatique de Metz
Faculté des Sciences de Metz
Ile du Saulcy, F-57045 Metz Cedex
{merel, zineb, herrmann, singer}@lrim.univ-metz.fr

1 Introduction

In this paper, we compare definitions of n-ary consistency introduced by Dechter & van Beek [1] and Jégou [2]. We show the duality between relational-k-consistency and hyper-k-consistency. The algorithm CBT: Constraint-based BackTracking, results from this comparison study. It is a dual approach with respect to the standard backtrack algorithm (variable-based BT).

2 Definitions and Notations

Definition 1 [3]. A **Constraint Satisfaction Problem** $\mathcal{P}$ is a tuple $\mathcal{P} = (X, D, C, R)$. $X = \{X_1, ..., X_n\}$ is a set of n *variables*. $D = \{D_1, ..., D_n\}$ is a set of n *domains*. Each D_i is associated with a X_i. $C = \{C_1, ..., C_m\}$ is a set of m *constraints*. Each constraint C_i is defined by a set of variables $\{X_{i_1}, ..., X_{i_{n_i}}\} \subseteq X$. $\{C_1, ..., C_m\}$ is called the *scheme* of C. $R = \{R_1, ..., R_m\}$ is a set of m *relations*. Each relation R_i defines a set of l n_i-tuples on $D_{i_1} \times ... \times D_{i_{n_i}}$ compatible w.r.t. C_i.

Definition 2.

- A CSP $\mathcal{P} = (X, D, C, R)$ is **relational-k-consistent** [1] iff
  $\forall C_1, \ldots, C_{k-1} \in C, \forall x \in \cap_{C_i}^{1 \le i \le k-1}$,
  $\rho((\cup_{C_i}^{1 \le i \le k-1}) \setminus \{x\}) \subseteq (\bowtie_{R_i}^{1 \le i \le k-1})[(\cup_{C_i}^{1 \le i \le k-1}) \setminus \{x\}]$.
- A CSP $\mathcal{P} = (X, D, C, R)$ is **hyper-k-consistent** [2] iff
  $\forall C_1, \ldots, C_k \in C, (\bowtie_{R_i}^{1 \le i \le k-1})[(\cup_{C_i}^{1 \le i \le k-1}) \cap C_k] = R_k[(\cup_{C_i}^{1 \le i \le k-1}) \cap C_k]$.
  $\forall C_i \in C, R_i \neq \emptyset$.

We recall the notations: *projection* is denoted $R_1[A]$, *join* is denoted $R_1 \bowtie R_2$, and $\rho(A)$ is the set of all consistent instanciations of the variables in A.

3 Comparative Study of N-ary Consistencies

Theorem 3. *hyper-k-consistency $\not\Rightarrow$ relational-k-consistency.*

Theorem 4. relational-k-consistency $\not\Rightarrow$ hyper-k-consistency.

Relational-k-consistency filters domains. Conversely, hyper-k-consistency filters all the relations, but leaves domains unchanged. We claim that relational-consistency and hyper-consistency are not equivalent but *dual*. This duality induces a new algorithm, named CBT, dual of the variable-based backtrak algorithm. CBT exploits filtering effects of hyper-k-consistency.

4 Constraint-based Backtracking: CBT

The general principle of CBT is to change instanciated objects in one step of a backtrack procedure: each time the procedure CBT is called, a set of variables (instead of one in procedure BT) is instanciated. The set corresponds to the scheme of one constraint. Actually, CBT is a dual version of BT.

Theorem 5. *The complexity of CBT is in $O(l^m)$. Recall that $m = |C|$: the number of constraints in the network, and l is the maximum number of tuples in a relation.*

- CBT will be trivialy faster than BT when $n > m$.
- The ratio $\frac{n}{m}$ seems to be a good parameter to choose between CBT and BT.
- CBT is faster than BT when $n \approx m$.
- As relational-k-consistency filters CSPs for variable-based backtracking, hyper-k-consistency filters CSPs for constraint-based backtracking.

Corollary 6. *Let $\mathcal{P} = (X, D, C, R)$ be an hyper-k-consistent CSP, and let $o(C)$ be an order on C. If the width of $o(C)$ on the dual graph of $\mathcal{P}$ is smaller than k, than CBT is backtrack-free relatively to $o(C)$.*

5 Perspectives

New methods should be developed to enhance Constraint-based BackTracking. The main directions to reduce cost of CBT we are looking for, are: *order heuristics, looking-back methods, looking-ahead methods* and *filtering methods.*

References

1. R. Dechter and P. van Beek. Local and global relational consistency. In *Proceedings of the 1st International Conference on Principles and Practice of Constraint Programming*, pages 240–257, Cassis, 1995.
2. Ph. Jégou. *Contribution à l'étude des problèmes de satisfaction de contraintes : algorithmes de propagation et de résolution – propagation de contraintes dans les réseaux dynamiques.* PhD thesis, Université des Sciences et Techniques du Languedoc, Montpellier, 1991.
3. U. Montanari. Networks of constraints: Fundamental properties and applications to pictures processing. *Information Sciences*, 7:95–132, 1974.

Global Behaviour for Complex Constraints

Stéphane N'Dong * and Michel Van Caneghem **

LIM - URA CNRS 1787
Case 901 - 163, avenue de Luminy
13288 Marseille Cedex 9 - FRANCE
Tel : (33)-91-26-86-36. Fax : (33)-91-41-96-37.
e-mail : {Stephane.N.Dong,Michel.Van.Caneghem}@lim.univ-mrs.fr

This work takes place in the field of Constraint Logic Programming based on the Interval Propagation Approach [3, 6, 7, 9]. The Fix Point Algorithm (the kernel part of Propagation techniques) is based on local consistency through networks of relations. These relations belong to a set of particular relations, known by the system : the primary relations. Complex constraints are systematically decomposed into primary relations. The major weakness of this technique is that only local reasoning at the level of primary relations is done.

When this level of consistency is too weak (non-convex relations, multiple occurrence variables, ...), it should be very interesting to have the ability to tackle some delicate constraints globally (*i.e.* handling such complex constraints as a whole). We introduce here a very simple methodology to cope with this problem: it only relies on basic Interval techniques (Fix Point and Enumeration) and the global semantics of the solver is unchanged (*i.e.* the characterization of the solutions is the same). The main idea can be stated as follows: if we want to handle a complex constraint (C) globally, then a specific enumeration will be called to narrow the domains of the variables of (C) each time the projections of this constraint have to be re-evaluated (the basic step of the Fix Point Algorithm). The Fix Point Algorithm and the Enumeration are, in that case, mutually recursive, contrarily to usual enumeration techniques.

Let's show now some experimental results we can obtain solving Broyden-Banded equation systems, in Prolog IV [1], using this technique. The results will be compared to Newton's results (a sophisticated CLP(Intervals) language using formal handling, Newton derivative methods, and a specific consistency notion: the box-consistency [2]).

The Broyden-Banded problem, made of polynomial equations of the third degree, is a classical benchmark for testing numerical solvers. Here is the definition of the problem H_n (n equations, n variables) [4, 5] :

$$\forall i \in 1..n,\ x_i(2+5x_i^2)+1-\sum_{j\in J_i} x_j(1+x_j)=0$$
$$\text{with } J_i=\{j\in I\!N \mid j\neq i, MAX(1,i-5)\leq j\leq MIN(n,i+1)\}$$

* This work is done under a CIFRE contract with PrologIA, and an ANVAR research contract.

** Computer Science Professor, Université de la Méditerranée.

We solve this problem applying complex constraint handling on the equations $y_i = x_i(2 + 5x_i^2)$ and $z_i = x_i(1 + x_i)$ which cannot be solved efficiently by a simple decomposition into primary relations because x_i occurs twice in both expressions. Execution times (in seconds), that are given in the following table, were obtained on a PC Pentium (Prolog IV) and a Sun SS 10/20 (Newton).

n	Prolog IV (Normal)	Prolog IV (With complex constraint handling)	Newton
5	2.9	4.4	1.2
10	10.9	13.4	8.8
20	279.8	29.9	25.9
40	> 10,688	66.6	61.6
80	-	134.4	127.7
160	-	274.7	264.6

Execution times of Prolog IV (with complex constraint handling) and Newton are similar, and seem to be linear, contrarily to the usual technique which is clearly exponential.

The methodology we have briefly presented here is useful to tackle problems like Broyden-Banded equation systems. Solving this kind of problems, the efficiency is comparable to more sophisticated systems like Newton. Considering the fact that our system only relies on a global treatment of complex expressions, done in a very straightforward manner, we can conclude that this mechanism, and only this mechanism, is fundamental solving such problems. This methodology can also be seen in another way : it can easily be used to design prototypes that gives the ability to test the behaviour of global constraints without any implementation effort.

References

1. F. Benhamou, Touraïvane. *Prolog IV : langage et algorithmes.* JFPL, 1995.
2. F. Benhamou, D. McAllester, P. Van Hentenryck. *CLP(Intervals) revisited.* ILPS, 1994.
3. J.G. Cleary. Logical arithmetic. *Future generation computing systems.* 1987.
4. E.R. Hansen, R.I. Greenberg. *An interval Newton method.* Appl. Math. Comput, 1983.
5. E.R. Hansen, S. Sengupta. *Bounding solutions of systems of equations using interval analysis.* BIT, 1981.
6. A.K. Mackworth. *Consistency in Networks of Relations.* Artificial Intelligence, Vol 8, No 1, 1977.
7. R.E. Moore. *Interval analysis.* Prentice-Hall, Englewood Cliffs, 1966.
8. W. Older, F. Benhamou. *Programming in CLP(BNR).* PPCP, 1993.
9. W. Older, A.J. Velino. *Constraint arithmetic on real interval in CLP.* Selected research. F. Benhamou and A. Colmerauer, eds MIT Press, 1993.

To Guess or to Think? Hybrid Algorithms for SAT (Extended Abstract) *

Irina Rish and **Rina Dechter**

Information and Computer Science
University of California, Irvine, CA 92717, U.S.A.
{*irinar,dechter*}*@ics.uci.edu* *http://www.ics.uci.edu/{~irinar,~dechter}*

Complete algorithms for solving propositional satisfiability fall into two main classes: backtracking search (e.g., the Davis-Putnam Procedure [1]) and resolution (e.g., the original Davis-Putnam Algorithm [2] and Directional Resolution [4]). Backtracking may be viewed as a systematic "guessing" of variable assignments, while resolution is inferring, or "thinking". Experimental results show that "pure guessing" or "pure thinking" might be inefficient. We propose an approach that combines both techniques and yields a family of hybrid algorithms, parameterized by a bound on the "effective" amount of resolution allowed. The idea is to divide the set of propositional variables into two classes: *conditioning* variables, which are assigned truth values, and *resolution* variables, which are resolved upon. We report on preliminary experimental results demonstrating that on certain classes of problems hybrid algorithms are more efficient than either of their components in isolation.

The well-known Davis-Putnam Procedure (DP) is a backtracking algorithm enhanced by unit resolution at each level of the search. Directional Resolution (DR)[4] is a variable-elimination algorithm similar to adaptive-consistency for constraint satisfaction. Its worst-case time and space complexity is exponential in *induced width*, w^*, of the interaction graph of a propositional theory. The time complexity of DP is worst-case exponential in the number of variables, while its space complexity is linear. However, on average DP is relatively efficient, while DR's average complexity is close to its worst-case. Consequently, DR is significantly less efficient than DP when applied to uniformly generated 3-cnfs having large w^*, while outperforming DP by many orders of magnitude when applied to theories with bounded w^* [4]. This time- and space-wise complementary behavior of the two algorithms prompted the idea of combining DP and DR.

We propose a family of hybrid algorithms, called *Dynamic Conditioning + DR (DCDR)*, parameterized by a bound, b, that controls the balance between resolution and backtracking. Given b, the algorithm DCDR(b) selects a subset of conditioning variables, or *cutset*, C_b, such that w^* of the resulting (conditional) theory does not exceed b. The hybrid algorithm searches the space of truth assignments for the conditioning variables and resolves upon the rest of the

* This work was partially supported by NSF grant IRI-9157636, by the Electrical Power Research Institute (EPRI) grant RP8014-06, and by Rockwell MICRO grants UCM-20775 and 95-043.

variables. Dividing the set of variables into the cutset and resolution variables is accomplished during run time, i.e. dynamically. We have also experimented with a static version of the algorithm (for details see the full paper available through *http://www.ics.uci.edu/~irinar*). We show that the time complexity of both algorithms is $O(exp(c+b))$, where c is the largest cutset size encountered during run time.

We tested DCDR(b) on *uniform k-cnfs* and on structured problems having bounded w^*, such as (k,m)-trees. A *(k,m)-tree* is a tree of cliques, each having $k+m$ nodes, where k is the size of intersection between each two neighboring cliques. We observed three different behavior patterns depending on $w*$ (see Figure 1): 1. on problems having large w^*, such as uniform 3-cnfs around the 50% solvable crossover point (the transition region from satisfiable to unsatisfiable problems), the time complexity of DCDR(b) is similar to DP when b is small (obviously, a bound $b=-1$ does not allow any resolution, making DP equivalent to DCDR(-1)), however, when b increases, the CPU time for DCDR(b) grows exponentially; 2. theories having very small w^* (such as (k,m)-trees with $k \leq 4, m \leq 6$) are easier for DCDR(b) with a large b, since DCDR(b) coincides with DR for $b \geq w^*$; 3. on (k,m)-trees with larger clique size, we observed an intermediate region of b's values yielding a faster algorithm than both DP and DR. The averages for uniform 3-cnfs are computed on 100 problem instances, while for (k,m)-trees we ran only 25 experiments per point. We therefore view our results as preliminary. However, they indicate the general promise of the approach.

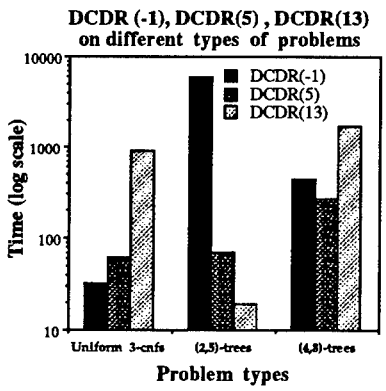


Figure 1

We see that w^* provides a reasonable predictor of b. When $w*$ is very large, we choose $b \leq 1$; when w^* is very small (less than 4), we choose large b; for intermediate levels of $w*$ it is better to choose a bounded level of b. The algorithms having b in the range of 5 to 8 seem promising, since they behave similarly to DP on uniform instances, to DR for small w^*, while for intermediate values of w* they exploit the benefits of both DP and DR. The hybrid algorithms trade space for time [3], and output a compiled theory from which a portion of the solution set rather than one solution can be generated in linear time.

References

1. M. Davis, G. Logemann and D. Loveland, A machine program for theorem proving, *Communications of the ACM*, 5, 1962, pp. 394-397.
2. M. Davis and H. Putnam, A computing procedure for quantification theory, *Journal of the ACM*, 7, 1960, pp. 201-215.
3. R. Dechter, Topological parameters for time-space tradeoff, In *Proceedings of UAI-96*, 1996.
4. R. Dechter and I. Rish, Directional Resolution: The Davis-Putnam Procedure, Revisited. In *Proceedings of KR-94*, pp. 134-145, 1994.

A Local Simplification Scheme for cc Programs

Vincent Schächter

Ecole Normale Supérieure
45, rue d'Ulm
75230 Paris Cedex 05
e-mail : schachte@dmi.ens.fr

1 Introduction

We introduce a method for adding a limited form of *local control* for cc programs in a potentially distributed context.

In settings where large number of agents interact through (possibly many distinct) shared stores, it is desirable to have a local simplification mechanism whereby agents whose "modifying power" is exhausted disappear. "Local" means here that simplification in a given store should not be dependant on knowledge of information located in remote stores, nor on knowledge of the whole multiset of agents active in the store. Rather, we envision the simplification mechanism as a kind of reaction rule, in the spirit of Berry and Boudol's ChAM "chemical" rules[1] : a given agent reacts to its perceived, local environment by reducing to `true`. The simplification is thus parametered by the scope of the perceived environment, i.e. the degree of 'locality'. In order for a simplification scheme to be as "local" as possible, the information that is used for control has to be carried by the agents, i.e. integrated at the language level.

2 A Language Extension : cc^{i^+,i^-}

We define a strict extension of the determinate cc languages by marking agents or multisets of agents with tags. The basic idea is the following : an agent may be marked by a pair (i^+, i^-) of tags, interpreted as upper bound/lower bound on the information the agent will eventually add to the store in the course of a complete computation. The upper bound is used to eliminate the agent when all the information it may add to a store is already entailed by it, while the lower bounds are used to guarantee that a group of agents will eventually bring as much information as a given agent and to eliminate the latter.
An adequate modification of the deterministic cc operational semantics ensures transmission of tags and simplification.
More formally, the *potential information* $I^+_{P,c}(A)$ of an agent A in context (program) P is defined as the upper bound along all computational paths, starting in c and ending in the terminal store, of (the entailment closure of) the set of constraints produced by A. The *minimal information* $I^-_{P,c}(A)$ is the similarly defined lower bound.
These quantities are usually hard to determine exactly : tag i^+ (resp. i^-) is used

to keep track of approximate information on I^+ (resp. I^-). This information enables dynamic simplification of two types of agents :

1. *quiescent* agents, whose potential information is entailed by the current store
2. *redundant* agents, whose potential information is entailed by the minimal information of a group of other agents of the program

Tagging a cc program amounts to selecting a certain subset of (hopefully truncated) computation paths of the initial program, and therefore trivially preserves correctness.
We distinguish two complementary approaches to program tagging : programs may be "manually" tagged by a programmer using expert knowledge of the specific application, or one may rely on a procedure that partially "automathizes" the tagging of a program and preserves completeness.

3 Static Analysis

To affix positive tags to a program P, we make a nonstandard use of the classical adjoint framework of abstract interpretation [2]. We compute upper approximations of the potential information of agents on an abstract constraint system and use these approximations to positively tag the agents in a *sound* manner, i.e. in such a way that the tagged program terminates in the same store as the initial program. We reason on a 'reverse' abstract interpretation of P, i.e. P is syntactically transformed into a program P' computing on an abstract constraint system $\mathcal{A}$ that returns constraints *stronger* than the ones returned by the concrete computation. P' is then executed and each agent in P is tagged by the information produced by its abstract counterpart during the abstract computation.

4 Future Work

The very nature of this simplification scheme makes it suitable for use in a distributed framework. We are developing a distributed cc execution model that would help assess local 'mobile' control techniques of the type introduced here as opposed to central control strategies (or central control strategies local to a given processor), and in general local control techniques depending on the 'degree of locality'.
We are also designing more involved abstract interpretation techniques that address the issue of *negative* tagging.

References

1. G. Berry and G. Boudol.: The chemical abstract machine. *Theoretical Computer Science*, 96 (217-248), 1992.
2. Roberto Giacobazzi, Giorgio Levi, and Enea Zaffanella.: Abstracting synchronization in concurrent constraint programming. In *Proc. of PLILP'94*, 1994.
3. Vincent Schachter.: cc and Task Intervals.. Technical report, LIENS, 1995.

From Evaluating Upper Bounds of the Complexity of Solving CSPs to Finding All the Solutions of CSPs

Gadi Solotorevsky

Dept. of Mathematics and Computer Science
Ben-Gurion University of the Negev
Beer-Sheva, 84-105, Israel
Email: gadi@cs.bgu.ac.il

Constraints Satisfaction Problems (CSPs) belong to the family of NP complete problems. The complexity of finding one solution for a CSP is lower or equal to the complexity of finding all the solutions of a CSP, this papers focus on the latter task. Lower upper complexity bounds of solving CSPs were found for CSPs that are represented as graphs of constraints with special properties [1, 3]. This paper presents a new method, for finding upper bounds on the complexity of solving CSPs. In many cases this method achieves better bounds than the other methods. An algorithm for finding all solutions of a CSP, and based on this method, is presented. Algorithms previously developed for finding all the solutions of CSPs ([2], [4]) do not manage well loose CSPs. We compare DAP to them, and show that DAP outperforms them in many cases, moreover it is the only algorithm that has a good behavior (both in time and space requirements) for loose CSPs.

The main idea is based on problem partitioning. Given a CSP divided into m groups, $G_1, G_2, ..., G_m$, with n_i nodes, e_i internal constraints, v_i largest domain size of group G_i, and c_i the number of nodes in G_i connected by external constraints to other groups, we will find all the solutions of the CSP. Assume that S_i is the group of all the solutions of G_i. Note that we won't require explicit representation of all the solutions (because the representation itself may be of exponential size), we will be content with a solution in which each S_i is divided into K_i groups and in which for any tuples $(k_1, .., k_n | k_i \in K_i)$ it is written either if all the of solutions members in those groups are solutions for the CSP or none of them is.

At the first stage, find all the internal solutions for each component. The complexity of this process is $O(\sum_{i=1}^{m} e_i v_i^{n_i})$. Then divide the solutions for each group into groups according to the values in the nodes that participate in external constraints; the complexity of this stage is of $O(\sum_{i=1}^{m} c_i v_i^{n_i})$.

After the internal solutions are found and divided into groups according to the values in **all** the nodes that participate in external constraints, proceed to find the groups of solutions that are competent. This will be done in the following way: start with an empty group of groups of partial solutions, combine at each step the groups of solutions of another G_i with the partial solutions found so far. The combination of the groups of solutions in group G_i with the groups of partial solution is done by testing for each group in G_i what groups of partial

solutions are legal. In the next stage the group of partial solutions will include all the legal combinations between the groups in G_i and the previous groups of partial solutions. After all the m groups were combined with the group of partial solution, they will include all the groups of solutions. The total complexity of combination process is $O(\prod_{i=1}^{m} c_i v_i^{c_i})$. Thus the total complexity is $O(\prod_{i=1}^{m} c_i v_i^{c_i} + \sum_{i=1}^{m}(c_i + e_i)v_i^{n_i})$ (1).

We will now consider the problem of finding all the solutions of a CSP. Finding all the solutions is of importance when trying to find an optimal solution for a CSP, then all the solutions can be generated and tested one by one[1]. Traditionally two approaches have been used for finding all the solutions of a CSP synthesis algorithms, and exhaustibly applying the backtrack based algorithms used for finding one solution until all the search space is traversed (see [5]). As Tsang [5] pointed out both the synthesis methods and the backtrack based methods can be an acceptable selection when the CSP is tight, however when the CSP is loose then none of them is suitable for the task. The DAP algorithm is aimed to handle the loose case. The DAP algorithm is strongly based on the method for evaluating complexity previously presented, basically: First, part the vertices of the CSP using a hill climbing based algorithm into disjoint sets trying to achieve a low value for the formula (1). Afterwards, find all the solutions of the CSP using the partition previously found and applying the same method we used for evaluating the complexity.

The behavior of the DAP algorithm was tested and compared using a set of random CSPs with 10 variables and 6 values. For sparse CSPs, the DAP algorithm outperforms the backtrack based algorithms by an order of 2 to 3 magnitudes. For tight CSPs, the partition in DAP yields just one group, therefore DAP and the backtrack based algorithms perform similarly. Note that an empirical comparison with the synthesis algorithms was impractical due to their huge space requirements. Similarly, increasing the variables or domain size of the CSPs make the backtrack based algorithms completely impractical.

References

1. Dechter, R. and Pearl, J., "Network-Based Heuristics for Constraint-Satisfaction Problems", *Artificial Intelligence*, No. 34, 1988, pp. 1-37.
2. Freuder E. C., "Synthesizing constraint expressions" *Communications ACM*, Vol. 21, No. 11, pp 958-966, Nov. 1978.
3. Freuder E. C, "Complexity of K-Trees Structured Constraint Satisfaction Problems", *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pp. 4-9, 1990.
4. Seidel, R. "A new method for solving constraint satisfaction problems", *Proceedings 7th International Joint Conference on AI*, pp. 338-342, 1981.
5. Tsang E.P.K., Foundations of Constraints Satisfaction, Academic Press, pp. 184-187.

[1] Branch and Bound methods should be preferred whenever the target function allows it

Modeling and Solving Distributed Constraint Satisfaction Problems (DCSPs)

Gadi Solotorevsky, Ehud Gudes and Amnon Meisels

Dept. of Mathematics and Computer Science
Ben-Gurion University of the Negev, Beer-Sheva, Israel
Email: {gadi,ehud,am}@cs.bgu.ac.il

Constraint satisfaction problems (CSP) are part of many real world domains, such as computer vision and scheduling problems. Often, CSPs are solved in real life by several agents, each of them working on a part of the problem [3, 4]. A distributed CSP can be viewed as a set of constraint networks(CN), each CN being solved by a different agent, where the CNs are connected by constraints. A major assumption of the present paper is that checking constraints *inside* the distributed components has a much lower cost than checking constraints *across* different components. The latter check involves some kind of *message passing* that the solving algorithm would like to minimize.

The processing of CNs have been studied extensively in the last decade [1, 2], usually within the standard model which is sequential. Several attempts have been made at studying the processing of CNs in parallel The most relevant study of distributed CSPs has been made by Yokoo [5]. The basic difference between our approach and Yokoo's approach is that our algorithms try to take advantage of the differences between the DCSPs components.

The model of a DCSP of the present paper uses agents that are connected by a communication network (i.e., no common memory, just message passing). The number of agents is equal or larger by a small constant, to the number of subproblems in the given division of the DCSP. Based on this we state the following goals for our multi-agent algorithms:

- Try to optimize the performance of the slowest agent, rather than optimizing each individual agent.
- Minimize the amount of backtracking each agent performs as a result of actions of other agents.

In general, a DCSP may be represented in two ways. The *Explicit* representation is the original one, where variables in one component may be connected by a constrains to any other variable in the same or in different component. In the *Canonical* representation, a new, central component is added. This component contains copies of all variables which are connected by inter-component constraints, such that solving the CSP of this central component guarantees that all global constraints are satisfied. The equivalence of the two representations can be shown easily.

Four algorithms for solving DCSPs, that are sound and complete are proposed. Two of these algorithms are sequential and two algorithms operate in parallel and are inherently distributed. They can be summarized as follows:

1. *Algorithm 1.* Look at the DCSP as a regular CSP and solve it by one of the commonly known techniques disregarding the distribution to components. Note that at most one agent is working at any time.
2. *Algorithm 2.* Same as 1, except that agents first work on their own sub-problem.
3. *Algorithm 3.* The first agent find first a solution to the central component. It then broadcasts this solution to all other agents. These agents search for solutions to their corresponding sub-problems in parallel. If all of them find a consistent solution to their sub-problems we are done, else the central agent must backtrack and broadcasts a new solution to the central component.
4. *Algorithm 4.* Here, the peripheral agents search for solutions in parallel, and send their solution to the central agent. If the central agent can find a consistent solution we are done, otherwise, the first agent that caused the failure is asked to backtrack, and send its new solution back to the central agent. The backtracking is done sequentially to assure completeness.

Algorithms 3 and 4 were designed for two opposite cases of DCSPs: a dominant central component seems natural for algorithm 4, while dominant peripheral components calls for algorithm 3.

The behavior of the proposed algorithms was tested by generating and solving a set of random DCSPs. The main parameters which were changed in the experiment were the number and tightness of internal and external constraints. The two main measures that were measured were the number of constraint checks performed, and the number of messages passed. The latter measure is particularly important in a distributed environment. The results show Algorithm 2 to be the worst. Algorithms 3 and 4 are much better when there aregreat differences between the tightness of local vs. external constraints, while Algorithm 1 is good only when this tightness is equal for all constraints (its really does not pay to decompose the problem!). As expected, Algorithm 3 is much better when the central component is tighter, while Algorithm 4 is better when the peripheral components are tighter. In the future we plan to use these results in solving real-life distributed resource allocation problems such as in [4].

References

1. R. Dechter and J. Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.
2. P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.
3. P. Prosser, C. Conway, and M. Muller. A constraint maintenance system for the distributed resource allocation problem. *Intelligent Systems Engineering*, pages 76–83, 1992.
4. K. Sycara, F. Roth, N. Sadeh, and M. Fox. Resource allocation in distributed factory scheduling. *IEEE Expert*, pages 29–40, 1991.
5. M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proc. 1st Intrnat. Conf. on Const. Progr.*, pages 88 – 102, Cassis, France, 1995.

Scheduling an Asynchronously Shared Resource

Douglas R. Smith and Stephen J. Westfold

Kestrel Institute, 3260 Hillview Avenue, Palo Alto, California 94304, USA
{smith,westfold}@kestrel.edu

This note describes one aspect of our exploration of the machine synthesis of scheduling algorithms [1, 2]. The approach involves several stages. The first step is to develop a formal model of the scheduling domain, called a *domain theory.* Second, the constraints, objectives, and preferences of a particular scheduling problem are formally stated within the language of the domain theory as a *problem specification.* Finally, an executable scheduler is produced semi-automatically by applying a sequence of *transformations* to the problem specification. The transformations embody programming knowledge about algorithms, data structures, program optimization techniques, etc. The result of the transformation process is executable code that is correct by construction. Furthermore, the resulting code can be extremely efficient.

In this note we focus on scheduling a class of resources that we call *Asynchronously Shared Resources* (ASRs) (see also [3]). An ASR can be shared simultaneously by many users or tasks whose usage patterns are not necessarily synchronized. The resource is assumed to have finite capacity and the tasks are assumed to use the resource for finite periods. A typical example of an ASR is an automobile parking lot having n parking slots. Users can come and go independently, but a scheduler should never assign more than n users to the parking lot at the same time. More generally any pool of individual resources can be treated as an ASR: ramp space at airports, machining tools in manufacturing, computer processors running in parallel, fleets of transportation vehicles, personnel in a skill pool, etc. Power sources (e.g. generators, batteries) provide nondiscrete examples of ASRs. ASR scheduling can be seen as a special case of multi-capacitated job-shop scheduling where there is one multi-capacity machine.

The motivation for this work was to ease the burden of creating domain theories, by building a library of abstract theories for various classes of resources. Included in each theory would be various axioms, lemmas, and theorems that facilitate the inference (at design time) of constraints for efficient propagation.

Suppose we are given a set of tasks where each task *tsk* has an earliest start time, latest start time, duration *dur(tsk)*, and demand *demand(tsk)*. The problem of scheduling an ASR can be defined as follows: given a set T of tasks and an ASR with capacity c, find an assignment of start times *st(tsk)* to each task that satisfies the ASR capacity constraint: at no time does the demand on the ASR exceed its capacity; formally $\forall(t : time)\ \ demand(T,\ t) \leq c$ where

$$demand(T,\ t) = \sum_{\substack{tsk \in T \\ st(tsk) \leq t < st(tsk)+dur(tsk)}} demand(tsk)$$

computes the aggregate or net demand of the tasks in T at time t.

The objective of this study was to work out how to synthesize a variety of algorithms for ASR scheduling. Our intent was more to represent the design knowledge necessary for deriving the best possible scheduling algorithms, rather than to derive new algorithms.

Two classes of algorithms were derived which we call *discrete* and *aggregate*. A *discrete* algorithm assumes that the capacity bound on the ASR is integral and that each task consumes one unit of capacity. The data structures and constraint propagation techniques are well-known from the literature on unit-capacity machine scheduling.

In *aggregate* algorithms we treat the scheduling of an ASR as a whole, assuming no internal structure to the capacity of an ASR. The key idea here is to maintain (via finite differencing [1]) data structures that represent lower and upper bounds on $demand(T,\ t)$ for all t, called respectively the definite and possible demand maps. The main disjunctive constraint that we derived states that whenever a block of tasks has *definitely* reserved the ASR at some time (i.e. no other task could feasibly be executed at that time), then any other task must either precede or succeed some task in the block. There are various strategies for choosing how to apply this constraint. The possible demand map is used to drive branching at potentially oversubscribed times.

Our overall experience with these algorithms is that they will either find a feasible schedule quickly or else take a very long time to complete. Finding a schedule quickly means that little or no backing up occurs during search – mainly descendants and siblings are ever explored. The aggregate algorithms can be tuned either to find better solutions or else to solve harder problems, although they also tend to be somewhat slower than the discrete algorithms due to the expense of maintaining the demand maps and the extra complexity of the deciding how to branch.

This work is intended to bring the goal of machine support for synthesizing customized high-performance scheduling algorithms one step closer to practical realization.

References

1. SMITH, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (September 1990), 1024–1043.
2. SMITH, D. R., PARRA, E. A., AND WESTFOLD, S. J. Synthesis of high-performance transportation schedulers. Tech. Rep. KES.U.95.6, Kestrel Institute, March 1995.
3. SMITH, D. R., AND WESTFOLD, S. J. Scheduling an asynchronous shared resource. Tech. rep., Kestrel Institute, February 1996.

The Generalized Railroad Crossing: Its Symbolic Analysis in CLP($\mathcal{R}$)

Luis Urbina

Abstract. The symbolic simulation and analysis method for hybrid systems presented in [2] is illustrated by means of the benchmark problem "The Generalized Railroad Crossing" [1].

1 The Generalized Railroad Crossing (GRC)

The GRC can be described as follows: The system operates a gate at a railroad crossing. The railroad crossing *intersection* lies in a region of interest *region*. A set of trains travel through *region* on multiple tracks in both directions. A sensor system determines when each train enters and exits *region*. We define the time-dependent *gate function* $g(t) \in [0, 90]$, where $g(t) = 0$ means the gate is down and $g(t) = 90$ means the gate is up, and the set $\{[\mu_i, \nu_i]\}$ of occupancy intervals $[\mu_i, \nu_i]$, where each *occupancy interval* $[\mu_i, \nu_i]$ is a time interval during which one or more trains are in *intersection*. μ_i is the time of the i-th entry of a train into the crossing when no other train is in the crossing and ν_i is the first time since μ_i that no train is in the crossing. Given two constants $\gamma_1 > 0$ and $\gamma_2 > 0$, develop a system to operate the gate in such way that it satisfies the following two properties: Safety Property: 'The gate is down during all occupancy intervals.' $\forall\, t \in \bigcup_i [\mu_i, \nu_i] \Rightarrow g(t) = 0$. Obviously, this property can be easily verified by blocking the crossing. Thus, a stronger property is given, which does not permit the realization of simple designs of the problem. Utility Property: 'The gate is up when no train is in the crossing.' $\forall\, t \notin \bigcup_i [\mu_i - \gamma_1, \nu_i + \gamma_2] \Rightarrow g(t) = 90$. This further requirement ensures that, at γ_2 time after the start of a non-occupancy interval, the gate must be open, $g(t) = 90$, and remains open until γ_1 time before the end of that interval, i.e., γ_1 time before the beginning of the next occupancy interval. The GRC is modeled as the product of $m + 1$ linear hybrid systems, i.e., $GRC = RAIL_1 \times \ldots \times RAIL_m \times GATE$ $(m \geq 1)$. Each of them is firstly modeled by a linear hybrid automaton and then transformed into a CLP($\mathcal{R}$) program. The product is a CLP($\mathcal{R}$) program as well.

Rail. The rail component RAIL_i is shown in Figure 1 a). c_i is the clock for RAIL_i. k is a counter variable for the number of trains being inside the railroad crossing *intersection*. $enter_i$ and $exit_i$ are the sensor signals when a train enters *region* and when it leaves *intersection*, respectively. σ_i is the minimal time that must occur between the exit of a train from *intersection* and the entry of the next train into *region*. ς_{i1} (ς_{i2}) is the lower (upper) bound of the time that a train takes from *region* to *intersection*. τ_{i1} (τ_{i2}) is the lower (upper) bound of the time that a train takes to leave *intersection*. Some restrictions: $\sigma_i \geq 0$, $0 < \varsigma_{i1} \leq \varsigma_{i2}$ and $0 < \tau_{i1} \leq \tau_{i2}$.

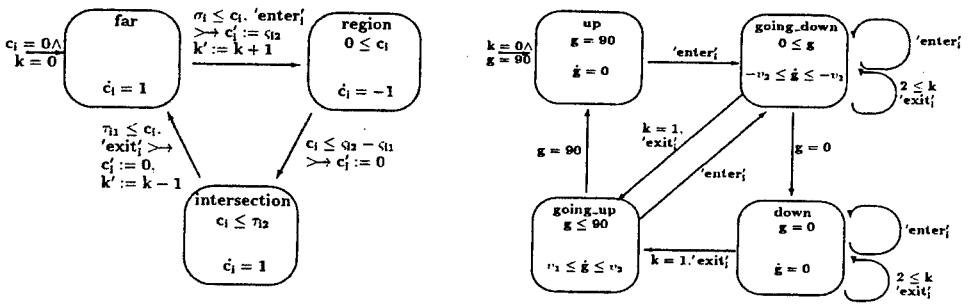


Fig. 1. a) RAIL$_i$ and b) GATE

Gate. The gate component GATE is shown in Figure 1 b). g is the gate function. v_1 (v_2) is the lower (upper) bound of the rate to lower the gate completely.

We have revised both requirements. The locations far^i, $region^i$ and $intersection^i$ are the corresponding locations for RAIL$_i$. Safety Property: 'Whenever the train is in the railroad intersection the gate is down.' $\neg(intersection^1 \wedge \cdots \wedge intersection^m \wedge \neg down)$. The GRC satisfies the safety requirement provided it never reaches a state where a train on RAIL$_i$ is in $intersection^i$ and the GATE is not in *down*. We introduce an extra clock x_i which serves to count the delay time since a train on RAIL$_i$ has left *intersection*. Thus, in the modified hybrid system RAIL'$_i$ x_i is reset in the transition from *intersection* to *far*. Utility Property: 'Whenever no train is in the railroad intersection the gate is up.' $\neg[\bigwedge_{i=1,\ldots,m}[(going\_down \wedge far^i) \vee (down \wedge far^i) \vee (going\_up \wedge far^i) \vee (going\_down \wedge region^i) \vee (down \wedge region^i) \vee (going\_up \wedge region^i)] \wedge (c_1 \le \varsigma_{12} - \gamma_1 \wedge \cdots \wedge c_m \le \varsigma_{m2} - \gamma_1 \wedge x_1 \ge \gamma_2 \wedge \cdots \wedge x_m \ge \gamma_2)]$.

2 Symbolic Analysis

Example 1 The GRC with one rail. We set $\sigma_1 = 8, \varsigma_{11} = 4, \varsigma_{12} = 6, \tau_{11} = 4, \tau_{12} = 6, v_1 = 1$ and $v_2 = 2$. The GRC violates the safety property.

Example 2 The GRC with two rails. We set $\sigma_i = 8, \varsigma_{i1} = \tau_{i1} = 4, \varsigma_{i2} = \tau_{i2} = 6, v_1 = 30$, and $v_2 = 40$. The correctness of the safety property can be seen by symbolic simulation. It can also be proved by bottom–up evaluation of the CLP($\mathcal{R}$) program. This implies to start the bottom–up evaluation with the set of facts: $A = \{(\_, \_, going\_up) \leftarrow x \ge 10., (\_, \_, up) \leftarrow x \ge 10., (\_, \_, going\_down) \leftarrow x \ge 10.\}$. The GRC meets the safety property iff the *initial goal* $(far, far, up) \wedge c_1 = 0 \wedge c_2 = 0 \wedge k = 0 \wedge g = 90 \wedge time = 0$ (corresponding to the initial fact) fails in the fixpoint B of the execution $A \rightsquigarrow \cdots \rightsquigarrow B$. Verification of the utility property for $\gamma_1 = 5$ and $\gamma_2 = 4$ for the modified GRC is made as for the correctness proof of the safety

property. A train takes from 4 to 6 time units to reach *intersection* and the gate takes up to 3 time units to lower. Note that $\gamma_1 \geq 6-4+3=5$ because the moment a train enters *region* is the moment the gate begins to lower. Since the gate takes up to 3 time units for going down after the last train has left *intersection*, $\gamma_2 > 3$. This is proved by bottom–up evaluation starting with the set of final facts generated by the utility property (for instance, $(far, \_, going\_down) \leftarrow c_1 \leq 1 \wedge c_2 \leq 1 \wedge x_1 \geq 4 \wedge x_2 \geq 4$ is a subset of such facts) and checking that the initial goal $(far, far, up) \wedge c_1 = 0 \wedge c_2 = 0 \wedge k = 0 \wedge g = 90 \wedge time = 0$. fails for the fixpoint of the bottom–up evaluation. Fortunately, the executions above terminate.

References

1. C. Heitmeyer, R. Jeffords, and B. Labaw. A Benchmark for Comparing Different Approaches for Specifying and Verifying Real–Time Systems. In *Proc. of the 10th Int. Workshop on Real–Time Operating Systems and Software*, May 1993.
2. L. Urbina. *Analysis of Hybrid Systems in CLP(R).* This volume.

A Stochastic Approach to Solving Fuzzy Constraint Satisfaction Problems

Jason H. Y. Wong, Ka-fai Ng, Ho-fung Leung

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, New Territories
Hong Kong
E-mails: {hywong, kfng, lhf}@cs.cuhk.edu.hk

Traditionally, constraint satisfaction problems (CSP's) [1] are so defined that "*all the constraints are satisfied simultaneously.*" However, this is not always true. Many CSP's in real-life are "*soft* CSP's," *i.e.*, an assignment of values to the variables is considered to be a solution even if some constraints are violated. Some of the practical CSP's are fuzzy: they are fully satisfied by some value assignments to the variables in the constraint, and they are considered to be "partially" or "less" satisfied, instead of "violated," by some other assignments. Sometimes a real-life CSP may consist of a mixture of hard constraints and soft constraints. In these cases we are required to find assignments that fully satisfy the hard constraints and fully or partially satisfy the soft constraints.

A constraint satisfaction problem is defined as a tuple (Z, D, C^c). Z is a finite set of variables and D is a finite set of domains one associated with each variable in Z. C^c is a set of constraints. Each constraint is a crisp relation among the domains of a subset of the variables in Z. Each constraint restricts the combination of values that these variables can take. The goal of a CSP is to find a consistent assignment of values to the variables in Z that satisfies all the constraints in C^c. A *fuzzy constraint satisfaction problem* (FCSP) is defined as a tuple (Z, D, C^f). C^f is a set of *fuzzy constraints.* Each fuzzy constraint is a fuzzy relation among the domains of a subset of the variables in Z. *Satisfaction index* of a fuzzy constraint tells us to what extent a constraint is satisfied. *Solution index* of an FCSP (Z, D, C^f) shows its overall satisfaction. It is based on the satisfaction indexes of all the constraints in C^f and obtained by a user-defined function called *satisfaction function. Threshold* is a user-defined lower bound of the acceptable solution index of an FCSP. The goal of an FCSP (Z, D, C^f) is to find an assignment of values to all variables in Z so that the solution index is not less than the threshold. The difference between FCSP and CSP lies on the set of constraints they involve. For a CSP (Z, D, C^c), the constraints in C^c are Boolean. An assignment of a tuple to the variables in C^c return true (1) or false (0). For an FCSP (Z, D, C^f), the constraints in C^f has a range of return values from 0 to 1. Obviously, CSP is a restricted instance of FCSP. The domain of problems that CSP can model is a subset of the problems that FCSP can model.

A generic neural network model called GENET has been proposed by Tsang and Wang [2] for solving CSP's with binary constraints. GENET solves CSP's by iterative improvement and incorporates a learning strategy to escape local

minima. Lee, Leung and Won [3] later propose E-GENET, an extended GENET to solve non-binary CSP's. We have also developed a model called fuzzy GENET based on GENET for solving binary FCSP. In this paper, we propose fuzzy E-GENET by merging the idea of E-GENET and fuzzy GENET. Fuzzy E-GENET allows the representation of nonbinary FCSP's.

We have built a fuzzy E-GENET simulator. Benchmarking results show that fuzzy E-GENET is as efficient as GENET in solving non-fuzzy problems, both binary and non-binary. We have also use the $N \times (N-1)$-queens problem, a fuzzy binary problem, to test our simulator. In this problem, N queens are placed on an $N \times (N-1)$ chessboard so that there exists at least one pair of queens attacking each other. We define that it is better for two queens attacking each other to be separated by a greater vertical distance. The result of fuzzy E-GENET running on a SPARCstation 10 on the $N \times (N-1)$-queens problem are shown in table 1.

Table 1. Results on $N \times (N-1)$ queens problem

threshold	0.9	0.8	0.7	0.6	0.5
20×19 queens	0.40s	0.13s	0.10s	0.08s	0.05s
30×29 queens	2.63s	0.88s	0.28s	0.22s	0.17s
40×39 queens	5.03s	0.88s	0.33s	0.32s	0.32s
50×49 queens	6.22s	1.70s	0.92s	0.77s	0.65s
60×59 queens	9.28s	3.70s	1.85s	1.33s	1.32s
70×69 queens	16.91s	4.54s	2.49s	2.10s	2.03s

Acknowledgement

This research is partially supported by the Croucher Foundation Research Grant CF 94/21.

References

1. Mackworth, A. K.: Consistency in Network of Relations. *Artificial Intelligence* 8 (1977) 99–118
2. Tsang, E. P. K., Wang, C. J.: A Generic Neural Network Approach For Constraint Satisfaction Problem. *Neural Network Applications* (1992) 12–22
3. Lee, J. H. M., Leung, H. F., Won, H. W.: Extending GENET for Non-Binary CSP's. In *Proceedings of the Seventh IEEE International Conference on Tools with Artificial Intelligence*, Wahington DC, November 1995. USA. IEEE Computer Society Press.

Branch-and-Price for Solving Integer Programs with a Huge Number of Variables: Methods and Applications

George L. Nemhauser

Georgia Institute of Technology

Abstract

Many interesting discrete optimization problems, including airline crew scheduling, vehicle routing and cutting stock, have "good" integer programming formulations that require a huge number of variables. Good means that the linear programming relaxations give tight bounds which implies small search trees. We will discuss classes of problems for which this type of formulation is desirable, and special (non-standard) methodology that is needed to solve these integer programs.

Constraint Databases

Dina Q. Goldin
Brown University

Abstract

Paris Kanellakis's pioneering paper in 1990 provided a framework for constraint databases by combining concepts from constraint logic programming and relational databases. The principal idea is to generalize a tuple (or record) data type to a conjunction of constraints from an appropriate language; for example, order constraints or linear arithmetic constraints. Such a tuple can be seen as representing a large, possibly even infinite, set of points in a compact way (e.g., for spatial databases and GIS). Constraint databases have since become a very active area of database research.

After a brief introduction to relational databases, we explain the semantics of constraint database relations and queries, providing complexity results for several specific constraint classes. We consider the various relational querying paradigms (declarative, procedural, and logic programming) and their reinterpretation in the presence of constraints as first-class data. We highlight the basic design principles for constraint databases, such as query closure and safety, efficiency of data representation and data access, and query optimization.

We discuss Paris Kanellakis's more recent work, including work on indexing, and constraint query algebras, and survey other developments in the area (aggregation, complex objects, expressive power). The ultimate goal of Paris Kanellakis's research was to enable commercial-quality implementations of constraint databases. We look at some possible applications for constraint databases, and at the implementational efforts currently under way. We conclude by considering the issues and the challenges that lay ahead.

Complexity-Theoretic Aspects of Programming Language Design

Harry G. Mairson

Brandeis University

Abstract

We survey three of Paris Kanellakis's contributions to the complexity-theoretic analysis of constructs in the design of programming languages. These are (1) his result that first-order unification is complete for polynomial time; (2) his contributions to the complexity analysis of type inference for polymorphically-typed functional programming languages, which was proven to be complete for exponential time; and (3) his work on expressibility of simply typed lambda calculus when used as a functional database programming language. These research investigations are interrelated, emphasizing common themes and difficulties in the design of programming languages, with concrete implications that can be appreciated by language designers. First-order unification is a ubiquitous building block in implementations of sophisticated programming languages. It is, for example, the workhorse of logic programming engines, and the essential component of compile-time type analysis. The research on unification provided a secure foundation for understanding the complexity of automatic polymorphic type inference in functional languages such as ML and Haskell. Modern functional languages are based on the typed lambda calculus, so it is then natural to consider its computational expressiveness. We describe how the degree of higher-order functionality in simply typed terms can be related directly to well known complexity classes.

Author Index

Lecture Notes in Computer Science

For information about Vols. 1–1053

please contact your bookseller or Springer-Verlag

Vol. 1054: A. Ferreira, P. Pardalos (Eds.), Solving Combinatorial Optimization Problems in Parallel. VII, 274 pages. 1996.

Vol. 1055: T. Margaria, B. Steffen (Eds.), Tools and Algorithms for the Construction and Analysis of Systems. Proceedings, 1996. XI, 435 pages. 1996.

Vol. 1056: A. Haddadi, Communication and Cooperation in Agent Systems. XIII, 148 pages. 1996. (Subseries LNAI).

Vol. 1057: P. Apers, M. Bouzeghoub, G. Gardarin (Eds.), Advances in Database Technology — EDBT '96. Proceedings, 1996. XII, 636 pages. 1996.

Vol. 1058: H. R. Nielson (Ed.), Programming Languages and Systems – ESOP '96. Proceedings, 1996. X, 405 pages. 1996.

Vol. 1059: H. Kirchner (Ed.), Trees in Algebra and Programming – CAAP '96. Proceedings, 1996. VIII, 331 pages. 1996.

Vol. 1060: T. Gyimóthy (Ed.), Compiler Construction. Proceedings, 1996. X, 355 pages. 1996.

Vol. 1061: P. Ciancarini, C. Hankin (Eds.), Coordination Languages and Models. Proceedings, 1996. XI, 443 pages. 1996.

Vol. 1062: E. Sanchez, M. Tomassini (Eds.), Towards Evolvable Hardware. IX, 265 pages. 1996.

Vol. 1063: J.-M. Alliot, E. Lutton, E. Ronald, M. Schoenauer, D. Snyers (Eds.), Artificial Evolution. Proceedings, 1995. XIII, 396 pages. 1996.

Vol. 1064: B. Buxton, R. Cipolla (Eds.), Computer Vision – ECCV '96. Volume I. Proceedings, 1996. XXI, 725 pages. 1996.

Vol. 1065: B. Buxton, R. Cipolla (Eds.), Computer Vision – ECCV '96. Volume II. Proceedings, 1996. XXI, 723 pages. 1996.

Vol. 1066: R. Alur, T.A. Henzinger, E.D. Sontag (Eds.), Hybrid Systems III. IX, 618 pages. 1996.

Vol. 1067: H. Liddell, A. Colbrook, B. Hertzberger, P. Sloot (Eds.), High-Performance Computing and Networking. Proceedings, 1996. XXV, 1040 pages. 1996.

Vol. 1068: T. Ito, R.H. Halstead, Jr., C. Queinnec (Eds.), Parallel Symbolic Languages and Systems. Proceedings, 1995. X, 363 pages. 1996.

Vol. 1069: J.W. Perram, J.-P. Müller (Eds.), Distributed Software Agents and Applications. Proceedings, 1994. VIII, 219 pages. 1996. (Subseries LNAI).

Vol. 1070: U. Maurer (Ed.), Advances in Cryptology – EUROCRYPT '96. Proceedings, 1996. XII, 417 pages. 1996.

Vol. 1071: P. Miglioli, U. Moscato, D. Mundici, M. Ornaghi (Eds.), Theorem Proving with Analytic Tableaux and Related Methods. Proceedings, 1996. X, 330 pages. 1996. (Subseries LNAI).

Vol. 1072: R. Kasturi, K. Tombre (Eds.), Graphics Recognition. Proceedings, 1995. X, 308 pages. 1996.

Vol. 1073: J. Cuny, H. Ehrig, G. Engels, G. Rozenberg (Eds.), Graph Grammars and Their Application to Computer Science. Proceedings, 1994. X, 565 pages. 1996.

Vol. 1074: G. Dowek, J. Heering, K. Meinke, B. Möller (Eds.), Higher-Order Algebra, Logic, and Term Rewriting. Proceedings, 1995. VII, 287 pages. 1996.

Vol. 1075: D. Hirschberg, G. Myers (Eds.), Combinatorial Pattern Matching. Proceedings, 1996. VIII, 392 pages. 1996.

Vol. 1076: N. Shadbolt, K. O'Hara, G. Schreiber (Eds.), Advances in Knowledge Acquisition. Proceedings, 1996. XII, 371 pages. 1996. (Subseries LNAI).

Vol. 1077: P. Brusilovsky, P. Kommers, N. Streitz (Eds.), Mulimedia, Hypermedia, and Virtual Reality. Proceedings, 1994. IX, 311 pages. 1996.

Vol. 1078: D.A. Lamb (Ed.), Studies of Software Design. Proceedings, 1993. VI, 188 pages. 1996.

Vol. 1079: Z.W. Raś, M. Michalewicz (Eds.), Foundations of Intelligent Systems. Proceedings, 1996. XI, 664 pages. 1996. (Subseries LNAI).

Vol. 1080: P. Constantopoulos, J. Mylopoulos, Y. Vassiliou (Eds.), Advanced Information Systems Engineering. Proceedings, 1996. XI, 582 pages. 1996.

Vol. 1081: G. McCalla (Ed.), Advances in Artificial Intelligence. Proceedings, 1996. XII, 459 pages. 1996. (Subseries LNAI).

Vol. 1082: N.R. Adam, B.K. Bhargava, M. Halem, Y. Yesha (Eds.), Digital Libraries. Proceedings, 1995. Approx. 310 pages. 1996.

Vol. 1083: K. Sparck Jones, J.R. Galliers, Evaluating Natural Language Processing Systems. XV, 228 pages. 1996. (Subseries LNAI).

Vol. 1084: W.H. Cunningham, S.T. McCormick, M. Queyranne (Eds.), Integer Programming and Combinatorial Optimization. Proceedings, 1996. X, 505 pages. 1996.

Vol. 1085: D.M. Gabbay, H.J. Ohlbach (Eds.), Practical Reasoning. Proceedings, 1996. XV, 721 pages. 1996. (Subseries LNAI).

Vol. 1086: C. Frasson, G. Gauthier, A. Lesgold (Eds.), Intelligent Tutoring Systems. Proceedings, 1996. XVII, 688 pages. 1996.

Vol. 1087: C. Zhang, D. Lukose (Eds.), Distributed Artificial Intelligence. Proceedings, 1995. VIII, 232 pages. 1996. (Subseries LNAI).

Vol. 1088: A. Strohmeier (Ed.), Reliable Software Technologies – Ada-Europe '96. Proceedings, 1996. XI, 513 pages. 1996.

Vol. 1089: G. Ramalingam, Bounded Incremental Computation. XI, 190 pages. 1996.

Vol. 1090: J.-Y. Cai, C.K. Wong (Eds.), Computing and Combinatorics. Proceedings, 1996. X, 421 pages. 1996.

Vol. 1091: J. Billington, W. Reisig (Eds.), Application and Theory of Petri Nets 1996. Proceedings, 1996. VIII, 549 pages. 1996.

Vol. 1092: H. Kleine Büning (Ed.), Computer Science Logic. Proceedings, 1995. VIII, 487 pages. 1996.

Vol. 1093: L. Dorst, M. van Lambalgen, F. Voorbraak (Eds.), Reasoning with Uncertainty in Robotics. Proceedings, 1995. VIII, 387 pages. 1996. (Subseries LNAI).

Vol. 1094: R. Morrison, J. Kennedy (Eds.), Advances in Databases. Proceedings, 1996. XI, 234 pages. 1996.

Vol. 1095: W. McCune, R. Padmanabhan, Automated Deduction in Equational Logic and Cubic Curves. X, 231 pages. 1996. (Subseries LNAI).

Vol. 1096: T. Schäl, Workflow Management Systems for Process Organisations. XII, 200 pages. 1996.

Vol. 1097: R. Karlsson, A. Lingas (Eds.), Algorithm Theory – SWAT '96. Proceedings, 1996. IX, 453 pages. 1996.

Vol. 1098: P. Cointe (Ed.), ECOOP '96 – Object-Oriented Programming. Proceedings, 1996. XI, 502 pages. 1996.

Vol. 1099: F. Meyer auf der Heide, B. Monien (Eds.), Automata, Languages and Programming. Proceedings, 1996. XII, 681 pages. 1996.

Vol. 1100: B. Pfitzmann, Digital Signature Schemes. XVI, 396 pages. 1996.

Vol. 1101: M. Wirsing, M. Nivat (Eds.), Algebraic Methodology and Software Technology. Proceedings, 1996. XII, 641 pages. 1996.

Vol. 1102: R. Alur, T.A. Henzinger (Eds.), Computer Aided Verification. Proceedings, 1996. XII, 472 pages. 1996.

Vol. 1103: H. Ganzinger (Ed.), Rewriting Techniques and Applications. Proceedings, 1996. XI, 437 pages. 1996.

Vol. 1104: M.A. McRobbie, J.K. Slaney (Eds.), Automated Deduction – CADE-13. Proceedings, 1996. XV, 764 pages. 1996. (Subseries LNAI).

Vol. 1105: T.I. Ören, G.J. Klir (Eds.), Computer Aided Systems Theory – CAST '94. Proceedings, 1994. IX, 439 pages. 1996.

Vol. 1106: M. Jampel, E. Freuder, M. Maher (Eds.), Over-Constrained Systems. X, 309 pages. 1996.

Vol. 1107: J.-P. Briot, J.-M. Geib, A. Yonezawa (Eds.), Object-Based Parallel and Distributed Computation. Proceedings, 1995. X, 349 pages. 1996.

Vol. 1108: A. Díaz de Ilarraza Sánchez, I. Fernández de Castro (Eds.), Computer Aided Learning and Instruction in Science and Engineering. Proceedings, 1996. XIV, 480 pages. 1996.

Vol. 1109: N. Koblitz (Ed.), Advances in Cryptology – Crypto '96. Proceedings, 1996. XII, 417 pages. 1996.

Vol. 1110: O. Danvy, R. Glück, P. Thiemann (Eds.), Partial Evaluation. Proceedings, 1996. XII, 514 pages. 1996.

Vol. 1111: J.J. Alferes, L. Moniz Pereira, Reasoning with Logic Programming. XXI, 326 pages. 1996. (Subseries LNAI).

Vol. 1112: C. von der Malsburg, W. von Seelen, J.C. Vorbrüggen, B. Sendhoff (Eds.), Artificial Neural Networks – ICANN 96. Proceedings, 1996. XXV, 922 pages. 1996.

Vol. 1113: W. Penczek, A. Szałas (Eds.), Mathematical Foundations of Computer Science 1996. Proceedings, 1996. X, 592 pages. 1996.

Vol. 1114: N. Foo, R. Goebel (Eds.), PRICAI'96: Topics in Artificial Intelligence. Proceedings, 1996. XXI, 658 pages. 1996. (Subseries LNAI).

Vol. 1115: P.W. Eklund, G. Ellis, G. Mann (Eds.), Conceptual Structures: Knowledge Representation as Interlingua. Proceedings, 1996. XIII, 321 pages. 1996. (Subseries LNAI).

Vol. 1116: J. Hall (Ed.), Management of Telecommunication Systems and Services. XXI, 229 pages. 1996.

Vol. 1117: A. Ferreira, J. Rolim, Y. Saad, T. Yang (Eds.), Parallel Algorithms for Irregularly Structured Problems. Proceedings, 1996. IX, 358 pages. 1996.

Vol. 1118: E.C. Freuder (Ed.), Principles and Practice of Constraint Programming — CP 96. Proceedings, 1996. XIX, 574 pages. 1996.

Vol. 1119: U. Montanari, V. Sassone (Eds.), CONCUR '96: Concurrency Theory. Proceedings, 1996. XII, 751 pages. 1996.

Vol. 1120: M. Deza. R. Euler, I. Manoussakis (Eds.), Combinatorics and Computer Science. Proceedings, 1995. IX, 415 pages. 1996.

Vol. 1121: P. Perner, P. Wang, A. Rosenfeld (Eds.), Advances in Structural and Syntactical Pattern Recognition. Proceedings, 1996. X, 393 pages. 1996.

Vol. 1122: H. Cohen (Ed.), Algorithmic Number Theory. Proceedings, 1996. IX, 405 pages. 1996.

Vol. 1123: L. Bougé, P. Fraigniaud, A. Mignotte, Y. Robert (Eds.), Euro-Par'96. Parallel Processing. Proceedings, 1996, Vol. I. XXXIII, 842 pages. 1996.

Vol. 1124: L. Bougé, P. Fraigniaud, A. Mignotte, Y. Robert (Eds.), Euro-Par'96. Parallel Processing. Proceedings, 1996, Vol. II. XXXIII, 926 pages. 1996.

Vol. 1125: J. von Wright, J. Grundy, J. Harrison (Eds.), Theorem Proving in Higher Order Logics. Proceedings, 1996. VIII, 447 pages. 1996.

Vol. 1126: J.J. Alferes, L. Moniz Pereira, E. Orlowska (Eds.), Logics in Artificial Intelligence. Proceedings, 1996. IX, 417 pages. 1996. (Subseries LNAI).

Vol. 1129: J. Launchbury, E. Meijer, T. Sheard (Eds.), Advanced Functional Programming. Proceedings, 1996. VII, 238 pages. 1996.

Lecture Notes in Computer Science

This series reports new developments in computer science research and teaching, quickly, informally, and at a high level. The timeliness of a manuscript is more important than its form, which may be unfinished or tentative. The type of material considered for publication includes

- drafts of original papers or monographs,
- technical reports of high quality and broad interest,
- advanced-level lectures,
- reports of meetings, provided they are of exceptional interest and focused on a single topic.

Publication of Lecture Notes is intended as a service to the computer science community in that the publisher Springer-Verlag offers global distribution of documents which would otherwise have a restricted readership. Once published and copyrighted they can be cited in the scientific literature.

Manuscripts

Lecture Notes are printed by photo-offset from the master copy delivered in camera-ready form. Manuscripts should be no less than 100 and preferably no more than 500 pages of text. Authors of monographs and editors of proceedings volumes receive 50 free copies of their book. Manuscripts should be printed with a laser or other high-resolution printer onto white paper of reasonable quality. To ensure that the final photo-reduced pages are easily readable, please use one of the following formats:

Font size (points)	Printing area (cm)	(inches)	Final size (%)
10	12.2 x 19.3	4.8 x 7.6	100
12	15.3 x 24.2	6.0 x 9.5	80

On request the publisher will supply a leaflet with more detailed technical instructions or a TEX macro package for the preparation of manuscripts.

Manuscripts should be sent to one of the series editors or directly to:

Springer-Verlag, Computer Science Editorial III, Tiergartenstr. 17,
D-69121 Heidelberg, Germany

ISBN 3-540-61551-2

9 783540 615514

ISSN 0302-9743